Hindawi
Wireless Communications and Mobile Computing
Volume 2021, Article ID 9926619, 15 pages
https://doi.org/10.1155/2021/9926619

Research Article

Disruption-Free Load Balancing for Aerial Access Network

Na Li,[1] **Yue Zhao**,[2] **Ning Hu**,[3,4] **and Jing Teng**[5]

[1]*Department of Special Medicine, Xiangya 3rd Hospital, Central South University, China*
[2]*Science and Technology on Communication Security Laboratory, China*
[3]*Cyberspace Institute of Advanced Technology, Guangzhou University, China*
[4]*Peng Cheng Laboratory, China*
[5]*School of Control and Computer Engineering, North China Electric Power University, China*

Correspondence should be addressed to Jing Teng; jing.teng@ncepu.edu.cn

Received 3 March 2021; Accepted 11 May 2021; Published 24 May 2021

Academic Editor: Haitao Xu

A fundamental issue of 6G networks with aerial access networks (AAN) as a core component is that user devices will send high-volume traffic via AAN to backend servers. As such, it is critical to load balance such traffic such that it will not cause network congestion or disruption and affect users' experience in 6G networks. Motivated by the success of software-defined networking-based load balancing, this paper proposes a novel system called Tigris, to load balance high-volume AAN traffic in 6G networks. Different from existing load balancing solutions in traditional networks, Tigris tackles the fundamental *disruption-resistant challenge* in 6G networks for avoiding disruption of continuing flows and the *control-path update challenge* for limiting the throughput of updating load balancing instructions. Tigris achieves disruption-free and low-control-path-cost load balancing for AAN traffic by developing an online algorithm to compute disruption-resistant, per-flow load balancing policies and a novel bottom-up algorithm to compile the per-flow policies into a highly compact rule set, which remains disruption-resistant and has a low control-path cost. We use extensive evaluation to demonstrate the efficiency and efficacy of Tigris to achieve zero disruption of continuing AAN flows and an extremely low control-path update overhead, while existing load balancing techniques in traditional networks such as ECMP cause high load variance and disrupt almost 100% continuing AAN flows.

1. Introduction

The next-generation cellular networks (*i.e.*, 6G networks) can interconnect devices in space, air, and ground networks with the help of aerial access networks (AAN) to provide users Internet access with a substantial higher coverage than traditional network technologies (*e.g.*, 5G networks) and hence have drawn much attention from both academia and industry [1–5]. Figure 1 gives an abstract architecture of 6G networks with AAN as a key component. After receiving data traffic from end devices, AAN forwards traffic to edge servers, which process and forward traffics to backend servers.

A fundamental issue of this 6G architecture is that with higher coverage of user devices and more ultra-high-bandwidth, ultra-low-latency applications such as VR/AR and remote medical, AAN needs to deliver high-volume data traffic involving a high number of flows from user devices to the corresponding backend servers via edge and backbone networks. As such, it is critical to load balance such traffic before they enter the backbone networks, such that it does not cause any congestion or network disruption in 6G networks.

Specifically, motivated by the recent success of flexible load balancing using software-defined networking (SDN), *e.g.*, [6–14], in this paper, we investigate the feasibility and benefits of load balancing AAN traffic in 6G network using a software-defined edge (SDE), which we term as SDE-LB. Specifically, continuing the trend of moving from dedicated load balancing appliances to native network switches for load balancing (*e.g.*, [6–8, 12, 15–19]), SDE-LB takes advantage of the flexibility of a logically centralized controller to collect load statistics from both the network and the servers to compute load balancing flow rules and install them on programmable switches (also called *load balancing switches*), whose TCAM flow tables allow high-speed packet processing [20–22] to forward packets to different backend servers, based on the matching results [12, 16, 18, 23, 24].

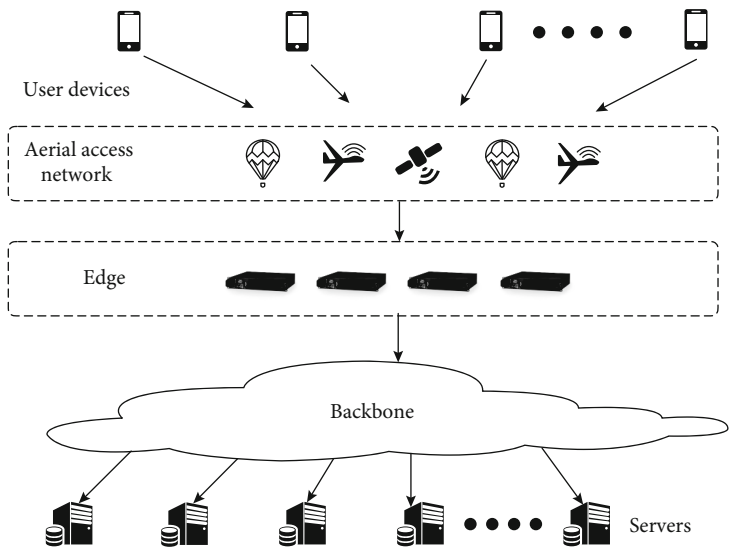

Figure 1: Architecture of 6G network with aerial access network.

Despite the potential of SDE-LB, key challenges remain. In particular, we found that directly using existing load balancing solutions in SDN to SDE-LB encounters substantial challenges under dynamicity, where dynamicity can happen either on the data path or on the control path.

The key dynamicity challenge on the data path is the *disruption-free challenge*. Specifically, as the load of AAN from the flows contained in each flow rule changes over time, imbalance occurs and rebalancing is needed. Although one may apply previous studies [16, 18, 25] to conduct rebalancing, since they do not consider existing assignments of continuing flows from AAN, *i.e.*, flows with open TCP connections, they can result in unacceptable disruptions of continuing flows. Although one may use migration to reduce the impacts of disruption, migration is considered adding substantial system complexity and hence is not preferred by many operators. Another possibility is to install exact machines to pin the assignment of continuing flows, but flow table size constraints make this approach infeasible. For example, one commodity edge server equipped with Dell Z9100 switch has only 2304 flow entries [26], while the number of data flows with different source IP addresses is much larger. Utilizing TCAM wildcard rules to aggregate flow rules may reduce the number of rules (*e.g.*, [16, 18, 21, 24, 27–29]) but would again result in a mass disruption of continuing flows.

The key dynamicity challenge on the control path is the *control-path update challenge*. Specifically, rebalancing AAN traffic load among servers requires an SDE-LB controller to send flow-mod instructions to update flow tables on the load balancing switches. Unfortunately, state-of-the-art SDN controllers have limited throughput in sending flow-mod instructions, putting a limit on control-path update frequencies.

In this paper, we cope with both dynamicity challenges by proposing the first disruption-resistant, low-control-path-cost, dynamic SDE load balancer for AAN traffic in 6G networks. Tigris provides *two key novel insights* for addressing the dynamicity challenges: (1) it shifts a small number of incoming flows among servers to achieve load balancing without disrupting continuing flows and (2) it leverages the small number of shifted flows, the decomposition of aggregation of a large rule set into parallel aggregation of multiple smaller rule sets, and the cached intermediate rule aggregation results from previous time slots to substantially improve the efficiency of flow rule aggregation and reduce the control-path update cost. This work sheds light for future research on system and protocol design in AAN and 6G networks, such as traffic engineering, resource orchestration, and network-application integration [30].

The *main contributions* of this paper are as follows.

(i) We study a fundamental problem for 6G networks with AAN, the disruption-free load balancing problem for AAN traffic, identify the disruption-resistant and the control-path-cost challenges, and design a novel dynamic load balancer at the edge called Tigris to address these challenges

(ii) We design DR-LB, an online algorithm, as the first phase of Tigris to compute per-flow disruption-resistant load balancing policies and prove that DR-LB achieves a competitive ratio of $(2 - 1/N)$ for load balancing, where N is the number of back-end servers

(iii) We develop Tree-Agg, an incremental, bottom-up rule aggregation algorithm as the second phase of Tigris to compile per-flow load balancing policies into a highly compact, disruption-resistant rule set which also has a low control-path update cost

(iv) We conduct extensive evaluations to show that Tigris achieves zero disruption of continuing flows in AAN of 6G networks, a close-to-1 competitive ratio on load balancing, a less than 5% rule update ratio between time slots, and up to 53x flow rule

compression ratio, while other state-of-the-art approaches such as ECMP can cause 2-4x load variance and disrupt almost 100% continuing flows in AAN

The remainder of this paper is organized as follows. We discuss related work on different load balancing approaches in Section 2. We present the system settings and problem definition in Section 3. In Section 4, we propose the Tigris load balancer, introduce its overall workflow and the details of its key components, and discuss its generality and overhead. We evaluate the performance of Tigris with extensive simulation in Section 5 and finally conclude our work in Section 6.

2. Related Work

Data traffic load balancing is a well-studied problem for which not only many algorithms have been designed and thoroughly analyzed [31–33] but also many systems have been developed in supporting scalable, reliable network services [6–8, 12, 15–19]. Modern networks increasingly rely on switches to develop load balancing solutions. In this section, we give a brief review of existing studies on load balancing and discuss the key limitations of these techniques for load balancing traffic from AAN.

2.1. Hash-Based Load Balancing. Among various load balancing systems, hash-based solutions such as Equal-Cost Multipath (ECMP) [25] and Weighted-Cost Multipath (WCMP) [7] are the most widely used ones. ECMP-based systems [25] evenly partition the flow space into hash buckets and use a hardware switch to redirect incoming packets to different software load balancers based on the hash value of the packet header. WCMP-based systems [7] achieve an unevenly weighted partition of the flow space by repeating the same software load balancers as the next hop multiple times in an ECMP group. However, these hash-based solutions split the traffic based on the size of flow space rather than the actual volume of flows, resulting in load imbalance due to the unequally distributed and dynamically changing traffic contribution from different flow spaces. Because ECMP hash functions are usually proprietary [18, 25], users have a limited customization capability in rebalancing to adapt to such dynamic flow statistics.

2.2. SDN Load Balancing. Compared with hash-based load balancing, SDN load balancing is a more powerful and flexible load balancing technique. As such, it is a more suitable solution to AAN load balancing, because balancing the dynamic traffic in AAN and 6G networks requires more flexible load balancing policies. SDN supports the programming of the flow rule table on switches using wildcards and hence enables a more flexible, fine-grained load balancing service [12, 16, 18, 23, 24]. The select group table defined in OpenFlow specification [21] can be used for load sharing, but it requires the controller's guidance to respond to events not detectable by the switch, *e.g.*, server failures. Solutions in [23, 24] send the first packet of every flow back to the controller for calculating the flow rules. Benson et al. [12] integrate the per-flow load balancing rule generation with WCMP to minimize the traffic congestion. Sending the first packet of each flow to the controller would add extra delay for these packets. Kang et al. [18] and Wang et al. [16] study the TCAM size-constrained traffic splitting problem and design an algorithm to generate efficient flow rules to partition the flow space into weighted subspaces for different servers. In addition, TCAM wildcards are also utilized to reduce the number of flow rules used for expressing network policies (*e.g.*, TCAMRazor [27] and CacheFlow [28]), including the load balancing policy.

One important limitation of applying existing SDN load balancing solutions to load balance AAN traffic at an SDE of 6G networks is that they do not consider the key dynamic SDE load balancing challenges, *i.e.*, the *disruption-resistant challenge* and the *control-path update challenge*, leading to the unacceptable disruption of continuing flows and a high control-path update cost. Miao et al. [34] design a stateful data plane load balancer, but it requires the support of expensive, customized programmable hardware. On the contrary, Tigris addresses the above challenges to design the first disruption-resistant, low-control-path-cost SDE load balancer using commodity SDN switches.

3. Dynamic SDE Load Balancing: System Settings and Problem Definition

We consider a dynamic SDE-LB system shown in Figure 2: it provides a service using multiple servers, indexed by $i = 1, 2, \cdots, N$. Clients access the service through a single public IP address, reachable via AAN. For any incoming packet, the load balancing switch at SDE finds the matched flow rule based on its source IP address, rewrites its destination IP addresses to that of the assigned server, and forwards it correspondingly to achieve load balancing. One may extend our system to multiple switches and to match on other fields for more complex load balancing scenarios for better scalability and fault tolerance, but we focus on the single load balancing switch and the source IPv4 address matching for clarity.

We consider that our system operates in discrete time where time is divided into slots, indexed by $t = 1, 2, \cdots, T$. Different time slots can have different duration to support a combination of periodic operations and event-driven operations, where events can include server up/down and the burst of data flows. At the beginning of each slot t, the SDE controller computes a set of load balancing flow rules denoted as $R(t)$ and update the flow table of load balancing switch accordingly.

Given a flow rule r, it has three basic attributes: the set of source IP addresses it matches, the forwarding action, and priority, denoted as r.match, r.act, and r.pri, respectively. Given a packet, its source IP may be matched by multiple rules. In this case, the switch will rewrite its destination IP and forward it following the action of the rule with the highest priority, as specified in OpenFlow specification [21]. We use an attribute named r.load to denote the total estimated *load* of the flows that followed the action of r. Our system allows the load metric to be a generic metric, considering

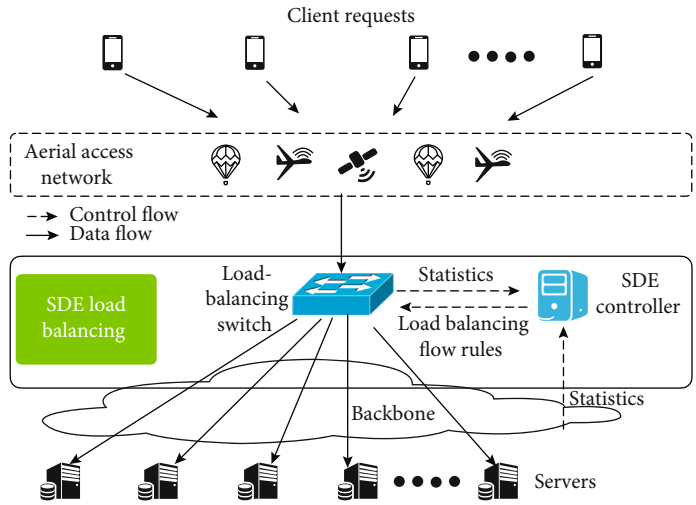

Figure 2: Architecture of SDE load balancing.

factors such as data volume and CPU. The controller collects related data from servers and switches to compute the load metric.

A set of flow rules R can be *aggregated* into a single rule r_a using wildcard. We call R the source rules of r_a, denoted as r_a.src. We also define some parameters to assist rule aggregation. We use r.lev to record the *aggregation level* of rule r, where the first $32 - r$.lev matching bits of r *must not contain any wildcard*. As we will show in Section 4.3, r.lev can be increased even if it fails to be aggregated with another rule.

An efficient SDE-LB system must satisfy multiple constraints, which we separate into two categories: (1) traditional constraints (*i.e.*, table size, full-coverage, and zero-ambiguity) and (2) dynamicity constraints (*i.e.*, disruption resistance and low control-path update).

3.1. Traditional Constraints. The first traditional constraint is the *table size constraint*. Specifically, the set of flow rules installed on the load balancing switch in any time slot must not exceed its flow table capacity C, as expressed in Inequality

$$|R(t)| \le C, \quad \forall t = 1, 2, \cdots, T. \tag{1}$$

Secondly, for any incoming packet, there must be at least one flow rule r that matches its source IP address and does not have punt as its action. This is the *full-coverage constraint*. It ensures that no packet will be punted back to the controller, eliminating the switch-to-controller delay for all incoming data traffic, and is expressed as

$$\left| \cup_{r \in R(t)} r.\text{match} \right| = 2^{32}, \quad \forall t = 1, 2, \cdots, T. \tag{2}$$

Thirdly, for any incoming packet, if there exists more than one flow rule matching its source IP, only one rule's action will be taken. This is the *zero-ambiguity constraint*. We use r.dom to denote the set of source IP addresses that belong to r.match but will not follow r.act and express this constraint as

$$(r_2.\text{match} \in r_1.\text{dom}) \wedge (r_1.\text{match} \in r_2.\text{dom}) = 0, \quad \forall r_1, r_2 \in R(t), t = 1, 2, \cdots, T, \tag{3}$$

where $\wedge$ is the logical conjunction.

3.2. Dynamicity Constraints. Other than the traditional constraints, SDE-LB under dynamicity also introduces additional constraints on both data path and control path. For the data path, any flow whose TCP connections to the servers are currently open should not be shifted to another server unless the current server fails. We call it the *disruption-resistant constraint*. Denoting a continuing flow as f_a and its server time slot t as $f_a.s(t)$, this constraint can be expressed as

$$f_a.s(t) = f_a.s(t-1), \quad \forall f_a, \text{where } f_a.s(t-1) \text{ is in } t, \forall t = 2, 3, \cdots, T. \tag{4}$$

For the control path, the number of rules updated (*i.e.*, deleted or inserted) in each time slot should be within the capacity of the controller, which is called the *control-path update constraint*. Using D to denote this constraint, we have

$$|R(t) \cup R(t-1) - R(t) \cap R(t-1)| \le D, \quad \forall t = 2, 3, \cdots, T. \tag{5}$$

Denoting the data flow forwarded to server i in time slot t as $L_i(t)$ and the number of time slots that server i is working till time slot t as $T_i(t)$, we formulate the *disruption-resistant*, *low-control-path-cost*, *dynamic load balancing* problem as

$$\min \max_i \lim_{T \to \infty} \frac{\sum_{t=1}^{T} L_i(t)}{T_i(T)}, \tag{6}$$

subject to constraints (1), (2), (3), (4), and (5). We prove the NP-hardness of this problem via a transformation from the classic multicore balancing problem [31]. Note that in Equation (6), we use the time-averaged load balancing as the system objective. This is the typical objective of load balancing systems [31–33], and it reflects the requirement of dynamic SDE-LB, *i.e.*, long-term, online load balancing. As we will discuss in Section 4.4, we design the Tigris system to be modular so that it provides users the flexibility to define different load balancing objectives and methods while maintaining the benefits of Tigris, *e.g.*, low-control-path-cost and disruption resistance.

4. Tigris: A Disruption-Resistant, Low-Control-Path-Cost, Dynamic SDE Load Balancer

We now fully specify Tigris, in which a controller computes disruption-resistant load balancing policies and generates compact flow rules with low control-path cost correspondingly. We will start with an overview of Tigris in Section 4.1. Then, we will give the design details of its key components: policy generation in Section 4.2 and rule aggregation in Section 4.3. We also discuss the generalization of Tigris in Section 4.4.

4.1. Overview of Tigris. In practice, Tigris operates in both periodic mode and event-driven mode in response to events such as the up/down of servers and the burst of data flows. For simplicity of presentation, we assume a periodical operation mode, where the controller executes Tigris at the beginning of every slot t. Figure 3 presents the workflow of Tigris. Specifically, each invocation of Tigris can be divided into two phases: (1) policy generation and (2) rule compilation.

4.1.1. Policy Generation. During the policy generation phase, Tigris first collects load statistics and related flow information, *e.g.*, TCP connection status, from each server i, and estimates the *target load* of i, *i.e.*, the estimated load of incoming flows which should be assigned to i in time slot t for load balancing purpose (Step 1). A flow is identified by a TCP/IP 5-tuple. It then uses the DR-LB algorithm to compute a set of disruption-resistant load balancing policies for all servers available in time slot t and generates a set of per-flow rules $R_{\text{pf}}(t)$ to express the computed load balancing policies (Step 2). In particular, DR-LB keeps the forwarding action of continuing flows on available servers and adopts an online, greedy approach to shift incoming flows from overloaded servers to underloaded servers based on the target load of each server and the similarity of source IPs between continuing flows and incoming flows. This design is disruption-resistant. It achieves load balancing by shifting a small number of flows among servers and increases the chance of reusing rule aggregated results from last slot $t-1$ during the rule aggregation phase, substantially increasing the aggregation efficiency and reducing the control-path update cost. And we prove that it achieves a $(2-1/N)$ competitive ratio for Equation (6) in certain cases.

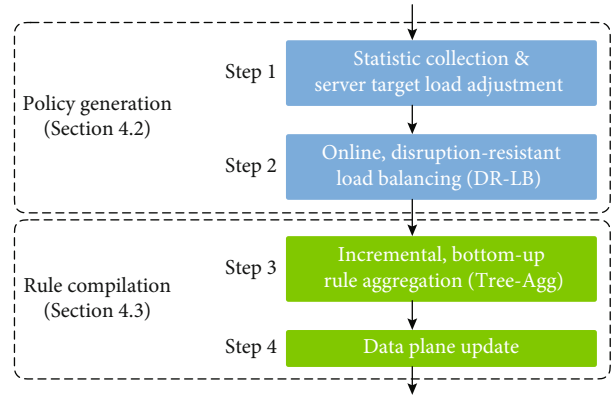

Figure 3: Workflow of Tigris.

4.1.2. Rule Compilation. In the second phase, Tigris adopts a bottom-up rule aggregation algorithm Tree-Agg to iteratively aggregate the large per-flow rule set into a highly compact rule set $R(t)$ along a binary IP tree (Step 3). Tree-Agg has three novel design points: (1) only traversing from IP leaves of shifted flows to the root of the IP tree, (2) decomposing the aggregation of a large rule set into parallel aggregation of multiple smaller sets, and (3) using cached intermediate rule aggregation results from slot $t-1$ during aggregation. These design points substantially increase the efficiency of rule aggregation and yield a compact rule set $R(t)$ with a high similarity with $R(t-1)$, which reduces the control-path cost for updating the data plane. After getting $R(t)$, Tigris updates the data plane by deleting the set of obsolete rules from the switch and installing the set of new rules (Step 4).

4.1.3. Addressing the SDE-LB Constraints. We show in the next subsections that the compact rule set $R(t)$ expresses the same policies as the per-flow rule set $R_{\text{pf}}(t)$ and satisfies the disruption-resistant constraint, the full-coverage constraint, and the zero-ambiguity constraint. Because it is NP-hard to decide if a rule set can be aggregated into a smaller set of a given size [35], Tigris uses a software switch as a safety measure to install extra rules if the compact rule set still exceeds the table size of the hardware switch. In this way, when packets arrive, the hardware switch first tries to find a matching rule. If no matching rule is found, the packet is forwarded to the software switch (*e.g.*, OpenvSwitch) for matching and processing. We show through extensive evaluation in Section 5 that this measure is rarely needed in practice since the aggregated rule set computed by Tigris is highly compact and outperforms the state-of-the-art rule aggregation solution, *i.e.*, TCAM Razor. Furthermore, we also show that the compact rule set computed by Tigris has an extremely low control-path update cost.

4.2. Online, Disruption-Resistant Policy Generation. We now give the details of the policy generation phase of Tigris. It involves two steps: statistics collection and target load adjustment and online, disruption-resistant load balancing, marked as Step 1 and Step 2 in Figure 3, respectively.

Step 1 (statistics collection and target load adjustment). At the beginning of time slot t, Tigris first collects $L^i(t-1)$, the actual load of each server i in slot $t-1$, and related info about its flows, *e.g.*, TCP connection status. In practice, such statistics can be retrieved from the log or the monitor process of the servers. It then estimates the total incoming load for all servers in slot t as $\bar{L_{\text{total}}}(t)$, the load of continuing flows (*i.e.*, flows with open TCP connection) at each server i in slot t as $\bar{L_i^a}(t)$, and the load of incoming flows at each server i in slot t as $\bar{L_i^{ia}}(t)$ when the load balancing policy stays the same in t as in $t-1$, using methods adopted in [21, 28]. Next, Tigris calculates a key metric L_{av}, the "target share" of each available server at each slot till the current time slot t, in

$$L_{\text{av}} = \frac{\sum_{i=1}^{N} \boxtimes \sum_{k=1}^{t-1} \boxtimes L_i(k) + \bar{L_{\text{total}}}(t)}{\sum_{i=1}^{N} \boxtimes T_i(t)}, \tag{7}$$

where $T_i(t)$ is the number of time slots server i is available among all t slots. Note that one can extend it to the case where different servers have different capacities.

With L_{av}, Tigris then computes the *target load* assigned to individual server i which is available in slot t, denoted as $\bar{L_i}(t)$ in

$$\bar{L_i}(t) = L_{\text{av}} \cdot T_i(t) - \sum_{k=1}^{t-1} \boxtimes L_i(k) - \bar{L_i^a}(t). \tag{8}$$

Specifically, according to L_{av}, i should serve $L_{\text{av}} \cdot T_i(t)$ in all slots until t. Since it has already served $\sum_{k=1}^{t-1} \boxtimes L_i(k)$ and will serve $\bar{L_i^a}(t)$ in slot t for continuing flows, these "credits" are deducted. Note that the way Tigris computes $\bar{L_i}(t)$ depends on different load balancing objectives. This paper focuses on Equation (6).

Step 2 (online, disruption-resistant load balancing DR-LB). With the targeting load $\bar{L_i}(t)$ for every server i, we design the DR-LB algorithm, summarized in Algorithm 1, to compute a set of disruption-resistant, per-flow load balancing policies and the corresponding per-flow rule set $R_{\text{pf}}(t)$.

4.2.1. Basic Idea. We design DR-LB as an online algorithm to achieve load balancing by shifting a small number of incoming flows among servers. This design ensures disruption resistance of continuing flows. It also increases the chance of reusing cached aggregation results from the last time slot $t-1$ during the rule aggregation phase (Section 4.3), substantially increasing the aggregation efficiency and reducing the control-path cost to update the flow table on the switch.

At the beginning of the whole system, *i.e.*, $t = 0$, it sets the load balancing policies to evenly divide flows from the whole source IP space to all N servers and generates a per-flow rule set $R_{\text{pf}}(0)$ (Line 1). At each time slot $t = 1, 2, \cdots$, it takes the collected load statistics from Step 1 and $R_{\text{pf}}(t-1)$, the per-flow rule set for slot $t-1$ as input (Line 3). It then iteratively shifts flows from overloaded servers to underloaded servers to achieve the target load for each server (Lines 4-18). During this process, it replaces the per-flow rule for every shifted flow with a new per-flow, eventually getting the new per-flow rule set $R_{\text{pf}}(t)$ (Lines 19-23).

4.2.2. Categorizing Overloaded and Underloaded Servers. DR-LB first categorizes servers into overloaded and underloaded based on their target load (Lines 4-11). Given an available server i at time slot t, it is *overloaded* (*underloaded*) if its target load $\bar{L_i}(t)$ is smaller (larger) than the estimated load of incoming flows when load balancing policies stay the same in slot t as in $t-1$ (Lines 4-8). For any server i that is unavailable in slot t, we set its target load as 0 and consider all the incoming flows destined to i incoming (Lines 9-10). Hence, every unavailable server is an overloaded server (Line 11).

4.2.3. Shifting Incoming Flows from Overloaded Servers to Underloaded Servers. Next, DR-LB adopts a greedy approach to balance the load among servers. For every overloaded server i, it iteratively finds the incoming flow f^* with the largest estimated load and shifts it to an underloaded server j which is the assigned server for the flow with the largest IP prefix match with f^*.srcIP (Lines 14-18). This process stops when i is no longer overloaded or there is no underloaded server (Line 13). The rationale of this approach includes that (1) it achieves the load balancing among servers by shifting a small number of flows among servers and (2) it increases the chance of rule aggregation for the flow rules for shifted data flows. For instance, suppose f^* has the source IP 10.0.0.0 and was forwarded to an overloaded server A. And there exists a flow f' whose source IP is 10.0.0.1 and was forwarded to an underloaded server B. DR-LB will make the load balancing decision to forward f^* to B and generates a flow rule $10.0.0.0 \rightarrow B$ to express this decision. In this way, the rules for f^* and f' can be aggregated into $10.0.0.0\mathbf{x}0000000 * \rightarrow B$ during the rule aggregation process.

During flow shifting, DR-LB also generates the per-flow rule set $R_{\text{pf}}(t)$ for slot t. It initializes $R_{\text{pf}}(t)$ as the per-flow rule set $R_{\text{pf}}(t-1)$ for the last slot $t-1$ (Line 3). For each shifted data flow, DR-LB generates a flow rule r with a priority of 32, a level of 0 (Lines 19-22). r is also assigned a property r.new as 1 to indicate that r represents a newly generated load balancing policy. Then, it finds the flow rule in $R_{\text{pf}}(t)$ who has the same match field as r and replaces it with r (Line 23). In the end, we get the per-flow rule set $R_{\text{pf}}(t)$ representing the load balancing policies for slot t.

4.2.4. Performance Analysis of DR-LB. It is easy to see that $R_{\text{pf}}(t)$ satisfies the disruption-resistant and the full-coverage constraints. We then propose the following proposition on the performance of DR-LB.

Proposition 2. *When all servers are available, repeatedly generating load balancing policies via DR-LB at every time slot achieves a competitive ratio of* $(2 - 1/N)$ *on the objective function in Equation (6).*

$R_{pf}(0)$: a set of per-flow rules which approximately evenly divide the whole IP space to all N servers, where each rule r has $r.pri \leftarrow 32$ and $r.lev \leftarrow 0$
foreach $t \leftarrow 1, 2, \cdots$ **do**
  $S_{ul}, S_{ol} \leftarrow \varnothing$, $R_{pf}(t) \leftarrow R_{pf}(t-1)$.
  foreach *available server* i **do**
    if $L_i(t) < L_i^{ia}(t)$ **then**
      $S_{ol} \leftarrow S_{ol} \cup \{i\}$
    else if $L_i(t) > L_i^{ia}(t)$ **then**
      $S_{ul} \leftarrow S_{ul} \cup \{i\}$
  foreach *unavailable server* i **do**
    $L_i(t) \leftarrow 0$, $L_i^{ia}(t) \leftarrow L_i^{ia}(t) + L_i^{a}(t)$
    $S_{ol} \leftarrow S_{ol} \cup \{i\}$
  foreach $i \in S_{ol}$ **do**
    while $L_i(t) < L_i^{ia}(t)$ and $S_{ul} \neq \varnothing$ **do**
      $f^* \leftarrow_{f \in F_i^{ia}} \{f.load\}$
      $L_j^{ia}(t) \leftarrow L_j^{ia}(t) + f^*.load$
      $L_i^{ia}(t) \leftarrow L_i^{ia}(t) - f^*.load$
      if $L_j(t) \leq L_j^{ia}(t)$ **then**
      $S_{ul} \leftarrow S_{ul} - \{j\}$
      $r.match \leftarrow f^*.srcIP$
      $r.act \leftarrow j$, where $j \in S_{ul}$ and j is the assigned server of the flow with the longest IP prefix match of $f^*.srcIP$.
      $r.load \leftarrow f^*.load$, $r.pri \leftarrow 32$, $r.lev \leftarrow 0$, $r.new \leftarrow 1$
      $r.src \leftarrow \varnothing$, $r.\mathrm{dom} \leftarrow \varnothing$
      $replaceRule(R_{pf}(t), r.match, r)$
  return $R_{pf}(t)$

Algorithm 1: DR-LB: an online, disruption-resistant load balancing algorithm.

Proof. When all servers are available, any instance of our load balancing problem can be transformed into an instance of the classic load balancing problem, which is aimed at minimizing the maximal task completion time across servers. And in the transformed instance, all tasks arrive at the same time. With this transformation, we can see that the load balancing policy computed by DR-LB is in the set of all possible policies computed by greedy Graham's algorithm [31]. Hence, the competitive ratio of $(2 - 1/N)$ of DR-LB for Equation (6) is a direct result of applying the technique in proving the competitive ratio of Graham's algorithm.

Although the complexity of DR-LB depends on the number of arriving flows, which can be large due to mouse flows, in practice, we can only focus on elephant flows for computing the load balancing decisions and assign mouse flows evenly across different servers.

4.3. Incremental, Bottom-Up Rule Compilation. We now give the details of the rule compilation phase of Tigris. It involves two steps: incremental, bottom-up rule aggregation and data plane update, marked as Step 3 and Step 4 in Figure 3, respectively.

Step 3 (incremental, bottom-up rule aggregation (Tree-Agg)). Directly installing the initial disruption-resistant per-flow rule set $R_{\mathrm{pf}}(t)$ computed by DR-LB into the load balancer switch is infeasible because the size of $R_{\mathrm{pf}}(t)$ is much larger than the size of the switch flow table, tens of thousands vs. a few hundreds or thousands. Next, we develop the Tree-Agg algorithm which adopts a bottom-up approach to aggregate $R_{\mathrm{pf}}(t)$ into a highly compact flow rule set $R(t)$ expressing the same load balancing policies as $R_{\mathrm{pf}}(t)$ does.

4.3.1. Basic Idea. We design Tree-Agg to iteratively aggregate rules in $R_{\mathrm{pf}}(t)$ along the 32-level binary tree for IPv4 addresses, starting from the leaf nodes, *i.e.*, level 0. At a first glance, this approach is impractical since the complete IPv4 address tree has 2^{32} leaf nodes, an extremely large number to traverse. However, we propose three novel design points in Tree-Agg. First, $R_{\mathrm{pf}}(t)$ and $R_{\mathrm{pf}}(t-1)$ typically only have small different flow rules, *i.e.*, rules where r.new is set to 1 in DR-LB, due to the online feature of the DR-LB. Hence, at each time slot t, Tree-Agg only needs to traverse from the leaves representing the new rules in $R_{\mathrm{pf}}(t)$ to the root of the binary tree. Secondly, Tree-Agg decomposes the aggregation of a large rule set into parallel aggregation on multiple disjoint subsets. Thirdly, Tree-Agg uses the cached intermediate aggregation results from the last time slot $t-1$ to aggregate with the new rules from $R_{\mathrm{pf}}(t)$. These design points have two major benefits: (1) they substantially reduce the traverse and aggregation scale, hence significantly increasing the efficiency of aggregation, and (2) they reuse the cached intermediate aggregation results from $t-1$ at maximum to ensure that $R(t)$ would have a high similarity with $R(t-1)$, the

installed rule set for time slot $t-1$, hence substantially reducing the control-path cost to update the flow table of the switch.

4.3.2. Rule Aggregation Operations. Before we present the details of the Tree-Agg algorithm, we first introduce some basic operations for rule aggregation, which are summarized in Algorithm 2. To aggregate a set of flow rules R into one rule r_a, we first need to decide the matching field of r_a. This is computed in the *matchFieldAgg* function, where a wildcard $*$ is used at every matching bit that is not the same across every rule $r \in R$, and an exact bit 0 or 1 is used otherwise (Lines 1-7). In this way, it is guaranteed that every flow that matches at least one rule in R will also match r_a.

After computing the matching field of r_a, we use the *ruleAgg* function to set other properties of r_a, including action and priority. Note that during the whole rule aggregation process, we only consider aggregating rules that have the same forwarding action to avoid altering the original forwarding policies (Line 10). And the priority of the aggregated rule r_a is set as the number of exact matching bits in its matching field (Line 11). The aggregated source src and r_a.load are set based on the definition in Section 3, and the aggregation level of r_a is also increased by 1 (Lines 12-15).

Given an aggregated rule r_a and a set of rules R', we can also compute the set of rules that direct a subset of the flow space of r_a to a different server, i.e., r_a.dom, using the *findDom* function. This process is straightforward by checking the matching field, the action, and the priority of every rule $r \in R'$ (Lines 19-22). In addition, as we will show next in the Tree-Agg algorithm, if an aggregated rule r_a cannot be inserted in the flow rule set due to the violation of the no-ambiguity constraint, the aggregation level of every rule r in r_a.src still needs to be increased by 1, by using the *ruleLevUp* function.

4.3.3. Bottom-Up Rule Aggregation along an IP Tree. Having introduced the basic operations for aggregating rules, we now present the details of the Tree-Agg algorithm. Tree-Agg iteratively aggregates rules in $R_{\text{pf}}(t)$ along the 32-level binary tree for IPv4 addresses, starting from the leaf nodes, i.e., $l = 0$. As stated earlier, Tree-Agg is an incremental aggregation algorithm. At each time slot, t, it only traverses from the leaves representing the new rules in $R_{\text{pf}}(t)$, *i.e.*, rules with $r.\text{new} = 1$, to the root of the binary tree. During the traverse, it uses $R_{\text{cache}}(t-1)$, the cached intermediate aggregation results from last slot $t-1$, to aggregate with the new rules from $R_{\text{pf}}(t)$.

The pseudocode of Tree-Agg is shown in Algorithm 3. In the beginning, we construct a rule set R_c by adding all newly generated rules in $R_{\text{pf}}(t)$ (Lines 2-4). We use this initialization of R_c to ensure that the rule aggregation process only traverses from the leaves representing the new rules to the root of the binary IP tree. And because the load balancing policies for time slot $t-1$ are different from slot t, intermediate aggregated rules from $R_{\text{cache}}(t-1)$ that conflict with the load balancing policies of slot t need to be removed (Line 5). For instance, suppose a rule $r : 10.0.0.1 \rightarrow A$ with $r.\text{lev} = 0$ is in $R_{\text{cache}}(t-1)$ and a rule $r' : 10.0.0.1 \rightarrow B$ with $r.\text{lev} = 0$ is in R_c. Because r and r' have the same location on the IP tree, *i.e.*, the same leaf node, r needs to be removed from $R_{\text{cache}}(t-1)$. And we define the function removeRule$(R_{\text{cache}}(t-1), R_c, k)$ to remove all the rules with $r.\text{lev} = k$ that have the same location as rules in R_c in the IP tree.

4.3.4. Decomposition of Rule Set into Disjoint Subsets Using 3-Node Subtree Representation. In each iteration of the main loop (Line 6-49), we aggregate the set of flow rules at the current level l of the IP tree into a more compact set and send it to the next level. In particular, every time, we randomly select a remaining flow rule r_0 from the current rule set R_c (Line 9). And we define the getCache$(R_{\text{cache}}(t-1) \cup R_c, r_0, k)$ function to return all the rules whose first k matching bits are the same as r_0 but the $k+1$th bit is different, from the union of R_c and $R_{\text{cache}}(t-1)$. Using this function, we construct two subsets R_1 and R_2 and remove their overlapping rules from R_c and $R_{\text{cache}}(t-1)$ (Lines 10-12). We can view R_1 and R_2 as two nodes with the same parent in the IP tree. Figure 4(a) gives an example of this subtree. Denoting the aggregated rule set of R_1 and R_2 as R_t and placing it in the parent node of this 3-node subtree, we can see that the aggregation process in each iteration of the main loop can be decomposed into the aggregation process of multiple 3-node subtrees. We prove the following property on the IP tree.

Proposition 4. *If R_1^a and R_1^b are two rule sets belonging to different 3-node subtrees on the same level of the IP tree, $\forall r_1^a \in R_1^a$ and $r_1^b \in R_1^b$, $r_1^a.\text{dom} \cap r_1^b.\text{dom} = \varnothing$.*

Proof. For any $r_1^a \in R_1^a$ and $r_1^b \in R_1^b$, they only share the same first $(31 - r_1^a.\text{lev})$ bits and differs on the $(32 - r_1^a.\text{lev})$. And we see that $r_1^a.\text{dom}$ and $r_2^a.\text{dom}$ only contain rules with lower levels. Therefore, it is impossible to have a rule in $r_1^a.\text{dom} \cap r_2^a.\text{dom}$ to intersect the flow space of both r_1 and r_2.

With this proposition, the aggregation of each subtree on the same level of the IP tree can be performed in parallel without violating the no-ambiguity constraint. Hence, such a decomposition substantially reduces the problem scale and improves the efficiency of rule aggregation. Next, we describe the aggregation process for this 3-node subtree.

4.3.5. Aggregating a 3-Node Subtree When R_2 Is Empty. If the set R_2 is empty, i.e., R_1 has no sibling with the same parent in the IP tree, it means that there are no flows with nonzero estimated load coming to the load balancing switch in the next time slot. In this case, we generate a new set of aggregated rules by changing the $(32-l)$th bit of the matching field of all rules in R_1 to the wildcard (Lines 14-15). It is straightforward that the newly generated aggregated rules do not increase the size of the rule set, cause no violation of the no-ambiguity constraint, and help ensure the full-coverage constraint by covering the flow space that has zero expected loads. As a result, it is a successful aggregation and we insert the updated R_1 with an increased level into the aggregated rule set R_t (Line 16).

```
1  Function matchFieldAgg(R)
2    forj ← 1 to 32do
3      if∀r ∈ R, r.match(j) is the same then
4        r_a.match(j) = r.match(j);
5      else
6        r_a.match(j) = ∗;
7    returnr_a.match;
8  Function ruleAgg(R)
9    r_a.match ← matchFieldAgg(R)
10   r_a.act ← r.act, ∀r ∈ R
11   r_a.pri ← number of exact bits in (r_a.match)
12   r_a.src ← R
13   r_a.load ← ∑_{r∈R} r.load
14   r_a.lev ← r.lev + 1, ∀r ∈ R
15   returnr_a;
16 Functionfind Dom(r_o, R′)
17   r_o.dom ← ∅
18   foreachr ∈ R′ do
19     ifr_o.match ∩ r.match ≠ ∅ and r_o.act ≠ r.act and r_o.pri ≤ r.prithen
20       r_o.dom ← r_o.dom ∪ {r}
21   returnr_o.dom
22 Functionrule LevUp(R)
23   foreachr ∈ Rdo
24     r.lev ← r.lev + 1;
25   returnR
```

Algorithm 2: Operations related to flow rule aggregation.

4.3.6. Aggregating a 3-Node Subtree When R_2 Is Nonempty. If the set R_2 is not empty, we move on to aggregate rules with the same forwarding server in R_1 and R_2 (Lines 18-45). To do this, we randomly select r_1 from R_1 and r_2 from R_2 whose forwarding actions are the same and generate the aggregated rule r^* (Line 22). Note that in Tree-Agg, we select r_1 and r_2 from different sets. This is because we can prove.

Proposition 6. *Given any two rules r' and r'' both from R_1 or R_2 whose forwarding actions are the same, they cannot be aggregated to reduce the number of rules without violating the nonambiguity constraint.*

Proof. Without loss of generality, assume that there exist such two rules r' and r'' with the same forwarding action and both from R_1. If they can be aggregated to reduce the number of flow rules, they should have been done during the aggregation of the 3-node subtree rooted at $R_1 = \emptyset$. The only reason they are not aggregated is that they cannot reduce the number of flow rules, i.e., aggregating them requires extra rules to eliminate ambiguity. So, at R_1 itself, they should not be aggregated. Now, assume that there is another rule $r \in R_2$ that shares the same forwarding action. Aggregating r', r'', and r together would cause the same issue because r only shares the same first $(31 - r.\text{lev})$ bits with r' and r''. Hence, r' and r'' cannot be aggregated to decrease the number of flow rules.

After generating r^*, we then compute r^*.dom using the *findDom* function defined in Algorithm 2 (Lines 23-24). Leveraging Proposition 4, we only search in the union of R_1, R_2, and R_t instead of the whole rule set R_c to find r^*.dom efficiently. Having computed r^*.dom, we can check if directly inserting r^* will lead to any violation of no-ambiguity constraint. To this end, we first check if there exists a rule r from $R_1 \cup R_2$ which violates this constraint with r^* and r^*.src. If so, r^* cannot be inserted into the aggregated rule set since such ambiguity cannot be removed even if r is later aggregated with another rule (Lines 25-26). If such an r does not exist, we then check if any ambiguity will happen between r^* and other newly aggregated rules $r^\diamond$ in R_t (Line 27) and take different actions in different cases.

Case 1. If such a rule $r^\diamond$ does not exist, r^* can be directly inserted into R_t as no ambiguity will be introduced by this insertion (Lines 28-31).

Case 2. If there are more than two such $r^\diamond$ rules, we do not insert r^* into R_t because at least 2 rules in R_t have to be unaggregated to avoid ambiguity, which would increase the size of the rule set (Lines 33-34).

Case 3. If there is only one such $r^\diamond$ from R_t, we compare the size of $r^\diamond$.dom and r^*.dom and only keep the one with a smaller set of flow-space intersection rules (Lines 35-39). The rationale of this strategy is that an aggregated rule with a smaller dom set would have a smaller chance to cause ambiguity with future aggregations in R_1 and R_2.

If the aggregated rule r^* eventually cannot be inserted into the aggregated rule set, we move on to select the next

```
1  l ← 0, R_c, R_cache(t) ← ∅
2  foreach r ∈ R_pf(t) do
3     if r.new is 1 then
4        R_c ← R_c ∪ {r}
5  removeRule(R_cache(t − 1), r, 0)
6  while l < 32 do
7     R_new ← ∅
8     while R_c ≠ ∅ do
9        Randomly select one rule r_0 ∈ R_c
10       R_1 ← {r_0} ∪ getCache(R_cache(t − 1) ∪ R_c, r_0, 32 − l), R_2 ← getCache(R_cache(t − 1) ∪ R_c, r_0, 31 − l)
11       R_c ← R_c \ R_1 \ R_2, R_t ← ∅
12       R_cache(t − 1) ← R_cache(t − 1) \ R_1 \ R_2
13       if R_2 == ∅ then
14          foreach r ∈ R_1 do
15             r.match(32 − l) ← ∗, r.pri ← r.pri − 1
16          ruleLevUp(R_1), R_t ← R_1
17       else
18          foreach server i do
19             foreach r_1 ∈ R_1 where r_1.act == i do
20                tmp ← 0
21                foreach r_2 ∈ R_2 where r_2.act == i do
22                   r* ← ruleAgg({r_1, r_2})
23                   r*.dom ← findDom(r*, R_1 ∪ R_2 ∪ R_t})
24                   R_r^c1 ← {r | r ∈ R_1 ∪ R_2, r ∈ r*.dom and r*.src ∩ r.dom ≠ ∅}
25                   if R_r^c1 ≠ ∅ then
26                      continue
27                   R_r^c2 ← {r° | r° ∈ R_t, r° ⊂ r*.dom and r* ⊂ r°.dom}
28                   if R_r^c2 == ∅ then
29                      R_1 ← R_1 \ {r_1}, R_2 ← R_2 \ {r_2}
30                      R_t ← R_t ∪ {r*}, tmp ← 1
31                      break
32                   else
33                      if |R_r^c| >1
34                         continue
35                      else if |r°.dom | >|r*.dom | then
36                         R_1 ← R_1 \ {r_1}
37                         R_2 ← R_2 \ {r_2}, tmp ← 1
38                         R_t ← R_t ∪ {r*} ∪ ruleLevUp(r°.src) \ {r°}
39                         break
40                if tmp ← 0 then
41                   r_1.lev ← r_1.lev + 1
42                   R_1 ← R_1 \ {r_1}, R_t ← R_t ∪ {r_1}
43             foreach r_2 ∈ R_2 where r_2.act == i do
44                r_2.lev ← r_2.lev + 1
45                R_2 ← R_2 \ {r_2}, R_t ← R_t ∪ {r_2}
46       R_new ← R_new ∪ R_t
47    l ← l + 1
48    removeRule(R_cache(t − 1), R_new, l)
49    R_c ← R_new, R_cache(t) ← R_cache(t) ∪ R_new
50 return R_c as R(t)
```

Algorithm 3: Tree-Agg ($R_{\text{pf}}(t)$): a bottom-up flow rule aggregation algorithm.

pair of rules from R_1 and R_2 for aggregation trial. For any rules $r_1 \in R_1$ and $r_2 \in R_2$ that has failed all possible aggregations, we increase their aggregated level by 1 and insert them directly into R_t (Lines 40-45). The aggregation process of a 3-node subtree stops when both R_1 and R_2 are empty. We insert the resulting aggregated rule set R_t for this subtree into a temporary set R_{new} and move on to select the next 3-node subtree for aggregation. The aggregation process for the current level l stops when every nonempty 3-node subtree on this level has been aggregated. And we then repeat the whole aggregation process for the next level on the complete aggregated set R_{new} (Lines 46-49) until we reach the root of the IP tree,

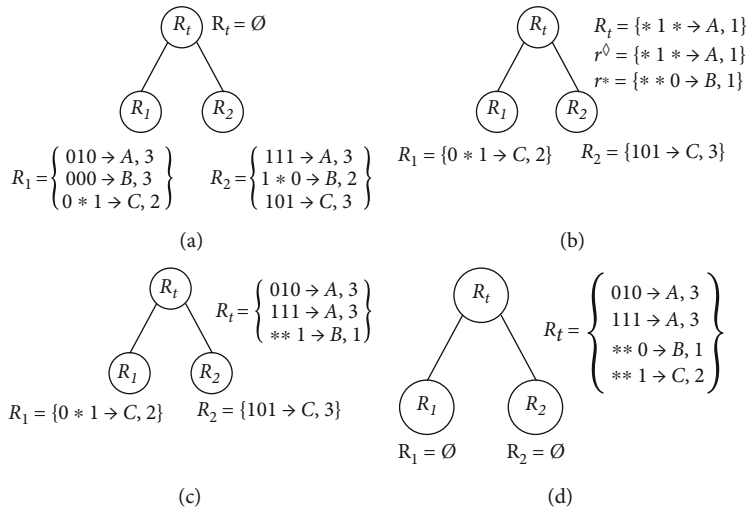

FIGURE 4: An example for aggregating a 3-node subtree: (a) original R_1, R_2, and R_t; (b) ambiguity conflict between r^* and $r^{\diamond}$; (c) $r^{\diamond}$ is unaggregated to satisfy no-ambiguity constraint; (d) the optimal R_t.

i.e., $l = 31$. Note that before each iteration of the main loop, the *removeRule* function is invoked to remove obsolete, conflicting rules from $R_{\text{cache}}(t-1)$ (Line 48). And we also cache R_{new} to assist the future rule aggregation for time slot $t+1$ (Line 49). In the end, Tree-Agg returns R_c as the compact rule set $R(t)$ (Line 50).

4.3.7. An Example. We use the example in Figure 4 to illustrate the whole flow rule aggregation process on a subtree. The original 3-node subtree is shown in Figure 4(a), and we omit the first 29 bits for simplicity. We start by aggregating the two rules forwarding to server A and get $*1* -> A$ with priority 1. Because this rule does not cause any ambiguity violation with either $R_1 \cup R_2$ or R_t, we insert it into R_t in Figure 4(b). Next, we generate another aggregate rule for server B, i.e., $**0 -> B$ with priority 1. Though this rule does not cause any ambiguity violation with $R_1 \cup R_2$, it conflicts with the newly inserted rule $*1* -> A$ in R_t. To decide which rule to keep, we compare the size of their dom set. $*1* -> A$.dom contains both $**0 -> B$ and $0*1 -> C$ from R_1, while $**0 -> B$.dom only contains $*1* -> A$. Therefore, we keep the latter to reduce the probability of ambiguity conflict with future aggregated rules, and unaggregate $*1* -> A$ back to its sources, and get the new R_t shown in Figure 4(c). Last, we generate an aggregate rule for server C and found that it has no ambiguity conflict with other rules. In this way, we get the minimal aggregated rule set R_t with only 5 rules in Figure 4(d). Readers may find that if we keep $*1* -> A$ in R_t, the minimal size of R_t will increase to 6, causing unnecessary waste of limited flow rule space.

4.3.8. Performance Analysis of Tree-Agg. From the previous propositions, we see that Tree-Agg does not change the action of any rule in $R_{\text{pf}}(t)$ when compressing it into $R(t)$. Hence, $R(t)$ expresses the same load balancing policies as $R_{\text{pf}}(t)$ does, *i.e.*, it satisfies the disruption-resistant and the full-coverage constraints. And Propositions 4 and 6 ensure that $R(t)$ satisfies the zero-ambiguity constraint. One may notice that this algorithm has a polynomial complexity of the number of shift flows. However, because the number of the shifted flow is usually small, the decomposition of flow rule aggregation and the cached intermediate rule aggregation results from slot t ensure that Tree-Agg is computationally efficient and that $R(t)$ has a high similarity to $R(t-1)$, substantially reducing the control-path update cost of Tigris.

Step 4 (data plane update). After computing the compact rule set $R(t)$, Tigris deletes the set of obsolete rules $R(t-1) \setminus R(t) \cap R(t-1)$ from the switch and then installs the set of new rules $R(t) \setminus R(t-1)$. It is proved NP-hard to decide if a given set of flow rules can be aggregated into a smaller set of a given size [35]. Hence, Tigris uses a software switch as a safety measure to install extra rules if the compact rule set still exceeds the flow table size of the hardware switch. As we will show in Section 5, however, it is rarely needed in practice since $R(t)$ is highly compact. And we will also show that $R(t)$ is highly similar to $R(t-1)$, *i.e.*, a low rule update ratio, yielding an extremely low control-path update cost.

4.4. Generalization of Tigris

4.4.1. Supporting Heterogeneous Servers and Other Load Balancing Objectives. For simplicity, we assume identical server machines in this paper, and the DR-LB algorithm makes online, disruption-resistant load balancing policies that achieve a good competitive ratio on the load balancing objective in Equation (6). However, Tigris can also be applied to scenarios where servers have different computation resources, e.g., CPU and memory. This is because Tigris adopts a modular design which separates the load balancing decision process from the rule compression process. With this design, users have the flexibility to define and implement different load balancing algorithms, while still leveraging the high rule set compressing capability of Tree-Agg.

4.4.2. Prefix vs. Suffix and Per-Flow Load Balancing vs. Per-Network Load Balancing. Though in the Tree-Agg algorithm we assume an aggregation process based on the IP prefix, it is

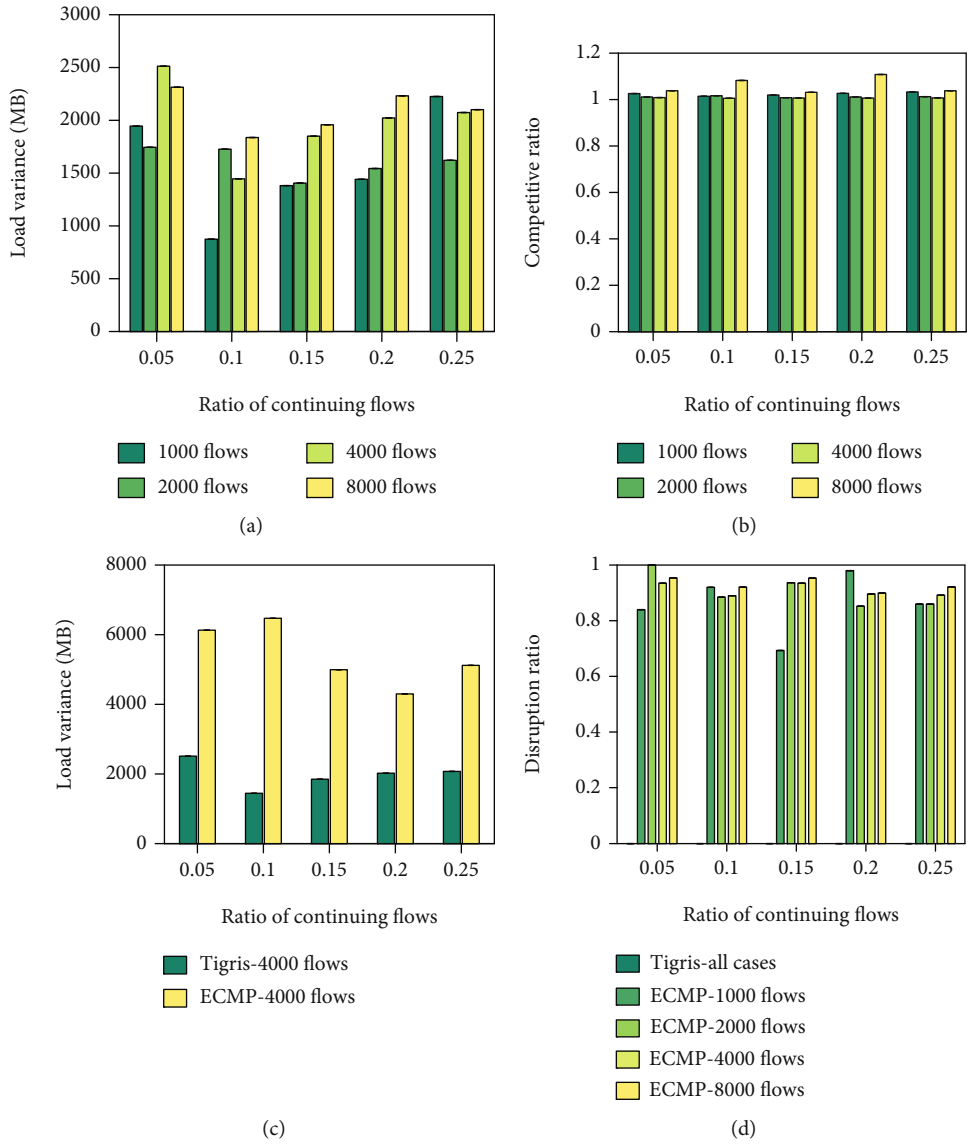

Figure 5: Load balancing performance of Tigris: (a) load variance of Tigris; (b) competitive ratio of Tigris; (c) load variance of Tigris and ECMP with $M = 4000$ flows; (d) disruption ratio.

straightforward to apply it for suffix-based aggregation. In addition, Tigris supports not only per-flow load balancing but also per-network load balancing. In the latter case, the DR-LB algorithm can make load balancing decisions for each network, *i.e.*, the traffic from the same network will be grouped and forwarded to the same server. With this type of load balancing decision, the Tree-Agg algorithm starts the aggregation from the network level, instead of from the leaf level of the IP tree. The computation overhead for per-network load balancing hence can be substantially reduced, with the trade-off of fine-grained load balancing policies.

5. Evaluations

We implement a prototype of Tigris and carry out extensive simulations to evaluate the efficiency of Tigris in achieving size-constrained, disruption-resistant load balancing. The evaluation is performed on a MacBook Pro with 4 2.2 GHz Intel i7 cores and 16 GB memory.

5.1. Methodology. In our evaluations, we assume the setting of Figure 1, where a load balancer switch direct clients' requests to a set of N servers. We generate M flows for the slot. Each flow has a source IP address chosen uniform randomly in the whole 2^{32} IP address space. Among the M flows, a ratio of α flows are considered continuing flows with randomly chosen forwarding servers in the previous slot and hence cannot be disrupted (*i.e.*, cannot be redirected to another server). We assume that the load of each flow is its traffic volume and the volume is uniformly distributed between 1 MB and 1000 MB. We evaluate different settings of $M = 1000, 2000, 4000$, and 8000 flows, $N = 10, 20$, and 30 servers, and $\alpha = 0.05, 0.1, 0.15, 0.2$, and 0.25 ratio of continuing flows. We compare Tigris with the state-of-the-art ECMP [25] system on the following load balancing metrics:

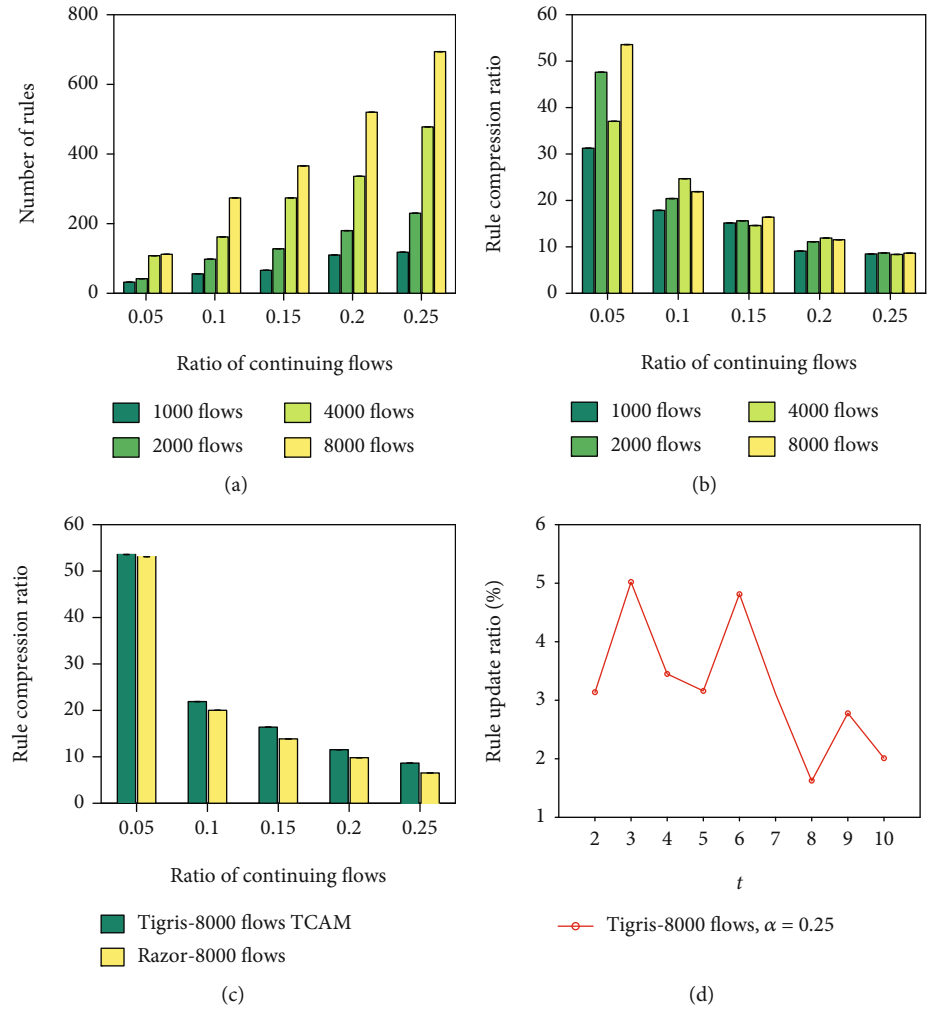

FIGURE 6: Rule aggregation performance of Tigris: (a) aggregated rule set size of Tigris; (b) rule compression ratio of Tigris; (c) rule compression ratio of Tigris and TCAM Razor with 8000 flows; (d) rule update ratio of Tigris with 8000 flows and $\alpha = 0.25$.

(i) Load variance: the standard derivation of servers' load over the load balancing target V_{av}

(ii) Disruption ratio: the percentage of disrupted continuing flows

We also compare Tigris with the state-of-the-art TCAM Razor system [27] on the following rule aggregation metrics:

(i) Rule compression ratio: the ratio of the size of per-flow rule set generated by DR-LB and that of the aggregated rule set computed by Tree-Agg

(ii) Rule set size: the size of the compact flow rule set computed by the Tigris

For these metrics, we run Tigris for $T = 10$ time slots and summarize the average results. In addition, we also study the following metric to measure the control-path cost of Tigris:

(i) Rule update ratio: the ratio of compact flow rules in slot t that are different from slot $t - 1$

5.2. Results. Due to the limited space, we only present the results with $N = 30$, which are more representative with a larger load balancing solution space and a more stringent requirement on rule aggregation, and omit the similar results of $N = 10$ and 20 servers. Figure 5 summarizes the load balancing performance of Tigris under different numbers of flows and ratios of continuing flows. From Figure 5(a), we see that Tigris achieves a stable, small load variance, i.e., between 1000 MB and 2500 MB, and the competitive ratio of the Tigris algorithm in all settings is close to 1, as shown in Figure 5(b). This is consistent with our theoretical finding in Proposition 2. Taking the case of 8000 flows as an example, we also compare the load variance of Tigris with that of ECMP in Figure 5(c). We see that the load variance of ECMP is between 2-4x higher than Tigris. These observations demonstrate the cumbersomeness of hash-based load balancing solutions in adapting to the dynamics of flow statistics, the necessity for developing a per-flow-based load balancing system, and the efficiency of Tigris in generating dynamic load balancing policies. We then plot the ratio of disrupted flows of ECMP and Tigris in Figure 5(d). We observe that Tigris

achieves a 0 disruption ratio of continuing flows in all cases of evaluations, while ECMP has a close-to-100% disruption ratio of continuing flows in all cases. This is because continuing flows are randomly distributed across all flows with random current servers, but ECMP simply divides the flow space evenly to different servers. This huge difference in the disruption ratio between Tigris and ECMP demonstrates the efficacy of Tigris for disruption-resistant load balancing.

We then study the capability of Tigris in generating highly compact flow rule sets in Figure 6. We see from Figure 6(a) that in most cases, Tigris is able to compute an aggregated rule set of less than 400 rules. Even in the worst case with 2000 continuing flows and a total of 8000 flows, Tigris yields an aggregated rule set of less than 800 rules. Figure 6(b) shows that Tigris achieves a high rule compression ratio, i.e., between 8 and 52. These observations show that Tigris is capable of computing a highly compact flow rule set that fits into typical commodity switches without the need to truncate any rules and hence achieves efficient utilization of the limited TCAM resource on commodity switches. Taking the case of 8000 flows as an example, we also compare the compression ratio of Tigris with that of TCAM Razor in Figure 6(c). We see that when the ratio of continuing flows is small, *i.e.*, 0.05, Tigris and TCAM Razor give almost the same compression ratio. As the ratio of continuing flows increases, Tigris outperforms TCAM Razor by yielding a higher rule compression ratio. This is because the design of Tigris shifts a small number of flows during rebalancing and leverages the cached rule aggregation results from the previous time slot for better rule aggregation performance, while TCAM Razor has to start from the per-flow rule set for a clean-slate aggregation. Furthermore, we plot the rule update ratio of Tigris over from $t = 2$ to $t = 10$ for the setting of $M = 8000$ flows and $\alpha = 0.25$ in Figure 6(d). We observe that Tigris yields a less than 5% rule update ratio, implying an extremely low control-path update cost of Tigris.

In summary, we demonstrate the efficiency of Tigris in providing disruption-resistant, low-control-path-cost, dynamic load balancing service. Results show that Tigris achieves zero disruption of continuing flows and a close-to-1 competitive ratio on load balancing objective. It also has a higher flow rule compression ratio, *i.e.*, up to 53x and higher than TCAM Razor, with an extremely low control-path update cost, while the state-of-the-art hash-based solution ECMP can cause 2-4x load variance and disrupt almost 100% continuing flows.

6. Conclusion and Future Work

In this paper, we explored the key challenges for dynamic SDE-LB of AAN traffic in 6G networks, *i.e.*, the disruption-resistant challenge and the control-path update challenge. We address these challenges and design Tigris, the first disruption-resistant, low-control-path-cost dynamic SDE load balancer. Tigris computes disruption-resistant load balancing policies on a per-IP basis and transforms the original large flow rule set into a highly compact rule set with a small size and a low control-path cost, while remaining disruption-resistant. Evaluation results show that Tigris simultaneously achieves a close-to-optimal load balancing performance, a high rule compression ratio, a low control-path update cost, and zero disruption of continuing flows. This work sheds light on future research in traffic management in AAN and 6G networks, such as traffic engineering, resource orchestration, and application-aware networking.

Data Availability

The synthesized and real evaluation data used to support the findings of this study have not been made available because it is proprietary due to nondisclosure agreements with research partners.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research is supported in part by NSFC grants #61976064 and #5207050731.

References

[1] N. Tafintsev, D. Moltchanov, M. Gerasimenko et al., "Aerial access and backhaul in mmwave b5g systems: performance dynamics and optimization," *IEEE Communications Magazine*, vol. 58, no. 2, pp. 93–99, 2020.

[2] N.-N. Dao, Q.-V. Pham, N. H. Tu et al., "Survey on aerial radio access networks:toward a comprehensive 6g access infrastructure," 2021, https://arxiv.org/abs/2102.07087.

[3] H. Zhang, L. Song, and Z. Han, *Unmanned Aerial Vehicle Applications Over Cellular Networks for 5G and Beyond*, Springer, 2020.

[4] T. Qi, W. Feng, and Y. Wang, "Outage performance of non-orthogonal multiple access based unmanned aerial vehicles satellite networks," *China Communications*, vol. 15, no. 5, pp. 1–8, 2018.

[5] Y. Cai, F. R. Yu, J. Li, Y. Zhou, and L. Lamont, "Medium access control for unmanned aerial vehicle (uav) ad-hoc networks with fullduplex radios and multipacket reception capability," *IEEE Transactions on Vehicular Technology*, vol. 62, no. 1, pp. 390–394, 2012.

[6] A. Greenberg, J. R. Hamilton, N. Jain et al., "Vl2: a scalable and flexible data center network," in *ACM SIGCOMM 2009*, New York, NY, USA, 2009.

[7] J. Zhou, M. Tewari, M. Zhu et al., "Wcmp: weighted cost multipathing for improved fairness in data centers," in *ACM EuroSys 2014*, New York, NY, USA, 2004.

[8] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *ACM SIGCOMM 2008*, New York, NY, USA, 2008.

[9] J. Cao, R. Xia, P. Yang et al., "Per-packet load-balanced, low-latency routing for clos-based data center networks," in *ACM CoNEXT'13*, New York, NY, USA, 2013.

[10] J. Guo, F. Liu, X. Huang et al., "On efficient bandwidth allocation for traffic variability in datacenters," in *IEEE INFOCOM*, pp. 1572–1580, Toronto, ON, Canada, 2014.

[11] F. P. Tso, K. Oikonomou, E. Kavvadia, and D. P. Pezaros, "Scalable traffic-aware virtual machine management for cloud

data centers," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, pp. 238–247, Madrid, Spain, 2014.

[12] T. Benson, A. Anand, A. Akella, and M. Zhang, "Microte: fne grained traffic engineering for data centers," in *ACM CoNEXT 2011*, New York, NY, USA, 2011.

[13] L. Rao, X. Liu, L. Xie, and W. Liu, "Minimizing electricity cost: optimization of distributed internet data centers in a multi-electricitymarket environment," in *2010 Proceedings IEEE INFOCOM*, pp. 1–9, San Diego, CA, USA, 2010.

[14] Z. Liu, M. Lin, A. Wierman, S. Low, and L. L. Andrew, "Greening geographical load balancing," *IEEE/ACM Transactions on Networking*, vol. 23, no. 2, pp. 657–671, 2015.

[15] P. Patel, D. Bansal, L. Yuan et al., "Ananta: cloud scale load balancing," in *ACM SIGCOMM 2013*, New York, NY, USA, 2013.

[16] R. Wang, D. Butnariu, and J. Rexford, "Openflow-based server load balancing gone wild," *Hot-ICE*, vol. 11, pp. 12–12, 2011.

[17] R. Gandhi, H. H. Liu, Y. C. Hu et al., "Duet: cloud scale load balancing with hardware and software," in *ACM SIGCOMM CCR 2015*, New York, NY, USA, 2015.

[18] N. Kang, M. Ghobadi, J. Reumann, A. Shraer, and J. Rexford, "Efficient traffic splitting on commodity switches," in *ACM CoNEXT 2015*, New York, NY, USA, 2015.

[19] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split / merge: system support for elastic execution in virtual middleboxes," in *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, New York, NY, USA, 2013.

[20] N. McKeown, T. Anderson, H. Balakrishnan et al., "Openflow: enabling innovation in campus networks," in *ACM SIGCOMM CCR 2008*, New York, NY, USA, 2008.

[21] A. Nygren, B. Pfaff, B. Lantz et al., "Openflow switch specification version 1.5. 1," *Open Networking Foundation, Tech. Rep*, 2015.

[22] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Dream: dynamic resource allocation for software-defined measurement," in *ACM SIGCOMM 2014*, New York, NY, USA, 2014.

[23] N. Handigol, M. Flajslik, S. Seetharaman, R. Johari, and N. McKeown, "Aster∗x: load-balancing as a network primitive," in *ACLD 2010*, Washington, DC, 2010.

[24] M. Bredel, Z. Bozakov, A. Barczyk, and H. Newman, "Flow-based load balancing in multipathed layer-2 networks using openflow and multipath-tcp," in *HotSDN 2014*, Chicago, 2014.

[25] D. Thaler and C. E. Hopps, "Multipath issues in unicast and multicast next-hop selection," in *RFC 2991*, 2000.

[26] "Specifications of Dell Z9100 Switches," http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell-Networking-Z9100-spec-sheet.pdf.

[27] A. X. Liu, C. R. Meiners, and E. Torng, "Tcam razor: a systematic approach towards minimizing packet classifiers in tcams," *IEEE/ACM Transactions on Networking*, vol. 18, no. 2, pp. 490–500, 2010.

[28] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Cacheflow: dependency-aware rule-caching for software-defined networks," in *Proceedings of the Symposium on SDN Research*, New York, NY, USA, 2016.

[29] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM 2011 Conference*, New York, NY, USA, 2011.

[30] D. Lachos, Q. Xiang, C. Rothenberg, S. Randriamasy, L. M. Contreras, and B. Ohlman, "Towards deep network & application integration: possibilities, challenges, and research directions," in *Proceedings of the Workshop on Network Application Integration/CoDesign*, pp. 1–7, New York, NY, USA, 2020.

[31] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, 1969.

[32] Y. Azar, *On-Line Load Balancing*, Online Algorithms, 1998.

[33] Y. Azar and L. Epstein, "On-line load balancing of temporary tasks on identical machines," in *Proceedings of the Fifth Israeli Symposium on Theory of Computing and Systems*, Ramat Gan, Israel, 1997.

[34] 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 *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, New York, NY, USA, 2017.

[35] A. Bremler-Barr and D. Hendler, "Space-efficient tcam-based classification using gray coding," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 18–30, 2012.