RESEARCH FEATURE

Balanced Service Chaining in Software-Defined Networks with Network Function Virtualization

Po-Ching Lin, National Chung Cheng University
Ying-Dar Lin and Cheng-Ying Wu, National Chiao Tung University
Yuan-Cheng Lai, National Taiwan University of Science and Technology
Yi-Chih Kao, National Chiao Tung University

Balanced Hash Tree (BHT) is a mechanism for service function (SF) chaining, service routing, and traffic steering that enables switches to select SF instances for load balancing without involving the controller. In an experimental evaluation, BHT decreased packet-in message-processing time by 92.5 percent and achieved near-perfect load-balancing performance.

Network architects often rely on a software-defined network (SDN) to decouple the control plane from the data plane for higher programmability.[1] They can then use network function virtualization (NFV) to extend the data plane to outsource service functions (SFs), which allows service providers to deploy virtual SFs on commodity servers instead of on specialized hardware. Virtualization facilitates the dynamic instantiation of SFs to meet resource requirements from input traffic, which effectively lowers deployment cost.

However, this strategy puts an extreme burden on the centralized controller in a large datacenter network, which manages a set of switches through a southbound interface such as that defined by the Open Networking Foundation's OpenFlow specification.[2] An SDN provides flexible connectivity between switches and hosts, but providing value-added services exclusively through the controller is not a scalable solution, and passing packets from the switches to the controller will incur excessive communication overhead.[3,4] The controller can chain SFs—order their sequence and bind them together in a group—by configuring the data plane to support NFV. However, load balancing with SF chains can rapidly overwhelm the controller in such a large network.

To address this issue, we developed Balanced Hash Tree (BHT), a load-balancing mechanism for SF chaining, service routing, and traffic steering. Rather than

0018-9162/16/$33.00 © 2016 IEEE

BHT BALANCES CHAINING BY REDIRECTING INCOMING FLOWS TO THE OPTIMAL SERVICE FUNCTION WHILE MITIGATING THE CONTROLLER'S WORKLOAD.

relying on the controller, BHT implements load balancing on the switches through the *select* group table, as described in the OpenFlow specification, which requires processing to be based on a switch-computed selection algorithm.[2] In BHT, switches use a hashing-based algorithm to determine the output port for load balancing among an SF's instances, and the switch uses the select group type to assign each flow to an action bucket. Flow distribution is similar to the well-known equal-cost multipath routing (ECMP) strategy, which balances loads using multiple equal-cost paths between two neighboring hops (en.wikipedia.org/wiki/Equal-cost_multi-path_routing). However, BHT is different because it also considers service chaining between SFs and can assign weights to different paths.

BHT has proven to be a successful alternative to load balancing through the controller—an approach taken by many existing load-balancing schemes. In a performance evaluation, BHT reduced packet-in message-processing time by 92.5 percent and its load-balancing performance was within 2.4 to approximately 5 percent of perfect.

SERVICE FUNCTION CHAINING AND LOAD BALANCING

A service chain, which is an ordered sequence of SFs, is typically used to build a required network service. For example, to configure web traffic to go through a firewall, an administrator can use SF chaining to combine and order intrusion-prevention and load-balancing functions; the controller can then instruct the switches to redirect traffic through the SFs in that chain.[5] The Internet Engineering Task Force (IETF) defines the architecture for SF chaining,[6] but in simple terms, for each SF chaining path, a tunnel is established between the roots of the trees of any two successive SFs—meaning that the entire SF chaining path consists of multiple tunnels.

Balancing through switches

OpenFlow switches process incoming traffic through a pipeline of one or more flow tables and determine the actions on a packet by matching the entries in the flow tables. If a switch cannot match any of the flow entries, it either drops the packet or sends a packet-in message to the controller to ask how to process the packet. In the latter case, the controller sends the switch a packet-out or a flow-modify message telling it how to process the packet and configure the flow table. The multipart message has the added benefit of allowing the controller to collect information from the switches, such as port statistics.

When a matched flow entry's actions specify that a packet should go to a specific group entry, the packet must be processed according to the actions specified in the group entry's action buckets, which are determined by the entry's group type. OpenFlow supports four group types: *all*, *indirect*, *fast failover*, and *select*. For the select type, one bucket in the group entry is executed, depending on the selection algorithm, but the algorithm's configurations and states are left unspecified.

Network function virtualization

NFV supports running virtualized SFs on commodity servers and makes it easier to deploy SF instances on the virtualization layer, increasing both scalability and flexibility.[7] The NFV infrastructure includes management and orchestration components, which determine the service chain for specific traffic. The controller is then programmed to enforce the orchestration.

When queried from the ingress switch that classifies incoming traffic, the controller runs service routing and then configures the switches to enforce the SF chain by traffic steering. Service routing is responsible for finding the optimal path through a particular service chain, given available resources in the NFV infrastructure. In traffic steering, the switches steer incoming traffic according to the flow entries set by the controller.

Network service header

The European Telecommunications Standards Institute (ETSI) specification on OpenFlow-based traffic steering[8] describes it as the process of matching five tuples in each incoming packet to set entries in the flow table. However, per-flow matching incurs a large number of flow entries because the same SF chaining path must be set multiple times for each group of five tuples.

To address that problem, the IETF has proposed using the network service header (NSH),[9] which enables traffic steering and service chaining to be carried out in the service plane.[10] The NSH carries two critical fields: the service path identifier (SPI), which identifies an SF chaining path; and the service index (SI), which identifies the SF's location in the path. The SI decreases by 1 each time a packet travels through an SF (after the SF is finished). The combination of SPI and SI determines the tunnels through which the packets will pass.

The NSH efficiently reduces the number of entries for traffic steering

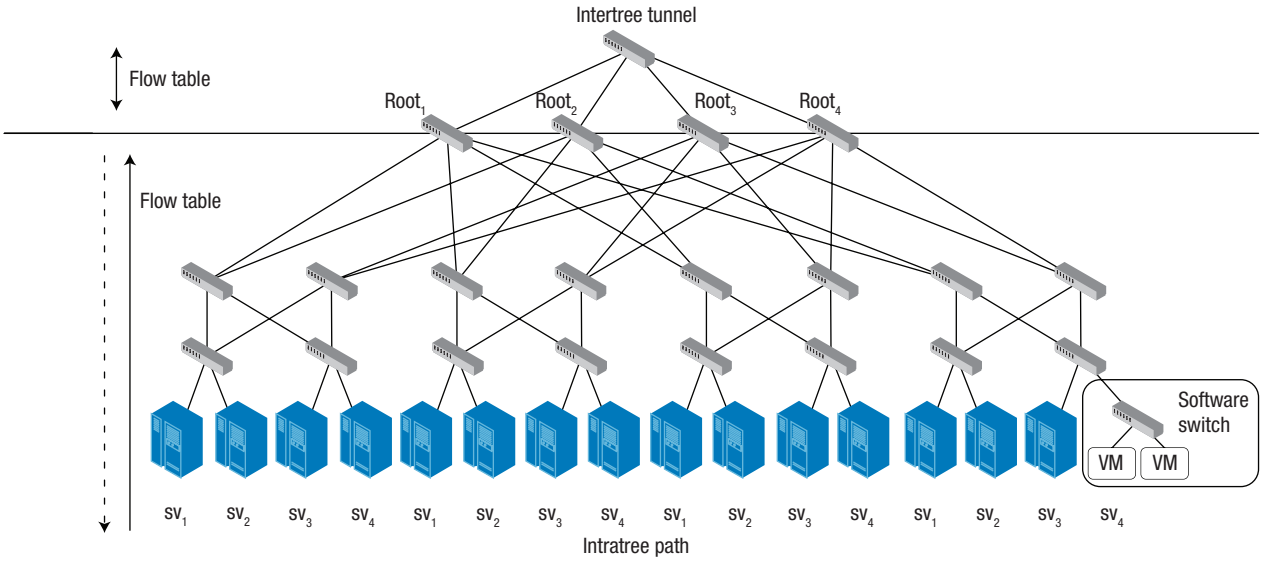


FIGURE 1. How Balanced Hash Tree (BHT) balances service functions (SFs). In this design, each physical server has one OpenFlow software switch, which runs two identical SF instances on two virtual machines (VMs), as shown in the enlarged box at far right. Multiple SFs make up an SF chain, represented as sv_1, sv_2, sv_3, ... sv_i. In the example, one server processes one chain. The SF path through the network is divided into an intertree tunnel, which interconnects the roots of successive SF trees through the flow entries (lines above the horizontal line), and the intratree path for distributing traffic (lines below the horizontal line). The arrows to the left represent packet-forwarding directions: up (solid), down (dashed), and both ways (double arrow).

because multiple SF chaining paths share the tunnels. The tunnel to be transformed for the next SF might consist of multiple equivalent instances for load balancing. The IETF draft does not specify the exact balancing method.

Controller-based balancing

Several proposed systems balance the load among SF instances, but all involve using the controller. One strategy implements multiple load-balancing algorithms on the controller, such as random, round-robin and load-based methods.[10] When it receives a packet-in message, the controller selects an SF instance according to the algorithms and then sets the flow entries. Another approach uses a dedicated controller for each service network to monitor SF instance loads for load balancing.[11] Yet another system monitors the server loads and the network status for processing packet-in messages when some flows request services.[12] A system for resource management and load balancing sets flow entries to divide incoming traffic.[13]

In all these systems, when the controller receives a packet-in message, it must track the loads of SF instances to select the appropriate one and then set the flow entries accordingly—tasks that greatly increase its workload.

HOW BALANCED HASH TREE WORKS

BHT has two main objectives: to balance SF chaining in a way that redirects incoming flows to the desired SFs, and to mitigate the controller's workload.

As Figure 1 shows, BHT establishes a tree of SFs and uses group entries to split incoming traffic among the SF instances. We assume that the datacenter network provides the NFV infrastructure, the network has a fat-tree topology, and the controller has information about both the network topology and the SF instances' locations. In this design, a service chain is mapped to a single service-chaining path that interconnects the roots of trees of SFs based on the NSH.

In Figure 1, SF locations are heterogeneous (different instances in the same pod), but BHT works equally well with homogenous locations (identical instances in the same pod). The intratree path is through the entries in the select group type, and the packets are redirected back to the root when an SF is finished. Thus, when the controller receives a packet-in message, it needs only to establish an intertree tunnel; it does not have to specify which instances will provide SFs.

Traffic steering

After classification to determine its traffic type (such as HTTP), a packet is mapped to the desired service chain according to the configured policy. It is then tagged with the NSH, which carries SPI and SI for traffic steering. SPI carries the service-chaining path identifier, and SI is initialized to the length of service chain.

Once SPI and SI determine the tunnels through which the packets will pass, BHT establishes the tunnels by imposing the NSH between the original packet and the Multi-Protocol Label Switching (MPLS) transport encapsulation in the outer network. We chose MPLS over a virtual extensible LAN (VXLAN) or generic routing encapsulation because its label format is simple and sufficient for our use. In an operation similar to that in MPLS, the switches in both ends of a tunnel will

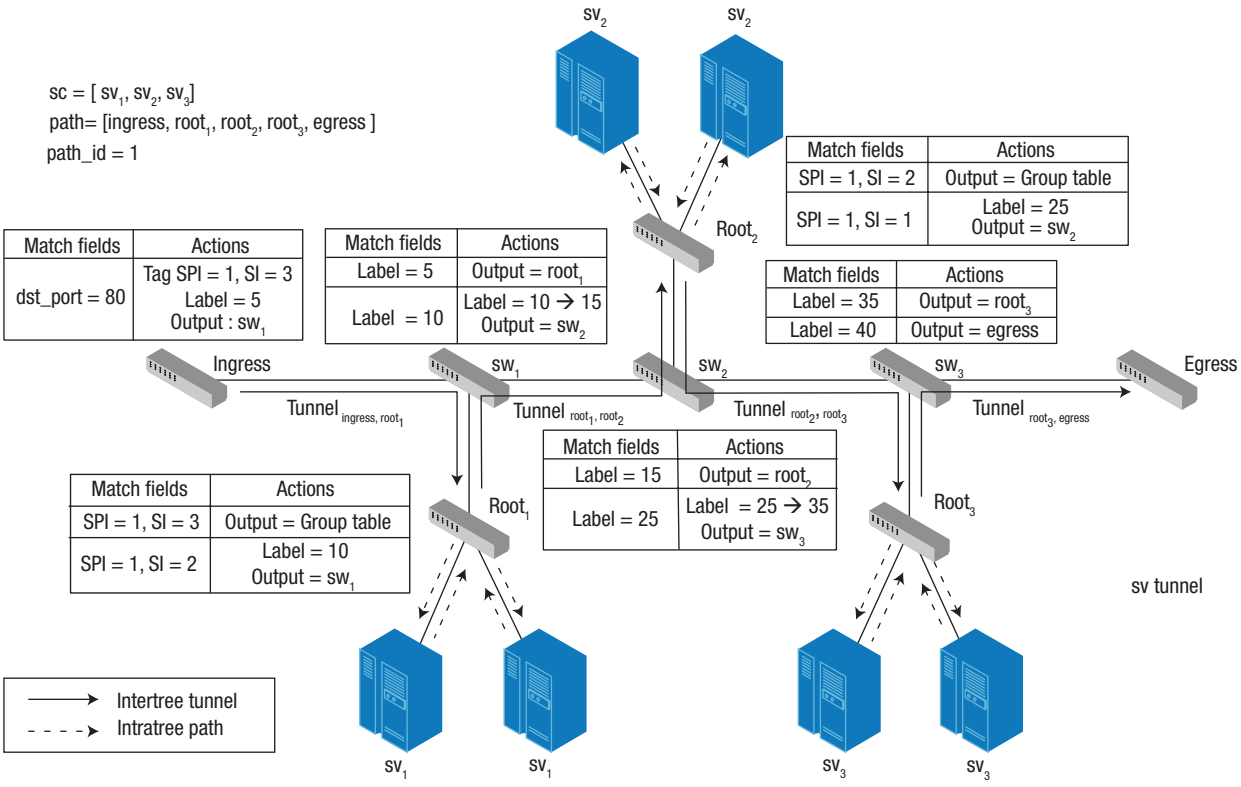


FIGURE 2. An example of traffic steering for an SF chain in which $Root_1$, $Root_2$, and $Root_3$ are the root switches for SFs sv_1, sv_2, and sv_3. The ingress switch first classifies incoming packets and initializes the service path identifier (SPI) and service index (SI) in the network service header as well as the label in the Multi-Protocol Label Switching (MPLS) transport encapsulation. The flow tables on the three root switches are in charge of changing the tunnel with a new label according to SPI and SI, while the other switches (sw_1, sw_2, and sw_3) perform label switching and forward packets. The egress switch removes the network service header and forwards the packets to their original destination.

distribute a locally unique label, and through repeated label switching the packets will move through the tunnel. After an SF processes a packet, the packet goes back to the root switch, which will change the tunnel and redirect the packet to the next SF depending on the SPI and SI.

The BHT module on the controller contains two functions related to traffic steering: `Packet-In_processing()` and `Service_chain_setting()`. We assume the controller knows the locations of the SF instances in terms of the attached switches and ports. It determines the group entries for each intermediate switch in the tree and then adds the group entries with the select group type (whether the SF locations are heterogeneous or homogeneous).

When the `Packet-In_processing()` function receives a packet-in message from the ingress switch, the controller assigns an SF chain to the flow specified in the message and checks whether the SF chain has been set. If it has, the controller sets a flow entry on the ingress switch to tag the flow's packets with the NSH. If it has not been set, the packet-in processing function calls the `Service_chain_setting()` function and assigns the new SF chain to the specified flow.

BHT assigns a unique path identifier to the SF chaining path and then checks whether every tunnel between any two successive SFs has been established. If it has, BHT uses SPI and SI as the match fields to set the entries on the root switch for changing the tunnel when the packets return to the root switch from a finished SF. Otherwise, it establishes a tunnel by distributing a locally unique label for the tunnel and then setting the entries for label switching.

Figure 2 shows a traffic-steering example for SF chain sc = [sv_1, sv_2, sv_3]. After an SF is finished, the corresponding root switch will change the tunnels (by changing the labels)—$Root_1$ for the tunnel from $Root_1$ to $Root_2$, $Root_2$ for the tunnel from $Root_2$ to $Root_3$, and $Root_3$ for the tunnel from $Root_3$ to the egress switch. The SF chain interconnects only the SF root switches, not the physical servers.

Load balancing

Because the controller knows the locations of SF instances, it can construct the tree for each SF and set the group entries according to the number of instances and their locations. In addition to the two traffic-steering functions, the BHT module on the controller contains the `Load_balancing()` function, which performs three tasks:

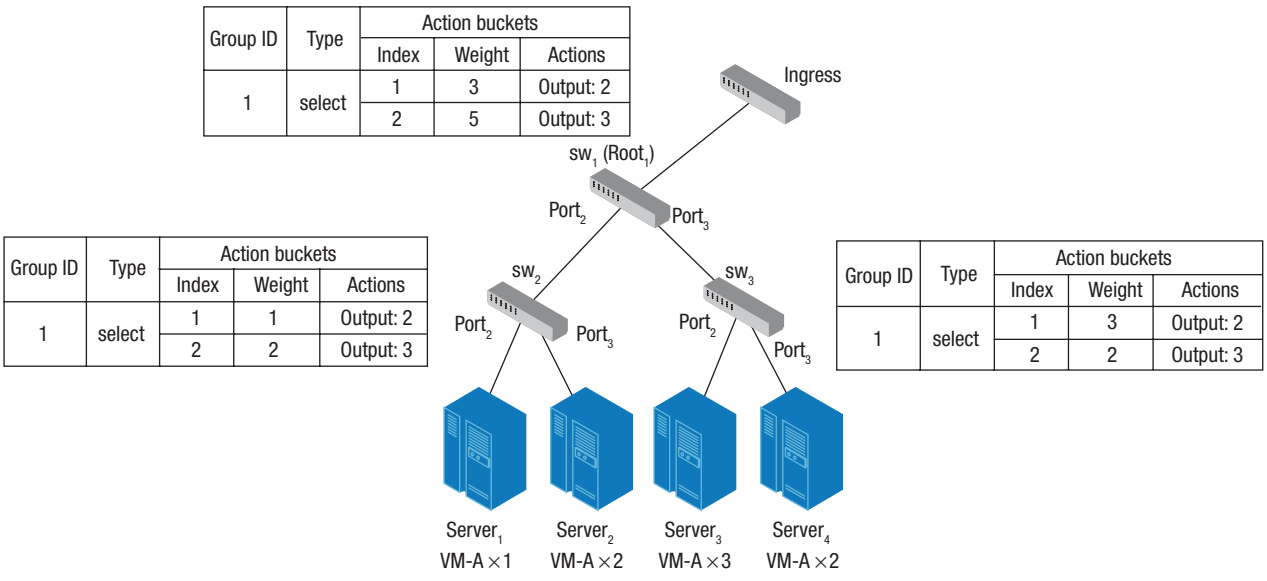


FIGURE 3. Load balancing in BHT. Physical servers can launch multiple VMs as SF instances, for example, SF-A in this case. After the tree has been established, the `Load_balancing()` function sets the group entries with multiple action buckets. The buckets consist of various forwarding ports, which are generated according to node branches. The corresponding weights are determined by the number of SF instances (VM–A × n) on the physical servers reachable from the output ports.

- For each SF, it calculates the paths from the switches to those SF instances attached to the ingress switch and leverages this information to find the SF's root switch (the switch nearest the ingress switch on the paths).
- It counts the number of instances to which an output port is attached for each intermediate node on the tree and records it as tree information.
- On the basis of the tree information obtained, it determines the actions and weights in the action buckets and sets the entries in the select group type.

Figure 3 shows an example of load balancing in BHT.

Selection mechanism of Open vSwitch

We assume that the network shown in Figure 2 uses Open vSwitch (openvswitch.org). The default selection mechanism on Open vSwitch is to calculate a score for each action bucket and have the switch select the action bucket with the highest score for the enforced action. When an incoming packet is assigned to an entry in the select group type, the first step is to retrieve the packet's destination media access control (MAC) address and then hash it to generate the basis value in the score calculation. The second step is, for each action bucket, hash its basis with index i and multiply the hashing result and the weight. The result is the action bucket's score. The score calculation is formulated as

```
basis = hash_mac(dst_mac)
score = (hash_int(i,basis) & 0xffff) *
        bucket→weight
```

The default selection calculation has a flaw, however. The hash functions are the same on every switch, so the packets with the same destination MAC address will get the same result with the same group entry, even on different switches. In other words, they will be all forwarded to the same port, and no packets will go through the other ports.

For that reason, BHT uses a modified calculation in which the destination MAC address and the switch's datapath identifier (dpid)—which allows the controller to manage switches—are hashed first to produce dpbasis, and dpbasis and the action bucket's index are hashed next. With this modification, selection varies with the switches even with the same group entry, but the selection for packets in the same flow is still uniform. The modification is formulated as

```
basis = hash_mac(dst_mac)
dpbasis = hash_int(dpid,basis) & 0xffff
score = (hash_int(i,dpbasis) & 0xffff) *
        bucket→weight
```

SAMPLE IMPLEMENTATION

To evaluate BHT's performance, we implemented it on the Ryu controller (osrg.github.io/ryu) using our modification of the Open vSwitch's default selection algorithm.

The BHT module on Ryu contained the `Load_balancing()`, `Packet_In_processing()`, and `Service_chain_setting()` functions. As implemented, the first function uses Dijkstra's algorithm[14] to calculate the SF trees and a list to store tree information. We applied `parser.OFPGroupMod()`—a function from

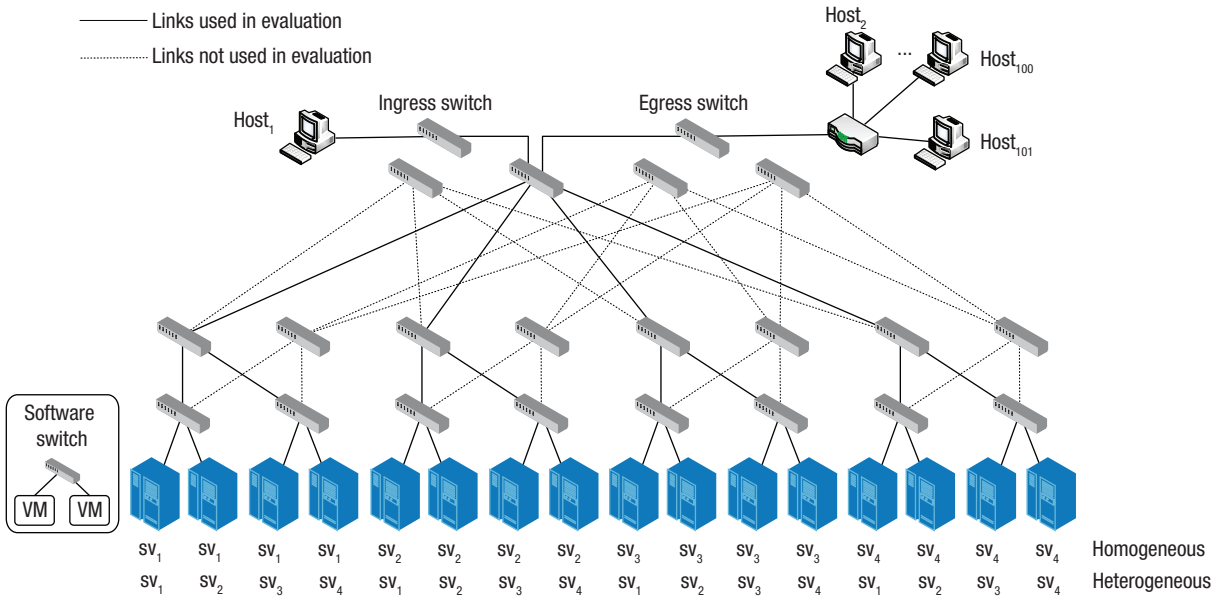


FIGURE 4. Network topology for our experiments to evaluate BHT's load-balancing ability. We employed Mininet 2.2 to emulate a virtual network. The VMs on Mininet emulate the SF instances. Mininet's iperf tool generated 100 TCP connections with different destination access control (MAC) addresses from $Host_1$ to $Host_{101}$ for emulating the flows that reach the desired SFs through SF chaining.

the Ryu APIs—to set group entries according to that information.

We had a packet-in message trigger the `Packet-In_processing()` as a thread, and used the `packet_in_handler()` function to process the message. If necessary, we could have the packet-in message trigger the `Service_chain_setting()` function, which relies on Dijkstra's algorithm to set a new SF chaining path.

We did not include the NSH in our implementation. Rather, we inserted VLAN tags to carry the SPI and SI fields, which simulated NSH use, and established tunnels in the transport encapsulation using the label value in the differentiated services code point (DSCP) field—a six-bit field in the IP header that specifies the per-hop behavior for a given packet flow.

EVALUATION RESULTS

We used two servers in our experiments. The first is an Intel Core i5-4590 running the Debian 7.7 OS at 3.30 GHz on a VMware workstation with the Ryu 3.15 controller and OpenFlow 1.3 switching protocol. The other server is an Intel Core i7-4790K also running Debian 7.7 but at 4.00 GHz and with Open vSwitch version 2.3. We chose the second server to emulate a datacenter network's tree topology by also running Mininet version 2.2 (mininet.org). Mininet creates a realistic virtual network that runs real kernel, switch, and application code on a single machine. Figure 4 shows the network configuration used in our experiments.

The SF types are irrelevant to our experimental results, and the locations of SFs can be either homogeneous or heterogeneous. The traffic was sent for 200 seconds, and the packet size was consistently 1,514 bytes. Once we sent all the traffic, we evaluated load-balancing performance in terms of the bytes received, which we considered the workload of an SF instance.

Our objectives in conducting the experiments were to evaluate packet-in message-processing time and load-balancing performance.

Packet-in message-processing time

To set a baseline for comparing packet-in message-processing times, we designed a mechanism that implements load balancing on the controller, as do many existing schemes. The load-balancing mechanism on the controller refers to the received bytes as the workload of an SF instance. When receiving a packet-in message, but before calculating the SF chaining path, the controller sends a multipart request message to get the port information of received bytes on the software switches to which the SF instances are attached. The fewer bytes a software switch has received, theoretically, the lighter the load of the SF instances attached to it will be.

After getting the received bytes in a time interval from the software switches, the controller selects the instances attached to a software switch with the lowest number of received bytes (the least loaded instances) to provide the SF for load balancing. Finally, the controller determines the SF chaining path and sets the flow entries.

In this experiment, we assumed the configuration in Figure 4: four SFs are in the datacenter network (sv_1 through sv_4), and each SF contains two instances on each of the four physical servers for that SF. We generated 100 TCP

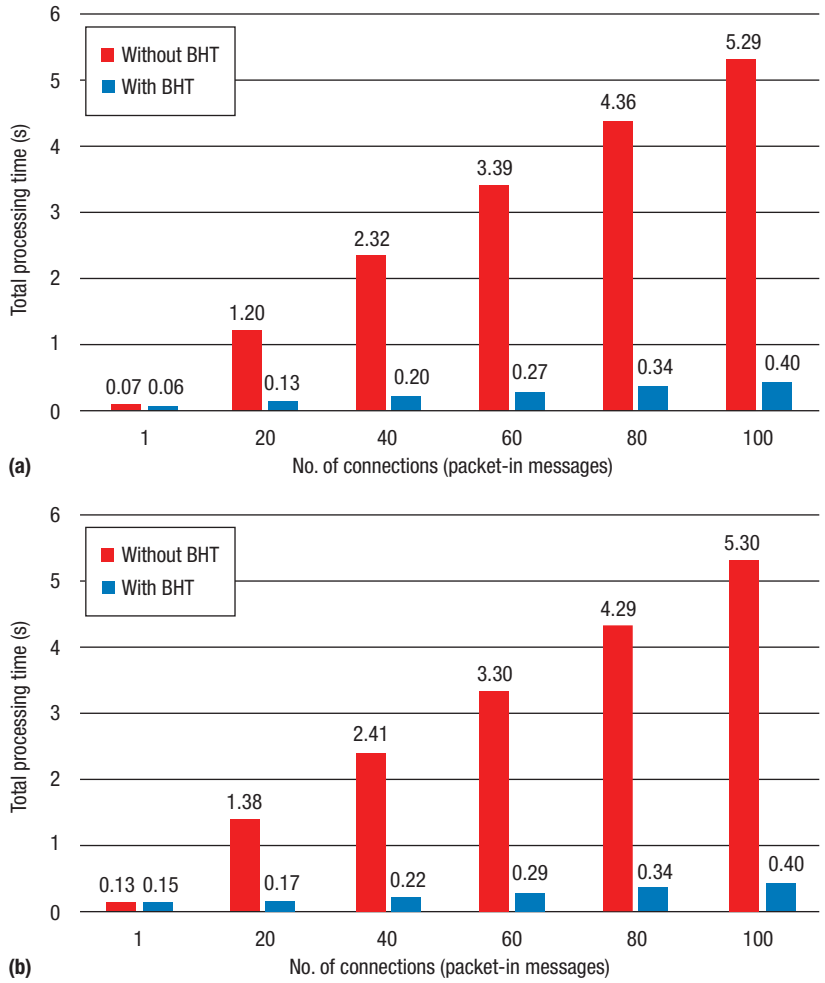


FIGURE 5. Processing times of packet-in messages with and without BHT. Times are in terms of connection number, which is synonymous with the number of packet-in messages, for (a) homogeneous and (b) heterogeneous SF locations. Time without BHT includes the latency from the queried switches reporting their load information back to the controller. With BHT, the controller outsources the selection of SF instances to the group table on the switches, which effectively mitigates the controller's workload. Total processing time without BHT was 5.3 seconds and, with BHT, 0.4 seconds—a reduction of approximately 92.5 percent. The result is similar for homogenous and heterogeneous SF allocation.

connections from $Host_1$ to enforce the same service chain over the four SFs.

Figure 5 shows processing times of packet-in messages with and without BHT. Packet-in processing time without BHT includes the latency from the queried switches reporting their load information back to the controller. We performed the queries in parallel by multithreading, and latencies were typically on the order of milliseconds. Considering the processing time of 5.3 seconds for 100 connections without BHT, almost all the processing time is still likely to be attributable to the controller's workload

Load-balancing performance

Figure 6 shows comparative load balancing for the same experiment. The comparison covers only load balancing among the four physical servers for each SF because the hypervisor handles balancing among the VMs inside the server. For each SF, the perfect load balancing is 25 percent on each of the four physical servers, so we evaluated load-balancing performance by calculating the average absolute difference between the actual load and the perfect load on each server.

Because the entry in the select group type is implemented by hashing, the loads are not as balanced as those with load balancing without BHT, but the absolute difference for BHT is within 2.4 to 5.0 percent of perfect performance. The load balancing with BHT is slightly better for heterogeneous SF allocation, but the difference is insignificant for any practical application. The results show that BHT efficiently balances loads among the servers, while simultaneously reducing the controller workload significantly.

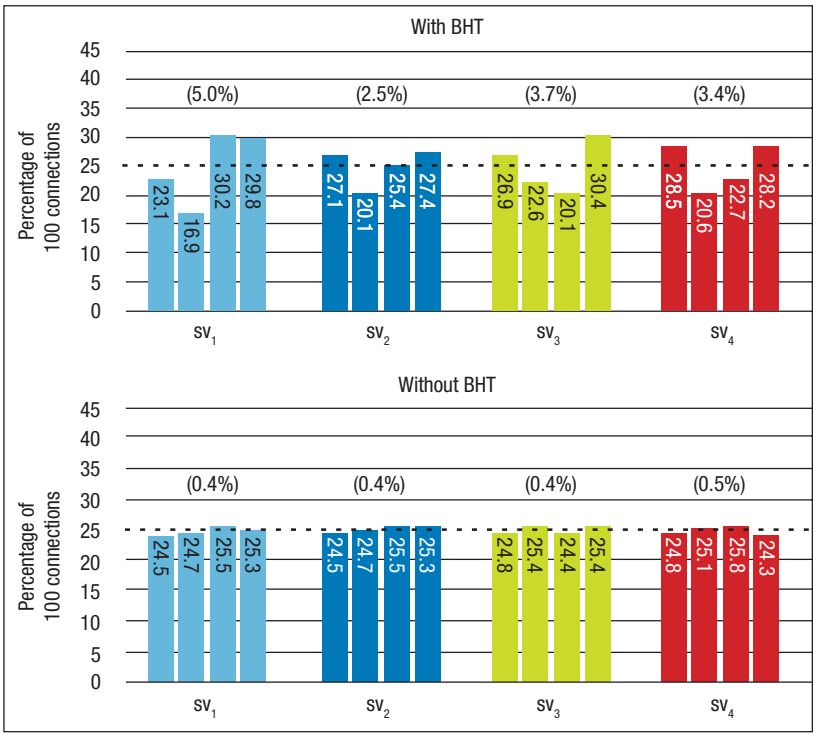
Initial evaluations show that BHT can be an effective-load balancing alternative to controller-based solutions, and we have already identified areas for extension. One is to accommodate multiple paths between adjacent SFs. In the current design, BHT considers only load balancing among the instances of the same SF. Extending BHT to cover load balancing among both the SF instances and paths simultaneously is an interesting issue for future work. Another open question for additional exploration is how to perform load balancing for diverse types of real-world traffic in a datacenter network.

ACKNOWLEDGMENTS

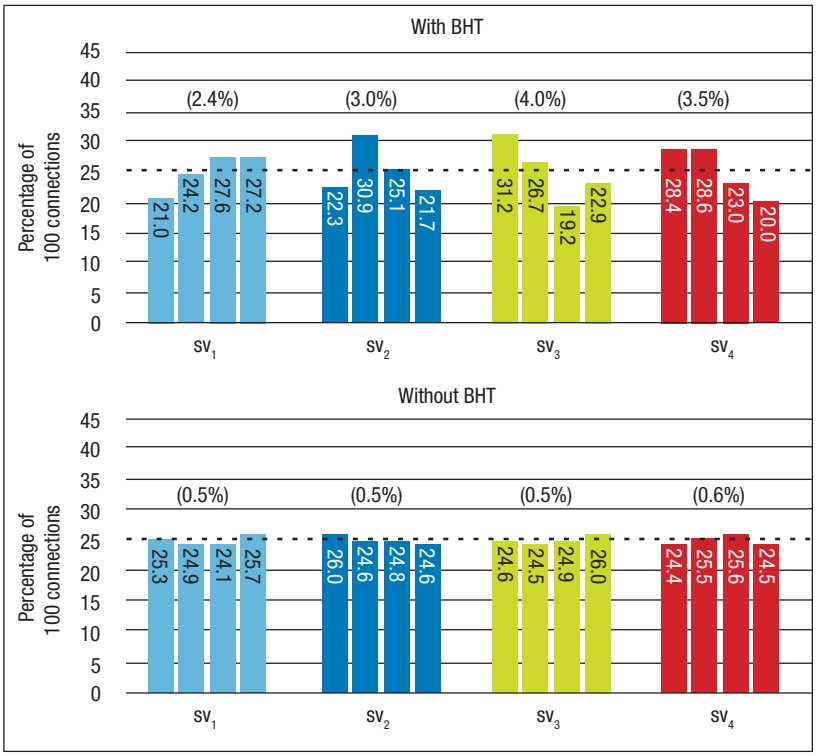
The work described in this article was supported in part by the Ministry of Science and Technology, Taiwan; Chunghwa Telecom and MediaTek; and the III Innovative and Prospective Technologies Project (1/1) of the Institute for Information Industry, which is subsidized by the Ministry of Economic Affairs, Taiwan.

REFERENCES

1. *Software-Defined Networking: The New Norm for Networks*, white paper, Open Networking Foundation, 2012; www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.
2. *OpenFlow Switch Specification, Version 1.5.0*, ONF TS-020, Open Networking Foundation, 19 Dec. 2014; www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf.
3. B. Nunes et al., "A Survey of Software-Defined Networking: Past, Present, Future of Programmable Networks," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 3, 2014, pp. 1617–1634.
4. Y.D. Lin et al., "An Extended SDN Architecture for Network Function Virtualization with a Case Study on Intrusion Prevention," *IEEE Network*, vol. 29, no. 3, 2015, pp. 48–53.
5. P. Quinn and T. Nadeau, *Problem Statement for Service Function Chaining*, IETF RFC 7498, Apr. 2015; tools.ietf.org/html/rfc7498015.
6. J. Halpern and C. Pignataro, *Service Function Chaining (SFC) Architecture*, IETF RFC 7665, Oct. 2015; tools.ietf.org/html/rfc7665.
7. R. Mijumbi et al., "Network Function Virtualization: State-of-the-Art and Research Challenge," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, 2016, pp. 236–262.
8. Group Specification: Network Functions Virtualisation (NFV); Virtual Network Functions Architecture, ETSI GS NFV-SWA 001 v1.1.1, European Telecommunications Standards Inst.; Dec. 2014; www.etsi.org/deliver/etsi_gs/NFV-SWA/001_099/001/01.01.01_60/gs_NFV-SWA001v010101p.pdf.



(a)



(b)

FIGURE 6. Comparison of load-balancing performance. We compared load balancing on four physical servers with (a) homogenous and (b) heterogeneous locations with and without BHT. The horizontal dashed line represents perfect balancing at 25 percent. The numbers in parentheses above the bars denote the percentage departure of the average load balancing (average of the numbers over the four bars) from a perfect load balancing for that SF. In (b) with BHT, the sv_1 balancing was within 2.4 percent of the perfect balance.

ABOUT THE AUTHORS

PO-CHING LIN is an associate professor in the Department of Computer and Information Science at National Chung Cheng University (CCU). His research interests include network security, network traffic analysis, and the evaluation of network systems performance. Lin received a PhD in computer science from National Chiao Tung University (NCTU). Contact him at pclin@cs.ccu.edu.tw.

YING-DAR LIN is a Distinguished Professor in the Department of Computer Science at NCTU and director of the Network Benchmarking Lab. His research interests include transforming networks into clouds, software-defined networks (SDNs), network function virtualization, 5G/mobile edge computing, and network security. Lin received a PhD in computer science from the University of California, Los Angeles. He is an IEEE Distinguished Lecturer, an Open Networking Foundation research associate, and coauthor of *Computer Networks: An Open Source Approach* (McGraw-Hill, 2011). Contact him at ydlin@cs.nctu.edu.tw.

CHENG-YING WU is an engineer at MediaTek. While conducting the research reported in this article he was a graduate student at NCTU. His research interests include network architecture, SDNs, and cloud computing. Wu received an MS in computer science from NCTU. Contact him at cywu.cs02g@nctu.edu.tw.

YUANG-CHENG LAI is a professor in the Department of Information Management at National Taiwan University of Science and Technology. His research interests include wireless networks, network performance evaluation, network security, and social networks. Lai received a PhD in computer science from NCTU. He is a member of IEEE. Contact him at laiyc@cs.ntust.edu.tw.

YI-CHIH KAO is director of the Network and System Division of the Information Technology Service Center at NCTU. His research interests include cyber forensics, network performance evaluation, SDNs, and service design. Kao received a PhD in industrial engineering and management from NCTU. He is a member of ACM and the Project Management Institute (PMI). Contact him at ykao@mail.nctu.edu.tw.

9. P. Quinn and U. Elzur, "Network Service Header," IETF Internet Draft, work in progress, 26 May 2016.
10. H. Uppal and D. Brandon, "OpenFlow-Based Load Balancing," cite seerx.ist.psu.edu/viewdoc/download?doi=10.1.1.168.5150&rep=rep1&type=pdf.
11. M. Koerner and O. Kao, "Multiple Service Load-balancing with OpenFlow," *Proc. IEEE 13th Int'l Conf. High-Performance Switching and Routing* (HPSR 12), 2012, pp. 210–214.
12. N. Handigol et al.,"Plug-n-Serve: Load-Balancing Web Traffic Using OpenFlow," *Proc. ACM SIGCOMM Conf. Data Communication* (SIGCOMM 09), 2009; conferences.sigcomm.org/sigcomm/2009/demos/sigcomm-pd-2009-final26.pdf.
13. Z. Qazi et al., "SIMPLE-fying Middlebox Policy Enforcement Using SDN," *Computer Communication Rev.*, vol. 43, no. 4, 2013, pp. 27–38.
14. E.W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, 1959, pp. 269–271.