Available online at www.sciencedirect.com

ScienceDirect

Future Generation Computer Systems 00 (2016) 1–10

FGCS

Network Measurement for 100Gbps Links Using Multicore Processors

Xiaoban Wu[1], Peilong Li[1], Yongyi Ran[1], Yan Luo[1]

University of Massachusetts Lowell, One University Ave, Lowell, MA, USA, 01854

Abstract

Network measurement has been playing a crucial role in network operations, since it can not only detect the anomalies, but also facilitate traffic engineering. With the fast development of high speed network of 100Gbps and beyond, how to efficiently monitor and measure the network at flow granularity has become a challenging problem. Although there are dedicated network instrumentation appliances, the flexibility of defining measurement network tasks on a programmable platform is very attractive. In this study, we mainly focus on the evaluation of sketch-based network measurement using a multicore platform supported by Intel DPDK, a fast packet I/O library. We describe the versatile system level design options available for implementing such a programmable measurement platform for 100Gbps network links. Through extensive experiments, we investigate these design options and compare their trade-offs. Specifically, we evaluate the performance of the sketch-based measurement in terms of packet drop rate, processing time per packet and delay per packet. Based on the evaluation over the collected data, we propose the best practical measurement implementation to sustain the line rate while achieving the highest level of programmability.

Keywords: Network Measurement, 100Gbps, DPDK

1. Introduction

Network measurement plays a key role in network management and development, since it is essential for trouble shooting, detection of anomaly, traffic engineering and load balancing etc.[1, 2]. With the recent development of ultra high speed networks, the line speed has reached 100Gbps and beyond [3]. To achieve scalable network measurement keeping up with the line rate has become increasingly important.

The measurement of high speed links demands online and low-cost designs that can support the counting of network metrics using streaming algorithms [4, 5]. In such a design, the network measurement platform does not store any packet traces, instead it strives to identify flows and use probabilistic "sketch" methods to estimate the flow metrics (e.g. volume) with an guaranteed error bound. These sketches include but not limited to Count-Min sketch [6], Reversible sketch [7], IBLT [8], Bitmap [9] and HyperLogLog [10] etc. They yield basic metrics of a network flow, with which one can then perform advanced analysis such as identifying the heavy hitters or DDoS attacks [11]. The deployment of the measurement sketches can be in either a single measurement device or a distributed environment [12, 13].

The implementation of sketch based measurement requires a modest amount of resources in particular memory space for storing the sketch data structures. Although possible and popular, hardware based implementation of sketches has several limitations: (a) the cost of memory components such as SRAM and TCAM increases as the size of the sketch data structure becomes prohibitively large for high line rate and fine-granularity measurement [12]; and (b) changing the built-in measurement functions is not an online procedure therefore it cannot adapt to run-time changes of requirements or new functionalities. As a result, there is a strong call for measurement tools that support the flexibility on measurement targets, metrics and granularity since a conventional fixed appliance with preset measurement capabilities often fall short to fulfill new measurement needs. Recently, programmable measurement is proposed to facilitate such flexibility leveraging programmable architectures such as FPGAs and multicore processors [11].

In this paper, we focus on the design of a programmable, high performance 100Gbps measurement platform using general purpose multicore processors due to their scalable performance and friendly programming environment. Modern mul-

Email addresses: Xiaoban_Wu@student.uml.edu (Xiaoban Wu), Peilong_Li@uml.edu (Peilong Li), Yongyi_Ran@uml.edu (Yongyi Ran), Yan_Luo@uml.edu (Yan Luo) ticore processors such as Intel Xeon can have 20+ cores on a single CPU, and the platform could have two or four sockets so the number of cores available for packet processing can reach more than 80 with hyperthreading enabled. The availability of 100Gbps NICs (e.g. Mellanox) can turn a high end x86 multicore server to a network measurement platform instantly. The familiarity of the development environment (for C or other high level languages) on a general purpose processor is very attractive to developers and operators because new measurement tasks can be programmed to the platform easily and at run-time. Such platforms open a variety of opportunities for extension and interaction with other frameworks. For example, one can interface it with SDN controllers or port with P4 [14] based protocol analyzers to examine new types of flows.

However, there exist a number of design challenges due to the hardware and software architecture of a multicore platform. (a) The packets arriving on a NIC needs to be brought over the PCIe bus to the system memory (DRAM), thus incurring bus transaction overhead. As the operating system manages the hardware resources, it typically relies on kernel drivers to receive and transmit packets. Yet the performance of the kernel drivers and protocol stack are shown to lag behind high line rates [15]. It is very challenging to receive packets at 100Gbps line rate on a Linux platform; (b) Without a dedicated queue management unit, the system need to spend CPU cycles on managing packet buffer and queues; (c) while the system memory is abundant on a multicore platform, the memory hierarchy (L1-L2-L3 cache and DRAM) introduces variations on memory access delays, making it nontrivial to bound the packet/flow processing latency; and (d) the multiple cores can be leveraged to process flows in parallel or pipeline, and it takes considerable effort to achieve an efficient design and keep fine tuning it.

In this paper, we study the design of a programmable network measurement platform for line rate of 100+Gbps using x86 multicore processors and off-the-shelf 100Gbps NICs with the support of DPDK [16], a high speed packet I/O library. We propose several design options for the aforementioned sketch-based heavy hitter detection methods (i.e. Count-Min, Reversible and Simple Hash Table). In particular, we optimize the sketch algorithms with concurrent primitives for improved parallelism on a multicore architecture. Then we compare and analyze the performance of our proposed schemes in terms of packet drop rate and packet processing delays. Finally we point out the best practical implementation according to the direct observation. To the best of our knowledge, our work presents the first x86 multicore based system for measuring network flows at 100Gbps rate, and provides in-depth analysis on the impact of system parameters on the performance. Our findings is instrumental to ultra high speed network processing using general purpose multicore microprocessors.

This paper is organized as follows. Section 2 describes the background of sketch-based measurement and DPDK. Section 3 introduces in detail several different parallel designs. The performance evaluation results are presented in Section 4. Finally, the paper is concluded in Section 5.

2. Background

In order to make it easy to understand our measurement designs based upon DPDK and sketches, in this section, we investigate the DPDK and three different sketches. First, we briefly introduce the architecture of the DPDK, which provides high throughput packet I/O in user space. Second, we introduce streaming algorithms and the Count-Min Sketch (CMS) [6], Reversible Sketch (RS) [7] and Simple Hash Table (SHT) [17], which are used to provide the data structure for network measurement.

2.1. DPDK

The Intel Data Plane Development Kit (DPDK) consists of a set of libraries, which can be used to provide high throughput packet I/O in user space, addressing the performance issues of OS kernel device drivers and protocol stacks. The involvement of the DPDK in our design is inspired by the recent development of DPDK applications. In [18], the authors integrate the DPDK with Cuckoo hash table to design the Cuckoo switch, which can reach very high throughput. In [19], the author compares the throughput of two different designs of OpenvSwitch (OVS), OVS and OVS-DPDK. The OVS-DPDK has much better throughput than the OVS. In [20], the authors propose a high speed Statistical Traffic Analysis tool that combines the Intel DPDK framework with Tstat, a passive traffic analyzer, to achieve 40Gbps of line rate processing.

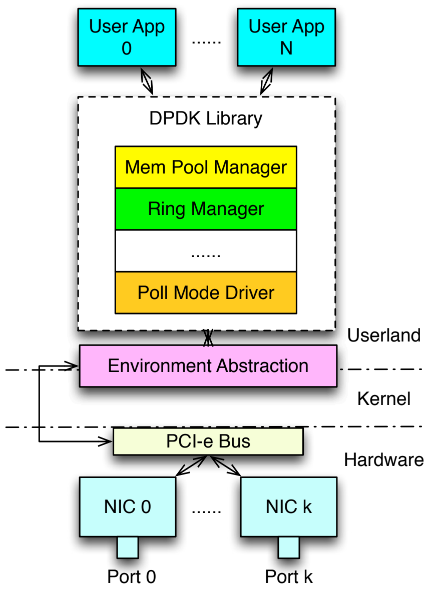

Figure 1. DPDK Architecture

The general architecture of DPDK is presented in Figure 1 with more details in [21]. The following aspects of the DPDK are mostly relevant to our designs: multicore framework, ring buffers and poll-mode drivers (PMD).

2.1.1. Multicore Framework

Nowadays, modern high-performance servers usually employ non-uniform memory access (NUMA) architecture. The

DPDK supports the NUMA architecture and provides the multicore framework. In the Environment Abstract Layer (EAL), according to the input parameters, it automatically detects the usable cores and sets the threads affinity to certain cores. Whenever the user's application needs memory, each thread can always choose the closest socket to allocate memory. Besides, the DPDK adopts huge page memory mechanism, all the key memory usage will be mapped to huge pages, for example, the memory of ring buffers and memory pools (mempools), hence the translation lookaside buffer (TLB) misses are greatly reduced. The DPDK allows multiple RX queues and TX queues at the same time, which can significantly increase packet I/O performance.

2.1.2. Ring Buffers

The DPDK implements the ring buffers specifically for each scenario, which includes single-producer, single-consumer, multiple-producer and multiple-consumer. The multiple-producer and multiple-consumer ring buffers are implemented by atomically updating index and parallel reading, writing operations. In default, all the RX and TX queues use multiple-producer and multiple-consumer ring buffers. Due to this fact, later in the evaluation section, we find that freeing time per packet is relatively large. For single-producer and single-consumer ring buffers, we use them to manage the packet transition in one of our designs.

2.1.3. Poll Mode Driver (PMD)

The DPDK provides a set of signed-off PMD for a set of network interface controllers (NICs) (e.g. Intel *ixgbe*, Netronome *nfp* and Mellanox *mlx5*, etc.) The DPDK provides 4 basic PMD RX modes, which are NONE, RSS (Receive Side Scaling), DCB (Data Center Bridging) and VMDQ (Virtual Machine Device Queues). The NIC vendors determine if their product can support each mode and provide the corresponding APIs. In our measurement platform, we mainly consider NONE and RSS as RX mode, since DCB and VMDQ are designed for the communications between Virtual Machine (VM) and hosts while our design does not involve any VMs. Moreover, the RX mode NONE and RSS will result in different parallel designs, since NONE implies only one RX queue and RSS implies multiple RX queues.

2.2. Streaming Algorithms and Sketches

Streaming algorithms are algorithms for processing data streams in which the input is presented as a sequence of items and can be examined in only a few passes (typically just one). Streaming algorithms fit in very well with network traffic analysis and monitoring, since there might be massive sequence of data within certain period of time and those data can only be processed once. In the following, we briefly present three streaming algorithms which are Count-Min sketch, Reversible sketch and Simple Hash Table. And, in our measurement designs all these three sketches are used to find heavy hitters which are defined as the flows that have the biggest number of packets within certain period of time.

2.2.1. Count-Min Sketch

The Count-Min sketch was first proposed by Graham Cormode and S. Muthukrishnan in 2003 [6]. The Count-Min uses a two dimensional matrix (d rows and w columns) to store and update the necessary flow information whenever a new packet arrives, and meanwhile it compares d counter values from each row and find the minimum as its appropriate estimate. The Count-Min sketch can be used in a streaming fashion to find the heavy hitter. In order to do so, a temporary global heavy hitter is maintained. Whenever each new packet comes in, the Count-Min sketch is used to find the estimated counter value for this new packet and then update the temporary global heavy hitter if the counter value is more than that of the temporary global heavy hitter. The main disadvantage of this sketch is that it cannot be used alone to find the heavy hitter in the distributed case, if the packets in the same flow are scattered into multiple different Count-Min sketches, which may result in the unreliable estimation.

2.2.2. Reversible Sketch

The Reversible sketch was first introduced by Schweller et al. in 2004 [7], and originally designed together with k-ary Sketch (similar to Count-Min sketch in the updating stage, but different from Count-Min in the finalizing stage) to detect the heavy change. Later Yu et al. integrated the Count-Min sketch with the Reversible sketch to find heavy hitters [11]. The "Reversible" means it can automatically find the corresponding flow information for the heavy hitters in the finalizing stage. To achieve this, the Reversible sketch adopts modular hashing, IP mangling and set intersection to store and update the flow information, and in the finalizing stage it uses set intersection to find the true heavy hitter among all the potential heavy hitters. The main disadvantage of Reversible sketch is that it consumes too much memory. In [7], Schweller et al. only consider the IP address which is only 4 bytes with 4 modular hashing functions. Even in this case, for each counter in the matrix, there are at most 512 ($4 \times 4 \times 32$) extra bytes associated with it, which can become even costly if we have to store additional information such as 5-tuples. Not surprisingly, some work such as [11] choose to sample and measure only a few source-destination pairs to reduce memory usage. In contrast to the disadvantage of the Count-Min sketch, Reversible sketch can work in distributed case, since the information of each flow has been recorded.

2.2.3. Simple Hash Table

The Simple Hash Table was studied by Alipourfard et al. in [17]. The authors emphasize that the Simple Hash Table has better performance in terms of throughput and latency than heap and sampling based solutions under 10Gbps network. The Simple Hash Table is similar to a mini version of the Count-Min sketch, since it has only 1 row in the matrix. The big difference between Simple Hash Table and Count-Min sketch is that in addition to a single counter contained in each bucket, each bucket of the Simple Hash Table contains a 5-tuple which is used to record the flow information. It comes no surprise that the Simple Hash Table is faster than the Count-Min sketch in the updating stage, but in the finalizing stage the estimate could go worse if the hash collision of the traffic occurs frequently. This sketch can be used in the distributed case, since the information of each flow would be recorded.

We summarize the key feature of each sketch in the table 1.

3. Measurement Design

In this section, we present our measurement design options in detail. Based on the selected DPDK RX modes (i.e. NONE and RSS) and the way each CPU core updates a sketch data structure, we propose several design options for the aforementioned sketch-based heavy hitter detection methods (i.e. Count-Min, Reversible and Simple Hash Table). The details of the proposed design options are presented in the following subsections.

3.1. Parallel Designs

In a multicore environment, a parallel design dictates the interaction among the CPU cores and how the measurement function is implemented. The two DPDK RX modes (i.e. NONE and RSS) influence our parallel designs. In the NONE mode, only one DPDK RX queue is present, hence only one core needs to be assigned to listen on the DPDK RX queue. Without loss of generality, we treat this listening core as the master core, and all the other cores as the slave cores. In contrast, using the RSS mode, multiple DPDK RX queues can be enabled at the same time, and all the cores can receive the packets independently in parallel.

We first study the design options in NONE mode, in which the interaction between the master core and a slave core can be either synchronous or asynchronous, as explained in Sections 3.1.1 and 3.1.2, respectively.

3.1.1. SYNC Design

"SYNC" means that all cores work together in a synchronous way. As shown in figure 2, when the master core receives a batch of packets, it informs the slave cores to fetch and start working on their corresponding packets, and then it keeps waiting until all the slave cores finish their work. After each slave core completes their work, it sends a feedback to the master core and then waits for a new round of notification from the master core. When the master core receives a feedback from all the slave cores, it can move on and receive the next batch of packets. More detailed procedures can be found in Algorithm 1.

3.1.2. ASYNC Design

Different from the SYNC design, "ASYNC" means that the master core and all the slave cores are working in an asynchronous way and do not wait for each other. To achieve this, as illustrated in figure 3, the master core needs to maintain a global ring buffer for each slave core. Whenever the master core receives a batch of packets, it splits all the packets into several groups first and then pushes each group into the corresponding ring buffer of the slave core. For each slave core, it keeps

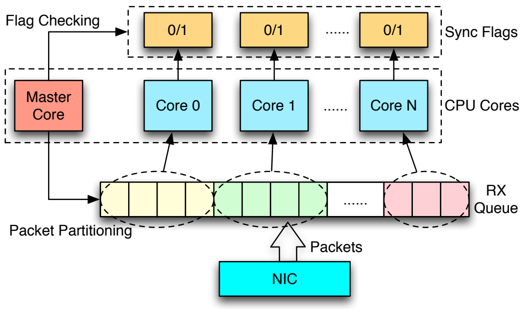

Figure 2. SYNC Design

Algorithm 1 SYNC Design

```
1:  Global Data: $slave\_start\_flag[]$, $slave\_end\_flag[]$, $received\_pkts[]$
2:  DPDK API: $rx\_burst()$
3:  function Master_Core_Subroutine( )
4:      while True do
5:          $rx\_burst(received\_pkts[])$;
6:          for each $flag$ in $slave\_start\_flag[]$ do
7:              $flag = 1$;
8:          while At least one $flag$ in $slave\_end\_flag[]$ is 0 do
9:              continue;
10:         for each $flag$ in $slave\_end\_flag[]$ do
11:             $flag = 0$;
12: function Slave_Core_Subroutine( )
13:     while True do
14:         while my $flag$ in $slave\_start\_flag[]$ is 0 do
15:             continue;
16:         Set my $flag$ in $slave\_start\_flag[]$ equal to 0;
17:         Work on my part of the $received\_pkts[]$;
18:         Set my $flag$ in $slave\_end\_flag[]$ equal to 1;
```

polling on its corresponding ring buffer and begins to work on the packets if there exists any. For the ring buffers, we take the advantage of the DPDK ring buffers and their associated APIs. The details are shown in the Algorithm 2.

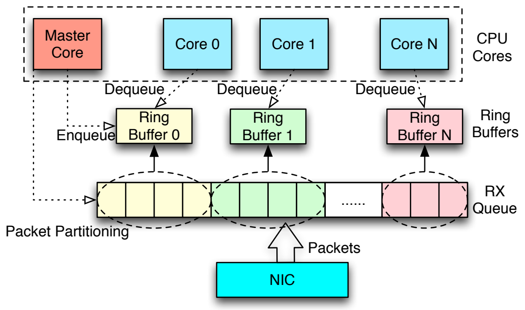

Figure 3. ASYNC Design

3.1.3. RSS Design

As shown in the Figure 4, the following steps are used to implement the RSS feature of the NIC. (1) Specify which fields of the received packet are chosen to be hashed. (2) Specify which hash function is going to be used to hash the chosen fields. (3) Choose the last 7 bits of the hash value to form the final hash value, since the redirection table has only 128 entries. (4) Find the core ID stored in the redirection table according to the final hash value. (5) By following the core ID, the received packet is directed to the corresponding DPDK RX queue.

Table 1. The key Feature of Sketches

Sketch	Data Structure	Pros	Cons
CMS	(1) Matrix with several rows (2) Each entry of the matrix consists of a counter	Ensure certain accuracy	Can not work alone in distributed measurement
RS	(1) Matrix with several rows (2) Each entry of the matrix consists of a counter and several sets	(1) Ensure certain accuracy (2) Can work in distributed measurement	Memory consumption is high
SHT	(1) Matrix with only one row (2) Each entry of the matrix consists of a counter and a 5-tuple	(1) Can work in distributed measurement (2) Low memory consumption	Accuracy is not guaranteed

Algorithm 2 ASYNC Design

```
1: Global Data: slave_ring_buffer[]
2: DPDK API: rx_burst(), enqueue(), dequeue()
3: function Master_Core_Subroutine( )
4:     while True do
5:         rx_burst(received_pkts[]);
6:         for each buffer in slave_ring_buffer[] do
7:             enqueue(some portions of the received_pkts[] into buffer);
8: function Slave_Core_Subroutine( )
9:     while True do
10:        while my buffer in slave_ring_buffer[] has packets do
11:            dequeue(the packets out of my buffer);
12:        Work on those dequeued packets;
```

After all these setup, as shown in Figure 5 and Algorithm 3, multiple DPDK RX queues can be kept at the same time, and each core can receive and handle the packets from each DPDK RX queue in parallel.

Since the NIC transfers each packet to its corresponding DPDK RX queue by applying the same hash function, packets within the same flow would be steered to the same DPDK RX queue. Earlier in section 2.2.1, we mentioned that the Count-Min sketch cannot be used alone to find the heavy hitter in the distributed case, if the packets in the same flow are scattered into multiple different Count-Min sketches. But, with the benefit from RSS, the disadvantage of Count-Min sketch can be remedied.

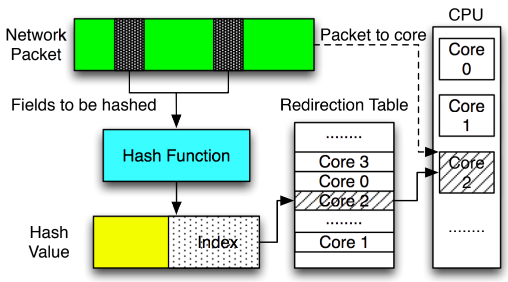

Figure 4. Principle of RSS

Algorithm 3 RSS Design

```
1: DPDK API: rx_burst()
2: function All_Core_Subroutine( )
3:     while True do
4:         rx_burst(received_pkts[]);
5:         Work on the received_pkts[];
```

3.2. Separate and Shared Design

Generally, there are two ways to store the sketch data structures accessed by multiple cores: i.e. "shared" or "separate" copies. For the case of separate copies, each core maintains and

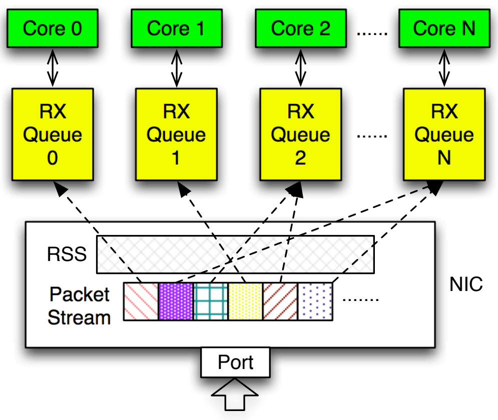

Figure 5. RSS Design

updates its own local sketch independently and safely, which is adopted in [6, 7, 17]. Using a shared copy can conserve memory space and take advantage of cache locality. However all the cores share the same global sketch, and care should be take to avoid conflicts when two or more cores are updating the same field.

In this work we identify and address three key issues out of the concurrent update of the sketches (CMS, RS and SHT), i.e. updating a counter, updating a set, and updating a critical section. It is simple to implement concurrent update by using *spinlock*, but *spinlock* usually consumes many CPU cycles, which is our major concern under the 100 Gbps network environment. Instead, we implement the concurrent update by atomic operation, which takes fewer CPU cycles. The details are as follows.

3.2.1. Update a Counter Concurrently

In order to update a counter concurrently, we take advantage of the DPDK API: *static inline int rte_atomic32_cmpset(volatile uint32_t *dst, uint32_t exp, uint32_t src)*. This API uses the assembly instruction *cmpxchgl* to implement the CAS (compare and swap) atomic operation. It compares *dst* with *exp*, if *dst* equals *exp*, it updates *dst* with *src* and return 1. Otherwise, it returns 0. The details are presented in Algorithm 4.

Algorithm 4 Update a Counter Concurrently

```
1: function Update_Counter( &counter, increment )
2:     success = 0;
3:     while success == 0 do
4:         current = counter;
5:         next = current + increment;
6:         success = rte_atomic32_cmpset(&counter, current, next);
```

3.2.2. Update a Set Concurrently

For the Reversible sketch in section 2.2.2, we need to update several associated sets when we update a counter. In our measurement design options, we implement these sets by the traditional hash table with a linked list. Therefore this issue is transformed into updating a hash table concurrently. Different from earlier work in [22, 23], for sketch based measurement, we do not need to delete an element from the hash table and do not need to resize the hash table, we can simply use atomic operation to concurrently update a hash table. The major problem happens at when multiple cores insert their local elements into the same linked list concurrently. We have to make sure that the size of the linked list grows serially. To achieve this, we use the following data structure for each bucket of the linked list. Among the fields, *data* is used to store the flow information,

```
struct Bucket {
    unsigned char* data;
    struct Bucket* next;
    uint32_t filled_flag;
    uint32_t finish_flag;
};
```

the *next* is used to form the linked list, the *filled_flag* indicates the bucket is filled when it equals 1, and the *finish_flag* denotes the insertion is complete when it equals 1.

For fast checking if an element has already existed in the linked list, define a function called *int find_element(linked_list, element, &element_location)*. If this function finds the *element* in the *linked_list*, it returns 1. Otherwise, it returns 0 and updates *element_location* with the last location of the marching pointer of the *linked_list*. To search the *element*, for the sake of safety, we only march the marching pointer if both *filled_flag* and *finish_flag* equal 1.

Another important operation is to insert an element into the set concurrently. To do this, first we need to find the corresponding linked list in the hash table. Second, we need to find if the element has already existed in the linked list. Third, if the element has not existed in the linked list, we can now insert the element into the linked list by atomic operation. The details are Algorithm 5, where lines 5-7 indicate that the *element* has

Algorithm 5 Update a Set Concurrently

```
1:  function Update_Set( hashtable, element )
2:      Find the linked_list for the element;
3:      success = 0;
4:      while success == 0 do
5:          status=find_element(linked_list, element, &element_location);
6:          if status == 1 then
7:              return;
8:          start = element_location;
9:          success = rte_atomic32_cmpset(&(start → filled_flag), 0, 1);
10:         if success == 1 then
11:             Allocate memory for start → data;
12:             Update start → data with the data from element;
13:             Allocate memory for start → next;
14:             start → finish_flag = 1;
15:             return;
16:         else
17:             while start → finish_flag == 0 do
18:                 continue;
```

already existed in the set, line 9 uses atomic operation to make sure that only one core can have *success* = 1 and grant this core the privilege to insert the *element*, lines 10-15 insert the *element* into the set and allocates one more bucket memory as the tail of the linked list, and lines 16-18 make sure all the other potential cores are waiting for the privileged core to finish the update of the linked list. All the cores can never jump out of the outmost while loop until it either finds that the *element* has already existed in the set or it successfully inserts the *element* into the set.

3.2.3. Update a Critical Section Concurrently

For the Count-Min Sketch (CMS) mentioned in section 2.2.1, a global heavy hitter (5-tuple) should be maintained and updated whenever the counter value of another flow exceeds the current heavy hitter. Since the atomic operation can only work up to 8 bytes, we cannot use a single atomic operation to finish a 17-bytes update (a 13-bytes 5-tuple and a 4-bytes counter). Hence, we would better resort to a critical section. The detailed implementation of a critical section is presented in Algorithm 6, where lines 4, 5, 9 ensure that only one core can change the flag

Algorithm 6 Update a Critical Section Concurrently

```
1:  Global Data: flag initialized with 0
2:  function Update_Critical_Section( )
3:      while True do
4:          success = rte_atomic32_cmpset(&flag, 0, 1);
5:          if success == 1 then
6:              Execute the critical section;
7:              rte_atomic32_cmpset(&flag, 1, 0);
8:              break;
9:          else
10:             while flag == 1 do
11:                 continue;
```

so that only this core can execute the critical section. Meanwhile all the other cores have to wait in the inner while loop until the *flag* turns 0 again, only after that they can start next round of competition.

4. Evaluation

With all design options discussed in section 3, we evaluate the performance of each design. The evaluation metrics we apply in this section include packet drop rate (PDR), packet processing time (PPT) of sketch update and total packet delay (the sum of RX time, processing time and freeing time).

4.1. Experiment Platform and Methodology

We employ two identical Dell Power Edge R730 servers in the experiments as depicted in Figure 6. Each server contains two 6-core CPUs that reside separately on 2 sockets (socket #0 and #1) connected with Intel QuickPath Interconnect (QPI). Both servers are equipped with a single 100GbE card on the server socket #1 over a PCIe x16 slot. The aforementioned two NICs are connected back-to-back via a QSFP-28 cable for testing purpose. We set one server as the RX end and the other server as the TX client. The detailed hardware and software specifications are listed in Table 2.

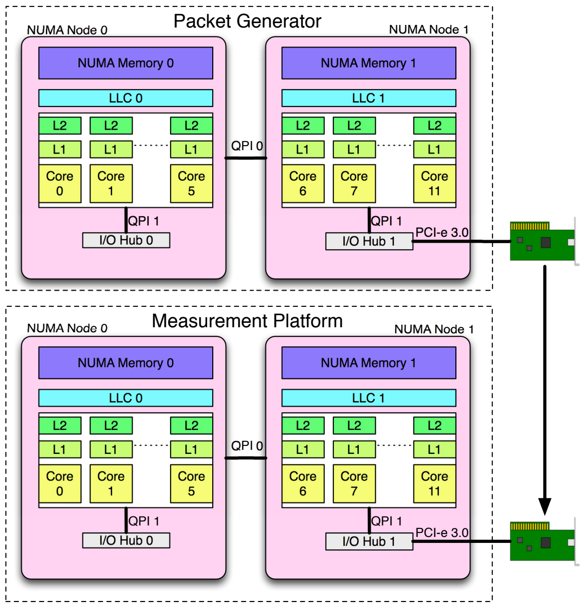

Figure 6. Experiment Platform and Test Methodology

Table 2. Experiment Platform Specification

Item	Specification
CPU 0 and 1	Intel Xeon E5-2643 6 cores @ 3.4GHz
L1i Cache	32 KB
L1d Cache	32 KB
L2 Cache	256 KB
Last Level Cache 0 and 1	20 MB
Memory 0	8 GB DDR3 @ 1.6 GHz
Memory 1	16 GB DDR3 @ 1.6 GHz
NIC 0 and 1	Mellanox ConnectX-4 EDR 100GbE
Host OS	Ubuntu 16.04 Desktop
DPDK	Version 16.04

4.2. *Packet Generator and Packet Size Study*

We adopt Intel's *pktgen-dpdk* [24] as a template packet generator, and extend it with the flexibility of generating packets with random L2/L3/L4 headers and random sized payload. This design intends to increase the packet diversity to serve for the hashing step in the aforementioned sketch algorithms.

To understand the impact of packet size on the experiment system, we study the system throughput and packet drop rate with three different packet sizes - small (64 bytes), medium (754 bytes), and large (1514 bytes). As shown in Figure 7, with this modified packet generator, we are able to generate random UDP packets with small size at 72 Mpps (or 34 Gbps) and large size at 8 Mpps (or 90 Gbps). The packet drop rate, however, increases dramatically when the packet size becomes small. This is because the very high packet rate saturates the receiving capability at the RX end with small packets. Therefore, to pressure-test the RX end and augment the difference in the evaluation results, we will employ 64-byte small packets in all the following experiments if not specified otherwise.

4.3. *DPDK Mempool Size*

Another important configuration in the experiments is the mempool size of each RX queue. In a DPDK application, each RX queue is configured with a predefined mempool size for packet buffering. To figure out an optimal mempool size for our evaluation, we study the effect of the mempool size for a DPDK application in terms of packet drop rate. As shown in the Figure 8, the x-axis denotes the mempool size with the unit of "number of packet slots" and the y-axis stands for the PDR in percentage. While with the difference less than 5%, we can observe that 16K mempool size renders the optimal PDR performance with the only exception for 1514-byte packets. These results indicate that it is performance-optimized and space-efficient to choose the 16K mempool size. We therefore define mempool size as 16K in the following experiments if not specified otherwise.

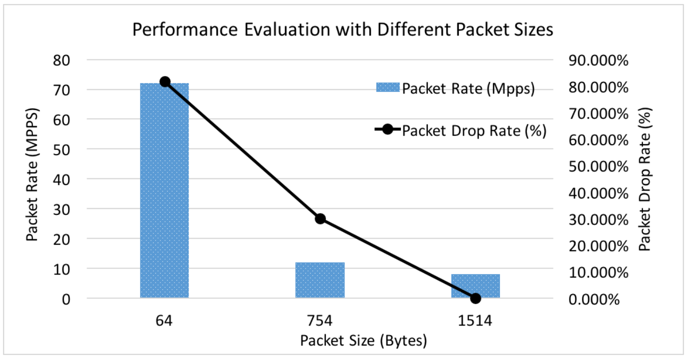

Figure 7. Performance Evaluation with Different Packet Sizes

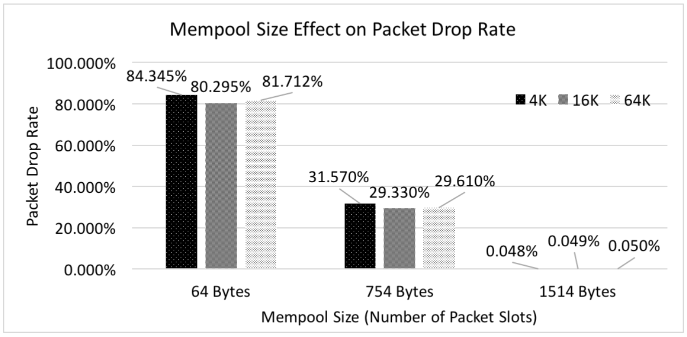

Figure 8. Mempool Size Effect on Packet Drop Rate

4.4. *Architecture Parameter Exploration*

We explore the performance effect of various architecture parameters such as number of cores, and memory hierarchy in this subsection. For annotation purpose, we use the format of "X-Y-Z" for experiment configurations, where "X" denotes for *SYNC*, *ASYNC* or *RSS*, "Y" being *Separate* or *Shared*, and "Z" means *Count-Min Sketch (CMS)*, *Reversible sketch (RS)* or *Simple Hash Table (SHT)*. For example, "SYNC-Shared-RS" stands for the synchronous design with shared Reversible Sketch data structure. The total number of evaluation subjects is therefore the combination of "X, Y, Z", with the exception that CMS cannot have SYNC and ASYNC design as explained in Section 2.2.1.

4.4.1. *NUMA and Cores*

As NUMA architecture becomes the mainstream hardware configuration in commodity multi-CPU servers and datacenters, it is critical to understand how NUMA settings and the number of cores can significantly affect the performance of network measurement. We employ two test cases - "SYNC-Separate-SHT" and "RSS-Separate-SHT" in this evaluation, and study the PDR and packet delay changes with different cores and NUMA configurations.

As shown in the Figure 9, the packet drop rate decreases by up to 50% when changing the number of processing cores from 2 to 6, regardless of the socket settings. Besides, the placement of running measurement tasks with regard to the NUMA sockets makes a huge difference (up to 20%) on the PDR. This is because if a running process has to access the remote device and memory on a different NUMA socket, it will trigger tremendous cache misses on the local memory hierarchy and thus increase the packet processing delay.

Then we study the latency performance with the "SYNC-Separate-SHT" case. As shown in the Figure 10, where "sX-cY" on the x-axis denotes number of Y cores on socket X, both RX latency and processing latency (updating SHT) can reduce by up to 50% if the measurement task is running on the NUMA socket with NIC (socket #1). However, in this case increasing the number of cores does not contribute too much on decreasing the total latency.

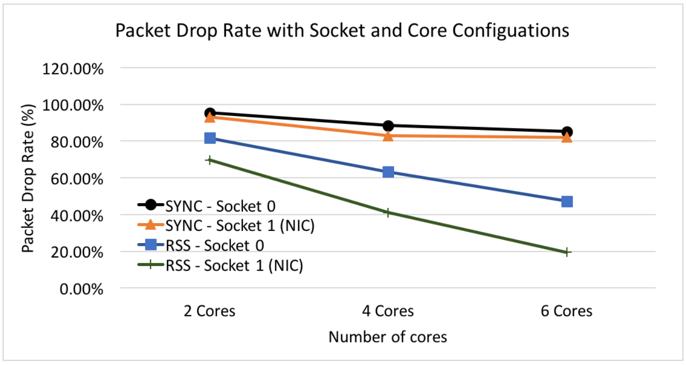

Figure 9. Packet Drop Rate with Different Socket and Core Configurations

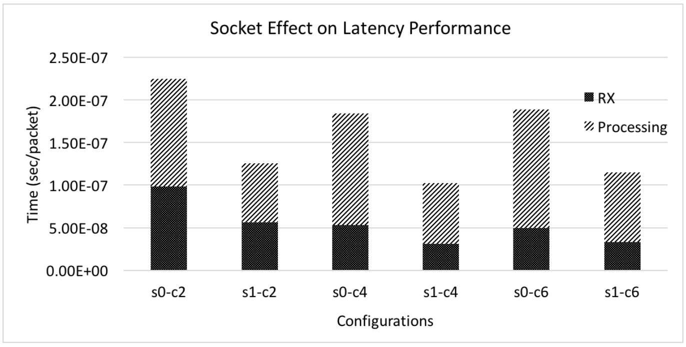

Figure 10. How Socket Effects Latency Performance

4.4.2. Memory and Cache

Memory and cache performance is critical in high speed network application since a single last level cache miss will normally bring more than 100 ns of delay in packet processing. In this experiment, we still use SYNC-Separate-SHT and RSS-Separate-SHT cases to evaluate the cache performance impact on the system. We use 1K, 256K, and 8M as the size of the simple hash table, where each bucket size is 17 bytes (5-tuple and a 4-byte counter). Since the experiment CPU contains 256 KB L2 Cache and 20 MB L3 cache, if using the "Separate" data structure with 6 cores, then 1K SHT can fit into both the L2 and L3 cache; 256K SHT can only fit into L3 cache; and 8M SHT is out of the capacity of L3 cache. Therefore, we would expect to see a huge performance degradation with the 8M SHT in the evaluation. However, as we can observe from Figure 11, an over-sized separate data structure maintained by CPU core cache can hardly affect the performance of the PDR. The explanation is as follows.

Firstly, since the mempool size is 16K and each mempool slot consumes 2 KB of aligned memory space to accommodate the largest possible packet (1514 bytes), the total amount of the memory of a single mempool is 32M. This mempool therefore becomes the largest memory space comparing with the SHT data structure. Thus, cache accesses are almost dominated and overwhelmed by the mempool instead of the simple hash table. Secondly, the actual processing time only contributes around 40% of the overall packet processing delay according to our experiments. This means even we manage to lower the cache miss rate and the packet processing time, the overall performance will not be greatly improved because of the percentage. These two reasons together explain why the size of the hash table only lays limited effect on the PDR.

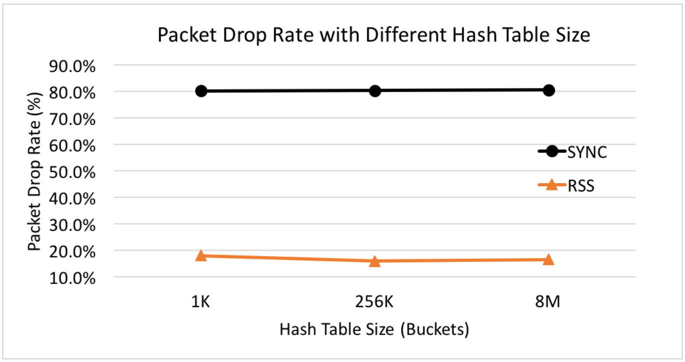

Figure 11. Packet Drop Rate with Different Hash Table Size

4.5. Shared/Separate Design Evaluation

As discussed in Section 3, every measurement sketch, i.e. SHT, CMS, and RS, can be shared across CPU cores or each CPU core can maintain a private local copy of the sketch. The difference between these shared and separate designs is two-fold. Firstly, a shared measurement sketch is efficient in cache space utilization. For a commodity x86 machine which only contains 20 to 30 MB of last level cache, a separate data structure may easily exceed the cache limit, and therefore degrades the cache performance. However, as demonstrated in the last subsection 4.4.2, cache efficiency may not contribute too much to the overall system performance. Hence, we wan to evaluate

Table 3. RSS: Performance of Shared and Separate

	PDR	PPT
RSS-Separate-SHT	1.66E-001	3.80E-008
RSS-Shared-SHT	2.02E-001	4.95E-008
RSS-Separate-CMS	3.86E-001	8.29E-008
RSS-Shared-CMS	9.44E-001	1.48E-006
RSS-Separate-RS	9.61E-001	2.14E-006
RSS-Shared-RS	9.91E-001	9.15E-006

Table 4. NONE: Performance of Shared and Separate

	PDR	PPT
SYNC-Separate-SHT	8.06E-001	7.90E-008
SYNC-Shared-SHT	8.04E-001	1.04E-007
ASYNC-Separate-SHT	8.03E-001	6.66E-008
ASYNC-Shared-SHT	8.02E-001	9.72E-008
SYNC-Separate-RS	9.93E-001	7.83E-006
SYNC-Shared-RS	9.92E-001	7.70E-006
ASYNC-Separate-RS	9.93E-001	7.19E-006
ASYNC-Shared-RS	9.92E-001	8.54E-006

to what extent can a separate data structure affect the performance results. Secondly, to guarantee mutual exclusive operations on the shared sketch, the design requires atomic control on the shared space accesses. The atomic operations may become very expensive especially in the 100 Gbps network environment.

We study the trade-off of applying shared or separate design within two different categories as follows.

4.5.1. If Using RSS Feature

If we leverage the RSS feature with the NIC, we want to compare the performance of shared versus separate design for all 3 sketches - SHT, CMS, and RS. In this case, we have 6 test scenarios annotated as RSS-Separate-SHT, RSS-Shared-SHT, RSS-Separate-CMS, RSS-Shared-CMS, RSS-Separate-RS, and RSS-Shared-RS. As we can observe from the Table 3, PDR is always lower if applying the separate design, and the PDR difference can reach up to 60% for the CMS case. Packet processing time is also much lower for the separate design. For the CMS measurement sketch, the separate design gains 17.8x packet processing speedup.

4.5.2. If Not Using RSS

If not using the RSS feature, we want to explore the other two parallel design - SYNC and ASYNC with shared or separate data structure for SHT and RS. We therefore have 8 test scenarios which are annotated as SYNC-Separate-SHT, SYNC-Shared-SHT, ASYNC-Separate-SHT, ASYNC-Shared-SHT, SYNC-Separate-RS, SYNC-Shared-RS, ASYNC-Separate-RS, and ASYNC-Shared-RS. As the data shown in the table 4 suggests, the only difference being less than 1%, shared/separate designs do not impact too much on the PDR. On the contrary, PPT is almost always better if with the separate design, with the only exception for synchronous RS.

4.6. Which Design to Choose?

With all possible different design options we discus thus far, it is beneficial to summarize a protocol to follow as a reference for the future implementation of a network measurement system. We therefore present a summary of performance data in terms of PDR, PPT, and total packet delay in Figure 12, 13 and 14, respectively.

4.6.1. Which Sketch to Choose?

If not considering the measurement accuracy boundary guarantee, SHT shows the minimum packet drop rate, packet processing time and total packet delay across all three performance figures. This is due to the simplicity of the data structure and data access pattern. However, if the system has to guarantee the measurement accuracy, then CMS and RS are the only two options as revealed in Table 1. The trade-off has to be made between a) measurement accuracy + space efficiency, and b) packet drop rate + packet delay.

4.6.2. Which Parallel Design to Choose?

For all three parallel designs - SYNC, ASYNC, and RSS, we can conclude from Figure 12, 13 and 14 that the RSS design is always the first option, especially for simple measurement primitives such as SHT. In addition, we can observe that RSS design works much better with the separate sketch data structure maintained by each core. The "RSS-Separate" combination ensures the optimal throughput at both the packet receiving stage and the packet processing stage.

However, if unfortunately a NIC does not provide RSS functionality, e.g. the COMBO-100G [25] NIC, then we should opt for the ASYNC design. This is because both SYNC and ASYNC design render similar PDR, but ASYNC design can increase packet processing speed by up to 8%.

4.6.3. Shared or Separate Data Structure?

As discussed in subsection 4.5, for both RSS-capable and RSS-incapable NIC, we should almost always opt for separate data structure for the optimal PDR and PPT.

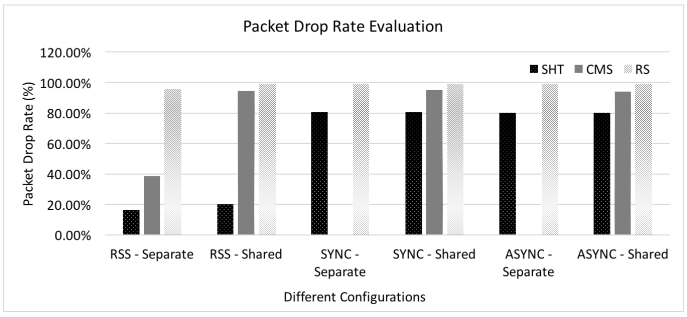

Figure 12. Packet Drop Rate Evaluation

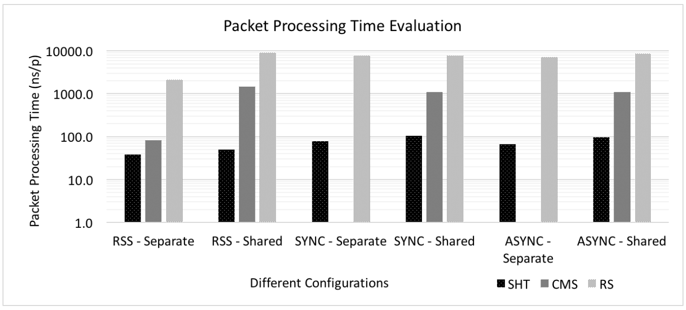

Figure 13. Packet Processing Time Evaluation

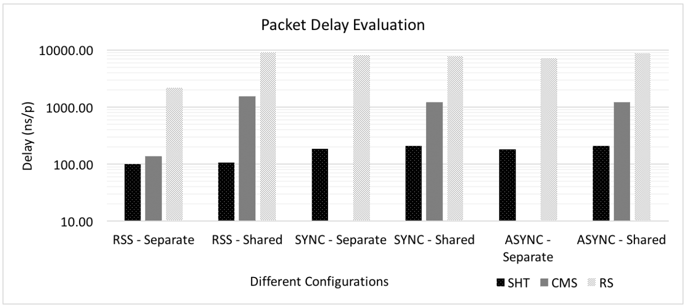

Figure 14. Total Packet Delay Evaluation

5. Conclusion and Future Work

In this paper, we focus on the design of a highly programmable network measurement platform for 100Gbps links using multicore processors. We compare DPDK RX modes and propose parallel designs using separate or shared copies of sketch data structures. With our system prototypes we compare and analyze the performances of these design options in terms of packet drop rate, packet processing time and total packet delay. Finally, we provide insights on the best practical implementation under each different DPDK RX mode. In the near future, we plan to investigate the measurement accuracy of the different sketches in such a multicore environment.

Acknowledgment

We acknowledge the technical assistance from Larry Wang, Marc Pepin and John Morgan of Intel Corporation. This work is supported in part by the National Science Foundation (No. 1547428, No. 1541434, No. 1440737 and No. 1450996) and a grant from Intel Corporation.

References

[1] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, A. Vahdat, Hedera: Dynamic flow scheduling for data center networks., in: NSDI, Vol. 10, 2010, pp. 19–19.
[2] Y. Chen, R. Griffith, J. Liu, R. H. Katz, A. D. Joseph, Understanding tcp incast throughput collapse in datacenter networks, in: Proceedings of the 1st ACM workshop on Research on enterprise networking, ACM, 2009, pp. 73–82.
[3] M. Balman, E. Pouyoul, Y. Yao, E. Bethel, B. Loring, M. Prabhat, J. Shalf, A. Sim, B. L. Tierney, Experiences with 100gbps network applications, in: Proceedings of the fifth international workshop on Data-Intensive Distributed Computing Date, ACM, 2012, pp. 33–42.
[4] Y. Liu, L. Zhang, Y. Guan, Sketch-based streaming pca algorithm for network-wide traffic anomaly detection, in: Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on, IEEE, 2010, pp. 807–816.
[5] A. Kumar, M. Sung, J. J. Xu, J. Wang, Data streaming algorithms for efficient and accurate estimation of flow size distribution, in: ACM SIGMETRICS Performance Evaluation Review, Vol. 32, ACM, 2004, pp. 177–188.
[6] G. Cormode, S. Muthukrishnan, An improved data stream summary: the count-min sketch and its applications, Journal of Algorithms 55 (1) (2005) 58–75.
[7] R. Schweller, A. Gupta, E. Parsons, Y. Chen, Reversible sketches for efficient and accurate change detection over network data streams, in: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement, ACM, 2004, pp. 207–212.
[8] M. T. Goodrich, M. Mitzenmacher, Invertible bloom lookup tables, in: Communication, Control, and Computing (Allerton), 2011 49th Annual Allerton Conference on, IEEE, 2011, pp. 792–799.
[9] C. Estan, G. Varghese, M. Fisk, Bitmap algorithms for counting active flows on high speed links, in: Proceedings of the 3rd ACM SIGCOMM conference on Internet measurement, ACM, 2003, pp. 153–166.
[10] P. Flajolet, É. Fusy, O. Gandouet, F. Meunier, Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm, DMTCS Proceedings (1).
[11] M. Yu, L. Jose, R. Miao, Software defined traffic measurement with opensketch, in: Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), 2013, pp. 29–42.
[12] M. Moshref, M. Yu, R. Govindan, A. Vahdat, Scream: Sketch resource allocation for software-defined measurement, CoNEXT, Heidelberg, Germany.
[13] M. Moshref, M. Yu, R. Govindan, A. Vahdat, Dream: dynamic resource allocation for software-defined measurement, in: ACM SIGCOMM Computer Communication Review, Vol. 44, ACM, 2014, pp. 419–430.
[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al., P4: Programming protocol-independent packet processors, ACM SIGCOMM Computer Communication Review 44 (3) (2014) 87–95.
[15] J. Ros-Giralt, A. Commike, D. Honey, R. Lethin, High-performance many-core networking: design and implementation, in: Proceedings of the Second Workshop on Innovating the Network for Data-Intensive Science, ACM, 2015, p. 1.
[16] D. Intel, Data plane development kit, URL http://dpdk. org.
[17] O. Alipourfard, M. Moshref, M. Yu, Re-evaluating measurement algorithms in software, in: Proceedings of the 14th ACM Workshop on Hot Topics in Networks, ACM, 2015, p. 20.
[18] D. Zhou, B. Fan, H. Lim, M. Kaminsky, D. G. Andersen, Scalable, high performance ethernet forwarding with cuckooswitch, in: Proceedings of the ninth ACM conference on Emerging networking experiments and technologies, ACM, 2013, pp. 97–108.
[19] OVS, Ovs-dpdk, URL https://software.intel.com/en-us/articles/using-open-vswitch-with-dpdk-for-inter-vm-nfv-applications.
[20] M. Trevisan, M. Mellia, M. Munafò, D. Rossi, Dpdk-stat: 40gbps statistical traffic analysis with off-the-shelf hardware, Tech. rep. (2016).
[21] W. R. Intel, High-performance multi-core networking software design options, White Paper 2011.
[22] Y. Liu, K. Zhang, M. Spear, Dynamic-sized nonblocking hash tables, in: Proceedings of the 2014 ACM symposium on Principles of distributed computing, ACM, 2014, pp. 242–251.
[23] J. Triplett, P. E. McKenney, J. Walpole, Resizable, scalable, concurrent hash tables via relativistic programming., in: USENIX Annual Technical Conference, 2011, p. 11.
[24] W. Keith, pktgen-dpdk, URL http://dpdk.org/browse/apps/pktgen-dpdk.
[25] Liberouter, Combo-100g, URL https://www.liberouter.org/combo-100g.