Dyssect: Dynamic Scaling of Stateful Network Functions

Fabrício B. Carvalho
UFMS and UFMT
fabricio.carvalho@ufms.br

Ronaldo A. Ferreira
UFMS
raf@facom.ufms.br

Ítalo Cunha
UFMG
cunha@dcc.ufmg.br

Marcos A. M. Vieira
UFMG
mmvieira@dcc.ufmg.br

Murali K. Ramanathan
Uber Technologies, Inc.
murali@uber.com

***Abstract*—Network Function Virtualization promises better utilization of computational resources by dynamically scaling resources on demand. However, most network functions (NFs) are stateful and require state updates on a per-packet basis. During a scaling operation, cores need to synchronize access to a shared state to avoid race conditions and to guarantee that NFs process packets in arrival order. Unfortunately, the classic approach to control concurrent access to a shared state with locks does not scale to today's throughput and latency requirements. Moreover, network traffic is highly skewed, leading to load imbalances in systems that use only sharding to partition the NF states. To address these challenges, we present Dyssect, a system that enables dynamic scaling of stateful NFs by disaggregating the states of network functions. By carefully coordinating actions between cores and a central controller, Dyssect migrates shards and flows between cores for load balancing or traffic prioritization without resorting to locks or reordering packets. Our experimental evaluation shows that *Dyssect* reduces tail latency up to 32% and increases throughput up to 19.36% when compared to state-of-the-art competing solutions.**

I. Introduction

Network Function Virtualization (NFV) promises to reduce hardware costs and increase network manageability by running network functions as software applications on general-purpose commodity servers. The flexibility provided by software implementations allows a network function (NF) to be dynamically scaled out by adding new instances or allocating additional resources (*e.g.,* CPU cores and memory) when the traffic load increases. Similarly, when the traffic load decreases, an NF should be scaled in by decreasing the number of instances or resources so that hardware can be reallocated to other tasks or put to sleep to save energy.

The vast majority of network functions are stateful and may require state updates on a per-packet basis [1], [2]. During a scaling operation, cores need to synchronize access to a shared state to avoid race conditions and to guarantee that NFs process packets in arrival order. The classic approach to control concurrent access to a shared state is the use of locks. Unfortunately, locks decrease performance [3] and prevent network functions from operating at line rate in the multi-gigabit links that are common today.

Several research proposals use state *sharding* to avoid the use of locks [4]–[8]. Sharding is a technique that partitions the (global) state of a network function into disjoint state subsets (*shards*). Flows are mapped to shards using the result of a hash function computed on a *flow key* (*e.g.,* the 5-tuple), and each shard is mapped to *one* CPU core to avoid concurrent accesses. Modern high-speed NICs can hash incoming packets and place them in per-core queues specified in a *Receive-Side Scaling* (RSS) indirection table. Cores poll their queues to receive and process packets for their shards at high throughput. In this approach, statically mapping shards to cores leads to load imbalance [9], [10], and using a centralized software packet dispatcher does not scale to the line speed of the current high-speed links [4], [11], [12].

A recent effort proposes dynamic reassignments of shards to balance the load across cores [10]. While this approach improves performance when compared with RSS, it is far from solving the problem. For instance, one shard might have multiple large-volume flows that a single core cannot handle, but unfortunately, the system cannot allocate more cores to handle the load, as all the flows in a shard are assigned to a single core. Moreover, packets from high-priority flows might experience higher delays in a queue behind packets from large-volume flows.

The current trend is to increase the NIC's RSS indirection table (*e.g.,* Mellanox ConnectX-5 EN has 512 entries [13]) and, consequently, allow more shards to reduce the likelihood of multiple large-volume flows in a single shard. Unfortunately, a large number of shards exacerbates load imbalance [14] between shards and requires migrations with higher frequency, leading to cache invalidation and, consequently, loss of performance. Moreover, a large number of shards reduces cache locality, as a core has to address a large number of different shards that might not all fit in its local cache. Also, some NICs may require hundreds of milliseconds to update their RSS indirection table (§IV), hindering or reducing the effectiveness of shard migrations.

A significant scalability barrier for deploying network functions in commodity servers is the difference between processor and memory speeds, also known as the memory-wall problem [15]. Cache memories mitigate this problem, but they work well only if network functions take advantage of temporal and spatial localities during packet processing. For example, Figure 1 shows the performance impact of the number of shards in CPU performance metrics. In this simple experiment, a single core runs a NAT and processes 32-packet batches from one million flows following a Zipf distribution with $\alpha = 1.1$, which is similar to α of the CAIDA trace we use in §IV-A. As we can see, the throughput drops up to 43.3%

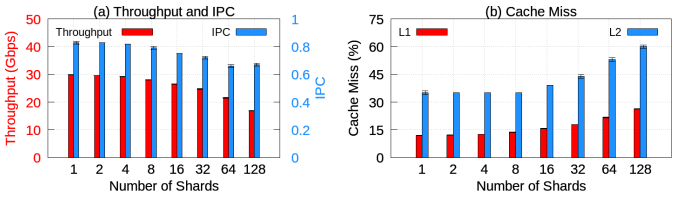

Figure 1: Throughput, instructions per cycle (IPC), and cache misses for different numbers of shards.

(29.8 vs. 16.9 Gbps) when we vary the number of shards from 1 to 128. We attribute this decrease in throughput to the loss of effectiveness of the CPU cache. We can see in the figure that cache-misses increase as the number of shards increases, leading to fewer instructions per cycle.

Overcoming the limitations above is easy in stateless network functions. We can increase or decrease the number of cores and freely move packets from one core to another without worrying about state consistency. However, the dynamic scaling of stateful NFs presents several challenges. Scaling may require state migration between cores to prevent race conditions. Moreover, it should preserve flow to core affinity to avoid packet reordering, use of locks, and inter-core communication, which degrade overall performance.

In this paper, we present the design and implementation of *Dyssect*, a system that improves NFV performance by allowing lock-less state-migration operations to achieve load balancing, prioritize traffic, and minimize resources while satisfying user-specified Service-Level Objective (SLO) requirements. By using a hardware-software codesign, *Dyssect* circumvents the many pitfalls of current approaches. First, it uses the hardware (the NIC's RSS indirection table) to steer packets to shards and to transfer shards between cores, avoiding the bottleneck of having a single core dispatching all the packets (*e.g.,* Shenango [12]). Second, *Dyssect* disaggregates state from network functions using a data structure that allows a packet to carry a reference to its flow state. This mechanism is crucial for allowing fine-grained migrations of individual flows between cores to prioritize traffic or achieve more even load distribution without introducing race conditions. By carefully coordinating migration actions between cores and a centralized controller, *Dyssect* prevents packet reordering and deadlocks. Third, because *Dyssect* can manage traffic at the flow level, it can use a smaller number of shards than other approaches (*e.g.,* RSS++ [10]) and avoid frequent shard transfers, overcoming the performance penalties discussed above (Figure 1). Finally, *Dyssect* assigns shards and flows to cores using optimization models that capture long- and short-timescale behaviors of the system to achieve operator-specified SLOs.

We implement *Dyssect* as user-level modules using BESS [16] with DPDK for fast packet processing and Gurobi [17] for solving our mathematical models. Experimental evaluations on a multi-core server running multiple NFs and processing synthetic and real traffic show that *Dyssect* reduces tail latency up to 32% and increases throughput up to 19.36% compared to the state-of-the-art approaches that use only sharding for partitioning the states of NFs.

II. Dyssect

Dyssect manages the workload of stateful VNF chains with the goal of processing packets at tens of Gbps using general-purpose hardware. We propose a novel architecture for packet processing. *Dyssect*'s key contribution is a set of techniques that provide fine control over flow-to-core assignments and overcome limitations that hinder cache locality (Figure 1), significantly improving packet processing throughput.

Dyssect achieves fine control over flow-to-core assignments by combining three techniques. First, *Dyssect* stores pointers to flow states in a data structure and attaches a reference to this structure in the packet metadata, amortizing lookup costs and enabling any core to process packets from any flow (§II-A). Second, we propose distributed, high-performance, lock-free synchronization mechanisms that allow a core to offload processing of a flow to another core while guaranteeing in-order packet processing (§II-B). Finally, we design optimization models that compute flow-to-core assignments to minimize operational costs, distribute traffic load, maximize throughput, and meet SLO latency constraints (§II-C).

Dyssect only supports network functions with *per-flow state*, *i.e.*, disjoint sets of flows can be allocated to different instances of a network function, and each instance has exclusive access to its flows' states, precluding the need of communication and synchronization across instances. Per-flow state partitioning is essential for a network function to process packets at high speed, as synchronizing accesses to the shared state does not scale, especially in network functions that need to update their states for every packet. This is the case for many common network functions, including stateful firewalls, NATs, and load balancers [1], [2]. *Dyssect* assumes that service chains employ the *run-to-completion model* (RTC), *i.e.*, all network functions in the service chain process the packet without yielding the CPU. The RTC model is also assumed in many frameworks, particularly those which aim at providing low latency and high throughput for packet processing [18]–[21].

A. State Management

Dyssect partitions flows, and thus the state of a network function, into S shards such that S is a power-of-two and $S \leqslant E$, where E is the maximum number of entries in the NIC's RSS indirection table. This partitioning allows us to efficiently identify the shard of a packet by using the $\log_2 S$ least significant bits of a hash value on the packet's flow key (*e.g.*, 5-tuple for TCP or UDP). *Dyssect* avoids computing the hash in software and instead uses the same RSS hash value that the NIC computes to direct packets to cores. As RSS places all packets of a shard on one core, *Dyssect* avoids the use of locks for accessing the shard.

1) State Initialization: A recent study shows that state lookup represents a significant cost for packet processing in software [22]. This cost compounds when we have a service chain with multiple network functions, as each network function does a lookup to find its state associated with the packet. *Dyssect* disaggregates the states from the network functions by keeping one hash table for each shard, containing

one entry for each flow; these are referred to as the *flow table* and *flow entry*, respectively. A flow entry is an array with n pointers, where n is the length of the service chain, and the i^{th} pointer points to the flow state of its i^{th} network function. *Dyssect* adds a reference to the flow entry into the packet metadata.

During the initialization of the service chain (*i.e.*, before any packet is processed), each network function calls the function *InitState()* to receive a handle from the framework. *Dyssect* uses this handle to associate each network function to one element in the array of pointers, and the network functions use the handles to call the other functions of the *Dyssect*.

When a network function receives a packet, it first calls *GetState()* to retrieve the associated state. If the packet is the first of its flow, *Dyssect* returns a null pointer. In this case, the network function allocates the state for the flow and invokes the function *InsertState()* with a pointer to the flow state just allocated, its handle, and the packet. *Dyssect* stores the flow's state pointer in the flow entry in the respective flow table. It uses the flow key (*e.g.,* 5-tuple for TCP and UDP) associated with the packet to determine the flow entry in the hash table.

2) State Lookup: For subsequent packets of the flow, *Dyssect* does a single lookup in its data structure when it retrieves a packet from the NIC queue. To avoid further lookups for the packet, *Dyssect* inserts into the packet metadata a pointer to the flow entry, which has a pointer to each NF's state and takes exactly eight bytes. To get its state, each network function calls *GetState()* passing as parameters its handle and the packet. With the handle and the metadata in the packet, *Dyssect* can return the state of the network function by dereferencing the pointer in the packet metadata without doing new lookups. Figure 2 shows an overview of this process.

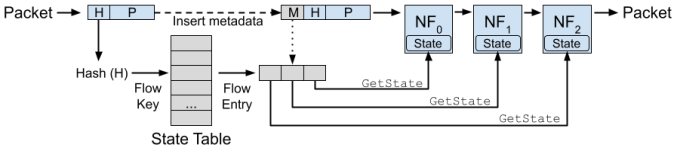

Figure 2: State management in *Dyssect*.

3) State Cleanup: When a network function finishes processing a flow (*e.g.,* by observing TCP FIN flags or from a timeout), it must call the *DeleteState()*. The *DeleteState()* function assigns null to the state pointer associated with the network function in the terminating flow's entry. When all pointers in a flow entry become null, *Dyssect* removes the flow entry from the shard's hash table. Each network function manages its own states, allocating the necessary amount of memory for its state and releasing memory when a flow ends; *Dyssect* only manages the state via (opaque) pointers.

B. Flow Assignment

Dyssect combines two mechanisms for assigning flows to cores (see Fig. 3). The first mechanism is coarse-grained and maps shards to cores. We leverage the NIC hardware and use its RSS capability. Each shard s comprises all entries of the RSS indirection table whose $\log_2 S$ least significant bits are

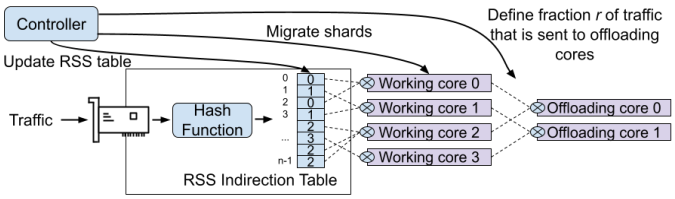

Figure 3: *Dyssect* overview.

equal to s. To assign shard s to a core, *Dyssect* updates all of s's entries in the RSS table to point to the core. *Dyssect* ensures all RSS entries of a shard s map to the same core.

The second mechanism is fine-grained and assigns a subset of flows in a shard to an offloading core. *Dyssect* employs the fine-grained flow assignment mechanism to balance the load between cores, prioritize certain types of traffic or meet operator-defined SLOs, as we discuss in §II-C.

Dyssect uses CPU cores as either *working* or *offloading* cores. A *working core* polls its NIC queue to retrieve batches of packets assigned to it using the coarse-grained RSS-based mechanism. For each packet in a batch, the working core performs a lookup in the shard's flow table to retrieve the corresponding flow entry, and adds a pointer to the flow entry into the packet metadata (§II-A). The working core then verifies if the packet belongs to a flow that should be sent to an offloading core (henceforth called an *offloading flow*) (see §II-B2). We use the term *offloading core* to refer to a core that processes offloading flows. Offloading cores do not have any extra functionality nor are more powerful than working cores.

Each working core is mapped to zero or one offloading core, *i.e.*, a working core either does not offload any packets or offloads packets to a single offloading core. This restriction simplifies coordination across cores during scaling operations, which change the mapping between working and offloading cores to redistribute traffic load. An offloading core receives packets from one or more working cores. Offloading cores maintain one queue for each working core it receives packets from and poll all queues in a round-robin fashion.

1) Dyssect Controller: The coarse- and fine-grained flow assignment mechanisms are centrally controlled, and flow assignment changes require coordination across cores to avoid packet reordering and race conditions in the data plane.

Algorithm 1 summarizes the operation of *Dyssect*'s centralized controller. The controller maintains the number of active working and offloading cores (n^{w} and n^{o}), a vector with the ratio (r) of each shard's traffic that should be offloaded to the corresponding offloading core, the association between shards and working cores ($\boldsymbol{A}$), and the mapping of working cores to offloading cores ($\boldsymbol{O}$).

The controller performs autoscaling operations periodically (Line 2). On each iteration, the controller builds a set of signals specifying autoscaling operations to be executed by each core (initialization in Line 3 and notification in Line 17). Over long timescales (Line 4), the controller runs an optimization that computes the number of working and offloading cores.

On every autoscaling operation, the controller runs a short-timescale solver. The short-timescale solver receives as input

Algorithm 1 Central Controller

Require: Short- and long-term optimization periods ϵ and τ
Require: Number of cores C and number of shards S
1: $n^{\text{w}} \leftarrow C, n^{\text{o}} \leftarrow 0, r \leftarrow \{0 \text{ for each } s \in S\}$
2: **at every** ϵ ms **do**
3: Reset signal[w] for all working cores
4: **if** τ s has elapsed **then**
5: $n^{\text{w}}, n^{\text{o}} \leftarrow$ longTimescaleSolver(C, S)
6: $\boldsymbol{A}, \boldsymbol{O}, r \leftarrow$ shortTimescaleSolver(n^{w}, n^{o})
7: **for each** $s \in S$ that r_s changed **do**
8: signal[shards[s].owner].offloadRatioChange $\leftarrow$ true
9: **for each** (s, w) assignment changed $\in \boldsymbol{A}$ **do**
10: shards[s].newOwner $\leftarrow w$
11: shards[s].hold $\leftarrow$ true
12: updateRSSTable()
13: **for each** (s, w) assignment changed $\in \boldsymbol{A}$ **do**
14: signal[shards[s].owner].migrations += (s, w)
15: **for each** (w, o) mapping changed $\in \boldsymbol{O}$ **do**
16: signal[w].newOffloadingCore $\leftarrow o$
17: Send signal[w] to each working core w

the current number of active working and offloading cores (computed by the long-timescale optimization), and updates the shard-to-core assignments, mapping between working and offloading cores, and offload ratios (Line 6).[1] For each offload ratio, assignment or mapping changes (Lines 7, 9, and 15), the controller updates signal accordingly (Lines 8, 14, and 16).

For each shard s that will migrate to another working core, the controller configures the working core w receiving the shard to hold packets from s in a queue for later processing (shards[s].hold, Line 11). This temporary delay allows s's previous working core (shards[s].owner) to finish processing any packets from s it may have retrieved from the network. The shards[s].hold operation takes effect immediately, before the controller updates the RSS indirection table (Line 12), which will cause packets from s to go into w's queue.

2) Dyssect Data plane: Working and offloading cores process packets continuously with efficient, lock-free synchronization. The main constraint on the data plane is ensuring packets from a flow are processed in order. Our insight is to delay processing of flows whose shards are assigned to different cores or flows that start or stop being offloaded. This delay allows the previous core to finish processing any pending packets for reassigned flows before the new core starts processing. To minimize delay, *Dyssect* only enqueues packets for flows that are reassigned during scaling operations. Processing of flows that are not reassigned is not delayed.

Dyssect maintains multiple queues (denoted with $\mathcal{Q}$ in the pseudocode). Queues are written to by a single core, and read from by a single core, although the writer and reader may be different for any single queue.[2]

Dyssect queues support a *cycling* operation, which allows cores to wait for packets queued before the start of the cycling operation to be processed while still being able to enqueue packets to the queue. A queue's writer core can start a cycling operation by setting a flag in the queue. On first seeing the flag, the reader process saves the current number of packets in the queue. The reader core clears the cycling flag after that many packets are processed from the queue, which signals to the writer core that the cycling has finished.

[1] It is possible that the optimization keeps offload ratios constant. However, updating offload ratios has an impact on packet processing performance that is proportional to the ratio differences (§II-B2), and there is no significant impact for signaling when there is no change in offload ratios.

Algorithm 2 Packet Processing at Working Cores

1: **procedure** PROCESSPACKETS
2: **for each** shard s assigned to this core **do**
3: **if** $|\text{self}.\mathcal{Q}_s^{\text{held}}| > 1$ **and not** shard[s].hold **then**
4: RunSFC(self.$\mathcal{Q}_s^{\text{held}}$)
5: **for each** packet p in self.$\mathcal{Q}^{\text{NIC}}$ **do**
6: $s \leftarrow$ getShard(p)
7: **if** shard[s].hold **then**
8: self.$\mathcal{Q}_s^{\text{held}} \ll p$
9: **else if** isOffloaded(p, r_s) **then**
10: self.$\mathcal{Q}^{\text{offloading}} \ll p$
11: **else if** self.offloadRatioChange **and**
12: isOffloaded(p, r_s^{old}) **then**
13: self.$\mathcal{Q}_s^{\text{pending}} \ll p$
14: **else**
15: RunSFC(p)

Algorithm 2 shows how working cores handle incoming packets. Packet processing is performed continuously. Before processing packets arriving from the network, the working core checks if there are any packets waiting to be processed for held shards that have finished migration to this core; these packets are processed immediately (Lines 2–4). The working core then processes packets arriving from the network (Line 5). For each packet, the working core first checks if the shard is under migration but still being processed by its previous core (Line 7); in this case, the working core enqueues the packet in a temporary queue to delay processing until the shard migration has finished (Line 8, see also Lines 2–4). Second, the working core checks if the packet belongs to a flow which is offloaded to the offloading core (Line 9), *i.e.*, a flow within the ratio r defined by the controller; in this case, the working core enqueues the packet in its offloading's core processing queue (Line 10). Lines 11–13 handle the case where shard offload ratios are being updated and some flows that were previously offloaded are no longer offloaded (*i.e.*, $r_s^{\text{old}} > r_s$). In this case, the working core needs to wait for the offloading core to finish processing any packets from these flows before proceeding. If the packet belongs to one such flow, the working core enqueues the packet in a temporary queue for processing after the offloading core has finished processing any previous packets from these flows. Finally, if none of the previous conditions apply, the packet can be immediately processed (Lines 14–15). Although Algorithm 2 processes one packet at a time for clarity, *Dyssect*'s implementation operates over batches of packets for improved performance.

Algorithm 3 shows pseudocode for how working cores handle scaling operations. These functions are called as needed, *in order*, whenever a signal is received from the controller (see Algorithm 1, Line 17). When updating offload ratios, the working core replaces self.$\mathcal{Q}^{\text{offloading}}$ with a new queue (Lines 2–3). The working core waits until its offloading core finishes processing all packets in $\mathcal{Q}^{\text{old}}$ (Line 4), and then swaps the

[2] Race-free writing and reading by multiple cores are achieved by using lock-free queues implemented using circular buffers.

Algorithm 3 Scaling Operations of a Working Core

```
1:  procedure UPDATEOFFLOADRATIO
2:      Q^old ← self.Q^offloading
3:      self.Q^offloading ← createQueue()
4:      wait until Q^old = ∅
5:      self.offloadingCore.swapQueue(Q^old, self.Q^offloading)
6:      self.offloadRatioChange ← false
7:      RunSFC(self.Q^pending)
8:  procedure CHANGEOFFLOADINGCORE
9:      Q^old ← self.Q^offloading
10:     self.Q^offloading ← createQueue()
11:     wait until Q^old = ∅
12:     self.offloadingCore.removeQueue(Q^old)
13:     self.offloadingCore ← self.newOffloadingCore
14:     self.offloadingCore.addQueue(Q^offloading)
15: procedure CHANGESHARDASSIGNMENTS
16:     wait until Q^NIC cycles
17:     wait until Q^offloading cycles
18:     for each (s, w) ∈ self.migrations do
19:         shards[s].hold ← false
20:     self.migrations ← ∅
```

offloading queue with the new one (Line 5). During this wait, PROCESSPACKETS executes normally and stores packets that will be offloaded to the offloading core in self.$\mathcal{Q}^{\text{offloading}}$ (see Algorithm 2, Line 10). After the offloading core has finished clearing $\mathcal{Q}^{\text{old}}$, the working core processes any pending packets in self.$\mathcal{Q}^{\text{pending}}$ (Lines 6–7, also Lines 11–13 in Algorithm 2).

Changing offloading cores is equivalent, with two differences: self.offloadRatioChange is false and self.$\mathcal{Q}^{\text{pending}} = \varnothing$, as the offload ratio is updated before changing the offloading core (functions are called in order); and queues are removed and added to different cores (Lines 12–14).

When changing shard assignments, remember that the controller updates the RSS tables before signaling working cores (Algorithm 1, Lines 13–14). Before telling other cores that they can start processing packets for the migrated shards, the working core processes all packets it received from the network and offloaded to its offloading core (Lines 16–17). The working core then updates the other cores' states to resume processing of the migrated shards (Lines 18–20, see also Algorithm 2, Lines 2–4).

Offloading cores have a significantly simpler data plane (no pseudocode shown). An offloading core has one queue for each working core that offloads traffic to it. Offloading cores poll queues in round-robin fashion and process one batch of packets from each queue before proceeding to the next queue. The participation of offloading cores in migration operations is limited to informing working cores when a queue cycling operation completes and implementing the addQueue, swapQueue, and removeQueue operations.

3) Correctness Analysis: To show that *Dyssect* does not introduce deadlocks or packet reordering, we define the following terms and then formally state its guarantees.

- Flow $\mathcal{F}$ constitutes a sequence of packets with the same flow key ordered by their time of insertion in $\mathcal{Q}^{\text{NIC}}$.
- For any packet $p \in \mathcal{F}$, e_I^p and e_R^p represent the events corresponding to insertion of packet p in $\mathcal{Q}^{\text{NIC}}$ and final processing of the packet in RunSFC(p), respectively.
- The relation $\prec$ is defined on any two events e_i and e_j such that $e_i \prec e_j$ iff the timestamp associated with e_i is less than e_j. Relation $\prec$ satisfies transitivity but is neither reflexive nor symmetric.

Property 1. *Deadlock freedom*: For any packet p in any network flow $\mathcal{F}$, if e_I^p exists, then e_R^p also exists.

Property 2. *Packet ordering*: For any network flow $\mathcal{F}$, for any two packets $p_i, p_j \in \mathcal{F}$, if $e_I^{p_i} \prec e_I^{p_j}$, then $e_R^{p_i} \prec e_R^{p_j}$

Theorem II.1 (Correctness)**.** *Dyssect algorithm guarantees deadlock freedom and packet ordering.*

Proof. We informally outline the proof sketch. We show that *deadlock freedom* and *packet ordering* are guaranteed by construction.

a) Deadlock freedom: There are five locations that can block the processing of packets – Line 3 in Algorithm 2 and Lines 4, 11, 16 and 17 in Algorithm 3.

Because the processing of packets can be disabled by the controller (Line 11, Algorithm 1), packets may be held in $\mathcal{Q}^{\text{held}}$. However, the duration of this hold is until their processing is enabled by the old owner of the shard (Line 19, Algorithm 3). Further, only the controller can disable the processing of packets, preventing any cyclic dependencies among working cores.

The *waits* in Algorithm 3 also cannot block the processing of packets. This is because Lines 4, 11 and 17 are associated with waiting on the packets in $\mathcal{Q}^{\text{offloading}}$ being processed. By construction, the offloading core never blocks, thereby bounding these waits. The *wait* at Line 16 is also bounded because the loop at Line 5 in Algorithm 2 never blocks.

Hence, e_R^p exists for every e_I^p event.

b) Packet ordering: Packet ordering is trivially satisfied when packets are processed by the same core. Packets can be processed by different cores when the controller reassigns shards, offloading cores, or changes the offload ratio. In each case, we show that the definition still holds.

UPDATEOFFLOADRATIO: Consider any two packets, p, $q \in \mathcal{F}$ such that $e_I^p \prec e_I^q$. There are four possibilities based on the cores where the packets are processed.

- If p and q are processed by the working core, $e_R^p \prec e_R^q$ holds due to Line 15 in Algorithm 2.
- If p and q are processed by the offloading core, $e_R^p \prec e_R^q$ holds due to Line 10 in Algorithm 2, and Lines 4–5 in Algorithm 3.
- If p and q are processed by the offloading and working cores respectively, $e_R^p \prec e_R^q$ holds because of Lines 4 and 7 in Algorithm 3.
- If p and q are processed by the working and offloading cores respectively, $e_R^p \prec e_R^q$ holds because of Lines 10 and 15 in Algorithm 2, and Line 5 in Algorithm 3.

CHANGEOFFLOADINGCORE: For any two packets, p, $q \in \mathcal{F}$, if p is already in the old offloading core and q needs to be processed by the new offloading core, *Dyssect* guarantees $e_R^p \prec e_R^q$ because the new offloading core adds the new offloading queue at Lines 13–14 in Algorithm 3 which happens only after the old offloading core is emptied at Line 11.

Table I: Optimization Variables and Parameters.

DECISION VARIABLES	
$\boldsymbol{A}_{s,c}$	Assignment of shards to working cores; 1 if shard s is assigned to working core c, else 0
$\boldsymbol{O}_{c,k}$	Mapping of working cores to offloading cores; 1 if working core c offloads traffic to offloading core k, else 0
r_s	Ratio of shard s's traffic to offloading core
$n^{\text{w}}, n^{\text{o}}$	Number of working and offloading cores, respectively
DEPENDENT VARIABLES	
w_c	1 if c is an active working core, else 0
f_k	1 if k is an active offloading core, else 0
$u_c^{\text{w}}, u_k^{\text{o}}$	Utilization of working core c and offloading core k
INPUT PARAMETERS AND CONSTANTS	
α	Scale value for the multi-objective function
$r_s^{\max}$	Total fraction of shard s's traffic to offload
$u_{\max}^{\text{w}}, u_{\max}^{\text{o}}$	Maximum working and offloading core utilization
T	Load target for working and offloading cores
L_s	Core utilization required to process shard s's traffic
$\text{CA}^{\text{p}}, \text{CS}^{\text{p}}$	CV of priority packet interarrival times and proc. times
$\text{CA}^{\text{r}}, \text{CS}^{\text{r}}$	As above, for regular flow packets
$T^{\text{p}}, T^{\text{r}}$	Mean processing time for priority and regular packets
$\text{SLO}^{\text{p}}, \text{SLO}^{\text{r}}$	Max. queuing delay for priority and regular packets
$\boldsymbol{A}^{\text{old}}$	Previous assignment of shards to working cores
$\boldsymbol{O}^{\text{old}}$	Previous mapping of working cores to offloading cores

CHANGESHARDASSIGNMENTS: Packets in $\mathcal{F}$ will flow into the new working core with the changed assignment (Line 12 in Algorithm 1), but are not yet processed as the controller disables their processing at Line 11 of Algorithm 1. Processing is enabled only at Line 19 in Algorithm 3 after the packets in $\mathcal{Q}^{\text{offloading}}$ and $\mathcal{Q}^{\text{NIC}}$ are cycled in Lines 16–17.

Hence, $e_R^p \prec e_R^q$ is guaranteed. □

C. Flow Assignment Optimization

1) Long-timescale Optimization: The long-timescale optimization problem is shown in Table II. The objective function (Eq. 1) minimizes resource utilization, expressed as the number of active working and offloading cores. Equation 2 ensures that there is at least one active working core and limits the maximum to all available cores. Equation 3 enforces that each shard s is assigned one working core c.

Equation 4 ensures no shards are assigned to inactive working cores. The first half of Equation 5 enforces that a working core with offloaded flows has one active offloading core; the second half enforces that no offloading core is assigned to an inactive working core and that at most one offloading core is assigned to an active working core. Equation 6 enforces that an active working core has at least one shard assigned to it. Equation 7 ensures that no working core offloads traffic to an inactive offloading core. Equation 8 defines the relation between working and offloading cores; it requires that each offloading core is assigned to at least two working cores, to ensure that the offload core can be fully utilized. Equation 9 restricts the fraction of shard s's traffic offloaded to the offloading core to $r_s^{\max}$.

The assignment of shards to working cores $\boldsymbol{A}$, mapping of working cores to offloading cores $\boldsymbol{O}$, and the fraction of each shard s's traffic offloaded to offloading cores r_s are ultimately defined by the maximum utilization of each core that still satisfies SLO requirements. Equations 10 and 11 define the utilization of active working and active offloading cores, respectively, as a sum across all shards assigned or offloaded to that core, taking into consideration the fraction of each shard that is offloaded to an offloading core.

Table II: Long-timescale Optimization Problem.

$$\text{minimize} \quad \sum_{i \in C} (w_i + f_i), \quad \text{subject to} \tag{1}$$

$$1 \leqslant \sum_{i \in C} (w_i + f_i) \leqslant |C| \tag{2}$$

$$\sum_{c \in C} \boldsymbol{A}_{s,c} = 1 \qquad \forall s \in S \tag{3}$$

$$\boldsymbol{A}_{s,c} \leqslant w_c \qquad \forall s \in S, c \in C \tag{4}$$

$$r_s \boldsymbol{A}_{s,c} \leqslant \sum_{k \in C} \boldsymbol{O}_{c,k} \leqslant w_c \qquad \forall s \in S, c \in C \tag{5}$$

$$\sum_{s \in S} \boldsymbol{A}_{s,c} \geqslant 1 \qquad \forall c \in C \text{ with } w_c = 1 \tag{6}$$

$$\boldsymbol{O}_{c,k} \leqslant f_k \qquad \forall c \in C, k \in C \tag{7}$$

$$\sum_{c \in C} \boldsymbol{O}_{c,k} \geqslant 2 \qquad \forall k \in C \text{ with } f_k = 1 \tag{8}$$

$$0 \leqslant r_s \leqslant r_s^{\max} \leqslant 1 \qquad \forall s \in S \tag{9}$$

$$u_c^{\text{w}} = w_c \sum_{s \in S} L_s \boldsymbol{A}_{s,c} (1 - r_s) \qquad \forall c \in C \tag{10}$$

$$u_k^{\text{o}} = f_k \sum_{c \in C} \left(\boldsymbol{O}_{c,k} \sum_{s \in S} L_s \boldsymbol{A}_{s,c} r_s \right) \qquad \forall k \in C \tag{11}$$

$$\left(\frac{u_c^{\text{w}}}{1 - u_c^{\text{w}}} \right) \frac{(\text{CA}^{\text{p}})^2 + (\text{CS}^{\text{p}})^2}{2} T^{\text{p}} \leqslant \text{SLO}^{\text{p}} \qquad \forall c \in C \tag{12}$$

$$\left(\frac{u_k^{\text{o}}}{1 - u_k^{\text{o}}} \right) \frac{(\text{CA}^{\text{r}})^2 + (\text{CS}^{\text{r}})^2}{2} T^{\text{r}} \leqslant \text{SLO}^{\text{r}} \qquad \forall k \in C \tag{13}$$

$$u_c^{\text{w}} \leqslant u_{\max}^{\text{w}} \qquad \forall c \in C \tag{14}$$

$$u_k^{\text{o}} \leqslant u_{\max}^{\text{o}} \qquad \forall k \in C \tag{15}$$

We model each core as a queue and limit core utilization such that packet wait (queuing) times in the system satisfy a target SLO. We make no assumptions on the packet arrival process or the NFV processing times, and model each core as a G/G/1 queue.[3] We compute the mean packet wait time as a function of core utilization using Kingman's formula [23]. Equations 12 and 13 capture the SLO requirement on the mean packet wait times.[4] The coefficients of variation of packet interarrival times and packet processing times (CA and CS) are estimated offline prior to model execution. The mean packet processing times (T^{w} and T^{e}) are computed as a function of the number of flows assigned to the core using a piecewise linear function estimated offline. If packet interarrival times and NFV processing times have low variation and can be modelled by an M/M/1 queue, then Equations 12 and 13 can be changed to use the exact distribution of packet wait times instead of Kingman's formula for more precise SLO bounds.

Core utilizations are bounded by Equations 14 and 15, which limit utilization to the thresholds $u_{\max}^{\text{w}}$ and $u_{\max}^{\text{o}}$. These thresholds can be used to prevent solutions from running cores too close to 100% utilization. We add this as a safety measure

[3] Although a G/G/1/k queue would capture the finite size of packet buffers, a G/G/1 queue has *higher* wait times for the same utilization. This makes our proposed model *conservative* compared to a G/G/1/k queue.

[4] The total packet processing time also includes polling time and the NFV processing time. SLOs can be defined on the total packet processing time by subtracting the polling time and NFV processing time prior to setting the mean packet wait time used in Equations 12 and 13.

Table III: Short-timescale Optimization Problem.

$$\text{minimize} \quad F(\boldsymbol{A} - \boldsymbol{A}^{\text{old}}) + F(\boldsymbol{O} - \boldsymbol{O}^{\text{old}}), \tag{16}$$

subject to Equations 2–13, and

$$u_c^{\text{w}} < 1 \qquad \forall c \in C \tag{17}$$

$$u_k^{\text{o}} < 1 \qquad \forall k \in C \tag{18}$$

$$\sum_{c \in C} w_c = n^{\text{w}} \tag{19}$$

$$\sum_{k \in C} f_k = n^{\text{o}} \tag{20}$$

to allow some spare processing capacity in case of traffic bursts and provide flexibility to the short-timescale optimization.

2) Short-timescale Optimization: The short-timescale optimization, shown in Table III, receives as input $\boldsymbol{A}^{\text{old}}$, $\boldsymbol{O}^{\text{old}}$, n^{w}, and n^{o}, the previous shard assignments, previous offloading core associations, the number of working cores, and the number of offloading cores, respectively. The short-timescale model does *not* change the number of active working and offloading cores (Eqs. 19 and 20). It computes $\boldsymbol{A}$, $\boldsymbol{O}$, and r_s to minimize the number of shard migrations and offloading core reassociations (Eq. 16) subject to SLO constraints. Function F computes the number of shard migrations and offloading core reassociations from the number of positive entries in the difference matrices $\boldsymbol{A} - \boldsymbol{A}^{\text{old}}$ and $\boldsymbol{O} - \boldsymbol{O}^{\text{old}}$, respectively.

The short-term optimization includes all restrictions in the long-timescale optimization (not shown), except restrictions given by Equations 14 and 15. These constraints are replaced by Equations 17 and 18, allowing the short-timescale optimization to use spare core processing capacity in case of traffic bursts. However, the core utilization is still limited by the SLO constraints in Equations 12 and 13.

III. Implementation

We implemented our *Dyssect* prototype using user-level BESS [16] modules with DPDK for fast packet processing. Similar to Click [24], BESS provides a flexible interface to develop different user-level modules to configure data and control paths. We extend the BESS implementation to allow the controller to dynamically change the number of cores during execution.

The *working core module* retrieves packets from NIC queues, fetches flow states, inserts the reference to the flow entry into the packet metadata, updates shard statistics, and processes packets that are not offloaded. The working core module also implements Algorithms 2 and 3. The *offloading core module* fetches and processes packets from the queues that connect it to working cores. Finally, the *controller module* collects statistics from the data path, executes the optimization models to compute shard assignment and offload ratios, updates the NIC indirection table, and signals the modules in the data path to transfer shards between working cores and flows to offloading cores. The controller uses Gurobi [17] for solving the optimization models and implements several system-level optimizations, such as allocating data structures in DPDK huge pages and observing cache alignment to avoid performance penalties in concurrent accesses to cache lines.

IV. Evaluation

We evaluate *Dyssect* with two use cases to show how fine-grained flow management reduces tail latency and increases throughput. We use a third use case to show that state disaggregation allows offloading some functionalities to a SmartNIC. We show the peroformance impacts in the function offloaded to the NIC and processor metrics, such as L1 and L2 cache misses and instructions per cycle.

Our testbed comprises two servers connected in a classical configuration of Traffic Generator and Device-Under-Test. Each server has two Intel(R) Xeon(R) Silver 4114 (10-cores @2.20 GHz), 128 GB of RAM, and a dual-port Netronome NFP-4000 40 GbE NIC. We disable hyperthreading, C-states and CPU frequency scaling, Turbo Boost, and uncore power scaling to reduce measurement variance and allow for reproducibility of our results. We use only one processor to avoid crossing the Ultra Path Interconnect that connects the two processors, isolate eight cores from the Linux scheduler for packet processing, and reserve 64 huge pages (1 GB each) for DPDK. We use the Linux's perf stat command for collecting processor statistics.

For all experiments, we set the warm-up time to $30\,s$ and the execution time to $60\,s$, and report results with 95% confidence intervals from 10 independent runs. We use real and synthetic traffic traces. The real trace is from CAIDA [25] and contains $1,835,436$ TCP flows with an average packet size of 1001 bytes. We also create synthetic traces of 1M flows with a Zipf flow size distribution by varying the exponent α. We set the packet size to 1001 bytes, the average in the CAIDA trace.

A. Use Case I: Traffic Class Prioritization

In this use case, *Dyssect* runs the optimization models (Tables II and III), with the long- and short-timescales set to $\tau = 1\,s$ and $\epsilon = 100\,ms$, respectively. We use traffic classes with two different priorities specified by the SLOs in Equations 12 and 13. The high-priority flows are always processed by working cores while working or offloading cores can process low-priority flows depending on the model output.

We use the CAIDA trace, which has a very skewed distribution with a few large and long-lived flows concentrating the bulk of the traffic load. To be conservative, we consider short flows and 0.1% of the traffic as high priority, which results in $753,725$ high-priority flows. A service chain with two network functions processes all the packets regardless of the core used. The first network function is a NAT that updates the source IP address and TCP port and recomputes both IP and TCP checksums. The second network function is an IDS that iterates over the TCP payload to represent CPU-intensive functions. We scale the traffic by changing inter-packet gaps to simulate throughputs from ~2.5 to ~22 Gbps, but we keep the same flows and packet ordering.

Figure 4 compares the performance results of *Dyssect* with the results of RSS++. *Dyssect* can process the same or higher traffic load as RSS++ (Figure 4a) using the same amount or fewer cores (Figure 4b), without causing traffic disruptions while migrating shards and flows (steps).

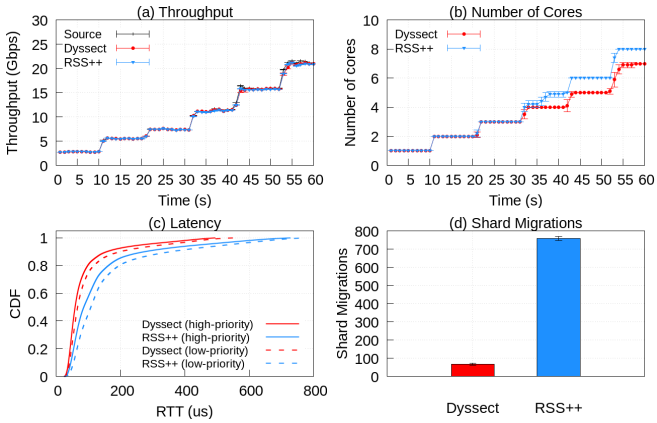

Figure 4: Performance results for the Use Case I using the CAIDA trace, with 95% confidence intervals.

Dyssect can prioritize some flows to have lower latency, something that RSS++ cannot do, as it handles traffic at the granularity of shards. Figure 4c shows the 99.5% tail latencies. Compared to RSS++, *Dyssect* reduces the tail latency of the high-priority traffic (solid lines) from 728.07 μs to 494.77 μs (a 32% reduction), without compromising the low priority traffic (dashed lines). Figure 4d shows that *Dyssect* achieves its results performing less than 9% shard migrations than RSS++ (67.4 *vs.* 758.3). This result is significant, as some NICs require a large amount of time to update their indirection tables. We evaluated the NICs BCM57416 NetXtreme-E Dual-Media (10G), Netronome NFP-4000 (40G), and Mellanox ConnectX-4 (100G) and obtained the following update times: 228.05 ($\pm$1.08) μs, 1073.67 ($\pm$2.97) μs, and 359,226.71 ($\pm$9,491.81) μs, respectively. Each update in the Mellanox NIC takes more than 350 ms as it has to restart the NIC.

B. Use Case II: Alternate Optimization Targets

The modularity and flexibility of *Dyssect* allow for different optimization models whose objective functions may maximize other performance metrics. We compare the performance of *Dyssect* using an optimization model that balances load across cores against RSS++, which also attempts to balance the load.

Table IV describes an optimization model that minimizes the quadratic difference between a target value T and the utilization of a core for all working and offloading cores. The third term of the objective function is Equation 16 scaled by a (small) factor α to reduce the number of shard migrations and offloading core reassociations when the load is balanced.

Table IV: Load Balance Optimization Problem.

$$\text{minimize} \quad \sum_{c \in C} (u_c^{\text{w}} - T)^2 + \sum_{k \in C} (u_k^{\text{o}} - T)^2 + \alpha(\text{Eq. 16}), \qquad (21)$$

subject to Equations 2 – 11 and Equations 19 – 20

We quantify the impact the load balancing model and the state disaggregation have on the performance in terms of the average number of shard migrations and throughput. For reference, we compare against the standard RSS load balancing scheme and RSS++. We also compare against a modified version of *Dyssect* that employs state disaggregation (labeled "State Disag") but uses RSS load balancing instead of an optimization model; our goal is to isolate the performance gains afforded by state disaggregation and the optimization.

For the traffic load, we send packets at 36 Gbps (90% of the link capacity) from synthetic traces generated using Zipf distributions with α varying from 0.7 to 1.1 in increments of 0.1. We use the same service chain as in Use Case 1 (§IV-A) and eight cores for packet processing for all four approaches (RSS, RSS++, State Disag, and *Dyssect*). Since *Dyssect* allows the transfer of flows to offloading cores, we use one as offloading core and seven as working cores. In this Use Case, we do not use the long-term model of Table II to minimize the number of cores, as we want to measure the throughput the system can achieve with a fixed number of cores under heavy load. We use 128 shards (default value of the NIC) for RSS++ and 16 for RSS and *Dyssect*. RSS++ does not accept a smaller number, and 128 is better for improving its load balancing, as it migrates only shards and not individual flows. We also quantify the performance when the batch size changes using four different sizes $(1, 4, 16, 32)$.

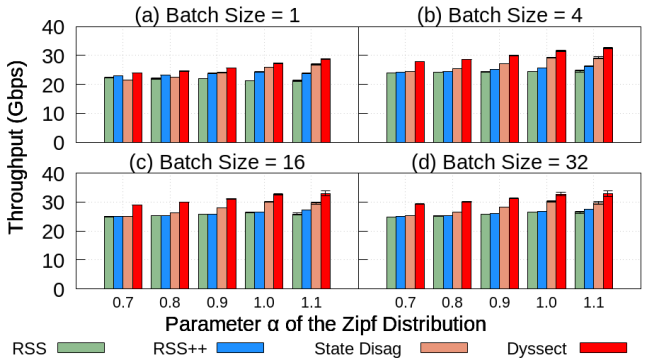

Figure 5: Throughput of *Dyssect* compared with RSS and RSS++ for different batch sizes and α parameters.

Figure 5 shows the throughput of the four approaches under different batch sizes and α values. As we can see, *Dyssect* outperforms all the other three approaches in all scenarios. We can also see that the performance gains come from both state disaggregation and the optimization model, with the state disaggregation having more impact on more skewed flow size distributions (higher α). The performance gains of *Dyssect* over RSS++ varied from 4.11% ($\alpha = 0.7$ and batch size 1) to 19.36% ($\alpha = 1.1$ and batch size 4), and from 6.67% to 25.97% in comparison with RSS. Note that the performance gains can be even higher in longer service chains, as *Dyssect* would save lookups in the other network functions. Table V shows the average number of shard migrations performed by *Dyssect* and RSS++. In the interest of space, we show only the results for α equal to 0.7, 0.9, and 1.1. *Dyssect* achieves higher throughput with fewer shard migrations in all scenarios.

C. Use Case III: SmartNIC Offloading

Our third use case shows another benefit of state disaggregation. We offload to a SmartNIC the lookup function for a

Table V: Average shard migrations of *Dyssect* and RSS++, varying the batch size (B) and α of the Zipf distribution.

Dyssect	$\alpha = 0.7$	$\alpha = 0.9$	$\alpha = 1.1$
B = 1	0.00 ± 0.00	0.00 ± 0.00	3.60 ± 0.57
B = 4	0.20 ± 0.37	0.00 ± 0.00	4.80 ± 1.23
B = 16	0.30 ± 0.56	0.10 ± 0.19	4.00 ± 0.83
B = 32	0.60 ± 0.74	0.40 ± 0.50	4.30 ± 1.47

RSS++	$\alpha = 0.7$	$\alpha = 0.9$	$\alpha = 1.1$
B = 1	15.50 ± 2.57	42.90 ± 6.87	44.20 ± 3.29
B = 4	47.20 ± 5.33	76.10 ± 6.92	75.00 ± 6.37
B = 16	42.10 ± 6.76	124.00 ± 13.68	264.40 ± 13.15
B = 32	294.40 ± 63.18	739.30 ± 137.09	1058.30 ± 93.19

subset of flows defined by the controller.[5] When receiving the first packet of a flow to be offloaded to the NIC, the working core inserts a rule in the NIC matching the flow 5-tuple and associates the rule with the address of the newly created flow entry. For subsequent packets, the NIC performs the lookup and inserts the address into the packet metadata. On receiving a packet, the working core skips the lookup if the metadata already contains an address. By offloading large-volume flows to the NIC [26]–[30] we can decrease packet processing times and free CPU cycles, allowing higher throughput.

To evaluate the performance gains of offloading the lookup function to a programmable NIC (Netronome NFP-4000), we use a single core to process traffic from our synthetic trace with Zipf parameter $\alpha = 1.1$. A single core allows us to collect precise statistics of (L1 and L2) cache misses and instructions per cycle without interference from other cores. The controller selects 1% of the flows with the largest volumes to be offloaded to the NIC ($10,000$ flows). We vary the packet size and set the load to 1 Gbps to collect performance metrics without overloading the single core.

Figure 6a shows that average lookup times decrease significantly when *Dyssect* uses a SmartNIC, as expected. However, the takeaway of this experiment is that offloading reduces not only the lookup time but also improves the CPU performance metrics, such as instructions per cycle (Figure 6b) and miss rates for L1 (Figure 6c) and L2 (Figure 6d) caches. Note that we show the average time of all lookups, including the ones performed on the NIC and the others performed on the CPU.

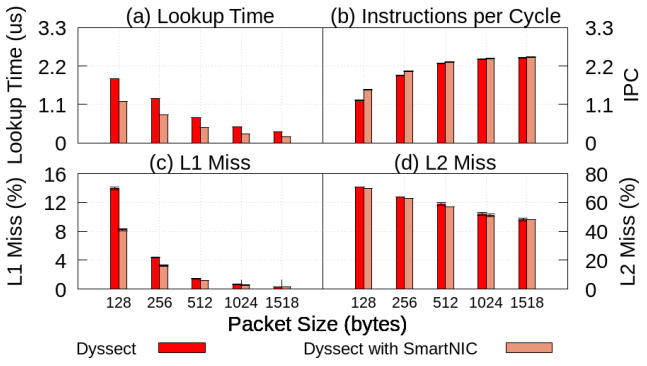

Figure 6: Average lookup times, instructions per cycle, L1 and L2 cache-miss rates of *Dyssect* with and without offloading to the SmartNIC.

[5]Handling each flow takes one rule in the Netronome NFP-4000 NIC, which supports about 12K rules in its fastest memory.

V. Related Work

Intra-Server Hardware Dispatcher. Recent efforts [9], [10], [31], [32] use RSS to direct packets to different cores and distribute the load, inheriting the limitations we discuss in §I. FlowDirector systems [33], [34] direct flows to specific cores, but only a few NICs support this functionality, and the ones that support can handle a small number of flows. *Dyssect* works on commodity NICs, redistributes traffic load at the flow level, and does not limit the number of flows that can be redirected to specific cores.

Intra-Server Software Dispatcher. Several works use a single core for fetching packets from the NIC and dispatching them to other cores [7], [11], [12], [35]. In *Dyssect*, multiple working cores send only a fraction of incoming packets to offloading cores, avoiding scalability limitations of having a single core dispatching all the packets. Also, these systems do not specifically target network functions and, therefore, do not provide strong guarantees of packet ordering as *Dyssect* does.

Inter-Server Load Balancing. Other works allow load distribution to multiple servers and provide state migration between servers [2], [36]–[39]. They are complementary to our work and can use *Dyssect* for intra-server packet processing.

System-Level Optimizations. Some efforts improve the processing of a service chain by exploiting hardware features [40], [41], performing code optimizations [42], or eliminating redundant computations [43] in NFs on a service chain inside and across servers. These efforts are orthogonal and complementary to our work, but we can incorporate some of these optimizations in a future version of *Dyssect*.

Hardware-Based Systems. A recent trend for accelerating network functions is the use of programmable NICs. Several recent systems [44]–[50] offload computation to programmable NICs, FPGA-based SmartNICs, or P4 switches. In §IV-C, we show an example of how *Dyssect* can use programmable NICs, but the rich features available on these devices open new possibilities for future optimizations.

VI. Conclusion

This paper presents new results on how sharding can have significant impacts on the performance of stateful network functions. *Dyssect* disaggregates states from network functions, which allows finer-grained management of network traffic. *Dyssect* can migrate shards and flows between cores without resorting to locks or reordering packets during migrations. Our experimental evaluations with real and synthetic traffic show that *Dyssect* increases throughput up to 19.36% and reduces tail latency up to 32% when compared with RSS++, the state-of-the-art for load-balancing within a server. We provide access to our code at *https://doi.org/10.5281/zenodo.5815715*.

Acknowledgements

This research is part of the INCT of the Future Internet for Smart Cities funded by CNPq proc. 465446/2014-0, CAPES Finance Code 001, and FAPESP procs. 14/50937-1 and 15/24485-9. This work was also supported by the Brazilian National Research and Educational Network (RNP) conv. 2956, and FAPESP procs. 2018/23085-5 and 2020/05183-0.

REFERENCES

[1] P. Zheng, W. Feng, A. Narayanan, and Z.-L. Zhang, "NFV Performance Profiling on Multi-core Servers," in *Proc. of IFIP Networking Conference (Networking)*, 2020.

[2] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic Scaling of Stateful Network Functions," in *Proc. of USENIX NSDI*, 2018, p. 299–312.

[3] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-Scalable Locks Are Dangerous," in *Proc. of the Linux Symposium*, 2012, pp. 119–130.

[4] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable Low Latency for Data Center Applications," in *Proc. of ACM SoCC*, 2012.

[5] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek, "CPHASH: A Cache-Partitioned Hash Table," in *Proc. of ACM SIGPLAN*, 2012, p. 319–320.

[6] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A Holistic Approach to Fast in-Memory Key-Value Storage," in *Proc. of USENIX NSDI*, 2014, p. 429–444.

[7] D. Didona and W. Zwaenepoel, "Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores," in *Proc. of USENIX NSDI*, 2019, pp. 79–94.

[8] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using RDMA Efficiently for Key-Value Services," in *Proc. of ACM SIGCOMM*, 2014, p. 295–306.

[9] H. Sadok, M. E. M. Campista, and L. H. M. K. Costa, "A Case for Spraying Packets in Software Middleboxes," in *Proc. of ACM HotNets*, 2018, p. 127–133.

[10] T. Barbette, G. P. Katsikas, G. Q. Maguire, and D. Kostić, "RSS++: Load and State-Aware Receive Side Scaling," in *Proc. of ACM CoNEXT*, 2019, p. 318–333.

[11] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive Scheduling for usecond-Scale Tail Latency," in *Proc. of USENIX NSDI*, 2019, p. 345–359.

[12] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads," in *Proc. of USENIX NSDI*, 2019, p. 361–377.

[13] "Mellanox ConnectX 5 EN," available at https://www.mellanox.com/products/ethernet-adapters/connectx-5-en.

[14] L. Saino, I. Psaras, and G. Pavlou, "Understanding Sharded Caching Systems," in *Proc. of IEEE INFOCOM*, 2016, p. 1–9.

[15] W. A. Wulf and S. A. McKee, "Hitting the Memory Wall: Implications of the Obvious," *ACM SIGARCH Comput. Archit. News*, vol. 23, no. 1, p. 20–24, 1995.

[16] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A Software NIC to Augment Hardware," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-155, May 2015.

[17] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2021. [Online]. Available: https://www.gurobi.com

[18] T. Barbette, C. Soldani, and L. Mathy, "Fast Userspace Packet Processing," in *Proc. of ACM/IEEE ANCS*, 2015, p. 5–16.

[19] L. Rizzo and G. Lettieri, "VALE, a Switched Ethernet for Virtual Machines," in *Proc. of ACM CoNEXT*, 2012, p. 61–72.

[20] S. Laki, D. Horpácsi, P. Vörös, R. Kitlei, D. Leskó, and M. Tejfel, "High Speed Packet Forwarding Compiled from Protocol Independent Data Plane Specifications," in *Proc. of ACM SIGCOMM*, 2016, p. 629–630.

[21] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. Maguire, "Metron: NFV Service Chains at the True Speed of the Underlying Hardware," in *Proc. of USENIX NSDI*, 2018, p. 171–186.

[22] Y. Wang, S. Gobriel, R. Wang, T.-Y. C. Tai, and C. Dumitrescu, "Hash Table Design and Optimization for Software Virtual Switches," in *Proc. of ACM KBNets*, 2018, p. 22–28.

[23] J. F. C. Kingman, "The Single Server Queue in Heavy Traffic," *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 57, no. 4, pp. 902–904, 1961.

[24] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, p. 263–297, 2000.

[25] CAIDA, "CAIDA Data Monitors," 2021. [Online]. Available: https://www.caida.org/catalog/datasets/monitors/

[26] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A Simple Algorithm for Finding Frequent Elements in Streams and Bags," *ACM Trans. Database Syst.*, vol. 28, no. 1, p. 51–55, 2003.

[27] R. Ben Basat, X. Chen, G. Einziger, R. Friedman, and Y. Kassner, "Randomized Admission Policy for Efficient Top-k, Frequency, and Volume Estimation," *IEEE/ACM Trans. Netw.*, p. 1432–1445, 2019.

[28] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting," in *Proc. of ACM SIGCOMM*, 2002, p. 323–336.

[29] T. Li, S. Chen, and Y. Ling, "Per-Flow Traffic Measurement through Randomized Counter Sharing," *IEEE/ACM Trans. Netw.*, vol. 20, no. 5, p. 1622–1634, 2012.

[30] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon," in *Proc. of ACM SIGCOMM*, 2016, p. 101–114.

[31] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A Protected Dataplane Operating System for High Throughput and Low Latency," in *Proc. of USENIX OSDI*, 2014, p. 49–65.

[32] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The Operating System is the Control Plane," in *Proc. of USENIX OSDI*, 2014, p. 1–16.

[33] Intel, "Introduction to Intel Ethernet Flow Director and Memcached Performance," https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf, 2016.

[34] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris, "Improving Network Connection Locality on Multicore Systems," in *Proc. of ACM EuroSys*, 2012, p. 337–350.

[35] G. Prekas, M. Kogias, and E. Bugnion, "ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks," in *Proc. of ACM SOSP*, 2017, p. 325–341.

[36] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless Network Functions: Breaking the Tight Coupling of State and Processing," in *Proc. of USENIX NSDI*, 2017, p. 97–112.

[37] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "OpenNF: Enabling Innovation in Network Function Control," in *Proc. of ACM SIGCOMM*, 2014, p. 163–174.

[38] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/Merge: System Support for Elastic Execution in Virtual Middleboxes," in *Proc. of USENIX NSDI*, 2013, p. 227–240.

[39] A. Oeldemann, F. Biersack, T. Wild, and A. Herkersdorf, "Inter-Server RSS: Extending Receive Side Scaling for Inter-Server Workload Distribution," in *Proc. of Euromicro International Conference on PDP*, 2020, pp. 46–53.

[40] A. Farshin, A. Roozbeh, G. Q. Maguire, and D. Kostić, "Make the Most out of Last Level Cache in Intel Processors," in *Proc. of ACM EuroSys*, 2019.

[41] A. Farshin, A. Roozbeh, G. Q. M. Jr., and D. Kostić, "Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks," in *Proc. of USENIX ATC*, 2020, pp. 673–689.

[42] A. Farshin, T. Barbette, A. Roozbeh, G. Q. Maguire Jr., and D. Kostić, "PacketMill: Toward per-Core 100-Gbps Networking," in *Proc. of ACM ASPLOS*, 2021, p. 1–17.

[43] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions," in *Proc. of ACM SIGCOMM*, 2016, p. 511–524.

[44] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein, "NICA: An Infrastructure for Inline Acceleration of Network Applications," in *Proc. of USENIX ATC*, 2019, p. 345–361.

[45] D. Firestone *et al.*, "Azure Accelerated Networking: SmartNICs in the Public Cloud," in *Proc. of USENIX NSDI*, 2018, p. 51–64.

[46] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy, "High Performance Packet Processing with FlexNIC," in *Proc. of ACM ASPLOS*, 2016, p. 67–81.

[47] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta, "Offloading Distributed Applications onto SmartNICs Using iPipe," in *Proc. of ACM SIGCOMM*, 2019, p. 318–333.

[48] R. D. G. Pacífico, L. F. S. Duarte, M. S. Castanho, L. F. M. Vieira, J. A. Nacif, and M. A. M. Vieira, "Application Layer Packet Classifier in Hardware," in *Proc. of IFIP/IEEE Symposium on Integrated Network and Service Management*, 2021.

[49] B. Li *et al.*, "ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware," in *Proc. of ACM SIGCOMM*, 2016, p. 1–14.

[50] J. Yen, J. Wang, S. Supittayapornpong, M. A. M. Vieira, R. Govindan, and B. Raghavan, "Meeting SLOs in Cross-Platform NFV," in *Proc. of ACM CoNEXT*, 2020, p. 509–523.