Scaling Stateful Network Services on Multicore Architectures

Fabrício B. Carvalho[1], **Ronaldo A. Ferreira**[2]

[1] FAENG – Universidade Federal de Mato Grosso (UFMT)

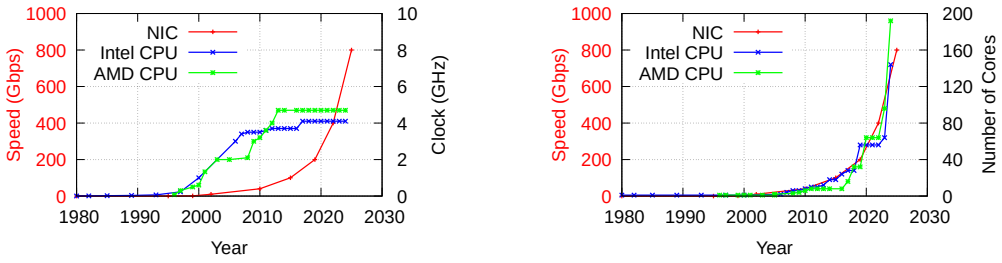
[2]FACOM – Universidade Federal de Mato Grosso do Sul (UFMS)

Abstract. *This thesis investigates the effective scheduling of TCP stacks alongside applications on multicore architectures, focusing on the trade-offs in allocating workers for both TCP and application processing. It explores the interplay between stateful network protocols with strong guarantees and the challenges of scheduling such protocols alongside multicore applications. To allow fair comparisons, we design and implement Demieagle, a benchmark framework that allows the execution of "apples-to-apples" experiments to uncover the trade-offs of different multicore scheduling policies and architectures.*
We also address the complexity of scaling stateful network functions, which require per-packet state updates. During a scaling operation, workers need to synchronize access to a shared state to avoid race conditions and to guarantee that network functions 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. To address these challenges, we design, implement, and evaluate Dyssect, a system that enables dynamic scaling of stateful network functions by disaggregating their states. Dyssect's state disaggregation allows the offloading of stateful network functions to programmable NICs and makes it easier to explore hardware-software trade-offs that better suit specific network functions and traffic loads. Our experimental evaluation shows that Dyssect reduces tail latency up to 32.04% *and increases throughput up to* 19.36% *compared to state-of-the-art competing solutions.*

1. Introduction

In the last decade, network speeds have grown exponentially while CPU speeds have remained stagnant. Figure 1a illustrates this mismatch and shows that the network speed has increased at least four times while the maximum CPU clock of the most popular processors used in datacenter servers did not increase at all. On the other hand, Figure 1b shows that the number of cores in a processor has increased significantly to compensate for the lack of higher clocks. As a result, kernel-based networking stacks are increasingly unaffordable. Recent work has explored user-level networking within the context of kernel-bypass operating systems [Peter et al. 2014, Zhang et al. 2021] and network functions [Carvalho et al. 2022, Barbette et al. 2019]. Since soon-to-be-released 800 Gbps NICs (Network Interface Cards) can deliver an MTU-size packet every 15 nanoseconds, scalable multicore network stacks and applications, such as network functions, will be critical to keeping pace with even faster networks.

Previous research has explored multicore scheduling of application workloads in detail [Ousterhout et al. 2019, Demoulin et al. 2021, Fried et al. 2020]. While these systems explored a range of multicore designs for managing application work queues,



(a) NIC speed *vs.* Proc. clock. **(b) NIC speed *vs.* Proc. cores.**

Figure 1. Evolution of NICs and processors over the years.

they each used a fixed design for scheduling the networking stack. For example, shared-nothing systems [Zhang et al. 2021, Peter et al. 2014] use per-core work queues and *Receive-Side Scaling* (RSS) to schedule both application work and network protocol processing onto the same core to reduce latency and synchronization. In contrast, some systems separate network processing and application work on distinct cores for various reasons, including fine-grained control over application work scheduling, extracting higher performance gains from batching and vectorization, or privilege-separation.

1.1. Stateful Transport Protocol

In this work, we argue that the network protocol stack should be treated as a service that is as important to multicore scheduling as the application. We focus on TCP stacks, which are crucial to the adoption of kernel-bypass systems in datacenters. TCP is a strict requirement for interoperability with legacy applications where the client-side networking stack cannot be changed. Also, an increasing number of applications rely on distributed services and require a microsecond-scale TCP stack to guarantee service-level objectives (SLOs).

Given that many microsecond-scale applications are relatively simple, TCP processing can consume many CPU cycles relative to the application while being notoriously difficult to schedule on multicore architectures because of its per-connection state. At microsecond scales, the multicore architecture of the network stack and application dramatically impacts performance, especially in *relationship* to each other. Slight differences in architecture have a large relative impact on performance, especially at the tail. Since previous systems either do not support all the features in TCP or do not evaluate the impact of network scheduling on performance, it is impossible to compare how their proposed scheduling policies would fare in the real world.

To allow fair comparisons, we design and implement *Demieagle*, a benchmark framework that includes *(i)* a flexible, kernel-bypassing, microsecond-scale TCP stack that schedules network processing and application requests with different multicore architectures and *(ii)* a benchmark suite with workloads across a range of characteristics that stress different trade-offs in multicore scheduling. *Demieagle* allows "apples-to-apples" experiments to uncover the trade-offs of multicore scheduling policies and architectures.

1.2. Stateful Network Functions

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) when the traffic load increases. NFs can be composed to provide network services in what is known as a service chain.

The vast majority of network functions are stateful and may require state updates on a per-packet basis. During a scaling operation, workers (*i.e.,* CPU cores or hyperthreads) need to synchronize access to a shared state to avoid race conditions and to guarantee that NFs process packets in arrival order. Several research proposals use state *sharding* to avoid race conditions. 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* worker to avoid concurrent accesses. High-speed NICs can hash incoming packets and place them in per-worker queues specified in a RSS indirection table. Workers poll their queues to receive and process packets for their shards at high throughput.

A recent effort proposes dynamic reassignments of shards to balance the load between workers [Barbette et al. 2019]. While this approach improves performance compared to RSS, it is far from solving the problem. For instance, a shard might have multiple large-volume flows that a single worker cannot handle, but unfortunately the system cannot allocate more workers to handle the load, as all the flows in a shard are assigned to a single worker. Moreover, packets from high-priority flows might experience higher delays in a queue behind packets from large-volume flows.

To address these challenges, we design, implement, and evaluate *Dyssect*, a system that enables dynamic scaling of stateful NFs by disaggregating their states. By carefully coordinating actions between workers and a central controller, *Dyssect* migrates shards and flows between workers for load balancing or traffic prioritization without using locks or reordering packets. Also, *Dyssect*'s state disaggregation allows the offloading of stateful network functions to programmable NICs and makes it easier to explore hardware-software trade-offs that better suit specific network functions and traffic loads.

2. Problem Statement and Contributions

This work investigates the challenges of scaling TCP network stacks and stateful network functions across multiple workers (*i.e.,* CPU cores or hyperthreads) while maintaining consistent state updates. We identify two critical issues: *(i)* the need for per-packet state updates, which often involve synchronization mechanisms that can degrade performance, and *(ii)* workload imbalances may overload some workers while leaving others underutilized, reducing the efficiency and potential benefits of multicore architectures. Therefore, the problem statement of this thesis can be summarized as follows:

Problem Statement: *Efficiently updating state on a per-packet basis while dynamically balancing the load between workers poses significant challenges to scaling stateful network services on multicore architectures.*

Several research efforts discuss the challenges a system must overcome to scale stateful network services on multicore architectures [Barbette et al. 2019, Jeong et al. 2014, Kaufmann et al. 2019]. For instance, depending on the service-time distribution of an application, the network stack may become a bottleneck and require more workers than the application, so determining the best assignment of workers to the application and network stack is challenging. Also, scaling may require synchronization mechanisms or state migration between workers to prevent race conditions, which may degrade performance.

Scaling should also preserve flow-to-worker affinity to avoid packet reordering and inter-core communication, which degrade overall performance. Packet reordering is particularly damaging to TCP performance, as it blocks the sending window and may reduce the congestion window, leading to lower goodput. Also, packet reordering can harm stateful network functions that rely on packets arriving in order, resulting in decreased performance or even service disruption when packets arrive out of order.

We overcome these challenges by addressing the following research questions:

Research Question 1: *How do worker assignments to network stack and application influence the response latency in kernel-bypass systems?*

We evaluate the trade-offs involved in assigning workers to the network stack and application [Carvalho et al. 2025]. We define the design space in three dimensions that include worker assignment to different stages of packet processing, classify existing systems in this design space, implement models that cover the main points in the design space in a framework with a fully-featured TCP stack, and evaluate combinations of worker assignment, queueing model, and worker placement under different workloads.

Research Question 2: *How does redistributing the load across workers impact the performance of stateful network functions?*

We address the second research question by presenting *Dyssect*, a system that disaggregates state from NFs using a data structure that allows a packet to carry a reference to its flow state [Carvalho et al. 2022, Carvalho et al. 2024]. This mechanism is crucial for allowing fine-grained migrations of individual flows between workers to achieve a more even load distribution without introducing race conditions. *Dyssect* prevents packet reordering, improves throughput, and reduces latency compared to competing solutions.

Research Question 3: *How can programmable NICs impact the performance of stateful network functions on multicore architectures?*

We explore further the idea of state disaggregation by offloading stateful network functions, or parts of them, onto programmable NICs, aiming to further optimize network performance [Carvalho et al. 2022,Carvalho et al. 2024]. This offloading strategy enables us to investigate hardware and software trade-offs that are specific to network functions.

2.1. Main Contributions

Our thesis contributes to effectively scaling stateful network services on multicore architectures in three aspects [Carvalho 2024]. First, we systematically study the trade-offs in multicore architectures for microsecond-scale networking stacks and applications, focusing on TCP for interoperability with legacy systems [Carvalho et al. 2025]. To this end, we design *Demieagle*, a benchmark framework to evaluate multicore designs through a flexible, kernel-bypassing TCP stack and scheduler. *Demieagle* also includes a workload generator that exercises our multicore classification system by changing workload parameters that are directly affected by the architecture.

Second, we design a system called *Dyssect*, which scales stateful network functions and enhances NFV performance by allowing lock-free state migration operations to balance the load, prioritize traffic, and optimize resource usage [Carvalho et al. 2022,Carvalho et al. 2024]. *Dyssect* uses RSS and sharding to steer packets and transfer shards between cores, avoiding single-core bottlenecks and disaggregates state from network

Table 1. Classification of existing systems in the design space.

System	Worker Assignment		Queueing Model			Spatial Scheduling		
	Inline	Dispatcher	cFCFS	dFCFS	WS	HT	SN	MN
Demikernel [Zhang et al. 2021]	✓		✓			✗	–	✗
Arrakis [Peter et al. 2014]	✓			✓		–	–	✗
Shenango [Ousterhout et al. 2019]	✓				✓	✓	✓	✗
Caladan [Fried et al. 2020]	✓				✓	✓	✓	✗
Shinjuku [Kaffes et al. 2019]		✓	✓			✓	✓	✗
mTCP [Jeong et al. 2014]		✓		✓		✓	–	✗
TAS [Kaufmann et al. 2019]		✓		✓		✓	–	✗
Perséphone [Demoulin et al. 2021]		✓	✓			–	–	✗
Demieagle	✓	✓	✓	✓	✓	✓	✓	✓

functions, allowing fine-grained migrations without introducing race conditions, packet reordering, or deadlocks. *Dyssect* distributes traffic at the flow level and uses optimization models to assign shards and flows to cores to achieve operator-specified SLOs.

Finally, we explore the flexibility provided by state disaggregation to allow *Dyssect* offloading the processing of NFs to a programmable NIC to reduce CPU load or satisfy latency guarantees of selected flows. We also evaluate the trade-offs of offloading processing of NFs to a programmable NIC. While the general belief is that offloading processing to a programmable NIC always pays off, our evaluation shows that, for some use cases, offloading a network function might have negative impacts on performance.

3. Multicore Scheduling in TCP Applications

In this section, we examine how the scheduling of the network stack affects the performance of microsecond-scale applications. We consider TCP network and application request processing equally important for multicore microsecond-scale scheduling. Investigating their interaction and placement on a multicore server is crucial for assessing a system's performance under different workloads. We outline the system model and define the design space dimensions used to classify and evaluate multicore architectures.

3.1. System Model

We assume a kernel-bypass system that runs at microsecond scale using commonly available datacenter hardware [Liu et al. 2019], similar to many recent systems [Fried et al. 2020, Zhang et al. 2021]. The hardware includes a high-performance network of at least 100 Gbps and a NUMA server with multicore CPUs. The TCP stack is fully featured and takes about $\sim$2 μs for total network processing, including retransmission and congestion and flow control. These times are based on the *Demieagle* TCP stack but are representative of other TCP stacks [Peter et al. 2014].

We also assume an RPC application where service times follow different distributions. Each request comprises four sequential stages: packet delivery, TCP inbound, application processing, and TCP outbound. We must schedule these four work units in order, but we have no additional restrictions on how or when we can schedule them.

3.2. Multicore Scheduling Design Classification

The design space for scheduling RPC processing spans three dimensions: *(i)* worker assignment, which maps processing stages to a set of one or more workers; *(ii)* queueing model, which defines the number of queues between workers and the queueing discipline; and *(iii)* spatial scheduling, which allocates workers to the server's hardware resources.

Worker Assignment. Each stage of request handling must be assigned to one or more workers for processing. The number of workers assigned to each stage may vary depending on the characteristics of the workload. We focus primarily on two predominant

models used by many existing systems [Ousterhout et al. 2019, Fried et al. 2020, Kaffes et al. 2019, Demoulin et al. 2021]: an *Inline* model where all stages are combined to run sequentially in each worker, and a *Dispatcher* model where network processing and application work are split between different sets of workers.

Queueing Model. For queueing models, we consider three approaches: *(i)* a centralized model (**cFCFS**), where workers for a given stage share a single centralized queue; *(ii)* a distributed model (**dFCFS**), where each worker in a given stage has its own queue from the previous stage; and *(iii)* a work-stealing model (**WS**), which is similar to **dFCFS**, but workers *steal* packets from other workers' queues when idle. Although other queueing disciplines are possible, they would significantly expand the design space, so we limit our investigation to FCFS for simplicity.

Spatial Scheduling. The spatial scheduling dimension maps workers to processing units. We classify spatial scheduling depending on where workers run using the following acronyms: using hyperthreads to share a physical core between workers (**HT**), using cores on a single NUMA node (**SN**), and using cores spanning multiple NUMA nodes (**MN**).

Table 1 shows how existing systems fit in our design classification. While *Demieagle* covers all points within this design space, our goal is not to compare the performance of *Demieagle* with these systems. Instead, our objective is to assess the impact of each design choice within the space.

3.3. Evaluation

We evaluate the impact of scheduling application workers across local and remote NUMA nodes in the Dispatcher model on latency. We use a memory-intensive application in which each worker iterates over its buffer with a 63-byte stride and a fixed number of iterations. The buffer size of each worker varies as a function of the LLC size. Using strided memory access patterns stresses the cache by frequently loading and unloading different data, exposing performance bottlenecks. In this experiment, we assign a single worker to handle the network stack on the local NUMA node, *i.e.,* the NUMA node where the NIC is connected, and distribute eight application workers as follows: *(i)* All application workers run on the local NUMA node where the NIC is attached (labeled as "L / L"); *(ii)* Half of the application workers run on the local NUMA node, and the other half in the remote NUMA node (labeled as "L / R"); *(iii)* All application workers run on the remote NUMA node, where the NIC is not attached (labeled "R / R").

Figure 2 shows the CCDF latency for the NUMA experiments with eight application workers and buffer sizes of 20%, 40%, 60%, and 80% of the LLC size. While the "L / L" configuration is competitive at 20%, it becomes much worse as the buffer size increases. Starting at 40%, the "L / R" configuration consistently provides much better latency. This improvement is attributed to using the additional LLC cache space on the remote NUMA node, which reduces cache misses. Furthermore, using the additional LLC cache space on the remote NUMA node helps the "R / R" configuration to achieve lower latency than "L / L". Note that the network worker is still on the local node.

We omit several results and findings due to space limitations. All experiments, results, and findings are in Chapter 4 of our thesis [Carvalho 2024]. *Demieagle*'s source code and the scripts for reproducing the results are publicly available.[1]

[1] `https://github.com/demieagle/demieagle`

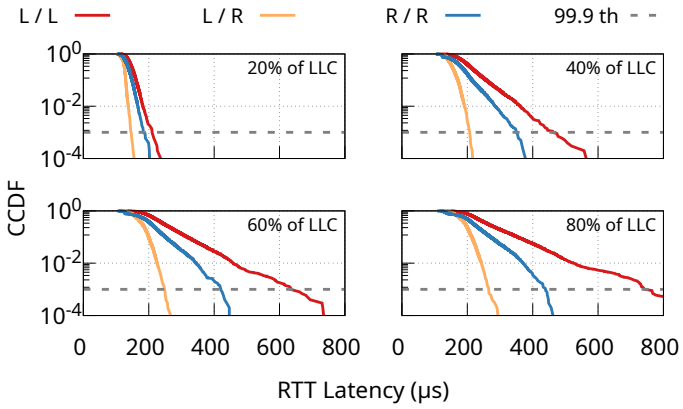


Figure 2. The Dispatcher model results using two NUMA nodes. Red, orange, and blue lines represent using only the local NUMA node, local and remote NUMA nodes, and only the remote NUMA node, respectively.

4. Dynamic Scaling of Stateful Network Functions

We present *Dyssect* [Carvalho et al. 2022, Carvalho et al. 2024], 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 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. 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. 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.

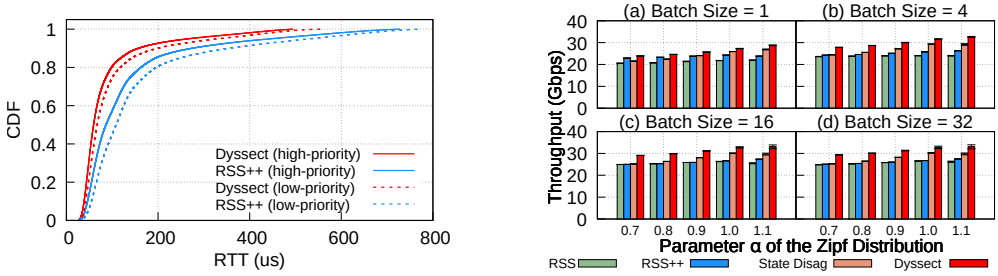
Dyssect partitions flows, and thus the state of a network function, into S shards such that S is a power-of-two and $S \leq E$, where E is the maximum number of entries in the NIC's RSS indirection table. This partitioning allows us to efficiently identify the shard of a packet by using the $\log_2 S$ least significant bits of a hash value on the packet's flow key (*e.g.*, 5-tuple). *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.

4.1. Flow Assignment

Dyssect combines two mechanisms for assigning flows to cores. 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 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, ensuring 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 types of traffic, or meet operator-defined SLOs.

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. 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*). 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.



(a) *Dyssect vs.* RSS++ latency. **(b) Throughput.**

Figure 3. Latencies and throughput.

4.2. Evaluation

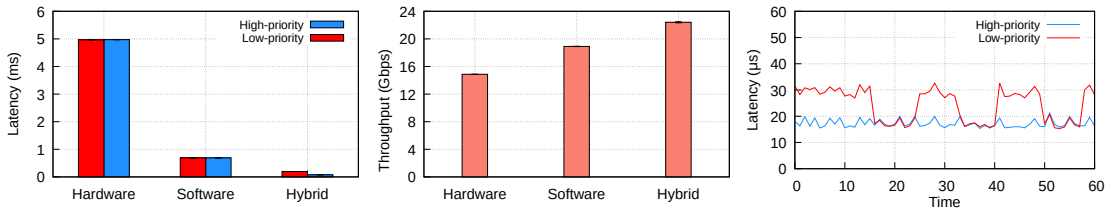
We evaluate *Dyssect* using a CAIDA trace, which exhibits a skewed distribution with a few large, long-lived flows carrying most of the traffic load. We consider short flows and 0.1% of the traffic as high priority, totaling 753,725 flows. The service chain includes NAT and IDS NFs. Unlike RSS++, which operates at the shard level, *Dyssect* prioritizes specific flows to reduce latency. As shown in Figure 3a, it lowers the 99.5% tail latency of high-priority traffic (solid lines) from $728.07\ \mu s$ to $494.77\ \mu s$ (32.04% reduction) without affecting low-priority traffic (dashed lines). Figure 3b shows the throughput of four approaches with varying batch sizes and α values in Zipf-distributed traces. *Dyssect* outperforms the other methods in all scenarios. Performance gains stem from both state disaggregation and the optimization model, with state disaggregation having a greater impact at higher α values (more skewed distributions). Compared to RSS++, *Dyssect* improves throughput by 4.11% ($\alpha = 0.7$, batch of 1) to 19.36% ($\alpha = 1.1$, batch of 4).

4.3. Hardware-Software Trade-offs

The general belief is that offloading network processing to the NIC always improves performance [Liu et al. 2019]. However, the performance depends on the network function and traffic workload, as NICs have limitations like lower clock and memory speeds. We evaluate the performance trade-offs of offloading to a programmable NIC by testing three setups: *(i)* running a NAT + IDS service chain entirely on the NIC, *(ii)* running it on a single server core, and *(iii)* a hybrid model, where the NIC handles high-priority flows while forwarding others to the server CPU.

Figures 4a and 4b show latency and throughput for three service chain execution models processing 1500-byte packets. Running the entire service chain on the NIC performs the worst due to the slower NIC processing units and state synchronization overhead. The hybrid model minimizes contention by offloading only low-bit-rate, high-priority flows to the NIC. These results emphasize the importance of selective NIC offloading, as some tasks perform better on the CPU.

Seamless Migration. *Dyssect* enables offloading functions and their states to a programmable NIC, which maps its memory to user space for seamless state migration between the host and NIC. To evaluate this migration, we measured the RTT of high- and



(a) Latency. (b) Throughput. (c) Packet latencies

Figure 4. Latency and throughput results for processing a service chain on the NIC (Hardware), on the CPU of the host (Software), or on both (Hybrid).

low- priority flows over 60 seconds, using a NAT NF, with the NIC handling high-priority packets and the CPU managing low-priority ones. Figure 4c shows the latencies of both flows. At 15 seconds, the *Dyssect* Controller offloads the low-priority flow to the NIC, then, migrates it back to the CPU 10 seconds later, repeating at 35, 40, and 50 seconds. When processed by the NIC, the low-priority flow achieves latencies similar to the high-priority flow. Additionally, state migration between the NIC and CPU occurs without traffic disruption or packet loss. Other experiments evaluating *Dyssect*'s performance can be found at [Carvalho et al. 2022, Carvalho 2024].

5. Related Work

Multicore Scheduling. Kernel-bypass systems enhance network application performance by allowing developers to bypass the kernel in packet processing, but they introduce challenges in coupling network stack and application processing. Two main models address these challenges: Inline [Peter et al. 2014, Fried et al. 2020], where network and application tasks run on a single worker, and Dispatcher [Kaufmann et al. 2019, Kaffes et al. 2019], which separates these tasks across multiple workers. Other works (see Chapter 3 in [Carvalho 2024]) investigate issues like scheduling, interference management, and network stack optimization, though performance evaluations often vary due to different stack implementations, complicating direct comparisons. *Demieagle* allows fair comparisons and facilitates a multitude of experiments to uncover trade-offs in scheduling policies for multicore architectures.

Stateful Network Functions. Recent research on stateful network functions focuses on packet distribution, load balancing, and offloading techniques to improve performance in high-speed network environments. Mechanisms like RSS and Flow Director assign packets to workers but face limitations such as load imbalance or vendor-specific constraints. Intra-server systems [Ousterhout et al. 2019, Fried et al. 2020] reduce latency and dynamically allocate workers, though they often lack support for stateful processing and packet ordering. Distributed solutions [Kablan et al. 2017, Gember-Jacobson et al. 2014, Woo et al. 2018] address state migration and load balancing across servers, ensuring state consistency during traffic spikes. Hardware-accelerated approaches, including CacheDirector and FPGA-based NICs, optimize data processing and reduce overhead for network functions. However, they do not address the challenges in maintaining packet ordering and state consistency when balancing the load for stateful network functions like *Dyssect* does.

6. Conclusion

This work investigates scaling stateful network services on multicore architectures, focusing on TCP stacks and network functions with per-packet state updates and dynamic workload balancing. We evaluate worker configurations using *Demieagle*, analyzing performance under synthetic and real-world workloads. Also, we propose *Dyssect*, which

dynamically scales cores and migrates state with zero-copy operations, improving latency and load balancing. We also examine offloading stateful functions to programmable NICs, showing that performance gains depend on NIC architecture and function requirements.

Acknowledgments

This work was partially funded by FAPESP procs. 2023/00812-7, 2023/00811-0, and 2020/05183-0; CNPq proc. 308101/2022-73; and CAPES Finance Code 001.

References

Barbette, T., Katsikas, G. P., Maguire, G. Q., and Kostić, D. (2019). RSS++: Load and State-Aware Receive Side Scaling. In *Proc. of ACM CoNEXT*.

Carvalho, F. B. (2024). *Scaling Stateful Network Services on Multicore Architectures*. PhD thesis, Federal University of Mato Grosso do Sul (UFMS).

Carvalho, F. B. et al. (2022). Dyssect: Dynamic Scaling of Stateful Network Functions. In *Proc. of IEEE INFOCOM*.

Carvalho, F. B. et al. (2024). State Disaggregation for Dynamic Scaling of Network Functions. *IEEE/ACM Transactions on Networking*.

Carvalho, F. B. et al. (2025). A Principled Approach to Multicore Scheduling in the Microsecond Era. To be submitted to ACM SOSP 2025.

Demoulin, H. M. et al. (2021). When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *Proc. of ACM SOSP*.

Fried, J., Ruan, Z., Ousterhout, A., and Belay, A. (2020). Caladan: Mitigating Interference at Microsecond Timescales. In *Proc. of USENIX OSDI*.

Gember-Jacobson et al. (2014). OpenNF: Enabling Innovation in Network Function Control. In *Proc. of ACM SIGCOMM*.

Jeong, E. Y. et al. (2014). MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *Proc. of USENIX NSDI*.

Kablan, M., Alsudais, A., Keller, E., and Le, F. (2017). Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *Proc. of USENIX NSDI*.

Kaffes, K. et al. (2019). Shinjuku: Preemptive Scheduling for usecond-Scale Tail Latency. In *Proc. of USENIX NSDI*.

Kaufmann, A., Stamler, T., Peter, S., Sharma, N. K., Krishnamurthy, A., and Anderson, T. (2019). TAS: TCP Acceleration as an OS Service. In *Proc. of ACM EuroSys*.

Liu, M. et al. (2019). Offloading Distributed Applications onto SmartNICs Using iPipe. In *Proc. of ACM SIGCOMM*.

Ousterhout, A. et al. (2019). Shenango: Achieving High CPU Efficiency for Latency-Sensitive Datacenter Workloads. In *Proc. of USENIX NSDI*.

Peter, S. et al. (2014). Arrakis: The Operating System is the Control Plane. In *Proc. of USENIX OSDI*.

Woo, S., Sherry, J., Han, S., Moon, S., Ratnasamy, S., and Shenker, S. (2018). Elastic Scaling of Stateful Network Functions. In *Proc. of USENIX NSDI*.

Zhang, I. et al. (2021). The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proc. of ACM SOSP*.