RSS++: load and state-aware receive side scaling

Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire Jr., and Dejan Kostić
KTH Royal Institute of Technology
Stockholm, Sweden
{barbette,katsikas,maguire,dmk}@kth.se

ABSTRACT

While the current literature typically focuses on load-balancing among multiple servers, in this paper, we demonstrate the importance of load-balancing **within** a single machine (potentially with hundreds of CPU cores). In this context, we propose a new load-balancing technique (RSS++) that dynamically modifies the receive side scaling (RSS) indirection table to spread the load across the CPU cores in a more optimal way. RSS++ incurs up to 14x lower 95th percentile tail latency and orders of magnitude fewer packet drops compared to RSS under high CPU utilization. RSS++ allows higher CPU utilization and dynamic scaling of the number of allocated CPU cores to accommodate the input load while avoiding the typical 25% over-provisioning.

RSS++ has been implemented for both (*i*) DPDK and (*ii*) the Linux kernel. Additionally, we implement a new state migration technique which facilitates sharding and reduces contention between CPU cores accessing per-flow data. RSS++ keeps the flow-state by groups that can be migrated at once, leading to a 20% higher efficiency than a state of the art shared flow table.

CCS CONCEPTS

• **Networks** → **Packet scheduling**; **Network resources allocation**; *Middle boxes / network appliances*; *Network servers*; *Packet classification*; *Network control algorithms*; Network design and planning algorithms; Network experimentation.

KEYWORDS

Load-balancing, intra server, state-aware, NIC indirection table.

ACM Reference Format:
Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire Jr., and Dejan Kostić. 2019. RSS++: load and state-aware receive side scaling. In *CoNEXT '19: International Conference On Emerging Networking Experiments And Technologies, December 9–12, 2019, Orlando, FL, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3359989.3365412

1 INTRODUCTION

Networking applications, such as web servers, databases, or network functions often use multiple CPU cores to serve multiple requests in parallel in order to meet high demands, while achieving low latency. Recent developments in key-value stores[9, 21, 22, 33, 39], Network Functions Virtualization (NFV)[25, 26], network stacks[12, 19, 49, 60], and high-speed networking platforms[4, 15, 48] all advocate the use of **sharding**. Sharding divides resources (*e.g.,* CPU cores and memory) into multiple shards that process requests in parallel, in a totally independent way. Each shard usually maintains a segregated **per-flow state**; *i.e.,* space allocated for packets sharing the same characteristics, such as Transmission Control Protocol (TCP) control blocks, Network Address Translator (NAT) entries, firewall statistics, *etc.* This results in higher CPU cache efficiency as sharding avoids inter-core communication.

However, sharding has a major drawback for network-driven workloads, as the amount of work per-CPU core (hereafter simply core) is dictated by the number of packets received by each core. This imbalance is depicted in the first 10 seconds of Figure 1. This figure shows the per core utilization of an 18-core server. This server runs iPerf2 receiving 100 TCP flows over a 100 Gbps link when using sharding. If the incoming load is not balanced across the cores, some cores will end up buffering more packets than others and therefore exhibit higher latency. Additionally, if the overload of a core exceeds its buffer capacity, packets will be dropped. Moreover, these high tail latencies and packet drops occur despite the fact that 3 cores are nearly idle, thus simply over-provisioning does not solve the problem.

1.1 Intra-server load-balancing challenges

Appropriately dispatching packets to shards is non-trivial as one needs to simultaneously ensure: (*i*) a **load-balance** which guarantees that no shard will starve and no shard will be overloaded; (*ii*) **flow to core affinity** is maintained, thus packets in the same flow are served by the same shard; and (*iii*) a **minimal amount of per-flow state transfers** between cores each time the sharding is rearranged.

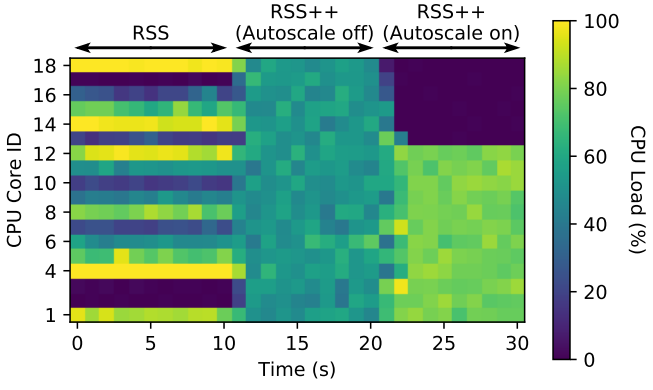


Figure 1: CPU load per core of an iPerf2 server receiving 100 TCP flows over a 100 Gbps link.

CoNEXT '19, December 9–12, 2019, Orlando, FL, USA

ACM ISBN 978-1-4503-6998-5/19/12…$15.00
https://doi.org/10.1145/3359989.3365412

In § 2, we review state of the art and textbook techniques to distribute packets to each of the shards. Receive-Side Scaling (RSS)[18] ensures flow to core affinity by hashing fields of the packets to select a "bucket" in an indirection table, as shown in Figure 2. Metron [25] dispatches packets to cores according to traffic classes, while Affinity-Accept[49] re-balances new connections. Later, § 5.4 will show those techniques are insufficient to ensure fairness between the shards. Sprayer [55] and RPCValet [8] use near-random and purely load-aware dispatching respectively to ensure greater fairness. Unfortunately, as shown in § 5.5 those methods introduce high packet reordering and do not cope well with any flow-based processing (*e.g.,* on top of TCP) or even read-mostly network processing, such as a NAT.

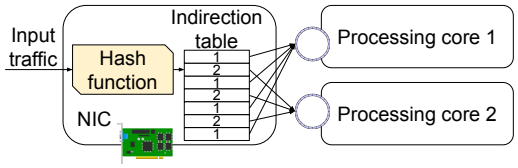


Figure 2: RSS load-balancing scheme.

1.2 Research contributions

Our technique, called RSS++, solves the packet dispatching problem by migrating the RSS indirection **buckets** between shards based upon the output of an optimization algorithm. RSS++ tracks the number of packets received by each RSS bucket. Then, knowing which buckets send packets to each core, RSS++ computes how much each bucket contributes to each core's load. The RSS indirection table is then modified to move excessive load from overloaded to underloaded cores. When needed, RSS++ can dynamically scale the number of cores, by moving buckets to a new core or reallocating buckets of a core scheduled for removal. By keeping per-bucket flow tables, when a bucket is re-assigned to another core, all of the corresponding flows can be migrated at once. To prevent packet reordering during migration, RSS++ tracks when a core has emptied its queue and then releases its associated per-flow state.

To the best of our knowledge, RSS++ is the first to achieve stateful near-perfect intra-server load-balancing, even at the speed of 100 Gbps links. It does so by enforcing flow to core affinity, minimizing & optimizing per-flow state transfers between cores, and exploiting the well established hardware-based load-balancing scheme of RSS present in commodity Network Interface Cards (NICs). Thanks to RSS, the complexity of our scheme is only proportional to the number of cores, and not to the number of flows. We only had to devise schemes for bypassing the update rate limits in some of the existing hardware. § 3 explains RSS++ in more detail.

We implement RSS++ using DPDK [34], a framework that enables fast I/O directly in user-level, therefore it is particularly well-suited to sharding as it does not need privileged system calls. We also implement the RSS++ load-balancing technique as a user-level daemon, working with today's Linux kernels. § 4 explains these two implementations in more detail and proposes a few small changes to the Linux kernel to allow flow migration.

In Figures 1 and 3, at 11 s we launch the RSS++ user-level daemon (with automatic scaling disabled) on a Linux system*. RSS++ has several positive effects on the iPerf2 server: (*i*) re-balances the load evenly among the available cores as shown by the heatmap in Figure 1 and (*ii*) reduces the average Round Trip Time (RTT) by ~30%, while reducing the standard deviation by a factor of 5, thereby tightly bounding tail latency (by 4.5x) as shown by the candlesticks in Figure 3. At 21 s we activate RSS++ automatic scaling which releases some cores by compacting the load, while incurring the same predictable latency. As a result, 6 cores could be allocated to other applications. Note that the bandwidth remains constant at 93 Gbps during the whole test.

Why RSS++?: RSS can provide good load-balancing when **both** of the following requirements are met: (*i*) input flows exhibit high entropy in the values of the header fields being hashed by the NIC (*e.g.,* 4-tuple), thus flows are well distributed among the available cores and (*ii*) input flows are of equivalent sizes and duration, thus posing similar processing requirements. Although such workloads might exist, typical Internet workloads follow the "mice and elephants" pattern[13, 32] (this pattern is also present in our campus trace). Even in a perfectly balanced situation, adding or removing cores **without** RSS++ will result in flows being continuously (at every balancing decision) migrated without their state, hence dramatically deteriorating TCP performance or losing state in a sharding context. §5 shows how quickly RSS++ balances various workloads (stemming from real traces) with large numbers of concurrent flows (80K) of various sizes.

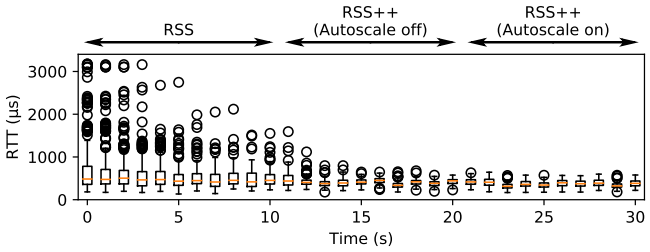


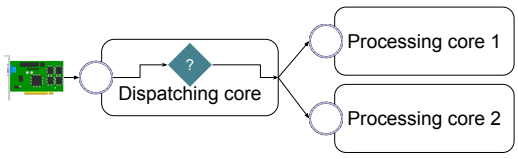
Figure 3: iPerf2 RTT in µs over time for 3 runs of the experiment shown in Figure 1. RSS++ efficiently uses CPUs, while bounding tail latency (black circles corresponding to RTT outliers).

2 INTRA-SERVER LOAD-BALANCING TECHNIQUES

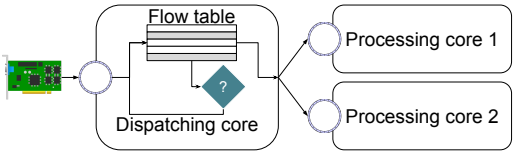
Multi-core computers achieve high performance by concurrently processing network traffic. However, traffic processing must be suitably shared among cores. In this section, we review intra-server load-balancing techniques, then compare them using multiple benchmarks in § 5. Other techniques will be discussed in § 6.

Perhaps the simplest scheme is to have one or more cores read packets from a device and put these packets into a software queue. We refer to this as **software queuing** in Table 1 and Figure 4a. Subsequently, multiple processing cores take packets

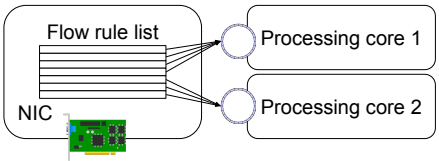
*The iPerf2 modifications to enable RSS++ and sharding are described in §4 while the testbed is further described in §5.



(a) Software queuing (*e.g.,* round-robin dispatching)



(b) Stateful queuing (*e.g.,* Flurries [61]) and E2 [47])



(c) Traffic-class based (*e.g.,* Metron [25])

Figure 4: Load-balancing techniques.

from this software queue. These inter-core transfers increase latency. Moreover, assigning core(s) to packet reception reduces the number of cores available for actual processing.

One way to realize flow to core affinity with software queuing is to have the dispatching cores open a software queue to each processing core. To select a processing core, one can utilize multiple schemes, such as round-robin or load-aware dispatching. The choice of processing core is stored in a flow table, thus subsequent packets of the flow are sent to the same core; however, building & maintaining this table has a huge performance cost. We refer to this technique as **stateful queuing** in Table 1 and Figure 4b. Affinity-Accept [6] modifies the Linux kernel to keep a per-core flow table and tries to use the same core for dispatching and processing. This improves performance, but load-balancing only occurs upon the arrival of new connections.

RSS is implemented by most recent high-speed NICs. Upon reception, a set of fields[†] defined per-protocol of every packet is hashed by the NIC. This hash is used to select a hardware queue. Each core opens one hardware queue to receive packets from the NIC. Directly using a modulo of the number of queues on the hash to select the queue would require that most of the flows be redirected to a new queue when the number of queues changes. This is unstable. Instead, RSS uses an indirection table (see Figure 2) to achieve a scheme similar to consistent hashing[23]. The hash selects an entry in the indirection table and this entry contains a queue index. As shown by [55], RSS may hash too many elephant flows to the same cores, while other cores are starved of work.

Sprayer [55] abuses the RSS functionality by computing the hash on packets' checksum instead of the 5-tuple. This allows it to mimic the round-robin software queuing method, but with the dispatching done entirely in hardware. Sprayer achieves good load-balancing, but breaks flow to core affinity; therefore, it is unsuitable for stream processing functions as will be shown in §5.5.

Metron [25] translates service chains' traffic classes (synthesized by SNF [26]) to NIC or OpenFlow classification rules (see Figure 4c). We refer to this method as **traffic-class based** in Table 1 and Figure 4b. Because packets can be dispatched directly to the queue of the core that will process them, inter-core transfers are avoided. In Metron, when a core is overloaded, half of the traffic classes currently sending packets to the queue assigned to the overloaded core are updated to dispatch packets to a new queue, and thus a new core. This scheme is stable; however, Metron cannot load-balance traffic classes which cannot be split. *e.g.,* if one runs only an HTTP server listening for TCP packets on port 80, the only traffic class will be "TCP dst port 80"; therefore, Metron can only use one core. Metron offloads some network functions and handles multiple servers using a centralized controller, but these features are outside the scope of this paper.

Table 1 summarizes the observations of existing load-balancing techniques and our technique (described in § 3). Stability is evaluated as P_{miss}, the probability for a core to receive a packet of a flow that was previously received by another core (excluding flows intentionally scheduled for migration). C is the number of cores.

Table 1: Overview of packet dispatching and load-balancing methods. RSS++ is the only method that meets both performance and load-balancing criteria.

Method	Performance	P_{miss}	Work division
Software queueing	~	✗ $1-\frac{1}{C}$	✓
Stateful queueing	✗	✓ 0	✓
RSS	✓	✓ 0	~
Sprayer	✓	✗ $1-\frac{1}{C}$	✓
Traffic-class based	~	✓ 0	~
RSS++	✓	✓ 0	✓

3 RSS++ DESIGN

The goal of RSS++ is to maximize flow affinity *and* fairness by moving *some* flows between cores, as described in § 3.1. In contrast, using more or fewer cores to spread or concentrate the load will be called CPU scaling (see § 3.2).

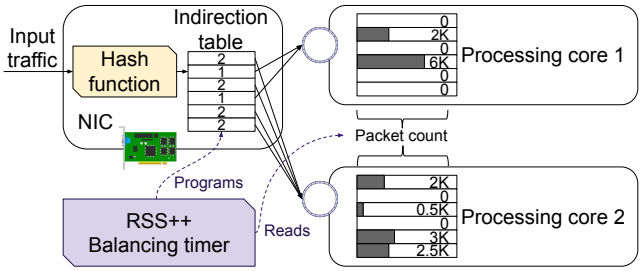


Figure 5: RSS++ overview.

3.1 Load-balancing among active cores

Figure 5 shows the overview of RSS++. *RSS++ does not require any modifications at the NIC*; instead, it leverages the fact that the

[†]Typically, source/destination IP address and source/destination ports for TCP and User Datagram Protocol (UDP).

RSS hash, that gets computed by commodity NICs, is stored in the packets' metadata. Each core tracks the number of packets received by each bucket. To do so, each core maintains a table of the same size as the number of RSS buckets and uses the low order bits of the hash as an index into the table and increments a simple counter. At a variable frequency, ranging from 1-10 Hz, RSS++ gathers the bucket counters and re-assigns some RSS buckets to cores that are not close to the average load. This is done by *re-programming the RSS indirection table using the standard API provided by the NIC.*

Figure 6 visualizes the RSS++ algorithm for stateful hardware-driven load-balancing. RSS++ gathers the cores' load, computes the average load, and builds two sets: overloaded (*i.e.,* with above average load) and underloaded (*i.e.,* with below average load) cores. The core load is a number ranging from 0% when the core is idle, to 100% when the core is completely busy. The exact metric behind the core load is explained in § 4. RSS++ assumes each bucket is responsible for a certain fraction of the load of the core serving the bucket, *i.e.,* the fraction is the number of packets received by the bucket divided by the total number of packets received by that core. In the following, the bucket's fractional load refers to its fraction of the core's load, rather than the number of packets. We then solve an optimization problem which assigns buckets of all overloaded cores to either stay in place or to migrate to an underloaded core, according to their fractional load.

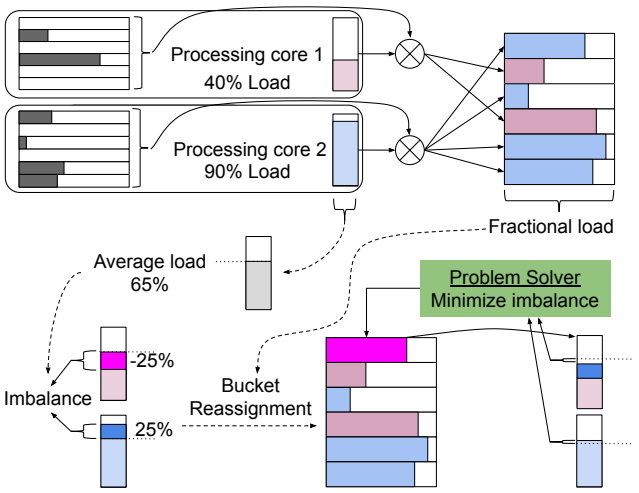


Figure 6: RSS++ algorithm to reassign load between cores.

Problem dimension: It is worth noting the dimension of the problem. Intel 82599 10 GbE NICs have a 128 bucket indirection table, while XL710 40 GbE NICs have a 512 bucket indirection table. Mellanox ConnectX-4 and ConnectX-5 100 GbE NICs can use more than 512 buckets, but system's limitations restrict the table to 512 buckets in DPDK and 128 buckets in the Linux kernel driver. The number of overloaded and underloaded cores is system dependent, but the algorithm should perform well, even for a hundred cores. Techniques for solving multi-way number partitioning, a similar optimization problem (without the transfer constraint and without considering the existing load of underloaded cores), would take tens of seconds[56]. However, an optimal solution is unnecessary. Instead, a quick sub-optimal solution is sufficient as a new re-assignment will be done soon. Moreover, the input of the problem is imprecise, as the load of RSS buckets varies over time - as does the amount of work per core; hence, even an optimal solution may not lead to an assignment that is optimal during the next interval.

Therefore, we adopt an iterative algorithm that stops as soon as (i) we find a solution leading to an 1% or less overall squared load imbalance or (ii) after 10 iterations while minimizing the number of re-assignments.

Formal optimization problem: We formalize the problem of balancing the load among a server's CPU cores as follows: C is the set of all cores and B is the set of buckets. M_i is the current load of underloaded cores and zero for overloaded cores. L_i is the fractional load of each bucket B_i and M is the average load that each core should reach. $T_{i,j}$ is the assignment matrix of core i to bucket j and $T^{old}_{i,j}$ a copy of the current assignment matrix, thus one can formulate an optimization problem that migrates the imbalance to achieve the best spreading of the load as follows:

$$\min_{T} \quad \sum_{i \in C} ((T * L^{\intercal})_i - M_i)^2 + \alpha * \frac{\sum_{i \in C, j \in B} T_{i,j} \neq T^{old}_{i,j}}{2}$$

$$\text{s.t.} \quad \sum_{j \in B} T_{i,j} = 1 \qquad \forall i \in C, \tag{1}$$

$$T_{i,j} \in \{0,1\} \qquad \forall i \in C, j \in B \tag{2}$$

The first constraint ensures that a bucket is assigned to one and only one core, while the second makes the problem binary, as we cannot migrate parts of a bucket to different cores. The minimization is done using a multi-objective function.

The first term of the objective function minimizes the resulting squared load imbalance by applying the assignment described by T. Without the second term of the objective function, the problem is similar to multi-way number partitioning[30]. Multi-way number partitioning finds the best way to split a set of numbers (such as the load of buckets) into subsets, such that all subsets exhibit the same sum of elements. In our case, the subsets represent cores. The difference from the classical problem is that some subsets are assigned to underloaded cores that already have some load. The second term minimizes the number of bucket transfers. Each migration has a cost, that should be reduced as much as possible (see § 5.6). The scalarization of the two objectives is done with the α factor.

Heuristic algorithm description: We start from the current assignment and move the heaviest buckets first. In the first iteration, we run a greedy algorithm that tries to solve a variant of the standard multi-way partitioning. We sort the overloaded cores (*i.e.,* those with above average load) in descending amount of load and then for each core we sort its buckets in descending amount of fractional load. Then, we sort the underloaded cores by ascending amount of load, *i.e.,* the most underloaded core first. We repeatedly select the most overloaded bucket of the most overloaded core and try to fit it in the most underloaded core (unless the underloaded core becomes overloaded). After an assignment, we change the

load of the overloaded and underloaded cores to account for this transfer. We stop taking buckets from overloaded cores when their load is within 1% above the average. The classical multi-way number partitioning algorithm considers all buckets of overloaded cores at once, while we keep buckets to move assigned as children of their current cores to balance the overloaded cores at the same time we solve the balance of the underloaded cores. Indeed, blindly taking buckets from the overloaded set could lead to emptying an overloaded CPU.

The first iteration leads to a situation where insufficient load is moved from the overloaded cores (as the desired balance is the average of all cores' load); therefore, the underloaded cores do not receive enough load. This is used as a lower bound. In the second iteration RSS++ tries to find an upper bound, by moving more than enough buckets, hence the overloaded cores become underloaded, while the underloaded cores become overloaded. This is done by allowing transfer of more buckets, by as much of a bucket's fractional load as the imbalance resulting from the first iteration. The goal of all subsequent iterations is to find the inflection point as shown in Figure 7, thus avoid moving too few or too many buckets. While there is no guarantee that this is the optimal solution of the formal problem, intuitively moving a few additional less-loaded buckets may lead to a better solution.

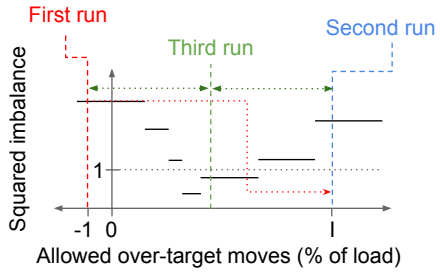


Figure 7: Three iterations of the RSS++ heuristic algorithm to solve the bucket assignment problem.

Evidence about the convergence of the RSS++ algorithm: During the experiment described in § 5 using 16 cores to process packets of a ~15 Gbps network trace, in ~85% of cases the algorithm stops at the first run, with an average completion time of ~25 µs and a total squared load imbalance of less than 0.5%. Including the query of CPU loads and preparation of the input to the algorithm, solving rarely takes more than 500 µs.

3.2 CPU scaling

When a core is overloaded and the *average* load is higher than a target load (currently, a default of 80% based upon the 95th percentile tail latency - see § 5.2), re-balancing among active cores as described in § 3.1 may be insufficient. When a new core is allocated to the application, this new core has no load and no assigned buckets, but the average load is lowered. Most cores will now exhibit an overload imbalance, hence the balancing algorithm migrates just enough load to the new core to re-balance the load.

In contrast, removing a core is a little trickier. This is because the balancing algorithm described in § 3.1 cannot ensure that a core would be emptied of its buckets. To decide when a core should be removed, we sum the difference between the target load and each core's current load. We remove one core if the sum is bigger than $1 + \alpha * N$ and $N > 2$, where N is the current number of cores and α a factor to prevent aggressive scaling decisions due to variance in the load of a core (by default, 5%). To remove a core, we solve the traditional multi-way number partitioning problem to assign all buckets of this core to existing cores in such a way that the load will be even. We sort the deactivated cores' buckets and migrate the most loaded buckets first, while maintaining a priority queue of the active processing cores, with the least loaded core first. As the smallest buckets are left to last, the imbalance is likely to be small. If not, the load-balancing algorithm will soon run and re-balance the remaining errors in initial re-assignments.

Most virtualization technologies, such as KVM [31], allow dynamic reduction/increase in the number of virtual CPUs (vCPUs). Given that cloud operators bill per-vCPU, scaling applications at a fine granularity of time reduces costs or at least allows better collocation. For instance, performing balancing at high speed allows Software Defined "Hardware" Infrastructure (SDHI) [53] to quickly free available cores. Once the utilization of a set of machines has decreased beyond a certain point, one may even begin deactivating some of these machines.

3.3 State migration

When a core is added to the set of processing cores, some packets will start to be received by the new core. Similarly, when a core is deactivated its packets will be sent to other active cores. Furthermore, when the load-balancing is unstable, some flows will be migrated from active cores to other active cores. RSS++ is stable as it relies on RSS's hashing, however RSS++ willingly migrates some flows to re-balance the load. In all cases, the flow's existing state will need to be accessed.

The easiest way to avoid migration is to have shared state management. *E.g.,* in the Linux kernel, a unique hash-table is used by all cores to access the common state of all flows. As the number of flows increases, so does the amount of contention to access this common state (see § 5.5).

Moreover, re-assigning RSS buckets can cause packet reordering, much as in FlowDirector [17] (a system to direct flows to NIC queues). This behavior is confirmed by Wu et al. [59]. For example, if packets of flow F are sitting in a queue, when F is redirected to a different queue that is less filled, the new packets of F could be handled before the enqueued packets are handled, leading to reordering. Therefore, we propose a new state management data structure that takes advantage of the fact that packets come with a *parent group* that is the source of balancing among cores. For RSS++, the grouping is the RSS buckets, but this technique is also usable for FlowDirector or traffic-class based migration [25].

The key idea is to keep one flow table per RSS bucket (hereafter referred to as a *per-bucket table*) as shown in Figure 8. When an RSS bucket is marked for migration, *e.g.,* when moving a bucket from core A to core B, the table is released only when A is sure to have handled all packets of the RSS bucket. Core B then takes control of the table. If packets are received by core B, while core A still holds the table, those packets are enqueued in a per-bucket

queue to resume their processing later. Since only core *B* will handle this queue, no lock is necessary. As the per-bucket table is released only when all packets of this RSS bucket have been handled, no re-ordering can occur. To detect when core *A* has processed all previously enqueued packets, we query the number of packets in the queue when the RSS table is rewritten. After *A* has processed that many packets, we know the new table has been applied. Alternatively, when core *A* receives packets from a bucket just assigned to *A*, it knows it is starting to receive packets of the new table and therefore can release its own tables to be migrated right away. At normal load levels, the number of packets sitting in a queue is low - as cores have enough cycles to process their incoming packets. Similarly, as RSS++ seeks to ensure an even balance between cores, *B* should have as much work in its queue as *A*, hence it will start processing new packets at the same time *A* releases those packet's flow table. Therefore, rarely will packets sit in the per-bucket queue, and if any do, the extra latency is expected to be very low.

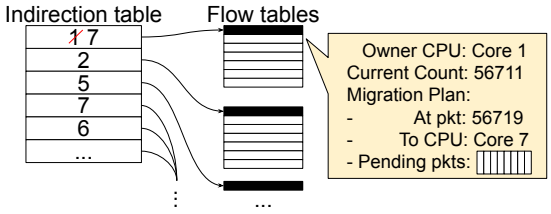


Figure 8: Non-reordering, non-blocking RSS++ migration scheme.

4 IMPLEMENTATION

To achieve interoperability with optimized and legacy network drivers, we implemented RSS++ both on top of DPDK[34] and directly in the Linux kernel.

For the DPDK prototype, we used FastClick[4] an enhanced version of the Click Modular Router[29]. The per-bucket packet counting is done directly as a function call, accessing the packet's metadata (rte_mbuf in DPDK) to read RSS's hash and incrementing the relevant entry in a counting table that keeps track of the number of packets per bucket for each core. To avoid synchronization problems, the counters are reset by using an epoch counter. After reading the counting tables, RSS++ increments the epoch. When updating the counting table, each core first verifies a local copy of the epoch number, and then resets the counter if it differs. Therefore, both implementations incur a slight memory overhead equal to the size of the indirection table times 8 bytes[‡] for the counter values and the epoch. For a 512 bucket indirection table, this is 2KB per core. As only $1/N$ (where N is the number of cores) of that table will be allocated to one given core, not all of that memory will need to reside in cache. If memory is an issue, a single table may be used for all cores, at the price of false sharing, or more fragmentation. The balancing timer runs as part of the user-level application, therefore it can directly access the counting tables and use DPDK's function to update the RSS indirection table. A frequency of 10 Hz was sufficient in our experiments. However, when the load is low, it is unnecessary to run the algorithm this often; therefore we use an exponential backoff to lower the frequency when the average load *and* the imbalance are lower than a threshold. As DPDK relies on polling, the traditional CPU load metric given by the Operating System (OS) is always 100%; therefore, we compute each core's load as the number of CPU cycles spent to process packets over the cycles spent to poll for packets with or without success.

Before applying the result of the optimization, the balancing timer propagates a message to the flow classification manager of the application to say "bucket X, Y and Z will move from core A to core B". The application may or may not register a function callback, though, in our implementation the message is used to implement the state migration scheme explained in § 3.3 and mark some of the per-bucket flow tables for migration.

Intel 82599 10 GbE and XL710 40 GbE NICs take roughly 20 µs to update the RSS table. However, the Mellanox ConnectX-4 and ConnectX-5 NICs have to restart the device to reprogram the global RSS table, which leads to loss of hundreds of thousands of packets at a 100 Gbps rate. Fortunately, we can install a flow rule that matches all IP traffic and add RSS as an action of that rule. As these NICs do not support updating actions, at the next even tick we install two rules that match a subclass of the first one (*e.g.*, by matching the last bit of the destination IP address) both with the new RSS action and remove the old one. At odd ticks we install a single rule and remove the two old rules. This process takes around 20 ms. Mellanox is working to decrease the flow installation time by a factor of a thousand[36].

In the Linux implementation, we insert a BPF program to count the number of packets per-bucket into the recently introduced Linux eXpress DataPath (XDP)[16]. BPF programs are verified and then compiled as native code by the kernel; therefore, they are very efficient. While BPF supports multiple data structures that can be used to share data with user-level programs, we opted for a per-CPU array map. The user-level daemon's balancing timer reads and resets the counter map. Re-programming the table is done through the ethtool API (known for the user-level client of the same name). The Mellanox ConnectX-4/5 driver does not exhibit the same problem as its DPDK counterpart, thus the table can be rewritten seamlessly.

As a first step towards sharding, Linux introduced the SO_REUSEPORT option to allow multiple sockets to listen to the same port. When a connection establishment packet is received, one of the sockets is selected, by default using hashing. This allows an application to receive packets using multiple receive queues, and therefore multiple threads[§]. One can select a per-core socket with a two instruction BPF program (shown in Figure 9). In Linux, when a packet is received for an established connection, the packet is put in a reception queue, then threads sleeping on the associated descriptors are awakened.

When RSS++ is re-balancing a bucket, the associated thread needs to be migrated to its new core. Therefore, we added a new socket option SO_AUTOMIGRATE that migrates the process to the last core that received a packet for its pending sockets before waking

[‡]At 100 Gbps, the worst case would be 150M packets per seconds, therefore a 32 bit counter is sufficient. One bit would be sufficient for the epoch, but padding leads to 8 bytes per bucket.

[§]In the DPDK implementation, the same process and the same thread receives packets and processes them to completion.

it up. To do so, when a packet is received, the reception core index is kept in the socket metadata. When a thread does an operation such as reading, we verify the current core index matches the index in the metadata. Otherwise, the current thread is rescheduled on the correct core. Figure 9 summarizes the most important changes made to iPerf2 to enable sharding. SO_AUTOMIGRATE is complementary to SO_REUSEPORT, as the latter allows opening one listening socket per core and spawning a newly established socket upon connection, while the former allows migration of established sockets between the cores as flows are load-balanced.

```
int boolean = 1;
setsockopt(sock, SOL_SOCKET, SO_REUSEPORT, (char*) &
    boolean, sizeof(boolean) );
setsockopt(sock, SOL_SOCKET, SO_AUTOMIGRATE, (char*) &
    boolean, sizeof(boolean) );
struct sock_filter code[] = {
    { BPF_LD  | BPF_W | BPF_ABS, 0, 0, (unsigned)(
    SKF_AD_OFF + SKF_AD_CPU)},
    { BPF_RET | BPF_A, 0, 0, 0 },
};
struct sock_fprog p = {
    .len = 2,
    .filter = code,
};
setsockopt(sock, SOL_SOCKET, SO_ATTACH_REUSEPORT_CBPF, &p
    , sizeof(p)))
```

Figure 9: C code to use SO_REUSEPORT, our new option SO_AUTOMIGRATE, and a BPF program to select the socket index equal to the current CPU.

To reduce the number of context switches, recent software does not use one thread per socket, but uses poll, epoll, or similar functions to listen to events from multiple sockets using a single system call. If more than one of those sockets are from different RSS buckets, RSS++ would migrate the listening task back and forth, which would be very inefficient. Therefore, we propose a second socket option, SO_SCHEDGROUPID, to get the "hardware scheduling ID" (*i.e.,* the RSS bucket index) of a given socket. Thus one can listen for all file descriptors of the same RSS bucket at once by grouping file-descriptors per RSS buckets and use one thread listening for events per group. As future work, we propose that the kernel manages and migrates file descriptors that have SO_AUTOMIGRATE enabled between epoll queues; therefore, a single thread or process per core can listen for all its assigned sockets.

The Linux TCP stack uses a single hash-table of listening sockets and a single hash-table of established sockets, hence these are potential sources of contention. We shard these hash-tables into per-core tables. Unfortunately, performance did not improve, thus more work is needed to remove the locks at numerous places in the Linux Kernel code to fully enable sharding. This is why we did not implement per-bucket flow tables in the kernel.

5 EVALUATION

In this section we evaluate performance, scalability, and quality aspects of intra-server load-balancing, comparing RSS++ with state of the art solutions using the DPDK implementation. § 5.1 discusses data used to evaluate our ideas. Using this data we answer the following questions: How well does RSS++ improve RSS? (§ 5.2) How quickly does RSS++ scale in response to input changes? (§ 5.3) How fair is RSS++? (§ 5.4) How important is flow-awareness? (§ 5.5) Can the state migration algorithm handle RSS++'s migrations? (§ 5.6) Can RSS++ handle realistic use cases? (§5.7)

5.1 Traces

Unfortunately, no datacenter traces that contain a realistic distribution of flows and enough packets to showcase load-balancing at high speed for several seconds are available. Traces from Facebook [54] or Azure [7] expose only aggregate flow statistics (*e.g.,* number of packets), while network headers are omitted; unfortunately, these headers are an essential input for load-balancing.

Therefore, we collected a campus trace called Campus with more than 20K users. Additionally, to showcase higher throughput than the ~4 Gbps of the Campus trace, we forged a novel trace (called Campus #4) by combining 4 windows of 90 seconds of Campus in parallel. The flows were rewritten to ensure no collisions. The result is a ~15 Gbps trace with four times more users.

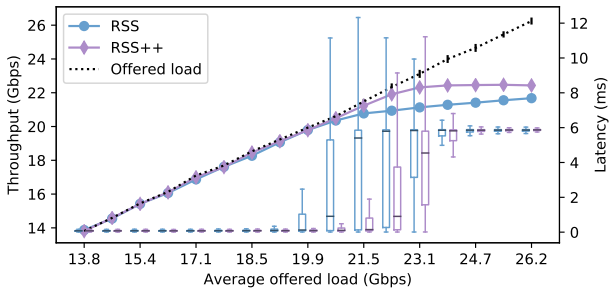
The first trace has around 20K active flows every second (flows that received at least one packet in a one second window). The Campus #4 trace has around 80K active flows every second, among which half are flows seen for the first time. These traces have a total of 661K and 2.7M flows respectively. 50% of the flows consist of a single packet, 80% of the flows are comprised of less than 10 packets, and the 95^{th} percentile flow size is 30 packets; the mean flow size is around 5900 packets for both traces. Roughly 20% of the packets are smaller than 74 bytes and 50% are MTU-sized. Appendix A gives further details of these traces.

To evaluate the performance of the load-balancing according to various metrics for each technique presented in § 2, we replay traces to a Device Under Test (DUT), that will do some processing on each packet. The DUT has an 18-cores Intel® Xeon® Gold 6140 CPU @ 2.3GHz with Hyper-Threading disabled and 256 GB of RAM. All machines use the Ubuntu 18.04 LTS OS with DPDK 19.02. To measure the latency, the generator rewrites part of the payload to include a unique packet number and keeps track of this number to timestamp when the packet is actually sent. After processing the DUT bounces packets back to the generator, so the latter can compute the overall round-trip time.

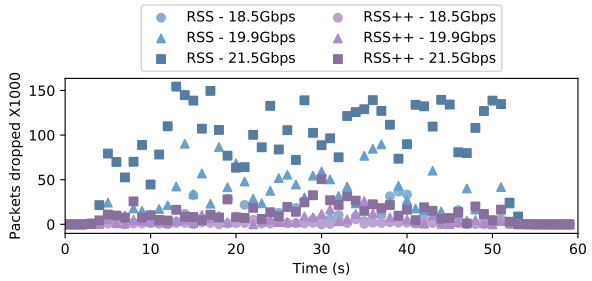
5.2 How well does RSS++ improve RSS?

We conducted a comparative performance study to show why RSS++ should be preferred over standard RSS-based systems.

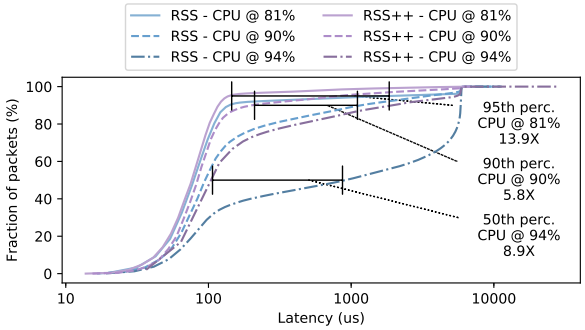
Figure 10a shows the performance of RSS++ versus RSS using the platform described in § 4. Each core is serving one queue of the NIC configured with 4096 buffers. As experiments are conducted using DPDK, there are no interrupts from the NIC. In the sharding context, processing is run-to-completion in a one-queue-for-one-core mapping. Both systems are tested with the same increasing offered load (by replaying the Campus #4 trace at different proportions of its original capture timing, while using a fixed per-packet workload). We use 4 cores as this leads to an average ~ 80% CPU load with the original trace speed, a value



(a) Throughput (lines) and latency (boxes) as a function of the average offered load.



(b) Packet drops (in thousands) for a subset of trace replay speeds.



(c) Latency distribution at three different average CPU loads.

Figure 10: Efficiency of RSS++ compared to RSS. RSS++ absorbs ~10% more load, while dropping orders of magnitude fewer packets and achieving lower latency.

sufficient to observe the increased latency, while limiting over-provisioning. As the offered load increases, the queues start to grow. The greater the imbalance, the more congested some queues will be, thus the respective packets will experience increased latency. Figure 10b shows the number of packets dropped over time. When the offered load is high, RSS++ drops fewer packets. Figure 10c shows the distribution of packet latency at offered loads leading to an average 81%, 90%, and 94% CPU load, corresponding to offered loads of 16.13 Gbps, 19.23 Gbps, and 20.61 Gbps. As the average CPU load increases, RSS starts to enqueue more packets in some queues - increasing tail latency. Figure 10c shows the tail latency ratio at one percentile for each CPU load. RSS++ exhibits up to an order of magnitude lower tail latency than RSS, allowing a higher average CPU load, while offering a comparable latency to users.

5.3 How quickly does RSS++ scale in response to input changes?

We inject the Campus #4 trace at a controlled proportion of its original capture timing to stress the resource allocation schemes of RSS, RSS++, and Metron.

As shown in Figure 11, all of the systems absorb the offered load, despite its variation; however, there is a key difference: RSS uses a constant number (*i.e.,* 15) of processing cores, while Metron and RSS++ dynamically adjust the amount of processing power according to the offered load. Changing the number of cores used by RSS in a sharding context would break connections as it relies on hashing, forcing the operator to over-provision. By comparing the purple dashed lines and green dotted lines we observe that RSS++ allocates 50% fewer cores on average compared to Metron's state of the art resource allocation scheme.

When a core is overloaded, Metron splits its traffic classes into two parts, *independently* of their load - which is "too harsh". Moreover, Metron can only merge/split pairs of underloaded/overloaded cores, while RSS++'s fast re-balancing scheme takes all overloaded and underloaded cores into account at once.

Unlike traffic-class based dispatching (*e.g.,* Metron), RSS++ does not require NICs with flow classification capabilities, instead it uses RSS's indirection table to re-balance some part of the load on the fly. RSS has been available for many years, while flow classification capabilities are often limited. *E.g.,* Intel 82599 NICs are limited to 8K flows with a single shared mask preventing any realistic traffic-class programming.

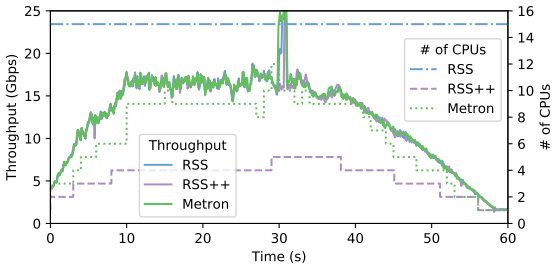


Figure 11: RSS, RSS++, and Metron under dynamic workload. RSS++ achieves the same throughput as Metron, with 2x fewer cores on average. Between 0-15 seconds the input trace (Campus #4) is replayed with an increasing replay rate, while a decreasing replay rate is applied between 31-60 seconds.

J. T. Araújo, et al. [1] suggest their services can receive hundreds of times their usual load due to traffic surges because of *e.g.,* Flashcrowds and DDoS attacks; therefore, it is necessary to quickly accommodate surges in load. Figure 11 shows RSS++ could implement NFV applications to drop malicious traffic that can quickly scale to absorb surges in load.

5.4 How fair is RSS++?

This section focuses on load-balancing performance in terms of fairness. A method that distributes an almost equal number of packets to all cores will be considered fair, while one that distributes many more packets to one or more cores will be considered skewed.

To evaluate fairness without (yet) considering flow affinity, the DUT generates W pseudo-random numbers for each packet received. We measured $W = 1$ to take around 8 CPU cycles. As there is no state, this test case allows us to use various load-balancing methods to dispatch packets to the DUT's processing cores.

Figure 12a shows the proportion of packets received by the most loaded core relative to the least loaded core, when using 8 processing cores for each technique presented in § 2. To keep the average core's CPU load around 50%, we set $W = 150$. The fairest load-balancing scheme is software round-robin. Sprayer is very close as it achieves nearly the same balancing as round robin, but in hardware. The flow-aware load-balancing methods generally perform worse than Sprayer or software round-robin. Stateful flow-aware load-balancing does not really improve performance, as balancing *new* flows according to the *current load* is insufficient to achieve a good balance. Note that all software-based methods need a dedicated core to do the load-balancing. Figure 12b shows the same test for a varying number of processing cores using the Campus #4 trace and when W is proportional to the number of cores. Single-queue work stealing ("SW Shared Queue" in Figure 12) exhibits a high imbalance when the number of cores increases as nothing guarantees ordering between cores trying to get packets from the queue.

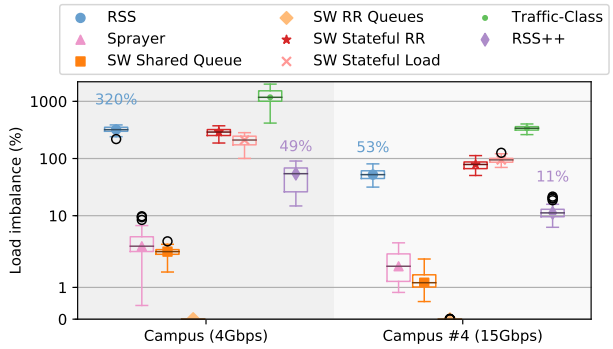
Unfairness itself is not a problem. However, a consequence of unfair load-balancing is that some core(s) will receive more packets than others and as the aggregate rate increases these packets will queue up, increasing latency and in the worst case lead to packets being dropped. Figure 13 shows the 95th percentile tail latency (the latency of the 5% longest RTTs) for the same test with 8 processing cores, under increasing offered load. While hardware-based methods meet the fairness trend, software-based methods take a performance hit due to inter-core communication and internal queuing.

Figure 14 compares the techniques' throughput with the same per-packet workload. This test is similar to Figure 12 but the CPU's clock frequency was reduced to 1 GHz to evaluate the impact of using more cores without unrealistically increasing the replay speed. Sprayer and RSS++ need at least 5 cores to handle the offered load, while RSS needed 8 cores. Due to the unfairness of RSS, some cores retain almost all their previous amount of work, while new cores may receive little traffic. In addition to requiring a dedicated core for dispatching, software-based dispatching does not scale as well as hardware-based methods. Due to cache invalidations across the CPU interconnect when exchanging buffers and inter-core buffer recycling [10], the non-sharded methods exhibit decreasing performance when more than 8-10 cores are used.

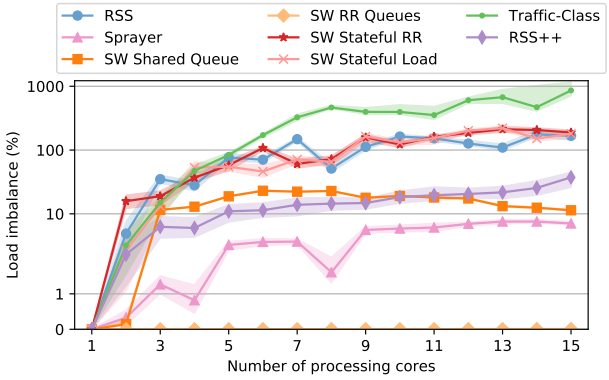
5.5 How important is flow-awareness?

While Sprayer seems to be suitable for spreading the work (as it nearly randomly dispatches packets to cores based upon their checksum), one must evaluate the impact of distributing packets to cores in a flow aware fashion. In Sprayer, only connection initiation packets have an assigned core and they are redirected to the correct core using software queues. While Sadok et al. argue that few NFV functions need state, we have a different viewpoint. While the state of simple functions, such as NAT is limited, many-core solutions will inevitably lead to cache contention, when all of the cores try to access the same part of memory to read and write state. Additionally, functions needing to reconstruct the payload and reorder packets, *e.g.*, Deep Packet Inspection (DPI), proxy caches, or Transport Layer Security (TLS) termination proxies need to handle the stream in, at least, a sequential fashion, and in general in an ordered fashion.

Figure 15 shows the performance of Sprayer (an example of hardware round-robin dispatching) and RSS methods for 4 test cases. Figure 15a shows the average number of cycles spent by the DUT to process a packet for an increasing number of concurrent UDP flows (each with 1000 UDP packets of 64 bytes). The generator repeatedly consumes a packet from a randomly chosen flow and sends it. While this data is synthetic, it allows us to precisely control the number of concurrent flows. The blue lines marked with dots are similar to the test case in § 5.4 generating $W = 100$ pseudo-random numbers per packet using 8 cores. The orange lines marked with triangles fetches per-flow state in a cuckoo-based [46] hash-table, using an implementation proposed by DPDK[34]. The green lines marked with squares do a write for each packet, while the red lines



(a) Load imbalance (%) when using 8 processing cores with two different input traces. Average values are shown for RSS and RSS++.



(b) Maximal load imbalance (%) observed with an increasing number of queues. The lines with markers show the median values, while the shaded zones around each line represent the 25th to 75th percentiles.

Figure 12: Load imbalance (the ratio of the number of packets of the most loaded queue relative to the least loaded queue) introduced by various load-balancing methods. RSS++ maintains a much better load-balance than any other flow-aware method.

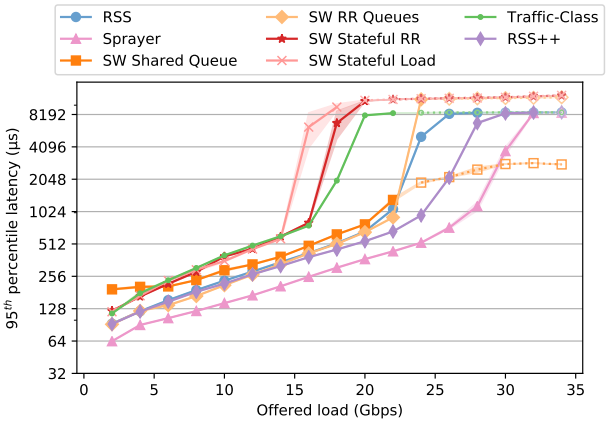


Figure 13: Tail latency (*i.e.,* 95th percentile) when using 8 processing cores. Dashed lines and empty markers indicate the corresponding throughput is below 90% of the offered load. RSS++'s performance is between RSS and Sprayer, trading a little latency for flow fairness.

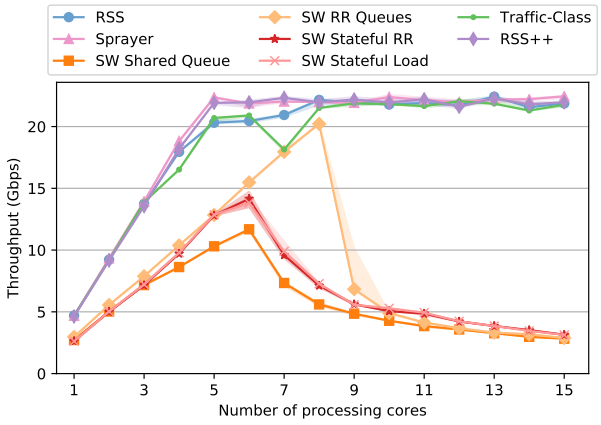
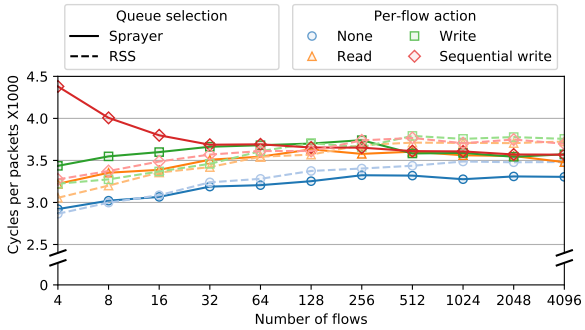


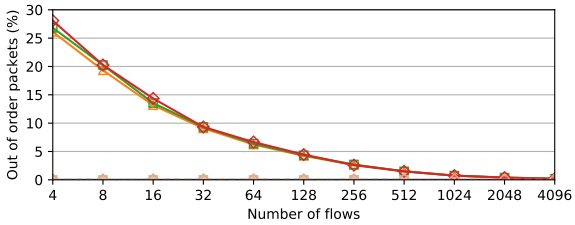
Figure 14: Forwarding performance using the Campus #4 trace while executing a fixed artificial per-packet workload. RSS++ performs similar to Sprayer, maintaining flow affinity (a 37% reduction in the number of CPU cores needed), while avoiding dropping packets as compared to RSS and Traffic-Class methods.

with diamonds do a lock-protected write, ensuring that processing is sequential. With Sprayer, for a small number of flows, packets of the same flow are dispatched to multiple cores and processed concurrently; hence, Sprayer generally performs worse than RSS, due to spending many more cycles/packet as cores are constantly invalidating cache lines used by other cores. The sequential write blocks other cores from doing any processing, while only one core processes packets of a given flow. As the number of flows increases, contention diminishes thus Sprayer performs better. With many flows, our generator sends packets of different flows back-to-back. In the Internet, packets tend to be forwarded in bursts, leading to potential state collisions even with a large number of flows.

Figure 15b shows the ratio of packets received out-of-order. To measure this, we introduce a sequence number inside the UDP payload. The DUT counts each time a sequence number is lower than the previously received one for each flow; hence, the DUT must read and write the sequence number per-flow. Therefore the out-of-order statistic is only computed for the write and sequential write test cases. As reordering does not depend on the DUT's processing, the two actions not shown in Figure 15b should follow the same trend. In the case of NFV, running such a system would be extremely selfish - as with 4 flows, reordering is ~30%; hence any stream processing, on the same machine or downstream of a DUT implementing a network function, would have to reorder nearly a third of all packets. Besides forcing clients and servers downstream to buffer packets in order to reorder them, reordering breaks Large Receive Offload (LRO) and Generic Receive Offload (GRO) (*i.e.,* hardware and software techniques to coalesce packets of a flow into a single large buffer); these could reduce the per-packet overhead of CPU processing time in TCP stacks by up to a 55%[38]. While flow unaware dispatching achieves fair load-balancing, it is *inapplicable* to most networking applications.



(a) Cycles per packet (in thousands).



(b) Ratio of packets received out of order by the DUT.

Figure 15: Impact of flow unaware load-spreading with an increasing number of UDP flows of 1000 packets when doing: (*i*) no per-flow action, (*ii*) reading a 4-byte per-flow space, (*iii*) writing to it, or (*iv*) writing, while holding a lock to force sequential action. Concurrently processing packets can double the CPU usage, but highly increases reordering.

5.6 Can the state migration algorithm handle RSS++'s migrations?

A purely sharded approach segregates flow state per-core, allowing efficient access. This section shows that the per-bucket data structure proposed in § 3.3 allows fast state access after a migration, to underline the importance of avoiding a shared flow table.

Figure 16 shows the throughput and average latency of the DUT forwarding 1024 concurrent flows of 1500-bytes UDP packets with

$W = 100$, potentially achieving 100 Gbps. We add a flow entry either using the cuckoo hash-table or the presented per-bucket technique. Every 2 seconds, the RSS indirection table is rewritten to use one more cores using consistent hashing (and not RSS++). Due to CPU contention, using more cores with the shared table does not lead to as much improvement as when using per-bucket tables. While 7 cores are sufficient to handle the load with per-bucket tables, adding more cores with the shared table approach does not increase the throughput up to the 100 Gbps line-rate of the NIC. Moreover, adding cores with a shared table increases the average latency, while the per-core latency decreases by an order of magnitude (see the orange lines in Figure 16). In contrast to using a shared table, the amount of reordering when using the per-bucket table is 4x lower and L2 cache misses are 2x lower. This shows that the proposed migration technique is suitable for high-frequency core scaling, which requires efficient compaction of multiple applications with variable offered loads.

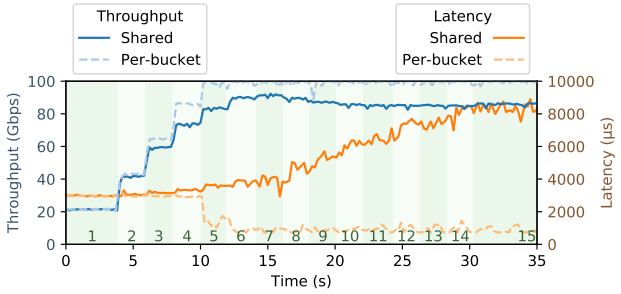


Figure 16: Performance comparison of per-bucket tables versus a single shared table, when adding an additional core every 2 seconds by reassigning the RSS indirection table. The per-bucket table is 20% more efficient than the shared table and enables seamless scaling, proving to be fit-for-purpose to back up RSS++'s migrations.

5.7 Can RSS++ handle realistic use cases?

To demonstrate that RSS++ can handle realistic use cases we execute three test cases with network functions running on 1 to 16 cores: (*i*) a firewall (labeled "FW") with 36K rules built from the traffic classes observed in the trace; (*ii*) chained with a NAT taken from [3], and (*iii*) a DPI using Hyperscan[58]; a pattern matcher that works in *streaming* mode, allowing per-stream state, while updating the state as packets go through. To reach higher speed, we preload the first 200M packets of the Campus #4 trace in memory and replay it in a loop at the NIC's 100 Gbps maximum line-rate.

Figure 17 visualizes the performance of RSS, Sprayer, and RSS++ when running the three service chains described above. As Sprayer dispatches packets of the same flow to multiple cores, each core accesses multiple paths of the firewall's classification tree. For this reason Sprayer exhibits 15% more cache misses than RSS++. RSS can achieve a steady throughput (almost) at 100 Gbps, but with 4 additional cores than RSS++ (*i.e.*, 12 versus 8).

As a NAT requires flow-based processing, Sprayer suffers from contention among the cores as mentioned in §5.5. Specifically, the throughput of Sprayer caps below 60 Gbps, as shown by the pink triangles and the pink dashed lines in Figure 17. When realizing the FW+NAT service chain, RSS cannot achieve line-rate even with 15 cores (while 10 are enough with RSS++) because RSS leaves some cores overloaded, dropping packets that could be processed by some other less-busy cores. In contrast, RSS++ demonstrates a linear increase in throughput with an increasing number of cores until the line-rate limit (see the purple diamonds with the purple dashed lines in Figure 17).

Finally, the dotted lines in Figure 17 depict the performance of the three systems when realizing the FW+NAT+DPI service chain. As also noted in the FW+NAT case, Sprayer's per-packet dispatching incurs excessive performance overhead due to locking, which prevents Sprayer from achieving more than 57 Gbps. RSS++ and RSS achieve a comparable linear throughput increase per additional core up to a maximum of 70 Gbps and 63 Gbps respectively.

Key outcome: RSS++ can realize stateful service chains (*i.e.*, FW+NAT) at line-rate 100 Gbps, while demonstrating up to 10% and 40% higher throughput than RSS and Sprayer respectively.

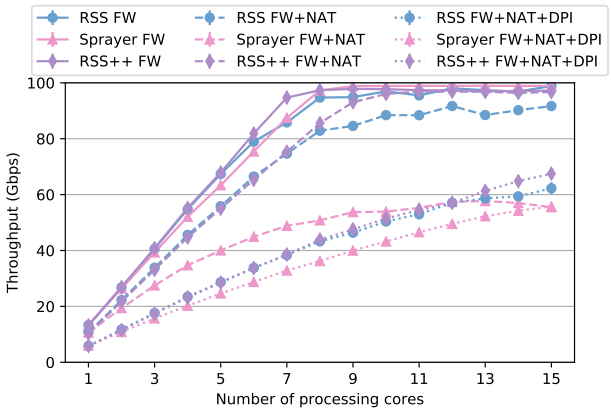


Figure 17: RSS++ better exploits the available cores, while it is the only solution (among the systems being tested) able to achieve line-rate 100 Gbps throughput for the stateful FW+NAT service chain.

6 RELATED WORK

Here, we discuss related efforts beyond the work mentioned inline throughout this paper.

6.1 Intra-server load-balancing

Shenango [45] uses a dedicated core to dispatch traffic to a set of cores per application. As described in § 5, this solution imposes an excessive performance cost compared to directly sending the packets to the correct core. Moreover, Shenango does not re-balance load between cores, but rather uses the RSS hash to dispatch packets to the processing cores, forcing either increased tail latency or a lower average CPU utilization. Finally, Shenango does not handle state migration when re-allocating cores, as it likely relies on a slow shared flow table per application.

ZygOS [50] uses an in-kernel software layer atop the network layer (which relies on RSS) to allow task exchange between cores. To do so, a single-producer multi-consumer shuffle queue per core is used to store ready-to-serve connections. Such connections can be either handled by a local core or stolen by a remote core which, at

the time, experiences low utilization. Shinjuku [20] uses RSS to split input load among multiple software-based dispatching cores, which in turn queue and schedule requests to worker cores. Shinjuku improves upon ZygOS for heavy-tailed and multi-modal load distributions, while achieves similar performance with ZygOS and IX [5] for light-tailed workloads. In contrast to ZygOS and Shinjuku, RSS++ allows more efficient connection "stealing" between cores using a hardware-level method (*i.e.,* tweaking the NIC indirection table) available in commodity NICs. Moreover, RSS++ reduces state migration overheads by batching migrated connections; such a feature is neither supported by ZygOS nor Shinjuku.

State migration can be done in advance, when the re-balancing is decided but before packets are reassigned to new shards. Split/Merge [51] scales out by selecting a shard to *split*. This shard is duplicated with its whole state. Consequently, half of this replicated state will eventually die off, (*i*) leading to some temporary memory overuse, while (*ii*) causing unnecessary delay to the migration. Split/Merge employs classification hardware (*e.g.,* an SDN switch) capable of maintaining per-flow entries to handle scaling. This needs a tight synchronization between servers and remote entities, such as an SDN controller.

Affinity-Accept [49] implements a sharded version of the Linux kernel to load-balance new connections from overloaded to underloaded cores. Every 100 ms a FlowDirector [17] entry is re-programmed (similar to our use of the RSS table) to migrate a flow group (*i.e.,* similar to RSS buckets) to direct packets to a core that handles connections mostly stolen by another core. Unfortunately, this scheme defeats sharding as it can end up with each flow group that is handled by a given core containing connections handled by other cores. This problem may get worse, especially with long-lived flows that become more common with the web moving towards HTTP/2. § 5.4 showed that a per-flow, load-aware dispatching strategy for new connections is insufficient to achieve fairness. Pesterev et al. propose a more efficient implementation of the same load-aware strategy. They propose that stealing is done according to a binary (*i.e.,* busy/non-busy) state rather than the actual load. In contrast, our scheme quickly computes a near-optimal assignment directly handled by the NIC. This allows RSS++ to start sharding at the DMA ring, enabling a run-to-completion approach *without* a queue to separate the network stack from the processing path. Our state migration algorithm involves per-core hash-tables with established connections and lock-free processing, while Affinity-Accept uses multiple hash-tables shared between all cores. Unlike RSS++, the Affinity-Accept source code is not publicly available, preventing further comparison.

MICA [33] clients *statically* encode core indices into UDP datagram headers by "hijacking" the destination port field. Then, MICA exploits NICs' perfect match classifiers (*e.g.,* FlowDirector) to quickly dispatch input requests to the correct shard. This design choice (*i*) restricts MICA to stateless UDP-only scenarios, (*ii*) hinders dynamic load-balancing due to the fixed data partitioning among cores, and (*iii*) complicates real deployments, *e.g.,* when Network Address and Port Translation (NAPT) middleboxes are present. RSS++'s flow dispatching mechanism is also hardware-based, but without the restrictions above.

6.2 NIC-assisted load-balancing

FlexNIC [28] uses software to emulate NICs with enhanced features, allowing finer programmability for core selection. RPCValet [8] proposes modifications to networking hardware by creating an API between the NIC and the system allowing better decision-making regarding which queues should be selected for each packet. This could potentially outperform Sprayer [55] (see § 5.5). Instead of relying on randomness, RPCValet proposes a load-aware queue selection scheme. As the scope of RPCValet is RPC (*i.e.,* mostly single-packet requests), they omit mentioning the problem of flow affinity; hence, we conjecture that either RPCValet's scheme is flow-unaware, thus exhibits the same performance problem as Sprayer for stateful network functions or needs an in-NIC flow table to store per-flow decisions combined with an eventual migration scheme.

Finally, SmartNICs [40–43, 52] could be used to offload additional parts of intra-server load-balancing schemes. For example, Netronome Agilo FX [43] SmartNICs can offload XDP's BPF programs; with such a NIC RSS++'s packet counting can be performed in hardware. In contrast, we show that RSS++ achieves near-optimal load-balancing and scaling across cores even at 100 Gbps links speeds with commodity hardware and with no additional cost.

6.3 Migration avoidance

RSS++ minimizes the number of flow transfers needed when a load balancing action is enforced (only a few transfers of per-bucket tables occur from time to time). Consequently, the performance impact of RSS++ is just a few L1/L2 cache misses that will occur at every migration. One can *send back* packets to the previously assigned core that still contains their state. This is similar to what Beamer[44] or Faild[1] do between servers (*i.e.,* "daisy chaining"). However, this would lead to inter-core transfers of packets, leading to buffering and requiring synchronization mechanisms between cores. With short flows, it is likely that moving the flow state is not worth the cache misses. E2 [47] installs rules in switches in front of a server to redirect existing flows once a load threshold is reached. Similarly, U-HAUL [35] only migrates elephant flows between NFV data plane instances, avoiding migration of mice flows using preceding Software-Defined Networking (SDN) switches to remember the mapping of the elephant flows. As future work, we will attempt to make RSS++ aware of the presence of mice/elephant flows. However, detecting mice and elephants necessitates keeping per-flow state, a technique shown to be very expensive (without specialized hardware) in § 5.4.

7 DISCUSSION AND FUTURE WORK

In this section, we discuss the context, application areas, and assumptions of this work as well as potentials to extend RSS++ in the near future.

7.1 Workload and processing matters

We target sharded systems with network-oriented workloads, where a set of cores is dedicated to one application. If background processing causes a core's load to increase, RSS++ will migrate buckets to other cores. RSS++ assumes that migrating packets will also migrate a proportional amount of load. It is possible that a core, even with only a single RSS bucket assigned, receives multiple

elephant flows and exhibits too much load. The more RSS buckets used, the less likely this situation is to happen. A potential solution to investigate would be to use NIC flow classification to move elephant flows out of the overloaded bucket. However, this was never been needed during the course of our experiments.

For the Linux implementation, we believe that RSS++ is already a better alternative than the Linux kernel's IRQ balance for spreading the network load across multiple cores. IRQ uses core pipelining to forward packets from a set of reception queues, served by a set of cores, to a second set of queues, served by another set of cores, handled by the applications. Pipelining involves inter-core communication; therefore, it is more expensive than sharding that runs-to-completion. Some DPDK-based packet processing frameworks (such as SoftNIC [14], E2 [47], Flurries [61], and NFP [57]) advocate pipelining for some use-cases. However, as network interfaces are currently reaching 400 Gbps and cores' operating frequency have peaked, it becomes clear that pipelining will dedicate *multiple* cores just for the first part of the pipeline (*i.e.,* packet reception and, in general, a part of the network stack processing). In this case, pipelining approaches could also benefit from RSS++. Specifically, input packets arriving at high speeds can be load-balanced across the reception cores of the pipeline in a more effective manner.

7.2 Short, numerous flows

As highlighted in §1, RSS is expected to evenly balance a high number of short flows among cores (still RSS++ provides support for scaling under sharding). While this scenario is not often encountered in the Internet[13, 32], it is typical of key/value store workloads. MICA[33] proposes tagging packets at the client to directly send packets towards the correct shard that contains the value for a given key. However, key popularity will create a high skew, thus load-imbalance. RSS++ can solve this problem by hashing on the key. The buckets containing more popular keys will then be automatically migrated to cores that contain less-popular buckets, without requiring any client-side support or exposing every single core to the replication manager. We will further investigate the usage of RSS++ with key/value stores in future work.

7.3 NUMA and NUCA awareness

As accessing memory from another CPU socket is more costly, the target assignment algorithm separates cores and buckets per Non-Uniform Memory Access (NUMA) node. The assignment is solved independently for each NUMA node, unless there is a large difference in the average load between the NUMA nodes, in which case the assignment will be solved on all cores and buckets at once to re-arrange the balance, allowing inter-NUMA transfers. Unfortunately, in recent x86 architectures a NIC is attached to a given socket, hence its traffic will go via this CPU's interconnect resulting in decreased performance when using cores in another NUMA node [24, 27]. Further evaluation of NUMA awareness and potential solutions, such as multi-socket NICs[37], are left for future work. RSS++ could also exploit Non-Uniform Cache Access (NUCA) awareness. Our algorithm could be augmented using the technique of [11] to re-assign the buckets of each overloaded core, first to a collocated hardware-thread and then assigned based upon the transfer times between cores.

7.4 Scaling multiple applications and virtualization

In the present design, RSS++ scaling is done at the granularity of a single RSS table. Having multiple applications served by the same RSS table will lead to context switches and will force all these applications to scale at the same time, using the same number of cores. We envision RSS++ with one RSS table per application, using a minimal number of cores to collocate as many applications as possible thus eliminate context-switching. This can be done with existing NICs using Virtual Functions (VFs). VFs are virtual NICs, exposed by a single physical NIC, supporting various traffic classification schemes, ranging from Medium Access Control (MAC) address or Virtual Local Area Network (VLAN) ID filtering to multi-header field flow classification on more evolved NICs. To associate multiple VFs with multiple virtual machines (thus potentially multiple applications) Single Root I/O Virtualization (SR-IOV) can be used. All NICs used for the experiments in this paper offer this possibility. Specifically, we can use Virtual Machine Device queues (VMDq) on Intel NICs or the flow classification engine of Mellanox NICs to perform fast RSS table updates on a per-VF basis. This way, RSS++ can load-balance each of the VFs independently, thus support multiple applications (virtualized or not) with minimal network setup modifications.

7.5 Outgoing connections

In this paper, we focused on evaluating *server* and *NFV* scenarios. To initiate new connections in a sharding context, one needs to ensure returning packets will correctly find their state. One can hash in software the 4-tuple of a new connection as RSS and mTCP[19] would do to find the future RSS bucket of a connection, and then add the entry in the correct flow table. As future work, we will evaluate this technique for client scenarios.

8 CONCLUSION

This paper bridges the gap between inter-server load-balancing and the fact that today's servers are not single entities but rather many-core machines that must locally balance requests across cores.

RSS++ combines NIC-based scheduling with sharding to remove the need for traditional scheduling; therefore, avoiding context switching and costly cache misses. The DPDK version of RSS++ achieves (*i*) 14x lower 95th percentile tail latency and (*ii*) orders of magnitude fewer packet drops compared to RSS under high CPU utilization. The Linux kernel implementation of RSS++ load-balances input flows at the rate of 100 Gbps across the correct number of cores, while bounding tail latency and transferring a minimal amount of state.

All software and automated scripts to reproduce the experiments shown in this paper are available at [2].

ACKNOWLEDGMENTS

We would like to thank our shepherd KyoungSoo Park and the anonymous reviewers for their insightful comments on this paper. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 770889). This work was also funded by the Swedish Foundation for Strategic Research (SSF).

REFERENCES

[1] João Taveira Araújo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. 2018. Balancing on the Edge: Transport Affinity Without Network State. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI'18)*. USENIX Association, Berkeley, CA, USA, 111–124. http://dl.acm.org/citation.cfm?id=3307441.3307452

[2] Tom Barbette. 2019. Public repository with all the experiments conducted in the course of the RSS++ paper. https://github.com/rsspp/experiments

[3] Tom Barbette, Cyril Soldani, Romain Gaillard, and Laurent Mathy. 2018. Building a chain of high-speed VNFs in no time. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE, Bucharest, Romania, 8 pages. https://doi.org/10.1109/HPSR.2018.8850742

[4] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast userspace packet processing. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '15)*. IEEE Computer Society, Washington, DC, USA, 5–16. http://dl.acm.org/citation.cfm?id=2772722.2772727

[5] Adam Belay, George Prekas, Mia Primorac, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2016. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst.* 34, 4, Article 11 (Dec. 2016), 39 pages. https://doi.org/10.1145/2997641

[6] Nicola Bonelli, Andrea Di Pietro, Stefano Giordano, and Gregorio Procissi. 2012. On Multi—gigabit Packet Capturing with Multi—core Commodity Hardware. In *Proceedings of the 13th International Conference on Passive and Active Measurement (PAM'12)*. Springer-Verlag, Berlin, Heidelberg, 64–73. https://doi.org/10.1007/978-3-642-28537-0_7

[7] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 153–167. https://doi.org/10.1145/3132747.3132772

[8] Alexandros Daglis, Mark Sutherland, and Babak Falsafi. 2019. RPCValet: NI-Driven Tail-Aware Balancing of µs-Scale RPCs. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. USENIX Association, NY, USA, 35–48. https://doi.org/10.1145/3297858.3304070

[9] Diego Didona and Willy Zwaenepoel. 2019. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 79–94. https://www.usenix.org/conference/nsdi19/presentation/didona

[10] Mihai Dobrescu, Katerina Argyraki, Gianluca Iannaccone, Maziar Manesh, and Sylvia Ratnasamy. 2010. Controlling Parallelism in a Multicore Software Router. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow (PRESTO '10)*. ACM, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/1921151.1921154

[11] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2019. Make the Most out of Last Level Cache in Intel Processors. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, USA, Article 8, 17 pages. https://doi.org/10.1145/3302424.3303977

[12] Massimo Gallo and Rafael Laufer. 2018. ClickNF: a modular stack for custom network functions. In *USENIX Annual Technical Conference (ATC'18)*. USENIX Association, Boston, MA, 745–757. https://www.usenix.org/conference/atc18/presentation/gallo

[13] Liang Guo and Ibrahim Matta. 2001. The war between mice and elephants. In *Proceedings of the 9th International Conference on Network Protocols (ICNP)*. IEEE, Riverside, CA, USA, 180–188. https://doi.org/10.1109/ICNP.2001.992898

[14] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. *SoftNIC: A Software NIC to Augment Hardware*. Technical Report UCB/EECS-2015-155. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

[15] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 135–148. http://dl.acm.org/citation.cfm?id=2387880.2387894

[16] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*. ACM, New York, NY, USA, 54–66. https://doi.org/10.1145/3281411.3281443

[17] Intel. 2016. Ethernet Flow Director. http://www.intel.com/content/www/us/en/ethernet-controllers/ethernet-flow-director-video.html

[18] Intel. 2016. Receive-Side Scaling (RSS). http://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf

[19] 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 *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 489–502. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong

[20] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for µSecond-scale Tail Latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19)*. USENIX Association, Berkeley, CA, USA, 345–359. http://dl.acm.org/citation.cfm?id=3323234.3323264

[21] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2014. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM SIGCOMM conference (2014)*. ACM, NY, USA, 295–306. https://doi.org/10.1145/2619239.2626299

[22] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. 2012. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*. ACM, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2391229.2391238

[23] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing (STOC '97)*. ACM, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660

[24] Georgios P. Katsikas. 2018. *NFV Service Chains at the Speed of the Underlying Commodity Hardware*. Doctoral thesis. KTH Royal Institute of Technology, School of Electrical Engineering and Computer Science, Kista, Sweden. http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-233629 TRITA-EECS-AVL-2018:50.

[25] 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 *15th USENIX Conference on Networked Systems Design and Implementation (NSDI'18)*. USENIX Association, Renton, WA, 171–186. https://www.usenix.org/system/files/conference/nsdi18/nsdi18-katsikas.pdf

[26] Georgios P. Katsikas, Marcel Enguehard, Maciej Kuźniar, Gerald Q. Maguire Jr., and Dejan Kostić. 2016. SNF: Synthesizing high performance NFV service chains. *PeerJ Computer Science* 2 (Nov. 2016), e98. https://doi.org/10.7717/peerj-cs.98

[27] Georgios P. Katsikas, Gerald Q. Maguire Jr., and Dejan Kostić. 2017. Profiling and accelerating commodity NFV service chains with SCC. *Journal of Systems and Software* 127C (Feb. 2017), 12–27. https://doi.org/10.1016/j.jss.2017.01.005

[28] Antoine Kaufmann, SImon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS '16*. ACM Press, Atlanta, Georgia, USA, 67–81. https://doi.org/10.1145/2872362.2872367

[29] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. Comput. Syst.* 18, 3 (Aug. 2000), 263–297. https://doi.org/10.1145/354871.354874

[30] Richard Earl Korf. 2009. Multi-way number partitioning. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence (IJCAI'09)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 538–543. http://dl.acm.org/citation.cfm?id=1661445.1661531

[31] KVM. 2019. Kernel-based Virtual Machine (KVM). https://www.linux-kvm.org/

[32] Kun-chan Lan and John Heidemann. 2006. A measurement study of correlations of internet flow characteristics. *Computer Networks* 50, 1 (2006), 46–62.

[33] Hyeontaek Lim, Donsu Han, David G Andersen, and Michael Kaminsky. 2014. MICA: A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[34] Linux Foundation. 2019. Data Plane Development Kit (DPDK). http://www.dpdk.org

[35] Libin Liu, Hong Xu, Zhixiong Niu, Peng Wang, and Dongsu Han. 2016. U-HAUL: Efficient state migration in NFV. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '16)*. ACM, New York, NY, USA, Article 2, 8 pages. https://doi.org/10.1145/2967360.2967363

[36] DPDK mailing list. 2019. doc: fix update release notes for Mellanox drivers. https://mails.dpdk.org/archives/dev/2019-May/132128.html

[37] Mellanox. 2019. Socket Direct. http://www.mellanox.com/page/products_dyn?product_family=285&mtag=socketdc

[38] Aravind Menon and Willy Zwaenepoel. 2008. Optimizing TCP Receive Performance. In *USENIX 2008 Annual Technical Conference (ATC'08)*. USENIX Association, Berkeley, CA, USA, 85–98. http://dl.acm.org/citation.cfm?id=1404014.1404021

[39] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. 2012. CPHASH: A Cache-partitioned Hash Table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 319–320. https://doi.org/10.1145/2145816.2145874

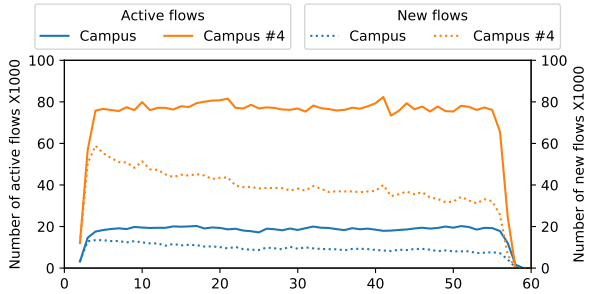
[40] Netcope Technologies. 2018. Netcope P4 Cloud: Online P4 to FPGA synthesis and in-hardware evaluation. https://www.netcope.com/en/products/netcopep4
[41] Netronome. 2017. Agilio LX 1x100GbE SmartNIC. https://www.netronome.com/m/documents/PB_Agilio_Lx_1x100GbE.pdf
[42] Netronome. 2019. Agilio CX. https://www.netronome.com/products/agilio-cx/
[43] Netronome. 2019. Agilio FX. https://www.netronome.com/products/agilio-fx/
[44] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless datacenter load-balancing with Beamer. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 125–139. https://www.usenix.org/conference/nsdi18/presentation/olteanu
[45] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout
[46] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (May 2004), 122–144. https://doi.org/10.1016/j.jalgor.2003.12.002
[47] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: a framework for NFV applications. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 121–136. https://doi.org/10.1145/2815400.2815423
[48] 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 Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Berkeley, CA, USA, 203–216. http://dl.acm.org/citation.cfm?id=3026877.3026894
[49] Aleksey Pesterev, Jacob Strauss, Nickolai Zeldovich, and Robert T Morris. 2012. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 337–350. https://doi.org/10.1145/2168836.2168870
[50] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 325–341. https://doi.org/10.1145/3132747.3132780
[51] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, Berkeley, CA, USA, 227–240. http://dl.acm.org/citation.cfm?id=2482626.2482649
[52] Ron Renwick. 2017. Increase Application Performance with SmartNICs. https://www.openstack.org/assets/presentation-media/Netronome-OpenStack-Summit-Marketplace-presentation.pdf
[53] Amir Roozbeh, Joao Soares, Gerald Q. Maguire Jr., Fetahi Wuhib, Chakri Padala, Mozhgan Mahloo, Daniel Turull, Vinay Yadhav, and Dejan Kostić. 2018. Software-Defined "Hardware" Infrastructures: A Survey on Enabling Technologies and Open Research Directions. *IEEE Communications Surveys & Tutorials* 20, 3 (thirdquarter 2018), 2454–2485. https://doi.org/10.1109/COMST.2018.2834731
[54] 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 Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 123–137. https://doi.org/10.1145/2785956.2787472
[55] Hugo Sadok, Miguel Elias M Campista, and Luís Henrique MK Costa. 2018. A Case for Spraying Packets in Software Middleboxes. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks (HotNets '18)*. ACM, New York, NY, USA, 127–133. https://doi.org/10.1145/3286062.3286081
[56] Ethan L. Schreiber, Richard E. Korf, and Michael D. Moffitt. 2018. Optimal Multi-Way Number Partitioning. *J. ACM* 65, 4, Article 24 (July 2018), 61 pages. https://doi.org/10.1145/3184400
[57] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. 2017. NFP: Enabling Network Function Parallelism in NFV. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. ACM, New York, NY, USA, 43–56. https://doi.org/10.1145/3098822.3098826
[58] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 631–648. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang
[59] Wenji Wu, Phil DeMar, and Matt Crawford. 2010. Why can some advanced Ethernet NICs cause packet reordering? *IEEE Communications Letters* 15, 2 (February 2010), 253–255. https://doi.org/10.1109/LCOMM.2011.122010.102022
[60] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-latency Networking with the OS Stack and Dedicated NICs. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 43–56. http://dl.acm.org/citation.cfm?id=3026959.3026964
[61] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, and Timothy Wood. 2016. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies (CoNEXT '16)*. ACM, New York, NY, USA, 3–17. https://doi.org/10.1145/2999572.2999602

A TRACES

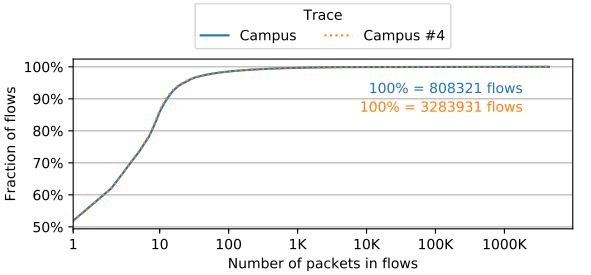
Table A-1 shows a summary of the characteristics of the first 90 seconds of each trace described in § 5.1, while Figure A-1 visualizes key statistics of these traces.

Table A-1: Characteristics of the traces.

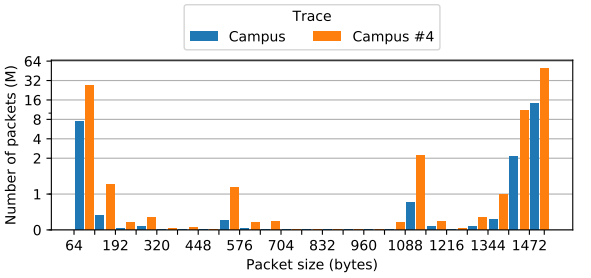
	Campus	Campus #4	*unit*
Total # of flows	808	3283	KFlows
Avg. active/new flows	18.1 / 8.7	71.3 / 34.9	KFlows/s
Avg. # of pkts/flow	4434	4423	pkts/flow
Bandwidth	4.1	15.1	Gbps
Avg. packet size	1011	1002	Bytes
Total data size	28.7	104.7	GB



(a) Number of active and new flows observed each second.



(b) Distribution of flow sizes.



(c) Distribution of packet sizes (in bytes).

Figure A-1: Statistics for Campus and Campus #4 traces.

The trace capture kept the beginning of every packet up to and including the L4 header (TCP or UDP), while dropping IPv6 packets. The trace is then randomly padded up to 128 additional bytes **or** the original capture length if it is less than 128 bytes. This randomness in the payload is added to be more realistic towards Sprayer. Sprayer would be penalized by only zero-padded payload as it selects the queue according to the packet checksum. When replayed, the remaining payload is padded up to the original capture length without changing the state of the memory, but the probability of the remaining payload being zero is high. Statistics are gathered by running live experiment, that is, collecting the information of interest on the DUT. To avoid artifact of the generator, unless stated otherwise, we replay traces with a 2 seconds speed increase at the beginning, and a 2 seconds speed decrease at the end.

Figure A-1a shows the number of active and new flows observed every second. Active flows are considered all established flows with at least one packet during the last second. Figures A-1b and A-1c show the distributions of flow and packet sizes respectively in the Campus and Campus #4 traces. Both traces are comprised of a large fraction of (almost) MTU-sized packets (*i.e.,* between 1000-1500 bytes) along with a substantial fraction of small packets (*i.e.,* between 64-200 bytes).