HALO: Accelerating Flow Classification for Scalable Packet Processing in NFV

Yifan Yuan
University of Illinois at Urbana-Champaign
yifany3@illinois.edu

Yipeng Wang
Intel Labs
yipeng1.wang@intel.com

Ren Wang
Intel Labs
ren.wang@intel.com

Jian Huang
University of Illinois at Urbana-Champaign
jianh@illinois.edu

ABSTRACT

Network Function Virtualization (NFV) has become the new standard in the cloud platform, as it provides the flexibility and agility for deploying various network services on general-purpose servers. However, it still suffers from sub-optimal performance in software packet processing. Our characterization study of virtual switches shows that the flow classification is the major bottleneck that limits the throughput of the packet processing in NFV, even though a large portion of the classification rules can be cached in the last level cache (LLC) in modern servers.

To overcome this bottleneck, we propose HALO, an effective near-cache computing solution for accelerating the flow classification. HALO exploits the hardware parallelism of the cache architecture consists of Non-Uniform Cache Access (NUCA) and Caching and Home Agent (CHA) available in almost all Intel® multi-core CPUs. It associates the accelerator with each CHA component to speed up and scale the flow classification within LLC. To make HALO more generic, we extend the x86-64 instruction set with three simple data lookup instructions for utilizing the proposed near-cache accelerators. We develop HALO with the full-system simulator gem5. The experiments with a variety of real-world workloads of network services demonstrate that HALO improves the throughput of basic flow-rule lookup operations by 3.3×, and scales the representative flow classification algorithm – tuple space search by up to 23.4× with negligible negative impact on the performance of collocated network services, compared with state-of-the-art software-based solutions. HALO also performs up to 48.2× more energy-efficient than the fastest but expensive ternary content-addressable memory (TCAM), with trivial power and area overhead.

CCS CONCEPTS

• **Hardware** → **Networking hardware**; • **Computer systems organization** → *Multicore architectures.*

KEYWORDS

flow classification, near-cache computing, network function virtualization, hash-table lookup

ACM Reference Format:
Yifan Yuan, Yipeng Wang, Ren Wang, and Jian Huang. 2019. HALO: Accelerating Flow Classification for Scalable Packet Processing in NFV. In *The 46th Annual International Symposium on Computer Architecture (ISCA '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3307650.3322272

ISCA '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6669-4/19/06…$15.00
https://doi.org/10.1145/3307650.3322272

1 INTRODUCTION

Network Function Virtualization (NFV), as a new generation of networking paradigm, has been prevailing in cloud platforms for virtualizing network functions. Along with software-defined networking (SDN) techniques, which decouple the data plane and control plane of a networking platform, NFV can manage a large number of network flows, and enables flexible and agile deployment of network services on general-purpose servers [38, 42, 59, 70].

These network services communicate with each other via the software virtual switches (see Figure 1). The key functionality of the virtual switches is the *software packet processing*, which typically involves a series of match-action operations. As network packets arrive, the virtual switch extracts the packet header information, and compares it with the predefined rules in multiple match-action tables. Subsequently, the packets will be classified into different flows (i.e., *flow classification*) and processed with the corresponding actions derived from the matched rules.

As the cloud server is scaling up to support increasing number of network services with more virtual machines or containers [31, 50, 52], the software packet processing is playing a critical role in sustaining the performance and scalability for network services in NFV.

To enable high-performance software packet processing, prior work proposed optimization techniques on both software and hardware. On the software front, researchers have proposed techniques such as DPDK [9] to avoid OS intervention and context switches [36, 60, 62], as well as algorithmic optimizations like cuckoo hash [57] and HiCuts series [26, 68, 78] to exploit the performance potentials of modern multi-core processors.

On the hardware front, prior studies typically explored one of two approaches. They either offload the software packet processing to hardware accelerators such as GPU [24, 37, 40, 74] and SmartNIC [7, 20, 21, 46], or use specialized memory architecture for fast data lookup, such as TCAM [4, 44, 71, 81], and its SRAM-based

versions [75–77]. The former approach leverages the extreme parallelism of hardware accelerators to speed up the software packet processing. However, it introduces PCIe communication bottleneck to the virtual switches, as well as extra power consumption and hardware cost [24, 40]. The latter approach like TCAM can execute one data lookup operation in a few clock cycles [58]. However, it involves expensive and inflexible update operations [67]. Moreover, its energy overhead is extremely high [2], which prevents it from being massively integrated in the general-purpose CPUs.

In this paper, we rethink the optimization approaches for the software packet processing. We first conduct a comprehensive and thorough study on a real-world well-developed virtual switch – Open vSwitch (OVS) [60] to understand the performance characteristics of the software packet processing, with representative NFV workloads in data centers (see details in § 3).

We break down the procedure of software packet processing into a few critical components, including packet transmission, packet pre-processing, and hash-table lookup. We observe that (1) the current software implementation of network packet processing with cuckoo hash has best utilized the LLC of modern server CPUs to cache a large portion of the useful data of the network flow tables, which provides us with the hint where we should optimize for the software packet processing. (2) The flow classification, especially the flow-rule lookup operation, contributes a significant portion (up to 77.8%) to the total execution time of the software packet processing in the virtual switch, which limits the capability of processing packets in modern servers. We also observe that (3) the concurrency control of data access in the virtual switch and the involved core-to-core communication introduce significant performance overhead to the flow classification, which further constrains its scalability.

Based on these observations, we propose Halo, a hardware-assisted, near-cache acceleration approach to accelerate the flow classification within modern general-purpose servers. We develop Halo based on the insight that most of the flow rules have already been cached in the LLC of modern server CPUs. Therefore, it falls naturally to move computing closer to the data in the LLC to reduce the data movement overhead. Thanks to the Non-Uniform Cache Access (NUCA) architecture [8, 30, 39] of modern LLC, and the near-cache nature of the Caching and Home Agent (CHA) that has been available in almost all Intel® multi-core CPUs, we associate the Halo accelerator with each CHA component to increase the parallelism of both data access and computation.

Unlike prior studies on near-cache accelerators [1, 41, 43] that were developed in a centralized manner, Halo focuses on the accelerator scalability for network flow classification by leveraging the existing cache architecture with minimal hardware modification. As Halo is mainly used to process data in LLC, it reduces the private cache pollution, and further avoids the performance inference with other collocated network services. Moreover, to facilitate concurrent data accesses, Halo has a simple but effective hardware-assisted locking mechanism to reduce the locking overhead.

Furthermore, instead of issuing hundreds of conventional x86-64 instructions to fulfill a single data lookup operation (see Table 1), we extend the x86-64 instruction set with three simple all-in-one instructions, which makes the Halo approach more generic. Such an extension significantly simplifies the programming effort with Halo accelerators, while reducing the execution cycles for each data lookup.

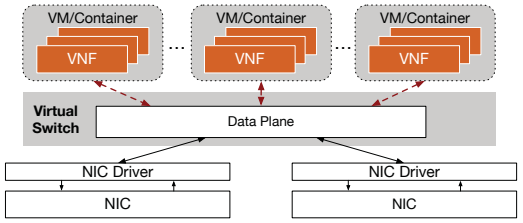


Figure 1: A typical NFV platform with network service consolidation. VNFs are connected to each other and the outside world via the software virtual switch.

We also propose a hybrid computation mechanism that allows Halo to work with software optimization solutions in a complementary fashion. Once Halo detects the number of flows can be fully cached in the faster L1 cache, it will allow programs to switch back to the software mode. Therefore, we can exploit the performance benefits of both faster L1 cache and the Halo accelerator adaptively. Overall, we make the following major contributions in this paper:

- To the best of our knowledge, we conduct the first detailed characterization study of the de-facto virtual switch widely used in the data centers today, and identify that the flow classification is the major bottleneck that limits the scalability of software packet processing in NFV.
- We propose a near-cache acceleration solution, named Halo, for flow classification. It exploits the hardware parallelism of the existing cache architecture to scale the flow-rule lookups with minimal hardware cost, while avoiding private cache pollution.
- We extend the x84-64 instruction set with three simple instructions for near-cache lookup operation, which makes Halo more generic and easier to use.
- We propose a hybrid computation mechanism that enables network services gain the benefits of both faster L1 cache and Halo accelerators in an adaptive manner.

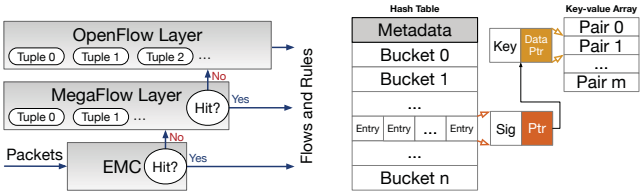
We develop and evaluate Halo within both the gem5 full-system simulator and a real NFV platform. Our experiments with a variety of NFV workloads demonstrate that Halo improves the throughput of the basic flow-rule lookup operations by 3.3×, and scales the typical flow classification method like tuple space search by up to 23.4×, while having negligible negative impact on the performance of collocated network functions. Halo also achieves up to 48.2× better energy-efficiency than the fastest TCAM-based solutions.

2 BACKGROUND

2.1 NFV and Virtual Switch

To facilitate flexible and agile deployment of network functions, data centers have been driven to virtualize the network functions, and leveraged virtual machines or containers to run virtual network functions (VNF) on the shared servers [27, 42, 66]. As shown in Figure 1, a typical NFV platform hosts multiple VNFs on a shared server, and the VNFs communicate network packets through the virtual switch.

To improve resource efficiency, the service providers tend to run multiple network services on the shared platform. In these network services, such as vEPC [61] and vB-RAS/vBNG [11, 55], a large



(a) Layered datapath in OVS. (b) A typical hash table.

Figure 2: A typical flow classification design and hash table data structure used in the virtual switch.

number of network functions are created. Therefore, an increasing amount of network traffic needs to be processed in the virtual switch, and thus it comes with higher requirement on the throughput of the virtual switches. Moreover, virtual switch processes are usually running together with the collocated VNFs on the shared server. This will inevitably causes the performance interference between the virtual switch and VNFs, which exacerbates the performance challenge of the software packet processing. Our experiments of co-running different VNFs such as DPDK-based access control (ACL) [14] and a scalable user-level TCP stack (mTCP) [36] show that the performance of VNFs will drop 17%–26% due to the cache pollution (see Figure 12 in § 6.3).

2.2 Packet Processing and Flow Classification

Processing network packets and classifying them into flows following a set of predefined rules are the core functionalities of virtual switches. When the virtual switch receives a network packet, it will extract its header information, and compare it with the predefined rules in the match-action hash tables. After that, the packet will be classified into different flows, and corresponding actions will be taken for each flow. Such procedure is called *flow classification*.

Hierarchical cache layers for flow-rule lookup. Taking OVS design as an example, a typical virtual switch uses a hierarchical cache structure to handle flow classification. As shown in Figure 2a, there are three software layers to cache the flow rules in hash tables. The first layer, Exact Match Cache (EMC), has a single hash table that will match the full header of each packet. EMC performs the fastest data lookup, as it requires only one table lookup without wildcard masking. However, its size is limited, and only a small number of hot flows can reside in this layer. The second layer, which is called MegaFlow layer, consists of a set of hash tables (tuples), and each tuple stores rules that share the same wildcarding pattern. Packets arriving at this layer will undergo the tuple space search for wildcard matching [69], and it will return once the first matching rule is found. The third layer, named as OpenFlow layer, is also implemented with tuple space search algorithm. However, OpenFlow layer performs slower than MegaFlow layer, because it has to search all the tuples and find the highest priority rules among all the matched rules.

Hash table data structure for storing flow rules. As discussed, the flow classification usually uses hash tables to store the packet headers and rules as shown in Figure 2b. The metadata stores the necessary information of the table, such as table size, key length, and hash function type. The hash table also has an array of buckets, each of which consists of several entries. Each entry has a signature hashed from the original key, and a pointer to the key-value pair. To reduce data access overhead, each bucket typically occupies and aligns with one CPU cache line (i.e., 64B). In a typical hash-table lookup procedure, the program will hash the key to get the index of the corresponding bucket in the table. And then, each entry inside it will compare with the signature. If the signature matches, the corresponding pointer will be used to acquire the key-value pair in the key-value array. If the key is matched, the value will be returned.

Cuckoo hash. To reduce the hash conflict as well as the storage cost, cuckoo hash [57] has been proposed and widely used in packet processing implementation. In cuckoo hash, the key will be hashed to two buckets using two different hash functions. For each data lookup, the entries of each bucket will be iterated to do the key comparison. When inserting a new key, cuckoo hash allows it to displace an existing key to its alternative bucket recursively. For example, when key A is inserted, if both buckets are full, key B residing in key A's bucket will be displaced to an alternative bucket of the key B to make space for key A. In such way, cuckoo hash achieves high table utilization without rehashing. It has been proven that the cuckoo hash can provide decent performance in terms of both data lookup and update, and it has been widely used in modern software virtual switches such as OVS [60], VPP [19] and DPDK. We use contiguous memory allocation for the hash table for performance reason. In this paper, we use such a software optimization by default.

3 OVS PERFORMANCE ANALYSIS

In this section, we present our characterization study on the popular virtual switch OVS used in modern data centers. The study results will shed light on the design of HALO and its optimization techniques.

3.1 Experiment Setup

Hardware platform. We set up a physical to physical forwarding testbed with a commercial off-the-shelf server (COTS) and the hardware traffic generator IXIA [35]. The COTS has two sockets, each has a 24-core Intel® Xeon® Platinum 8160 CPU running at 2.1GHz [34]. We allow a 40GbE dual-port Intel® XL710 NIC [10] (with DDIO enabled) to directly communicate with the cores on socket-0. We use IXIA to generate full 40Gb speed traffic with 64B UDP packets[1].

Software platform. We use OVS as an example to understand the performance characterization of the virtual switches. However, other virtual switch implementations, such as VPP and Tungsten Fabric vRouter, share the similar design basics with OVS. Since OpenFlow Layer is seldom accessed in practice in OVS, we focus on the EMC and MegaFlow layer in our performance analysis. We run OVS with user-space data plane DPDK [9] to minimize the I/O overhead caused by the operating system kernel stack. We use Intel® VTune [33] to profile the performance of the OVS workflow when running various workloads, and then demonstrate the performance breakdown with average execution times of processing one packet in each critical component in the workflow.

[1]Since virtual switches mainly deal with packet headers, their performances are not related to the payload size of packets.

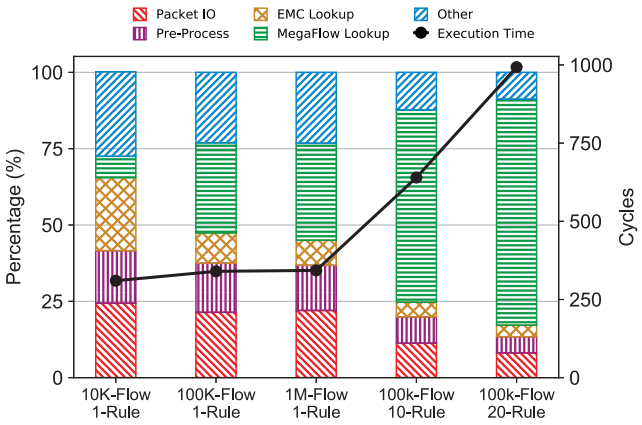


Figure 3: Performance breakdown of software packet processing with various network traffic in OVS.

3.2 Performance of Packet Processing

We follow the real-world workload characteristics in the data centers [5, 60, 61, 65] and use the traffic generator IXIA to generate three representative scenarios with five configurations:

- Small number of flows (under 100K flows): this represents the overlay network in which many flows are encapsulated with a single header, thus the total flow count becomes smaller [16].
- Many flows (100K–1M flows with 1–10 rules): this represents the situation that OVS routes traffic to multiple containers on the same server. While the the number of rules is small (limited by the number of network functions), the number of flows is large, since they are from different addresses.
- Many flows and rules (100K–1M flows with 20 hot rules): this represents a gateway or top-of-rack router that handles network communication destined to different group of servers in a data center.

For these five configurations, they increasingly take 340 – 993 cycles for processing one packet on average (see Figure 3). To further understand their performance characteristics, we divide the software packet processing procedure into five parts: packet IO (i.e., packet transmission, reception, and queueing), packet pre-processing (i.e., header extraction), EMC lookup, MegaFlow lookup, and others. As shown in Figure 3, the flow classification (EMC lookup and MegaFlow lookup) occupies 30.9%–77.8% of the total execution time of the packet processing. As the number of flows and rules increases, the flow classification is becoming the major bottleneck. This is mainly contributed by the hash-table lookup operations in the MegaFlow layer. As discussed in § 2.2, EMC performs much faster than MegaFlow layer. However, its size is limited, which can only cache a small number of flows. As the network traffic is increased, EMC will be mostly missed, and the packets have to conduct the tuple space search in the MegaFlow layer.

Observation: *The virtual switch uses multi-layer cache structure and cuckoo hash to optimize packet processing on general-purpose CPUs. However, due to the constraint of the software caching, the flow classification is the major performance bottleneck for the realistic use cases that have a large number of flows and rules in data centers.*

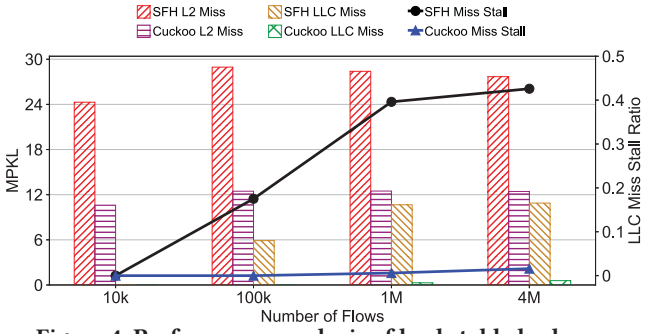


Figure 4: Performance analysis of hash-table lookups.

3.3 Performance of Hash-Table Lookup

As discussed, the hash table is the core data structure used in flow classification. For the hash table implementation, OVS takes advantage of the cuckoo hash algorithm. In this experiment, we evaluate both the cuckoo hash table (8-way set associative, which is the default setting in DPDK) and a regular hash table that has a single hash function (represented as SFH in Figure 4), with the goal of understanding their cache/memory access characteristics.

Given the same number of keys, cuckoo hash is much more efficient than SFH in terms of space utilization. As a result, the allocation of keys in cuckoo hash table is more compact (~ 95%) [17], which means less cache space is wasted. In terms of SFH, we find that most of the table buckets only have one or two entries occupied, which leads to a low table utilization (~ 20%). Therefore, we have to allocate a much larger SFH table to install the same number of flows.

To further validate our analysis, we profile the cache performance. We use the number of L2/LLC misses per thousand retired loads (MPKL) as the metric. To quantify the cache miss penalty, we calculate the ratio of the stall cycles caused by the L2/LLC miss to the total execution cycles. As shown in Figure 4, for the case having even four million flows, most of the load instructions from the cuckoo hash hit the LLC. In contrast, SFH has a significant number of LLC misses when the number of flows reaches 100K, resulting in many CPU stalls.

With the observation in § 3.2, we find that the flow-rule lookup operations occupy a large portion of the packet processing time, although the relevant data has been cached in LLC. This motivates us to conduct a further investigation on the performance overhead of the hash-table lookups. We quantify the number of instruction for each hash-table lookup in Table 1. As we can see, each lookup operation takes about 210 instructions on average. Among them, 48.1% are regular memory instructions (36.2% on *load* and 11.8% on *store*), 21.0% are arithmetic instructions, and 30.9% belong to others that include control flow instructions. This indicates that a large portion (69.1%) of the instructions for flow-rule lookup are related to data accesses with simple arithmetic operations.

Observation: *(1) The state-of-the-art hash table implementation like cuckoo hash successfully reduces memory accesses even with a large number of flows, and most of the useful data for flow classification can be cached in the LLC of a modern server CPU. (2) A significant portion of the instructions for the flow-rule lookup operation are on the data access with basic arithmetic operations. These provide us the hint: the flow-rule lookup in LLC is an ideal target for hardware acceleration.*

Table 1: Number of executed instructions of a single lookup, and its distribution among different types of instructions.

Solution	#instructions per lookup	Memory (Load/Store)	Arithmetic	Others
OVS/Cuckoo hash	210	48.1%	21.0%	30.9%

3.4 Concurrency Overhead

To scale up the throughput of packet processing, the virtual switch usually exploits the multiple CPU cores to increase the parallelism. To understand the concurrency overhead of the cuckoo hash, we use the optimistic locking [18] that has also been adopted in DPDK *rte_hash* library. We profile the performance of the hash table lookups when we run various flow classification workloads, as described in § 3.2 with different packet header size that ranges from 4 to 64 bytes[2]. The profiling results show that the locking mechanism used in the optimistic locking contributes to 13.1% of the total execution time.

Beyond the performance overhead caused by the software locking mechanisms, it is well-known that the core-to-core communication will also introduce significant overhead to the latency of accessing shared data structures. It could take up to more than 100 cycles for a remote core to access a cache line with exclusive or modified state. Accessing a hash table entry in LLC is 2× faster than accessing it in the remote private cache [53]. Therefore, for the shared hash tables that will be frequently accessed, keeping them in LLC can effectively avoid the core-to-core communication overhead.

Observation: *The flow classification suffers from concurrency overhead. Such overhead comes from two major sources: the hardware core-to-core communication and the software locking mechanisms.*

4 HALO DESIGN AND IMPLEMENTATION

According to our performance analysis in § 3, we show that the flow classification is the major bottleneck in virtual switches. The hash-table lookup, as the core operation of flow classification, is critical to scale the packet processing. Our observations motivate us to pursue a near-cache acceleration approach that can be ideally integrated into the general-purpose multi-core CPUs.

4.1 Design Goals and Principles

The goal of HALO is to achieve high-throughput packet processing in virtual switches, with minimal negative impact on the collocated network services on the NFV platform. In our design and implementation, we follow the following specific principles.

- First, the HALO accelerators should be integrated with CPU and LLC to reduce the data movement overhead. The observations in § 3 demonstrate that most of the data used for flow classification can reside in the LLC of modern server CPUs today, it is natural to move the accelerator closer to the data located in LLC.
- Second, the HALO design should be scalable to exploit the architectural parallelism of modern processors. A centralized accelerator itself could become the bottleneck in a multi-core processor, where many packet processing threads are running simultaneously.
- Third, the HALO design should be resource- and energy-efficient. As the shrinking process technology is reaching the physical limit of chip design, there is not much free resource on the CPU dies.

[2]They are the typical sizes of network protocol headers.

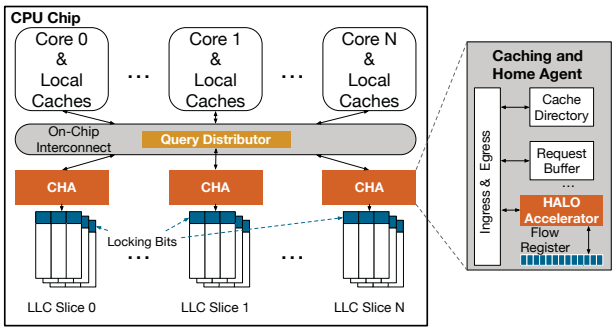


Figure 5: The overview of HALO architecture.

- Fourth, the HALO approach should be generic. We need to provide a simple abstraction for upper-level programs to exploit the benefits of HALO accelerators.

4.2 HALO Overview

To achieve the aforementioned goals, we develop HALO and demonstrate its architecture in Figure 5.

Key idea of HALO. HALO is developed based on the fact that: in modern CPUs with NUCA enabled, the LLC is physically separated into slices along with multiple cores [8, 30, 39]. Each LLC slice has one CHA which is responsible for handling requests and maintaining the cache coherency across the cores. We place HALO accelerator in each CHA to enable the parallelism of near-cache acceleration and data access. We offload the hash-table lookups to each HALO accelerator via the query distributor located in the on-chip interconnect.

To simplify the programming with HALO accelerator, we extend the x86-64 instruction set with three simple all-in-one instructions, which can quickly fulfill the data lookup with accelerators. To handle the read-write concurrency of the hash tables, we use a reserved bit in the metadata of each cache line as the lock bit to implement an efficient and lightweight hardware-assisted locking mechanism.

As discussed, HALO is mainly used to accelerate the flow classification in LLC. For the cases that the flow-rules can be cached in L1 cache, the software-based lookup operations may achieve higher performance due to the lower data access latency. We propose a linear-counting based profiling technique to record active flows within a short period of time, and use the number of active flows as the reference to intelligently decide the computation mode at runtime.

Key components of HALO. HALO is composed of four key components: a set of distributed near-cache accelerators with one query distributor in on-chip interconnect (§ 4.3), a hardware-assisted locking mechanism for improved concurrency support (§ 4.4), an x86-64 instruction extension to simplify the programming with HALO accelerator (§ 4.5), and a linear-counting based flow register for flow counting to support hybrid computation mode that supports both software-based and accelerator-based computing (§ 4.6). We will describe these components in details one by one in the following sections.

HALO workflow. A lookup query sent from the core contains three items: the key address, the table address, and the result destination. These items are specified (explicitly or implicitly) by the proposed lookup instructions. When a lookup query is generated by the core, it will be dispatched to one of the accelerators distributed across LLC

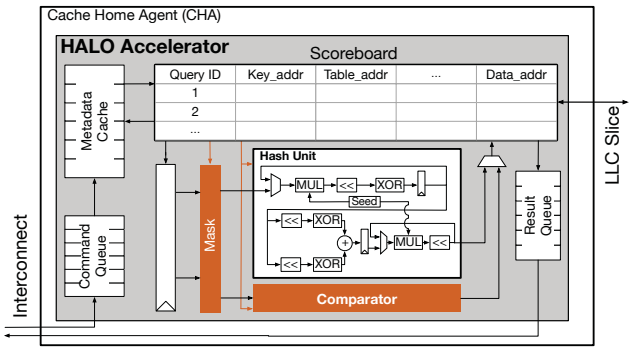


Figure 6: The architecture of HALO accelerator.

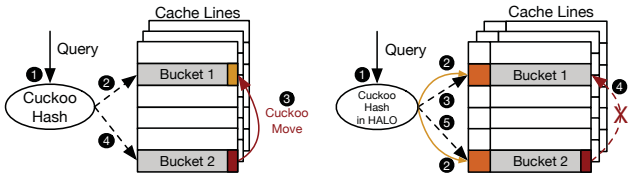
slices for processing. During the lookup procedure, the corresponding cache lines could be locked for concurrent write operations by setting the lock bit in the cache line's metadata. Meanwhile, the hash value of the key is used to update the flow register's bit array to record the number of active flows. The result of the flow lookup is then sent back to the core or written to a specified memory location.

4.3 Distributed Near-Cache Accelerator

Accelerator components. We depict the main components of the proposed HALO accelerator in Figure 6. The `Scoreboard` is responsible for the overall control. It keeps track of the execution progress of each on-the-fly query, generates data access requests to LLC or memory, and sends the results back to the designated destination. Along with the `Scoreboard`, HALO accelerator has three major units to conduct the computation: the `Hash Units`, `Mask Units`, and `Comparators`. Similar to a regular ALU, the `Hash Unit` is implemented with simple logics, such as boolean, shift, and other bit-wise operations. The `Command Queue` and `Result Queue` serve as buffers for the input and output data stream.

To exploit the spatial locality of data accesses, each HALO accelerator has a small Metadata Cache that stores the metadata of recently accessed hash tables. We include `Metadata Cache` in the cache coherence domain by adding one more core-valid (CV) bit to the snoop filter to indicate whether a cache line exists in the `Metadata Cache` or not. Once the cache line is brought into `Metadata Cache`, the CV bit for the corresponding address will be set to "1". The bit is reset when the line is evicted from `Metadata Cache`. When answering snoop request, if the CV bit of the `Metadata Cache` is "1", we snoop into the `Metadata Cache` in the corresponding CHA. With this design, the metadata of a hash table will be cached in the `Metadata Cache` using the existing interconnect logic (as used for distributing all LLC accesses). If the snoop request is a "Read for Ownership", we also invalidate the cache line. This follows the same cache coherence protocol as accessing a core's cache. We believe this additional bit causes minimal hardware cost. Since the metadata of a hash table is unlikely to change after creation, there are very infrequent snoops happen to the `Metadata Cache`.

Query procedure. When a lookup query arrives at the accelerator, the associated table address is used to fetch the table's metadata. The query is then inserted into the scoreboard where it generates several operations in sequence. First, based on the key address associated with the instruction, it generates a data request to fetch the key. Second, after the key has been returned, it issues a hash operation



(a) A typical concurrency error. **(b) Hardware-assisted locking.**

Figure 7: The concurrency issue and the proposed hardware-assisted locking in HALO.

with the key and calculates the bucket index. Third, it generates data request for the buckets. Fourth, it compares the signature in each of the entry of the first bucket, if the signature matches with the key's signature, the corresponding key-value pair will be retrieved by another data request. Finally, if the key in the key-value pair matches with the key obtained from the first operation, the attached data is stored as the result. If not, the same operation will be applied on the alternative bucket. After all the required operations have been finished, the scoreboard commits the query, pushes the result back to the result queue, and the result will be eventually returned to its designated destination.

Query dispatch. HALO solution is integrated in each LLC slice and thus can process multiple queries with multiple accelerators simultaneously. We propose a query distributor to conduct such operation with the on-chip interconnect. Queries from the same core could be dispatched to different accelerators to exploit the parallelism of both near-cache accelerators and data access. The query distributor hashes the table address of each query, and decides which slice the query should be sent to. Note that the current CPU architecture has already had such logic to evenly distribute memory accesses to LLC slices. We reuse such logic in HALO. To avoid congestion, when an accelerator is saturated with on-the-fly queries, it will set a "busy" bit in the query distributor. Until this "busy" bit is cleared, the query distributor will not send any query to the corresponding accelerator.

4.4 Hardware-Assisted Concurrency Lock

In multi-threading use cases, read-write concurrency needs to be maintained carefully for cuckoo hash. We demonstrate a typical example in Figure 7a to show how a function error could happen when such concurrency is not maintained correctly. When a lookup query is received, the program will first conduct cuckoo hash to get the indexes of the two alternative buckets (❶), and then search the buckets sequentially (❷ ❹). However, such operation is not atomic. Thus, before the query (read) thread reaching the second bucket for the matched result, another "update" (write) thread could have moved the entry back to the query thread's firstly searched bucket (i.e., cuckoo move, ❸), causing the "not found" error.

HALO leverages one reserved "locking" bit to achieve the atomicity of read/write operations. We illustrate this mechanism in Figure 7b. When a query reaches the accelerator and the bucket indexes have been calculated (❶), the corresponding cache lines which contain the targeted buckets will be locked by setting the lock bit, which is a reserved bit in the cache line's metadata (❷). During locking, any modification to these cache lines will be forbidden (❹). Specifically, if a core intends to modify an entry, it will first issue a "snoop invalidate

request" to invalidate the cache line in LLC, and request for the ownership. This will trigger a "snoop miss" response indicating the cache line is not successfully invalidated. The requesting core will then reissue a new snoop invalidate request. This request will be granted until the locking bit is reset. After the signature comparison, if a match is found, the corresponding cache line having the key-value pair will also be locked until the data is returned to HALO. The locked state of the cache line will not be cleared until the end of the query (❸ ❺).

With such a hardware-assisted locking mechanism, programmers do not need to implement a similar but costly software-based locking mechanism (see § 3.4).

4.5 Instruction Extension

To simplify the programming of hash-table lookup operations with HALO accelerator, we extend the x86-64 instruction set with three new instructions. As discussed, the accelerator needs to know the key address, table address, and the result destination for each query. To shorten the instruction length like many other x86-64 instructions, for the lookup instructions in HALO, we leverage the general-purpose register RAX/EAX as an implicit operand, and store the table address in the implicit operand. This is because multiple subsequent table lookups are usually sent to the same hash table in real-world applications, the value in the RAX/EAX can be reused. In the following, we describe the three new instructions respectively.

- `LOOKUP_B mem.key_addr reg.result`

`LOOKUP_B` sends the lookup query with the table address (stored in RAX/EAX register) and key address to the HALO accelerator, and returns the lookup result to a specified register. This instruction is executed in blocking mode, which is similar to the `load` instruction. It may block the execution pipeline while waiting for the lookup result from the accelerator. Therefore, it could limit the number of on-the-fly queries and also block the execution of other instructions.

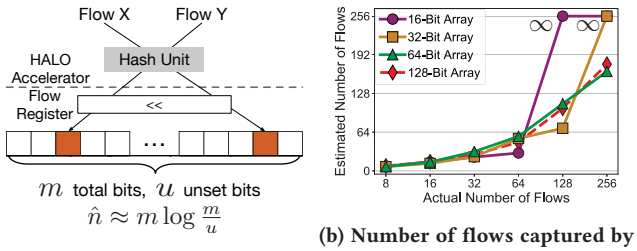
- `LOOKUP_NB mem.key_addr mem.result`

`LOOKUP_NB` is an non-blocking version of the lookup instruction, which behaves similarly to a `store` instruction. It issues the query to the accelerator with the input of the key address, table address, and a memory address for storing the lookup result. Instead of returning the result to the CPU core, it writes the result to a designated memory location. As a result, `LOOKUP_NB` does not block the execution pipeline, which can improve the throughput of data lookups.

- `SNAPSHOT_READ mem.result_addr reg.result`

As for the `LOOKUP_NB` instruction, it is necessary to check whether the query has been completed or not. However, the conventional methods such as polling are expensive for these fine-grained instruction operations. Therefore, we use a new instruction that reads a snapshot of the data without changing the ownership of the cache line.

`SNAPSHOT_READ` takes a "snapshot" of the current value from the memory location specified by the first operand (source operand), and put it to the general-purpose register given by the second operand (destination operand). For the cache line mapped to the source operand, if it is found in the cache hierarchy, its ownership state will not be modified. Since the accelerator is responsible for writing back the result in the non-blocking execution mode, `SNAPSHOT_READ` can help keep the cache line in the LLC to avoid the cache line bouncing between the private caches and LLC. Similar to the vectorized load instructions, `SNAPSHOT_READ` also has a vector version to snapshot an entire cache line.



(a) The flow register in HALO.

(b) Number of flows captured by different flow registers.

Figure 8: The design of the HALO flow register.

To efficiently use `SNAPSHOT_READ` along with `LOOKUP_NB`, programs can issue a batch of `LOOKUP_NB` instructions with the destination addresses mapped to the same cache line but at different offset. The program then snapshots the entire cache line and finds whether all the results are ready or not. For example, the original cache line are zero and HALO accelerators write non-zero pointers into the cache line when each lookup finishes. The program snapshots the entire cache line and uses SIMD instructions with Intel® AVX vectors for integer comparison (e.g., `_mm256_cmpeq_epi64`) to find whether all the results are ready (i.e., non-zero).

4.6 Linear-Counting Based Flow Register

Although HALO can offload hash-table lookup to hardware accelerators, we may prefer to apply the software-based implementation when the active flow count is small, and the majority of the active hash-table entries has already resided in the L1 cache. In such case, the software-based lookup may perform better with small cache footprint (see the details in § 6). Therefore, we propose a run-time monitoring mechanism that can switch the computing mode between HALO-based lookup and software-based lookup, according to the number of active flows.

To track the number of active flows, we propose a bit-array based flow register. We leverage the linear counting algorithm [79] to estimate the cardinality of a set of flows as it provides low time and space complexity.

When a HALO accelerator receives a query, the corresponding bit[3] in the bit array is set (see Figure 8a). The bit array will be scanned periodically (a defined time window), and the number of unset bits is stored in the flow register. After that, we can estimate the number of active flows in that time window with the following formula:

$$\hat{n} \approx m \log \frac{m}{u}$$

where m denotes the size of the bit array and u denotes the number of unset bits.

We demonstrate the efficiency of HALO's flow counting in Figure 8b. We measure the estimation accuracy with different bit-array sizes when feeding different numbers of randomly generated packet flows. As shown in in Figure 8b, a flow register can accurately estimate around 2× more flows than the size of its bit array. In essence, we can use a small bit-array to accurately estimate a larger number of flows that will happen. As we only need to track a small number

[3] Its location is indexed by the value of `(H mod S)`, where `H` is the primary hash value calculated in the lookup procedure, and `S` is the size of the bit array in the flow register.

Table 2: gem5 CPU model configuration.

Item	Configuration
Core	16 OoO cores, 2.1GHz
Caches	8-way 32KB L1D/L1I, 16-way 1MB L2, 16-way 32MB shared LLC, 20 MSHR
LQ/SQ/ROB Entries	128/128/192
Process	22nm
Memory	32GB DDR4_2400

of flows (64 flows according to our evaluation in § 6), a flow register with 32-bit bit-array is large enough to satisfy our requirement with negligible hardware cost.

To switch back to HALO lookup from software-based lookup, the program needs to keep a similar linear counting to estimate the number of active flows. As we only need to maintain a 32-bit bit-array, the software overhead is less of a concern.

4.7 HALO Implementation

We implement HALO solution in the full-system simulator gem5 [6]. We simulate a CPU that is similar to the Intel® Skylake-SP CPU [34]. We show its detailed configuration in Table 2. In our configuration, the LLC is partitioned into 16 slices, each of which is associated with one CHA. We develop the query distributor within the on-chip interconnect, and the HALO accelerator is associated with each CHA component. Each HALO accelerator has a metadata cache which can cache metadata information for 10 hash tables (i.e., 640B). HALO accelerator also enforces boundary check for each memory access. For the scoreboard, we limit the number of on-the-fly queries to 10. HALO has one fully pipelined hash unit for each accelerator. According to our experiments, such configurations maintain a decent balance between performance and hardware cost.

4.8 Discussion on the General Applicability

In this paper, we mainly focus on the hash-table lookup acceleration for flow classification in the virtual switches. The proposed solution could also benefit a wider range of applications such as network services and data structures such as trees. In NFV, many VNFs are hash-table based and require high performance. For instance, the Network Address Translation (NAT) leverages hash tables to quickly find the corresponding WAN IP and port for packets with the LAN IP and port; the packet filter checks whether the incoming packets match with any filtering rules using its internal hash tables. Beyond VNFs, many key-value stores also use hash table as their index. For example, MemC3 [18] applied exactly the same cuckoo hash table described in this paper to `memcached` [13] to achieve higher throughput. We believe HALO can be easily integrated into the aforementioned applications with the three extended x86-64 instructions. Moreover, HALO could also benefit other lookup operations against other data structures such as tree [45, 51, 78] as they share the similar data access procedure. For instance, EffiCuts [78] uses a decision tree for packet classification, it will walk through the decision tree for rule comparison. HALO accelerator can be used to conduct the comparison with the nodes in the tree.

5 EXPERIMENTAL SETUP

5.1 Baseline Configurations

We evaluate HALO with five configurations, described as follows.

- **Software.** For the reference software-based flow classification, we use the cuckoo hash implementation in DPDK's `rte_hash` library. It is highly optimized with software prefetching, memory alignment, and compiler optimization.
- **HALO Blocking.** We use the blocking execution mode of HALO as discussed for `LOOKUP_B` instruction in § 4.5. The CPU core will wait until it gets the lookup result before it issues the next instruction.
- **HALO Non-Blocking.** We use the non-blocking execution mode of HALO as discussed for `LOOKUP_NB` instruction in § 4.5. To exploit the hardware parallelism, we send the queries (each of which consists of eight hash-table lookups) to all the tuples at once and then use one `SNAPSHOT_READ` for every query to check the query results.
- **TCAM.** We compare HALO against TCAM which is used in network devices [4, 81]. TCAM allows fully parallel lookups across entire rule set, it can return the matching results in a few cycles [58].
- **SRAM-TCAM.** To improve the energy efficiency of TCAM, prior work developed a solution to add certain logic units inside the SRAM to emulate the parallelism of TCAM operations [75–77]. It partitions a TCAM table into multiple small sub-tables and stores each sub-table in a SRAM block. We implement a TCAM model and its SRAM-based version in the gem5 simulator.

5.2 Network Workloads

Flow classification workloads. To evaluate the performance of HALO, we first evaluate the performance of EMC flow classification, which issues single hash-table lookups. We also evaluate the MegaFlow flow classification that supports tuple space search.

We generate various flow classification workloads with different number of flows and hash-table sizes. More specifically, for the performance comparison with EMC flow classification with single hash-table lookup, we create the single hash table with the size that ranges from 2^3 to 2^{24} flow entries. We then fill the hash table with different occupancy ratio that ranges from 25% to 90% of the hash-table size to demonstrate the performance trend. For tuple space search, we evaluate the cases of 5, 10, 15 and 20 tuples respectively. Each tuple contains 1024 flow entries [4]. These cases follow the common use cases reported in OVS [60]. We issue 10K hash-table lookups to warm up the system before each experiment.

Network function workloads. To quantify performance benefits of HALO for various network functions (NFs), we use two real-world scenarios in our experiments:

- Collocating the virtual switch with the network services in the shared server for evaluating how flow classification will affect the performance of collocated network services. In this experiment, we emulate a switching process by implementing hash-table lookup functions without really accessing the data but waiting for hundreds of cycles to simulate the HALO lookup latency. We use three network functions that include ACL, Snort, and mTCP as shown in Table 3. These network functions are computation

[4]Note that the "flow" here is megaflow (with wildcard rules), which is different from the ones in EMC (without wildcard rules).

Table 3: The network workloads used in our experiments.

Name	Description	Configuration
ACL [14]	DPDK-based access control list library	Packets are randomly generated to match 6 rules and 1 route with various wildcarding.
Snort [64]	Network intrusion detection system	The traffic generator sends random TCP/IP packets with random payload against the default rules in Snort [72].
mTCP [36]	A scalable user-level TCP stack	Issue 5 million requests with 100 concurrent connections for downloading sample files.
NAT [15]	DPDK-based NAT table (exact match)	Have 1K, 10K and 100K entries for translation.
prads [22]	Passive real-time asset detection system	Have 1K, 10K and 100K entries for asset record.
Packet Filter [3]	Hash-table based IP packet filter	Have 100, 1K and 10K filtering rules selected from an open source ruleset [73].

intensive. We co-run each network function with the emulated switching process on the same core with hyper-threading enabled. We measure the L1D cache miss ratio and the processing throughput of each network function.

- Applying the HALO approach to other network functions for evaluating how HALO accelerator can benefit other hash-table based network services. In this experiment, we use three hash-table based network functions that include NAT, prads, and an IP packet filter, as described in Table 3.

6 EVALUATION

Our evaluation demonstrates that: (1) HALO not only improves the throughput of single hash-table lookup operations (§ 6.1), but also scales the flow classification for the virtual switch by exploiting the hardware parallelism of accelerator computing and near-cache data access (§ 6.2); (2) HALO has minimal negative impact on the performance of collocated network functions by alleviating the private cache pollution (§ 6.3); (3) HALO introduces negligible power consumption and area overhead to the CPU chip, which provides an much more efficient solution, compared to TCAM and its SRAM-based approaches (§ 6.4); (4) HALO is a generic approach that can benefit other hash-table based network functions (§ 6.5).

6.1 Benefit for Single Hash-Table Lookup

We first examine the performance of the EMC flow classification which mainly has single hash-table lookup operations. We run the experiments with different hash-table sizes that range from 2^3 to 2^{24} entries. For each configuration, we populate the hash table with various occupancy rate. We present the normalized performance to the software-based approach in Figure 9. According to the experimental results, we obtain the following observations.

First, when the hash table fits in the LLC (less than 2^{24} entries), HALO achieves up to 3.3× more throughput compared to the software-based approach, and its performance is close to that of the TCAM approaches. As we further increase the hash table size (i.e., the hash-table entries are partially cached in LLC), HALO performs 2.1× better than software-based approach on average. As for the cases with various occupancy rate, they share the similar performance trend.

Second, as we expected, the TCAM and its SRAM-based solutions always perform the best among all the approaches. This is because all the flow rules have been loaded in TCAM in advance, with the assumption that we have enough space of TCAM or SRAM-TCAM to host all the hash-table entries. However, this is not always true in practice due to their power and area restrictions. As we increase the capacity of TCAM, the hardware cost will be dramatically increased, making it become a less attractive choice. We will discuss the trade-off between HALO and TCAM-based approaches in details in § 6.4.

Third, as for HALO, its non-blocking execution mode performs slightly worse (less than 5.3%) than its blocking execution mode. This is because the non-blocking mode uses additional instruction (`SNAPSHOT_READ` per eight queries) to check the cache line for the lookup results, which introduces extra performance overhead for single hash-table lookup.

Finally, when the hash-table size is extremely small (e.g., less than 10), the performance of the software-based approach is better than hardware solutions. This is because most of the hash-table entries can reside in the L1 caches. Although the performance overhead (e.g., computing and locking overhead as shown in Figure 10) introduced by the software-based approach is larger than the hardware-based approaches, the latency of accessing L1 cache is much lower than that of LLC, which offsets the software overhead.

To further understand the performance overhead of each approach, we show the performance breakdown of the hash-table lookup operation using different approaches for different scenarios (i.e., hash-table entries are in LLC or DRAM) in Figure 10. Compared with the software-based approach, HALO reduces the computing time of hash-table lookup by 48.1% with the hardware accelerator design. Furthermore, it facilitate the data access for hash-table lookup operations. Thanks to the near-data nature, HALO directly accesses data in LLC slices from CHA, which is 4.1× faster than that from the CPU core. As for the case of accessing data in DRAM, HALO is 1.6× faster than that from CPU cores. As discussed in § 4.4, such near-data feature also helps HALO reduce the locking overhead by directly setting the locking bit in cache lines without relying on software-based locking mechanisms.

6.2 Benefit for Tuple Space Search

We further explore the performance benefit of HALO for the typical flow classification algorithm – tuple space search. We demonstrate the normalized lookup throughput of different approaches to the software-based solution in Figure 11. Because of their excellent capability of wildcard searching, TCAM and SRAM-based TCAM store all the classification rules with different wildcards in a single table, thus, each lookup requires only one wildcard search operation and each search operation takes only a few cycles. They perform the best among all the proposed approaches. Unlike TCAM-based approaches, HALO and the software-based implementation maintain multiple tuples. Each tuple represents one wildcard pattern.

With the blocking execution mode, the performance improvement of HALO is limited, as we increase the number of tuples. This is because the blocking mode has to serialize the lookup operations

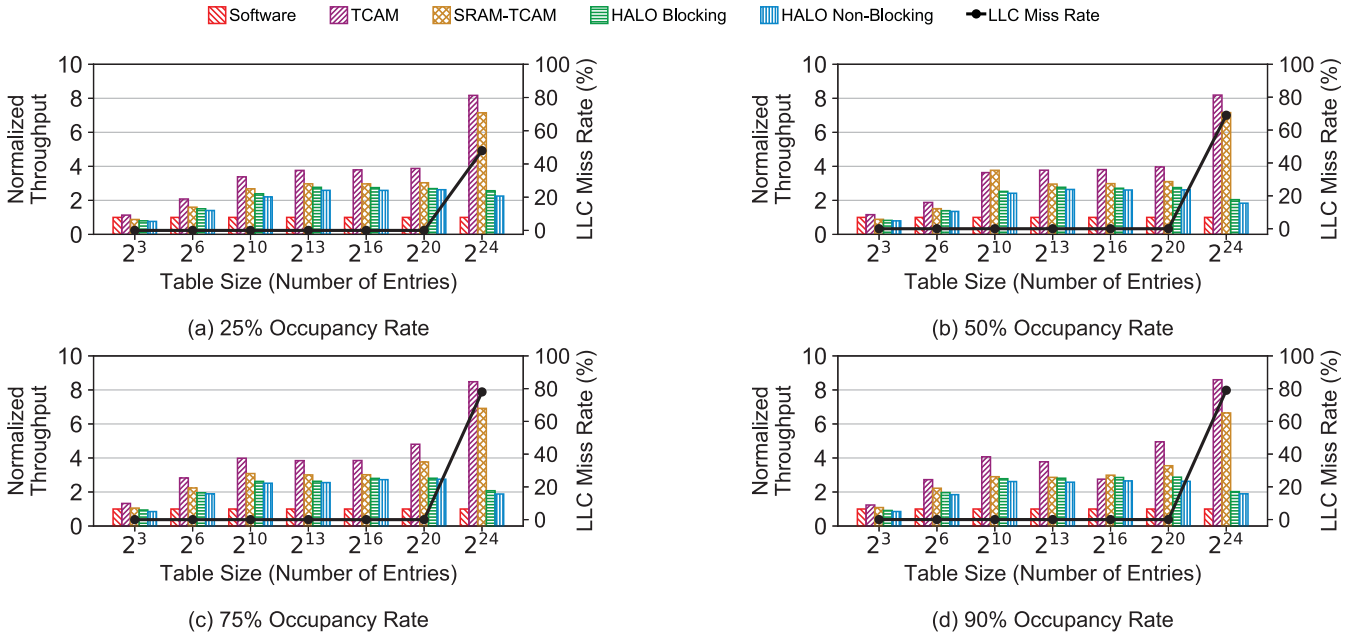


Figure 9: Performance of the single hash-table lookups with various table size and occupancy rate that ranges from 25% to 90%.

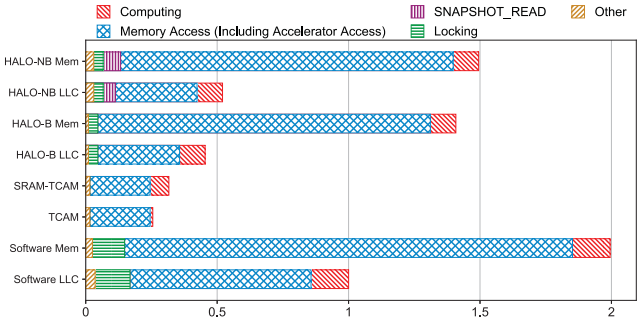


Figure 10: Performance breakdown for the hash-table lookup operation with different approaches in different scenarios. The performance is normalized to the latency of software-based approach when the accessed hash-table entry resides in LLC.

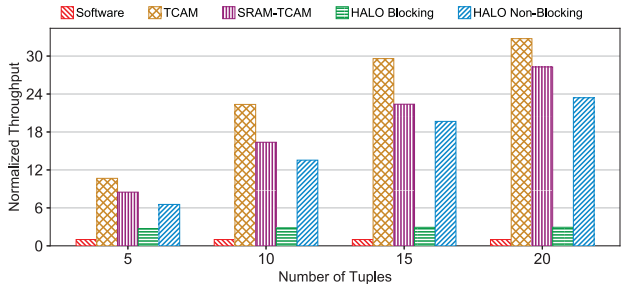


Figure 11: Performance comparison with tuple space search.

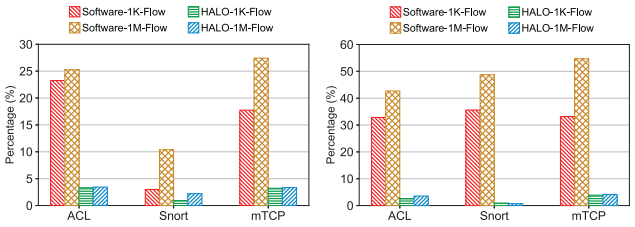
dispatched to different tuples. As the number of on-the-fly instructions in a core is limited, it is hard to fully parallelize the lookups on different tuples by taking advantage of multiple HALO accelerators.

Unlike the blocking mode, the non-blocking mode enables HALO to dispatch more queries in parallel without stalling the CPU core. As shown in Figure 11, the non-blocking mode of HALO scales the flow classification as we increase the number of tuples.

6.3 Benefit for Collocated Network Services

In this section, we discuss the performance interference when running both network functions and the virtual switch process on the same core with hyper-threading enabled. To be specific, we co-run three popular network functions ACL, Snort, and mTCP with the virtual switch respectively. As shown in Figure 12a, the performance of the network functions will decrease up to 23.3% when the network traffic in virtual switch is low (i.e., 1K flows). As we increase the number of flows, the performance interference is becoming more serious. This is mainly due to the resource contention on the shared CPUs. According to our profiling on the CPU caches (see Figure 12b), co-running the network functions with the software-based flow classification in the virtual switch will suffer from much higher L1D cache miss ratio.

HALO has trivial impact (i.e., less than 3.2%) on the performance of the collocated network functions, regardless of the traffic conditions in the virtual switch. HALO alleviates the resource contention on the core's private resources such as L1 and L2 caches, as it offloads the hash-table lookup operations to the accelerators associated with LLC slices. Essentially, each hash-table lookup operation occupies only one instruction slot and one load/store queue (LSQ) entry for execution, it almost does not consume private cache resource.



(a) NF throughput drop. (b) L1D cache miss rate increase.

Figure 12: Performance interference of co-running network functions with the virtual switch.

Table 4: Power consumption and area overhead of hardware-based flow classification approaches.

Solutions		Area/tiles	Power	
			Static /mW	Dynamic /(nJ/query)
TCAM	1KB	0.001	71.1	0.04
	10KB	0.066	235.3	0.37
	100KB	1.044	3850.5	13.84
	1MB	9.343	26733.1	84.82
HALO		**0.012**	**97.2**	**1.76**

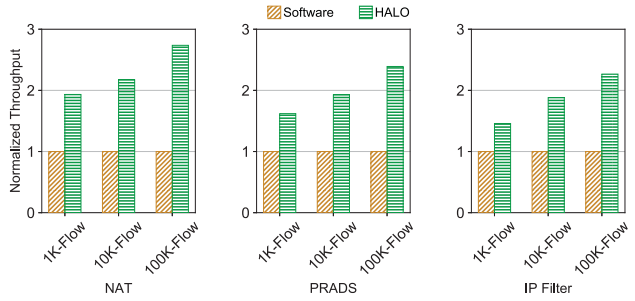


Figure 13: Throughput improvement of hash-table based network functions with HALO.

6.4 Power and Area Analysis

In the previous experiments, we assume TCAM-based solutions can host all the flow information without considering their constrains on power and area cost. In this section, we conduct the energy-efficiency analysis to pave the way for the practical use of HALO. We use McPAT [47] and CACTI [54] to evaluate the on-die power consumption and area cost. As shown in Table 4, the power and area cost of TCAM increase rapidly as we increase its capacity. Given 1MB TCAM, which could store the information for about 100K 5-tuple rules, it consumes an area of 9.3 tiles with more than 26W extra static power, which is challenging for CPU integration.

For the power and area model for SRAM-based TCAM, we follow the models used in [77] to estimate the overhead. Compared with a TCAM with the same capacity and query rate, its corresponding SRAM-based version typically consumes 45% less power, and 57% less area cost. However, it is still much less efficient and scalable, as we compare it with HALO approach.

As shown in Table 4, each HALO accelerator only adds 97.2mW static power, 1.76nJ/query dynamic power, and 1.2% tiles area overhead towards the entire chip budget.

6.5 General Applicability

As discussed in § 4.8, the HALO approach can also be applied to other hash-table based network functions and applications. In this section, we apply HALO to three different network functions NAT, prads, and Packet Filter (see Table 3). We illustrate the performance speedup of HALO compared with the software-based solution in Figure 13. HALO improves the performance of these network functions by 2.3–2.7×, which demonstrates that HALO would also bring significant performance benefits to other network functions in NFV.

7 RELATED WORK

Software optimization for packet processing. Prior studies proposed to accelerate NFV packet processing with software optimization techniques on general-purpose servers. They either bypass the OS kernel with the techniques like DPDK [9], Netmap [62], and mTCP [36] to reduce the context-switch overhead, or speed up the data transfer from network card to host machine with techniques like Intel® DDIO [32] and SR-IOV [12]. In addition, a large number of researches such as ELI [25], Vale [63], NetVM [31], Cuckooswitch [83], have been proposed to optimize the software stack within the virtualized network environment. Our characterization study of virtual switches is conducted on the NFV platform that has been optimized with these proposed techniques. Henceforth, we develop HALO with a focus on the near-cache acceleration for flow classification.

Hardware optimization for packet processing. To facilitate the packet processing, prior studies have exploited the parallelism of existing hardware accelerators such as GPU [24, 28, 40, 74] to process network packets. ClickNP [46] and Microsoft's SmartNICs [7, 20, 21] proposed to offload network functions to the intelligent network card. These approaches significantly improve the performance of processing network traffic across physical machines (i.e., inter-host traffic). However, for intra-host traffic, which resides in a single machine and does not go through the external devices, these solutions have a non-negligible drawback: talking with CPU through PCIe link. With consolidated VNFs, each packet will traverse to and from the device multiple times, causing undesired latency of multiple microseconds [24, 37, 56], and potentially creating bandwidth bottleneck. HALO focuses on improving the performance of processing intra-host traffic with near-cache accelerators. It is compatible with these proposed approaches, but has much lower energy overhead.

Algorithm optimization for packet processing. There are many studies focusing on software algorithms to improve the performance of flow classification. For example, HiCuts series [26, 68, 78] and SAIL [80] use tree-based algorithms and divide rules into multiple dimensions for efficiency. As the core component in the flow classification, the hash-table lookup has also been developed in the field of data management. Typical optimized data structures or systems include Masstree [51], Adaptive Radix Tree [45], MemC3 [18], and SuRF [82]. Our performance analysis of cuckoo hash is conducted based on these optimizations, and the results indicate that there is still space for performance improvement, especially with hardware techniques. Our work HALO is developed for accelerating the hash-table lookups by exploiting the hardware parallelism of the cache architecture of modern server CPUs, and proposing hardware-assisted techniques to reduce the software overhead such as locking.

Near-data processing. Many of the recent studies on near-data processing are on near-DRAM computing. For example, DRISA [48] and DRAF [23] apply simple configurable logic inside the DRAM chips. Lloyd et al. [49] and Hasan et al. [29] integrated hash-table lookup units inside the DRAM subsystem. Our work Halo focuses on the near-cache acceleration, based on the insight that most of the useful data for flow classification have been cached in LLC. Aga et al. proposed compute cache [1], which embeds specific computational logic into the SRAM arrays. However, it requires significant changes to the SRAM hardware, which inevitably increases the complexity of hardware implementation and manufacture. DASX [43] and Widx [41] proposed on-chip accelerators for specific data structures in a centralized manner. Halo exploits the modern NUCA architecture and CHA to develop the distributed near-cache accelerators with minimal hardware cost.

8 CONCLUSION

In this paper, we conduct a thorough performance analysis on the virtual switches. Our findings disclose that the hash-table lookup for the flow classification is the major bottleneck that limits the throughput of network packet processing, although most of useful data for the flow classification has been cached in LLC of modern server CPUs. These observations provide us the insight that hash-table lookup in LLC is an ideal target for hardware acceleration. To this end, we propose Halo, a near-cache acceleration approach that leverages the cache architecture in modern Intel® multicore CPUs. We associate an accelerator with each CHA component to increase the parallelism for data lookup. We extend the x86-64 instruction set with three lookup instructions to simplify the programmability of Halo. Compared with optimized software solutions, Halo improves the throughput of single hash-table lookup by up to 3.3×, and scales the tuple space search by up to 23.4×, while performing up to 48.2× more energy efficient than the fastest but expensive TCAM solution.

ACKNOWLEDGMENTS

We thank Charlie Tai, Andrew Herdrich, David Koufaty, Alex Bachmutsky, Raghu Kondapalli, Ilango Ganga, and Nam Sung Kim for their helpful discussions and suggestions. We also thank the anonymous reviewers for their insightful feedback and comments. The work was initiated and partially conducted when the first author was a research intern at Intel Labs.

REFERENCES

[1] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute Caches. In *Proceedings of the 23rd IEEE Symposium on High Performance Computer Architecture (HPCA'17)*. Austin, TX.

[2] Banit Agrawal and Timothy Sherwood. 2008. Ternary CAM Power and Delay Model: Extensions and Uses. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 16, 5 (2008).

[3] Rohit G Bal. 2017. IP Packet Filtering Using Hash Table for Dedicated Real Time Ip Filter. *International Journal of Wireless and Microwave Technologies* (2017).

[4] Anindya Basu, Girija J Narlikar, and Francis X Zane. 2002. Method and Apparatus for Performing Network Routing with Use of Power Efficient TCAM-Based Forwarding Engine Architectures. US Patent 7,356,033.

[5] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC'10)*. New Delhi, India.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhmmad Shoaib, Nilay Vaish, Mark D. Hill, and Dvid A. Wood. 2011. The gem5 Simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011).

[7] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. 2016. A Cloud-Scale Acceleration Architecture. In *Proceedings of the 49th IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. Taipei, Taiwan.

[8] Zeshan Chishti, Michael D Powell, and TN Vijaykumar. 2003. Distance Associativity for High-performance Energy-Efficient Non-Uniform Cache Architectures. In *Proceedings of the 36th IEEE/ACM International Symposium on Microarchitecture (MICRO'03)*. San Diego, CA.

[9] Intel Corporation. 2018. Data Plane Development Kit (DPDK). https://www.dpdk.org.

[10] Intel Corporation. 2018. Intel® Ethernet Converged Network Adapter XL710 10/40 GbE. https://www.intel.com/content/www/us/en/ethernet-products/converged-network-adapters/ethernet-xl710-brief.html.

[11] Thomas Dietz, Roberto Bifulco, Filipe Manco, Joao Martins, Hans-Joerg Kolbe, and Felipe Huici. 2015. Enhancing the BRAS Through Virtualization. In *Proceedings of the 1st IEEE Conference on Network Softwarization (NetSoft'15)*. London, UK.

[12] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. 2010. High Performance Network Virtualization with SR-IOV. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA'10)*. Bangalore, India.

[13] Dormando. 2018. memcached - A Distributed Memory Object Caching System. https://memcached.org/.

[14] DPDK. 2018. DPDK Programmer's Guide: Access Control. https://doc.dpdk.org/guides/prog_guide/packet_classif_access_ctrl.html.

[15] DPDK. 2018. DPDK Programmer's Guide: Packet Framework. https://doc.dpdk.org/guides/prog_guide/packet_framework.html.

[16] dpif-netdev: per-port configurable EMC. 2018. https://patchwork.ozlabs.org/patch/1000597/.

[17] Ulfar Erlingsson, Mark Manasse, and Frank McSherry. 2006. A cool and practical alternative to traditional hash tables. In *Seventh Workshop on Distributed Data and Structures (WDAS'06)*.

[18] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'10)*. Lombard, IL.

[19] FD.io. 2018. VPP (Vector Packet Processing). https://fd.io.

[20] Daniel Firestone. 2017. VFP: A Virtual Switch Platform for Host SDN in the Public Cloud. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. Boston, MA.

[21] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Lavier Jack, Lam Norman, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Madhan Silva, Ganriel nd Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, Dvid A. Maltz, and Albert Greenberg. 2018.

Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. Renton, WA.

[22] gamelinux. 2018. prads: Passive Real-time Asset Detection System. http://gamelinux.github.io/prads/.

[23] Mingyu Gao, Christina Delimitrou, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Christos Kozyrakis. 2016. DRAF: A Low-Power DRAM-Based Reconfigurable Acceleration Fabric. In *Proceedings of the 43rd IEEE/ACM International Symposium on Computer Architecture (ISCA'16)*. Seoul, Korea.

[24] Younghwan Go, Muhammad Asim Jamshed, YoungGyoun Moon, Changho Hwang, and KyoungSoo Park. 2017. APUNet: Revitalizing GPU as Packet Processing Accelerator. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. Boston, MA.

[25] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. 2012. ELI: Bare-Metal Performance for I/O Virtualization. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. London, UK.

[26] Pankaj Gupta and Nick McKeown. 1999. Packet Classification Using Hierarchical Intelligent Cuttings. In *Hot Interconnects VII*, Vol. 40.

[27] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. 2015. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine* 53, 2 (2015).

[28] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: A GPU-Accelerated Software Router. In *Proceedings of the 2010 ACM SIGCOMM Conference (SIGCOMM'10)*. New Delhi, India.

[29] J Hasan, S Cadambi, V Jakkula, and S Chakradhar. 2006. Chisel: A Storage-Efficient, Collision-Free Hash-Based Network Processing Architecture. In *Proceedings of the 33rd IEEE/ACM International Symposium on Computer Architecture (ISCA'06)*. Boston, MA.

[30] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W Keckler. 2005. A NUCA Substrate for Flexible CMP Cache Sharing. In *Proceedings of the 19th International Conference on Supercomputing (SC'05)*. Cambridge, MA.

[31] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood and. 2014. NetVM: High Performance and Flexible Networking using Virtualization on Commodity Platforms. In *Proceedings of 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. Seattle, WA.

[32] Intel Corporation. 2018. Intel® Data Direct I/O (DDIO). https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html.

[33] Intel Corporation. 2018. Intel® VTune™ Performance Ayalyzer. https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[34] Intel Corporation. 2018. Intel® Xeon® Platinum 8160 Processor. https://ark.intel.com/products/120501/Intel-Xeon-Platinum-8160-Processor-33M-Cache-2_10-GHz.

[35] ixia. 2018. IxNETWORK: L2-3 network infrastructure performance testing. https://www.ixiacom.com/products/ixnetwork.

[36] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. Seattle, WA.

[37] Anuj Kalia, Dong Zhou, Michael Kaminsky, and David G Andersen. 2015. Raising the Bar for Using GPUs in Software Packet Processing. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. Okaland, CA.

[38] Georgios P. Katsikas, Tom Barbette, Dejan Kostić, Rebecca Steinert, and Gerald Q. Maguire Jr. 2018. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *Proceedings of 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI'18)*. Renton, WA.

[39] Changkyu Kim, Doug Burger, and Stephen W Keckler. 2002. An Adaptive, Non-uUniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. San Jose, CA.

[40] Joongi Kim, Keon Jang, Keunhong Lee, Sangwook Ma, Junhyun Shim, and Sue Moon. 2015. NBA (Network Balancing Act): A High-Performance Packet Processing Framework for Heterogeneous Processors. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*. Bordeaux, France.

[41] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for In-Memory databases. In *Proceedings of the 46th IEEE/ACM International Symposium on Microarchitecture (MICRO'13)*. Davis, CA.

[42] Teemu Koponen, Keith Amidon, Peter Balland, Martin Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan Jackson, Andrew Lambeth, Romain Lenglet, Shih-Hao Li, Amar Padmanabhan, Justin Pettit, Ben Pfaff, Rajiv Ramanathan, Scott Shenker, Alan Shieh, Jeremy Stribling, Pankaj Thakkar, Dan Wendlandt, Alexander Yip, and Ronghua Zhang. 2014. Network Virtualization in Multi-Tenant Datacenters. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. Seattle, WA.

[43] Snehasish Kumar, Naveen Vedula, Arrvindh Shriraman, and Vijayalakshmi Srinivasan. 2015. DASX: Hardware Accelerator for Software Data Structures. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ISC'15)*. Newport Beach, CA.

[44] Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. 2005. Algorithms for Advanced Packet Classification with Ternary CAMs. In *Proceedings of the 2005 ACM SIGCOMM Conference (SIGCOMM'05)*. Philadelphia, PA.

[45] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases. In *Proceedings of the 2013 IEEE 29th International Conference on Data Engineering (ICDE'13)*. Brisbane, Australia.

[46] Bojie Li, Kun Tan, Layong Larry Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. 2016. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM'16)*. Florianopolis, Brazil.

[47] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42nd IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. New York City, NY.

[48] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2017. DRISA: A DRAM-Based Reconfigurable In-Situ Accelerator. In *Proceedings of the 50th IEEE/ACM International Symposium on Microarchitecture (MICRO'17)*. Boston, MA.

[49] Scott Lloyd and Maya Gokhale. 2017. Near Memory Key/value Lookup Acceleration. In *Proceedings of the 3rd International Symposium on Memory Systems (MEMSYS'17)*. Alexandria, VA.

[50] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than Your Container. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP'17)*. Shanghai, China.

[51] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-Value Storage. In *Proceedings of*

the 7th European Conference on Computer Systems (EuroSys'12). Bern, Switzerland.

[52] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*. Seattle, WA.

[53] Daniel Molka, Daniel Hackenberg, Robert Schone, and Matthias S. Muller. 2009. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor Systems. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*. Raleigh, NC.

[54] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. 2009. CACTI 6.0: A Tool to Model Large Caches. *HP laboratories* (2009).

[55] Balazs Nemeth, Xavier Simonart, Neal Oliver, and Wim Lamotte. 2015. The Limits of Architectural Abstraction in Network Function Virtualization. In *Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Managemen*. Ottawa, Canada.

[56] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 ACM SIGCOMM Conference (SIGCOMM'18)*. Budapest, Hungary.

[57] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo Hashing. *Journal of Algorithms* 51, 2 (2004).

[58] Kostas Pagiamtzis and Ali Sheikholeslami. 2006. Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey. *IEEE Journal of Solid-State Circuits* 41, 3 (2006).

[59] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. Savannah, GA.

[60] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. 2015. The Design and Implementation of Open vSwitch. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. Okaland, CA.

[61] Ashok Sunder Rajan, Sameh Gobriel, Christian Maciocco, Kannan Babu Ramia, Sachin Kapury, Ajaypal Singhy, Jeffrey Ermanz, Vijay Gopalakrishnanz, and Rittwik Janaz. 2015. Understanding the Bottlenecks in Virtualizing Cellular Core Network Functions. In *Proceedings of the 21st IEEE International Workshop on Local and Metropolitan Area Networks*. Beijing, China.

[62] Luigi Rizzo. 2012. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security'12)*. Bellevue, WA.

[63] Luigi Rizzo and Giuseppe Lettieri. 2012. Vale, A Switched Ethernet for Virtual Machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. Nice, France.

[64] Martin Roesch. 1999. Snort: Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th Systems Administration Conference (LISA'99)*. Seattle, WA.

[65] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. 2015. Inside the Social Network's (Datacenter) Network. In *Proceedings of the 2015 ACM SIGCOMM Conference (SIGCOMM'15)*. London, UK.

[66] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. 2012. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*. San Jose, CA.

[67] Devavrat Shah and Pankaj Gupta. 2000. Fast Incremental Updates on Ternary-CAMs for Routing Lookups and Packet Classification. In *Proceedings of Hot Interconnects*. San Francisco, CA.

[68] Sumeet Singh, Florin Baboescu, George Varghese, and Jia Wang. 2003. Packet Classification Using Multidimensional Cutting. In *Proceedings of the 2003 ACM SIGCOMM Conference (SIGCOMM'03)*. Karlsruhe, Germany.

[69] Venkatachary Srinivasan, Subhash Suri, and George Varghese. 1999. Packet Classification Using Tuple Space Search. In *Proceedings of the 1999 ACM SIGCOMM Conference (SIGCOMM'99)*. Cambridge, MA.

[70] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling Network Function Parallelism in NFV. In *Proceedings of the 2017 ACM SIGCOMM Conference (SIGCOMM'17)*. Los Angeles, CA.

[71] Lin Tan and Timothy Sherwood. 2005. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*. Madison, WI.

[72] Sourcefire's Vulnerability Research Team. 2018. VRT Rule Set. https://www.snort.org/talos.

[73] Emerging Threats. 2018. Emerging Threats Open Rulesets. https://doc.emergingthreats.net.

[74] Janet Tseng, Ren Wang, James Tsai, Saikrishna Edupuganti, Alexander W Min, Shinae Woo, Stephen Junkins, and Tsung-Yuan Charlie Tai. 2016. Exploiting Integrated GPUs for Network Packet Processing Workloads. In *Proceedings of the 2nd IEEE Conference on Network Softwareization (NetSoft'16)*. Seoul, South Korea.

[75] Zahid Ullah, Kim Ilgon, and Sanghyeon Baeg. 2012. Hybrid Partitioned SRAM-Based Ternary Content Addressable Memory. *IEEE Transactions on Circuits and Systems I: Regular Papers* 59, 12 (2012).

[76] Zahid Ullah, Manish Kumar Jaiswal, YC Chan, and Ray CC Cheung. 2012. FPGA Implementation of SRAM-based Ternary Content Addressable Memory. In *Proceedings of 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW'12)*. Shanghai, China.

[77] Zahid Ullah, Manish K Jaiswal, and Ray CC Cheung. 2015. Z-TCAM: An SRAM-based Architecture for TCAM. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23, 2 (2015).

[78] Balajee Vamanan, Gwendolyn Voskuilen, and TN Vijaykumar. 2010. EffiCuts: Optimizing Packet Classification for Memory and Throughput. In *Proceedings of the 2010 ACM SIGCOMM Conference (SIGCOMM'10)*. New Delhi, India.

[79] Kyu-Young Whang, Brad T Vander-Zanden, and Howard M Taylor. 1990. A Linear-Time Probabilistic Counting Algorithm for Database Applications. *ACM Transactions on Database Systems (TODS)* 15, 2 (1990).

[80] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X Liu, Qi Li, and Laurent Mathy. 2014. Guarantee IP Lookup Performance with FIB Explosion. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM'14)*. Chicago, IL.

[81] Fang Yu, Randy H Katz, and Tirunellai V Lakshman. 2004. Gigabit Rate Packet Pattern-matching Using TCAM. In *Proceedings of the 12th IEEE International Conference on Network Protocols (ICNP'04)*. Berlin, Germany.

[82] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *Proceedings of the 44th ACM SIGMOD International Conference on Management of Data (SIGMOD'18)*. Houston, TX.

[83] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G Andersen. 2013. Scalable, High Performance Ethernet Forwarding with Cuckooswitch. In *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT'13)*. Santa Barbara, CA.