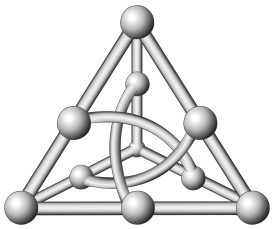
Scaling Stateful Network Services on Multicore Architectures

Fabrício Barbosa de Carvalho

Advisor: Prof. Ronaldo Alves Ferreira, Ph.D.



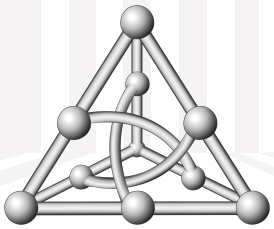
Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
November, 2024

Scaling Stateful Network Services on Multicore Architectures

Fabrício Barbosa de Carvalho

Advisor: Prof. Ronaldo Alves Ferreira, Ph.D.

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Computer Science



Faculdade de Computação
Universidade Federal de Mato Grosso do Sul
November, 2024

Abstract

In recent years, network speeds have surged while CPU speeds have plateaued, making kernel-based networking stacks increasingly impractical, particularly for multicore servers in datacenters. Consequently, kernel-bypass networking stacks and multicore applications have become necessary to keep up with faster datacenter networks. A large number of applications still require TCP for interoperability with legacy applications. However, TCP processing is resource-intensive compared to microsecond-scale applications, and its per-connection state complicates scheduling across multicore architectures. The interaction between network stack and application scheduling, whether on shared or different workers (*e.g.,* cores), strongly impacts performance by affecting cache usage, CPU pipeline efficiency, and memory access patterns. Given the demands of high-speed networks, optimizing these architectures is essential, yet many existing systems fail to address the performance of TCP and its interactions with the application under real-world conditions.

One of the goals of this thesis is to investigate the effective scheduling of a TCP stack alongside applications on multicore architectures, emphasizing the trade-offs involved in allocating workers for both TCP and application processing. This thesis takes a principled look at the relationship between a stateful network protocol with strong guarantees for scheduling such a stack with multicore applications. 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 the execution of "apples-to-apples" experiments to uncover the trade-offs of different multicore scheduling policies and architectures; thus, guiding application programmers toward an ideal scheduling policy for their workload.

In this thesis, we also address the complexity of scaling network functions. Network Function Virtualization (NFV) promises better utilization of computational resources by dynamically scaling resources on demand. However, most network functions are stateful and 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. 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. 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.

Keywords: Network Services, TCP, Stateful Network Functions, NFV.

"I have not failed.
I've just found 10,000
ways that won't work."

— Thomas A. Edison

Acknowledgements

First and foremost, I am deeply grateful to my parents, Elizabeth and Luiz Carlos; my siblings, Luiz Carlos, Nayara, and Tatiana; and my nephews, Dario and Natan. Your appreciation and encouragement of my academic achievements have inspired me constantly. I am forever grateful for your presence in my life, and this accomplishment is as much yours as it is mine.

I am sincerely grateful to my dedicated advisor, Professor Ronaldo Ferreira, for his invaluable guidance, feedback, and support throughout my doctorate. His extensive knowledge and experience were instrumental in completing this thesis, and I am truly thankful for all the guidance he provided during this journey.

Thanks to Ronaldo's encouragement, I had the opportunity to meet renowned researchers such as Dr. Adam Belay, Dr. Brian Kernighan, Dr. Dan Ports, Dr. Irene Zhang, Dr. Jennifer Rexford, Dr. Shan Lu, Dr. Simon Peter, Dr. Timothy Roscoe, and Dr. Walter Willinger. Additionally, I visited premier research institutions in the United States, including Columbia, Harvard, MIT, Microsoft Research, Princeton, and the University of Washington.

While it is challenging to name everyone who supported me during this journey, there are a few individuals whose significant contributions stood out at various stages and in different aspects. My heartfelt thanks go to my dear friends, in lexicographic order: André, Brivaldo, Diego (João Lucas), Luciana, Mayco, Nathalia, Rodrigo, and all K4 friends. Your unwavering support, encouragement, and friendship were a constant source of strength, especially during the challenging moments of my

doctorate journey.

I am also grateful to Dr. Ítalo Cunha, Dr. Marcos Vieira, and Dr. Murali Ramanathan for their invaluable contributions throughout my doctoral studies. Their insights greatly improved the quality of my work. Their guidance enriched my studies and was important in developing *Dyssect*, a work that I present in Chapter 5.

I am deeply grateful to Dr. Irene Zhang, Dr. Pedro Penna, and Mr. Anand Bonde for providing unimaginable experiences during my internship at Microsoft Research and to Mr. Joshua Fried and Mr. Matheus Stolet for their invaluable assistance in developing parts of my doctoral thesis.

I would like to thank the thesis committee, Dr. Ítalo Cunha, Dr. Marcos Vieira, Dr. Miguel Campista, and Dr. Weverton Cordeiro, for your time and effort in evaluating my thesis. I greatly appreciate your valuable feedback, support, and constructive criticism throughout this process. Thank you for participating in my thesis defense and for your commitment to advancing academic excellence.

Finally, I am grateful to my colleagues at the College of Engineering (FAENG) of the Federal University of Mato Grosso (UFMT) for their understanding and support, which made my academic journey more enriching and enjoyable. I also extend my thanks to RNP, the Brazilian National Research and Education Network, for funding a fellowship that allowed me to work on the research project "Researching Internet Routing Security in the Wild."

List of Acronyms

CAIDA	Center of Applied Internet Data Analysis
CCDF	Complementary Cumulative Distribution Function
CDF	Cumulative Distribution Function
cFCFS	Centralized First Come First Serve
CLS	Cluster Local Scratch
CPU	Central Processing Unit
CTM	Cluster Target Memory
DDIO	Data Direct I/O
dFCFS	Distributed First Come First Serve
DMA	Direct Memory Access
DPDK	Data Plane Development Kit
DRR	Deficit Round Robin
DWT	Discrete Wavelet Transform
eBPF	Extended Berkeley Packet Filter
EMEM	External Memory
FCFS	First Come First Serve
FPC	Flow Processing Core
FPGA	Field Programmable Gate Arrays
Gbps	Gigabits per Second
GRO	Global Reordering
HOL	Head-of-Line
IDS	Intrusion Detection System
IMEM	Internal Memory
IP	Internet Protocol
IPC	Inter-Process Communication
krps	Thousand Request per Second
KVS	Key-Value Store
LB	Load Balancer

LLC	Last-Level Cache
LMEM	Local Memory
LSB	Least Significant Bit
MANO	Management and Orchestration
MTU	Maximum Transmission Unit
NAT	Network Address Translation
NF	Network Function
NFV	Network Function Virtualization
NIC	Network Interface Card
NUMA	Non-Uniform Memory Access
PL	Pipeline
RPC	Remote Procedure Call
RQ	Research Question
RSS	Receive-Side Scaling
RTC	Run-to-Completion
RTT	Round-Trip Time
SLO	Service-Level Objective
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VNF	Virtual Network Function
WS	Work-Stealing

List of Algorithms

List of Figures

List of Tables

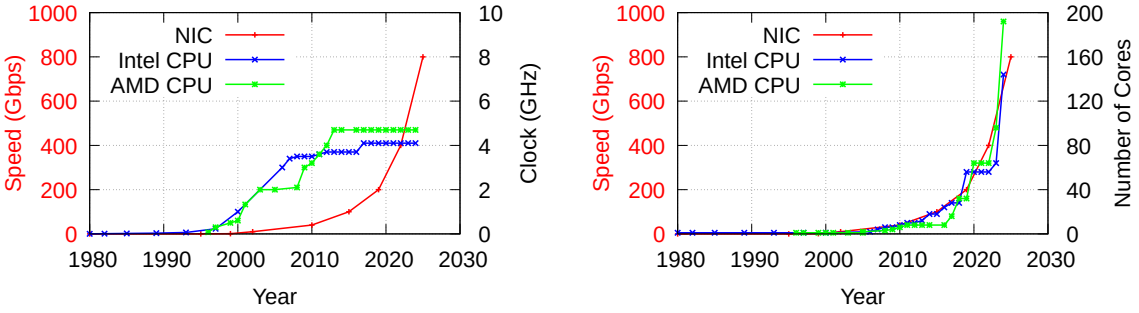
Contents

Chapter 1

Introduction

In the last decade, network speeds have grown exponentially, while CPU speeds have remained stagnant. Figure 1.1a 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 1.1b 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 [9, 108, 138] and network functions [33, 42]. 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 [25, 37, 99]. While these systems explored a range of multicore designs for managing application work queues, they each used a fixed design for scheduling the networking stack. For example, shared-nothing systems [9, 108] 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



(a) Evolution of NIC speed and clock rates for the AMD and Intel processors. The blue and green lines show the maximum base clock rates. .

(b) Evolution of the number of cores in the AMD and Intel processors.

Figure 1.1: NIC speeds, processor clock rates, and number of cores over the years.

synchronization. The NIC performs RSS by applying a hash function to the header of incoming network packets, using packet fields such as source and destination IP addresses and ports. It then uses the hash value's Least Significant Bits (LSBs) to index an indirection table, which directs the packet to the appropriate CPU core for further processing, as shown in Figure 1.2. At the other extreme, some systems separate network processing and application work on distinct cores for various reasons, including fine-grained control over application work scheduling [25, 57], extracting higher performance gains from batching and vectorization [65], or privilege-separation [82].

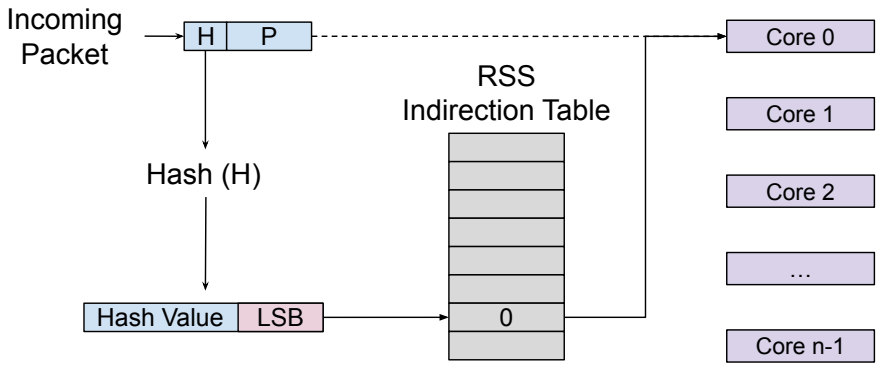


Figure 1.2: Illustration of the Receive Side Scaling (RSS) process.

1.1 Stateful Transport Protocol

In this thesis, 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. We do not consider network stacks that require a custom protocol [60] or hardware [114]. TCP is a strict requirement for interoperability with legacy applications where the client-side networking stack cannot be changed. While many applications run exclusively within the datacenter, an increasing number of applications rely on distributed services and require a microsecond-scale TCP stack to guarantee service-level objectives, such as Key-Value Stores (KVS) [28, 85] and real-time gaming applications [22, 26].

Given that many microsecond-scale applications are relatively simple, TCP processing can consume a large number of CPU cycles relative to the application while being notoriously difficult to schedule on multicore architectures as the protocol requires per-connection state. At microsecond scales, the multicore architecture of the network stack and application, especially in *relationship* to each other, has a large impact on performance. Small differences in architecture impact cache utilization, CPU pipeline efficiency, and memory accesses, which 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 actually fare in the real world.

Figure 1.3 shows an example of how multicore architecture affects tail latency (*i.e.,* 99.9th percentile) for application request times of 1 μs (upper graph) and 100 μs (lower graph). The latency measurements represent Round-Trip Times (RTTs) collected by the client, which sends several Thousand Requests per Second (krps) to evaluate the server's application performance under high request rates. Network

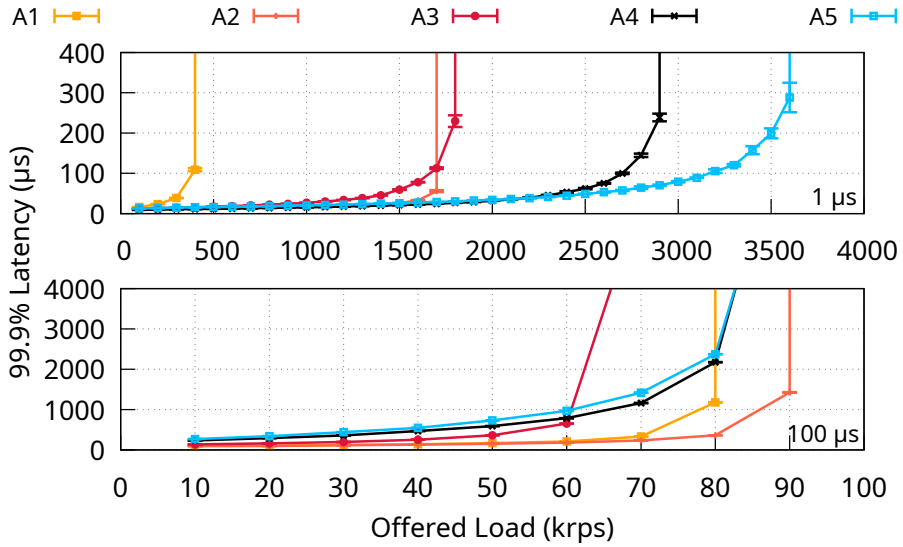


Figure 1.3: Tail latencies for five different 8-core arrangements (A1 to A5) to process requests with 1 μs (top) and 100 μs (bottom) service time and increasing offered load.

stack processing takes about 2 μs, which gives a processing ratio of 0.5 and 50, respectively, between the application and network stack. The figure compares five system configurations (A1 to A5) running on eight CPU cores. The configurations differ in: *(i)* queueing policies (A2 and A4), *(ii)* the number of allocated cores for network-stack processing (A1 and A3), whether the network processing and application are co-located (A2 and A4) *(iii)*, or separated (A1 and A3) on the same cores, and whether hyperthreading is used (A5). We do not detail the different configurations here (see Chapter 4); our primary goal is to show the wide range of performance. For example, while configuration A1 performs the worst with a service time of 1 μs, it outperforms three of the other four configurations when the service time increases to 100 μs.

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 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. Network functions can be composed to provide network services in what is known as a service chain.

The vast majority of network functions are stateful (*e.g.,* network monitors, stateful firewalls, or load balancers) 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. The classic approach to control concurrent access to a shared state is the use of locks. Unfortunately, locks decrease performance [15] and may prevent network functions from operating at line rate in the multi-gigabit links that are common today.

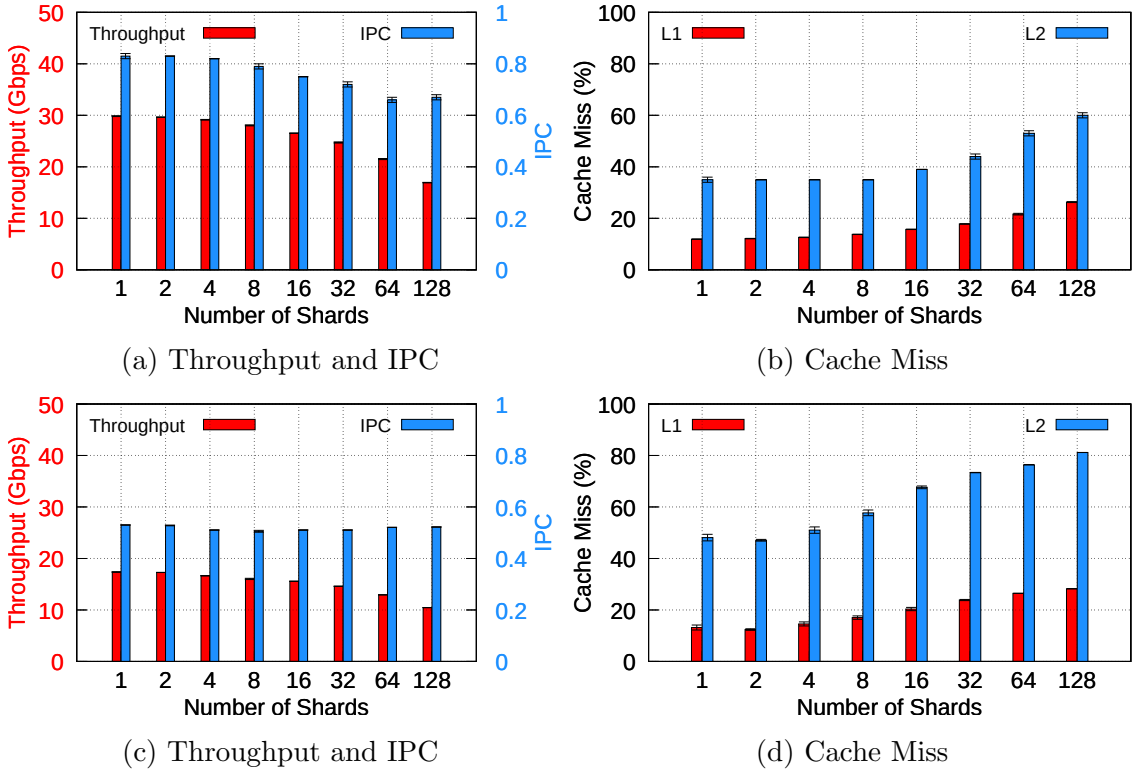
Several research proposals use state *sharding* to avoid the use of locks [27, 58, 61, 76, 86]. Sharding is a technique that partitions the (global) state of a network function into disjoint state subsets (*shards*). Flows are mapped to shards using the result of a hash function computed on a *flow key* (*e.g.,* the 5-tuple), and each shard is mapped to *one* worker to avoid concurrent accesses. Modern high-speed NICs can hash incoming packets and place them in per-worker queues specified in a RSS indirection table [51]. Workers poll their queues to receive and process packets for their shards at high throughput. In this approach, statically mapping shards to workers leads to load imbalance [6, 118], and using a centralized software packet dispatcher does not scale to the line speed of the current high-speed links [57, 61, 99].

A recent effort proposes dynamic reassignments of shards to balance the load between workers [6]. 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.

The current trend is to increase the NIC's RSS indirection table (*e.g.,* Mellanox ConnectX-5 EN has 512 entries [96]) and, consequently, allow more shards to reduce the likelihood of multiple large-volume flows (*i.e.,* elephant flows) in a single shard. Unfortunately, a large number of shards exacerbates the load imbalance [119] 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 worker 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, which may hinder or reduce 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 [134]. Cache memories mitigate this problem, but they work well only if the network services take advantage of temporal and spatial localities during packet processing. For example, Figure 1.4 shows the performance impact of the number of shards in CPU performance metrics. In this simple experiment, a single worker runs a NAT (Network Address Translation) and processes packets from one million flows following a Zipf distribution with $\alpha = 1.1$. As we can see, the throughput drops up to 43.3% (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 Figures 1.4b and 1.4d that cache misses increase as the number of shards increases, leading to fewer instructions per cycle (Figures 1.4a and 1.4c).



(a) Throughput and IPC

(b) Cache Miss

(c) Throughput and IPC

(d) Cache Miss

Figure 1.4: Throughput, instructions per cycle (IPC), and cache misses for different numbers of shards under workloads with Zipf (1.4a and 1.4b) and uniform (1.4c and 1.4d) distributions.

1.3 Problem Statement and Research Questions

The previous sections discuss two network services, TCP network stacks and network functions, that require an internal state for proper operation. Scaling these services while maintaining their states consistent across multiple workers (*i.e.,* CPU cores or hyperthreads) presents significant challenges, particularly when each packet demands a state update, as these updates may require synchronization mechanisms that can degrade performance. Moreover, workload imbalances may result in some workers being overloaded while others are 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 [1, 6, 40, 55, 65, 69, 101, 132, 137]. 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 [12, 41, 69]. Also, packet reordering can harm stateful network functions that rely on packets arriving in order (*e.g.,* stateful firewall), resulting in decreased performance or even service disruption when packets arrive out of order. Many efforts [40, 136, 137] address such issues by carefully orchestrating the migration of flows between stateful network functions.

In this thesis, we overcome these challenges by addressing the following three research questions:

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

We address the first research question in Chapter 4, where we systematically evaluate the trade-offs involved in assigning workers to the network stack and application. In particular, we consider the network stack processing and application request processing equally important for multicore scheduling at microsecond-scale. Investigating their interactions and placement on a multicore server is crucial for

assessing the performance of a system under different workloads. We define the design space in three dimensions that include worker assignment to different stages of packet processing, the queueing model, and worker placement in hardware resources, such as hyperthreads and cores on the same or different NUMA (Non-Uniform Memory Access) nodes. We classify existing systems in this design space, implement models that cover the main points in the design space in a unified framework with a fully-featured TCP stack, and evaluate the main combinations of worker assignment, queueing model, and worker placement under different workloads.

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

We address the second research question in Chapter 5, where we discuss the design, implementation, and evaluation of *Dyssect*, a system that 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 workers to achieve a more even load distribution without introducing race conditions. By carefully coordinating flow migration between workers and a centralized controller, *Dyssect* prevents packet reordering, improves throughput, and reduces latency compared to competing solutions.

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

After analyzing how redistributing load across workers affects the performance of stateful network functions (RQ-2), Section 5.3 investigates the role that programmable NICs may play in the performance of these functions, addressing RQ-3. In this case, we advance the idea of state disaggregation by offloading stateful network functions, or parts of them, onto programmable NICs, aiming to further optimize network performance. This offloading strategy enables us to investigate hardware and software trade-offs that are specific to network functions.

While offloading network functions to programmable NICs generally offers potential performance gains, Section 5.3 reveals that these gains are not always achieved. This finding is mainly due to the distinctive multicore architectures of programmable NICs and the unique processing requirements of some network functions. For example, some programmable NICs operate at significantly lower clock speeds than general-purpose server processors (*e.g.,* 800 MHz *vs.* 2 GHz). As a result, running a network function that requires intensive packet payload processing, such as an Intrusion Detection System (IDS), may lead to considerable performance degradation.

1.4 Main Contributions

In this section, we present our main contributions to scaling stateful network services on multicore architectures. In Section 1.4.1, we discuss our contributions to investigating trade-offs in scheduling the network stack and applications in kernel-bypass systems that seek to process requests at microsecond-scale. Section 1.4.2 presents this thesis' contributions to scaling stateful network functions.

1.4.1 Stateful Transport Protocol

This thesis is the first work to systematically study the trade-offs in various multicore architectures for microsecond-scale networking stacks and applications. For the network stack, we focus specifically on TCP, as it is a requirement for interoperability with legacy applications. However, the findings we discuss in Chapter 4 apply to other transport protocols that have to update state at every received or transmitted packet. We classify the multicore architectures and implement them in *Demieagle*, a benchmark framework that we designed to evaluate different points in the design space. *Demieagle* consists of a flexible, kernel-bypassing TCP stack and scheduler, built on the Demikernel [138] datapath operating system,

which provides a fully-featured TCP stack and flexibility for implementing different scheduling policies. Using *Demieagle*'s flexible scheduler, we can directly compare multicore architectures and scheduling strategies across various workloads, ensuring an apples-to-apples comparison of the different architectures.

Demieagle also includes a workload generator that exercises our multicore classification system by changing workload parameters that are directly affected by the architecture. This workload generator demonstrates which workload characteristics benefit from specific architectures and which are less efficient. In addition, it serves as a tool for exploring other systems and whether their performance characteristics match the expectations within our classification.

Finally, we offer guidelines to choose the best multicore architecture based on the queueing model, the offered load, the average application service time (relative to the network stack's time), and the application memory footprint (small or large).

1.4.2 Stateful Network Functions

Based on the main findings and guidelines on multicore architecture that we discuss in Chapter 4, we design, implement, and evaluate a system called *Dyssect*, which scales stateful network functions. This system enhances NFV performance by allowing lock-free state migration operations to balance the load, prioritize traffic, and optimize resource usage while effectively meeting user-defined Service-Level Objectives (SLOs). 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[1], avoiding the bottleneck of having a single core dispatching all the packets (*e.g.,* RAMCloud [100], Shinjuku [57], Perséphone [25]). 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 distribute traffic at the flow level, it can use a smaller number of shards than other approaches (*e.g.,* RSS++ [6]) and avoid frequent shard transfers, overcoming the performance penalties discussed above (Figure 1.4). Fourth, *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. Finally, the flexibility provided by state disaggregation allows *Dyssect* to offload the processing of network functions to a programmable NIC to reduce CPU load or satisfy latency guarantees of selected flows.

We implemented *Dyssect* as user-level modules using BESS [43] with DPDK (Data Plane Development Kit) for fast packet processing and Gurobi [98] for solving our mathematical models. We also ported *Dyssect* to FastClick [7] to make a fair comparison with RSS++, which is built on top of FastClick. Experimental evaluations on a multicore server running multiple NFs and processing synthetic and real traffic show that *Dyssect* reduces tail latency up to 32.04% 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. We also evaluate the trade-offs of offloading processing of network functions 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.

[1]To maintain a consistent nomenclature, we use the word worker to refer to a CPU core or hyperthread. However, in Chapter 5 and when discussing *Dyssect*, we use the term core with the same meaning as worker. We make this distinction because we define different types of cores in *Dyssect*, which makes sense in the context of network functions, as they generally run on cores with hyperthreading disabled.

1.5 Thesis Roadmap

The remainder of this thesis is organized as follows. Chapter 2 provides essential background information to establish the necessary foundation for the thesis. Chapter 3 reviews related work, emphasizing recent developments in kernel-bypass systems for network applications and functions. Chapter 4 examines multicore scheduling for network applications that use the TCP protocol, highlighting the equal importance of the network stack and application in the scheduling process. Chapter 5 discusses the dynamic scaling of stateful network functions through state disaggregation. Finally, Chapter 6 concludes the thesis and suggests directions for future research.

Chapter 2

Background

This chapter introduces the key concepts that are essential for understanding the contributions of this thesis. Section 2.1 discusses some aspects of the implementation of the TCP protocol that make it difficult to scale it on multicore architectures. Section 2.2 introduces network functions and their key features. Finally, Section 2.3 summarizes the main aspects of stateful transport protocol and network functions that are relevant to this thesis.

2.1 Stateful Transport Protocol

The Transmission Control Protocol (TCP) provides reliable, ordered, and error-free data delivery to applications over a network. Implementing TCP on multicore architectures is complex and involves many steps, from when a packet arrives from the network to when it is delivered to the application. To start, either the operating system or a kernel-bypass system must configure the NIC to deliver the packet to a specific worker. More specifically, the NIC's RSS indirection table is set up with a number of queues matching the number of workers. The NIC then places the packet in one of these queues based on a hash computed over the packet's five-tuple.

Upon receiving a packet, the worker reads it from the queue and initiates the

network-stack processing sequence. When the packet gets to TCP, the worker examines the TCP header to retrieve essential information, such as source and destination ports, that are necessary to locate the TCP Control Block (TCB). The TCB is a data structure maintained for each active TCP connection that stores the connection's current state, including sequence and acknowledgment numbers, window size, and pointers to the send and receive buffers. As packets are received, the worker updates the TCB to ensure that the protocol manages in-order delivery and handles retransmissions when necessary.

The process of sending packets mirrors that of the reception. To prepare a packet for transmission, the worker constructs a TCP segment, including the appropriate headers and payload, and places it in a send buffer for processing. Once ready, the network stack transmits the segment through the NIC.

In multicore environments, where multiple workers can access the TCB, synchronization may be required for handling shared data structures like the receive and send buffers. For example, the TCP implementation in Linux uses atomic operations, semaphores, readers-writer locks, and spinlocks to maintain data integrity and prevent race conditions [46]. These mechanisms ensure that only one worker can access or modify the TCB or buffers at any given time.

Scaling TCP applications requires adding more workers to process higher workloads, whether for TCP or application processing. As the number of workers increases, the complexity of coordinating and synchronizing them grows, especially when TCP and application run on separate workers. Therefore, special care must be devoted to the multicore architecture of the network stack to maintain low-latency responses, as even minor delays in state synchronization can significantly impact overall performance.

2.2 Network Functions

Network Functions (NFs), also called middleboxes or network appliances, are components of the network that perform well-defined procedures on network packets. Previously, network functions were black boxes with exclusive manufacturer procedures, making them expensive and inflexible to custom operation within a conventional network. However, 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.

There are two main classes of network functions: stateful and stateless. The first class includes network functions that need state to perform the application logic. For instance, a Network Address Translation (NAT) network function needs to store the mapping between IP addresses to modify network packets. On the other hand, stateless NFs do not need any additional state to process the packets (*e.g.,* checksum, compression, and decompression).

The vast majority of network functions are stateful (*e.g.,* network monitors, stateful firewalls, or load balancers) and may require state updates per packet. However, an NF can have more than one state and update the state on a per-flow basis, too. For example, a NAT has an Address Pool state, which is updated on a per-flow basis (when a new entry is created), and a NAT Entry state that is updated per packet [132].

In NFV, there are two execution models in the existing frameworks [141]: run-to-completion (RTC) and pipeline (PL). The RTC model [63, 104] executes all NFs of a service chain on a single worker. On the other hand, the pipeline model runs one NF per worker in a pipeline fashion [49, 81, 103]. Although RTC performs better (in terms of throughput and latency) than PL due to inter-worker communication, PL provides more flexibility during scaling operations.

Regardless of the execution model adopted by an NFV framework, packet

reordering is still a huge issue in packet processing. Network protocols assume that packet reordering, packet loss, and packet corruption are infrequent events that occur during data communication. For instance, TCP reduces throughput when detecting packet reordering because its congestion control algorithm interprets these issues as a signal that the network is congested [14, 105]. Several research efforts present different techniques to measurement packet reordering [10, 12, 125, 131, 133], indicating that this issue is present nowadays.

The flexibility provided by software implementations allows a network function to be dynamically scaled out by adding new instances or allocating additional resources (*e.g.,* workers 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 the hardware can be reallocated to other tasks or put to sleep to save energy.

During a scaling operation, workers 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 controlling concurrent access to a shared state is the use of locks. Unfortunately, locks decrease performance [15] 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 [27, 58, 61, 76, 86]. 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. Modern high-speed NICs can hash incoming packets and place them in per-worker queues specified in a *Receive-Side Scaling* (RSS) [51] indirection table. Workers retrieve incoming packets through these queues. Also, recent NICs provide a FlowDirector [53], a recent functionality to forward incoming packets to specific NIC queues based on the 5-tuple. In this case, packets from a flow are forwarded to a specific queue and,

consequently, a specific worker.

2.3 Summary

This chapter covers two foundational concepts: stateful transport protocols and stateful network functions. The Transmission Control Protocol (TCP) provides reliable data packet delivery to applications over a network. In multicore architectures, the implementation of TCP presents challenges, particularly in managing the TCP state represented by the TCP Control Block (TCB). With multiple workers accessing the TCB, synchronization may be necessary to prevent race conditions and maintain data integrity. Mechanisms like mutexes and spinlocks ensure that only one worker can access or modify the TCB at any time. As TCP applications scale, effective coordination becomes crucial for preserving performance and enabling low-latency responses.

Stateful network functions add complexity by maintaining network states crucial for services such as Network Address Translation (NAT), Load Balancers (LBs), and Intrusion Detection Systems (IDSs). Unlike stateless functions, which process each packet independently, stateful functions require consistent state tracking to retain connection information across distributed elements. This chapter addresses the scalability and consistency challenges of stateful network functions across multiple workers. Overcoming these challenges requires carefully designed strategies that balance state distribution, minimize latency, and maximize throughput to ensure efficiency and reliability in high-performance, large-scale systems.

Chapter 3

Related Work

This chapter reviews the most relevant related work for this thesis. Section 3.1 begins with the stateful transport protocol, focusing on in-memory key-value stores, which are essential to reduce response latency in datacenter applications. It then discusses advancements in kernel-bypass technologies, which are designed to eliminate operating system overhead and enhance network performance. The section then considers various scheduling policies and assesses their impact on resource allocation and system throughput across different workloads. Finally, the section reviews spatial scheduling strategies, which seek to optimize task placement to leverage hardware resources effectively within multicore architectures, such as allocating a task to a worker on a local core or on a remote node's core in a NUMA server.

The second section considers stateful network functions, beginning with hardware dispatchers that enhance packet processing by offloading critical tasks to specialized hardware. Next, the section discusses software dispatchers that provide flexible and scalable solutions through software-defined methods. It then reviews load-balancing techniques, with a focus on efficiently distributing workloads across network function instances, especially in contexts with large data flows. The section also covers system-level optimizations, highlighting their impact in reducing

latency and improving throughput. The section concludes with a discussion of hardware-accelerated systems, which leverage specialized processors to handle network functions at scale.

The chapter concludes by synthesizing the state-of-the-art in stateful network protocol and network functions, illustrating how these developments align with this thesis's objectives. The insights gained from this review lay the foundation for the approaches proposed in subsequent chapters.

3.1 Stateful Transport Protocol

3.1.1 In-memory key-value stores

Key-value storage systems increasingly adopt in-memory designs to meet the demands of high-throughput and low-latency large-scale Internet services. Systems such as RocksDB [28], Memcached [85], and Redis [115] use a range of techniques to improve performance, including advanced data structures and algorithms [66,74–76], system-level optimizations [32, 128], and new hardware capabilities [58, 59, 71, 71]. Our work, on the other hand, shows how the TCP stack and application scheduling impact performance on multicore systems, revealing another source for performance improvements on these systems.

3.1.2 Kernel-bypass systems

Kernel-bypass systems empower developers with full control over packet delivery and processing and offer flexibility for building network applications, such as implementing custom network stacks in user space. By bypassing the general-purpose kernel in the data plane, these systems enhance performance but also introduce new challenges. Recent research efforts have addressed issues such as scheduling [25, 57, 111], interference management [21, 37, 79], network stack and

service optimization [55, 60, 65], and efficient I/O processing [9, 61, 108]. A key challenge in developing kernel-bypass systems lies in effectively coupling network stack and application processing. There are two primary models for this purpose: *Inline* and *Dispatcher*. We briefly describe these models below, along with an overview of existing systems that adopt each approach. We detail the Inline and Dispatcher models and their configurations in Chapter 4.

3.1.2.1 Inline model

In the inline model, the network stack and application run on a single worker, which generally leads to performance gains since there is no need for inter-core communication. Arrakis [108] is the first system in this domain and bypasses the kernel for I/O operations to let applications directly manage network and storage resources, which results in lower overhead and increased throughput. IX [9] improves this approach by implementing a data-plane operating system that reduces context switching and cache contention, achieving ultra-low latency and high throughput for network-intensive workloads. ZygOS [111] introduces a scalable multicore architecture that balance network flows between workers, using work-stealing, and ensures that application and network processing occur on the same worker, resulting in lower latency and improved load balancing. Shenango [99] maximizes efficiency in low-latency applications by dynamically allocating workers to handle varying loads while keeping the application and network processing tightly coupled on the same worker to minimize inter-core communication. Finally, Caladan [37] extends these principles with a focus on high throughput and low latency. It runs the application and network stack on the same worker and uses a software-defined scheduler that optimizes core utilization.

3.1.2.2 Dispatcher model

In the Dispatcher Model, network processing and application work are split between different sets of workers. mTCP [55] provides a high-performance, user-level TCP stack for multicore servers, focusing on reducing context-switching overhead and inter-core locking. The primary contribution of mTCP lies in enhancing TCP efficiency for high-concurrency environments by employing lock-free data structures, cache-aware thread placement, and efficient per-core resource management. TAS [65] offers TCP acceleration as a service by moving typical TCP processing from the OS kernel to a fast-path OS service on dedicated CPU cores. Its primary contribution is isolating common-case TCP operations in a fast path while managing corner cases in a slow path, which enhances throughput. RAMCloud [100] is a distributed in-memory storage system for low-latency access to large datasets. Its main contribution is using DRAM for distributed storage, achieving microsecond-level access times through a log-structured mechanism for managing all storage and a networking layer that bypasses the kernel to communicate directly with the NIC using polling. Shinjuku [57] is a low-latency microservice execution system with hardware-level preemption to guarantee microsecond-scale response times. Its main contribution lies in exploring hardware support for virtualization to make frequent preemptions without degrading performance. It provides low tail latency and high throughput for various request distributions and service times. Perséphone [25] is a kernel-bypass OS scheduler that minimizes tail latency for microsecond-scale applications with service-time distributions with high dispersion. The primary contribution of Perséphone is to provide a non-work-conserving policy that profiles application requests and reserves cores for short requests, preventing them from being blocked by longer ones. It achieves lower tail latencies and can handle higher loads with fewer cores than existing kernel-bypass schedulers by mitigating the head-of-line-blocking problem.

A key limitation in previous research, discussed in Section 3.1.2.1 and in this section, is the use of different network-stack implementations when assessing performance gains. This inconsistency makes it difficult to determine whether reported gains come from the new approaches themselves or simply from differences in network-stack implementations. For instance, some works [89, 120] use a minimal implementation of TCP that ignores several corner cases, which may save a few nanoseconds. At microsecond-scale, small savings for processing each packet may yield significant performance gains for a specific approach. In Chapter 4, we address this issue by evaluating various configurations of the Inline and Dispatcher models using the same network stack and scheduling framework under different workloads. Using the same framework for evaluating the models allows us to isolate the performance gains of each model under different conditions and identify the best model and configuration for specific workloads.

3.1.3 Scheduling Policy Evaluation

Previous work has explored scheduling policies to understand their impact on datacenter applications, focusing on optimizing throughput, reducing latency, or evaluating different scheduling options. However, some works explore only a narrow subset of configurations in the design space [57, 111] or rely on simulations that do not account for independent scheduling of the network stack and application [83]. In contrast, we comprehensively evaluate various scheduling policies within a unified evaluation framework that includes a fully-featured TCP stack of a real datapath operating system. Our findings offer practical insights, guiding developers toward implementing the most effective scheduling policies to meet their specific requirements regarding workload and application characteristics.

3.1.4 Spatial Scheduling

Previous studies [29,38,57,94,95] have extensively explored spatial scheduling, often focusing on specific aspects of CPU core placement (*e.g.,* Caladan [37] optimizes worker placement to reduce interference). However, some approaches either disable hyperthreading or leave sibling hyperthreads idle, overlooking potential advantages or disadvantages. In contrast, we thoroughly investigate all possible worker placement strategies for both the network stack and the application while also evaluating the potential performance benefits of utilizing additional resources from a remote node in a NUMA server.

3.2 Stateful Network Functions

3.2.1 Intra-Server Hardware Dispatcher

Recent proposals, such as IX [8], Arrakis [108], and Sprayer [118], use RSS to direct packets to workers. However, RSS assigns flows to workers based on the hash of some fields of the packets, which may lead to CPU load imbalance and, consequently, higher packet delays and underutilization of some workers. RSS++ [6] tries to balance the load across workers by constantly updating the RSS indirection table. Still, it can only balance the load at the shard level and requires updates at a rate that some commercial NICs cannot sustain [6, 19].

Flow Director [53] allows a better traffic distribution than RSS by using a match table that associates flows and processing workers. Flow Director does not use a hash function to direct the packets but requires one rule for each flow to provide fine-grained flow management, limiting the number of flows it can handle. Also, Flow Director is a proprietary technology supported only by specific NIC models from some brands. Affinity-Accept [107] is an example of a recent system that uses Flow Director.

3.2.2 Intra-Server Software Dispatcher

Shenango [99] introduces a dynamic threading architecture that allows applications to efficiently use workers to adapt the system to varying workloads to maintain low latency and high performance. Caladan [37], on the other hand, enhances the efficiency of multicore systems by implementing scheduling algorithms that mitigate interference and reduce latency, also yielding high throughput. Perséphone [25] load balances the requests between workers and minimizes inter-thread interference to guarantee service-level objectives for latency-sensitive applications. Shinjuku [57] handles events to guarantee an ultra-fast response by managing task queues with precision to deliver sub-microsecond response times, essential for real-time processing needs. ZygOS [111] distributes flows evenly between workers to prevent bottlenecks and maintain a low-latency response consistently. Minos [27] integrates hardware and software to refine thread scheduling and resource management and ensure high efficiency in multicore environments to guarantee low latency for applications. These systems are intra-server software dispatchers that use a set of workers to fetch packets from the NIC and forward them to other application workers. While these works reduce tail latency, they neither support stateful network functions nor guarantee packet ordering.

3.2.3 Inter-Server Load Balancing

StatelessNF [56] manages stateless network functions and balances the load between the functions by avoiding the complexities of maintaining state information across servers, which facilitates scaling and fault tolerance. OpenNF [39] builds on stateless network functions and provides a flexible platform for managing network functions that dynamically balances load and migrates state between servers to optimize performance and resource utilization. Split/Merge [113] offers a mechanism for dividing and consolidating network flows across servers. It enables fine-grained

control for balancing the load and minimizing the impact of traffic spikes. S6 [132] focuses on efficient state management to scale stateful applications across servers without sacrificing performance, making it ideal for scenarios where state consistency is critical. isRSS [97] uses receive-side scaling to distribute incoming network traffic across multiple servers, optimize CPU usage, and reduce bottlenecks when the traffic load increases. E2 [103] introduces an elastic execution framework that adjusts resource allocation across servers in response to workload variations, which ensures balanced load distribution and maintains high performance in distributed network environments. All these systems implement state migration for stateful network functions to balance the load across servers, which differs from our approach in Chapter 5 to balance the load inside a server.

3.2.4 System-Level Optimizations

Some efforts improve the processing of a service chain by exploiting hardware features [34, 35], performing code optimizations [33], or eliminating redundant computations. Reframer [41] delays some packets and explores system caches to improve throughput. CacheDirector [34] places the first 64 bytes of the packet in the Last-Level Cache (LLC) slice closest to the processing core and [35] optimizes Data Direct I/O (DDIO) [52] for reducing processing delays. PacketMill [33] uses code-optimization techniques and a new metadata management model to eliminate redundant NF computations inside a server. OpenBox [16] separates control and data planes to eliminate redundant computations in NFs on a service chain inside and across servers.

3.2.5 Hardware-Based Systems

A recent trend to accelerate network functions is the use of programmable or (FPGA – Field Programmable Gate Array)-based NICs. NICA [30] accelerates the

application data plane on FPGA-based Programmable NICs (F-NICs), offloading part of the network traffic to be processed by the NIC. EBPFlow [102] offloads eBPF (Extended Berkeley Packet Filter [129]) programs to the FPGA programmable NIC. AccelNet [36] offloads networking stack to hardware using custom Azure Programmable NICs based on FPGAs. FlexNIC [64] proposes a flexible DMA (Direct Memory Access) programming interface for network I/O, inserting packet processing rules into the NIC. iPipe [77] offloads distributed applications onto Programmable NICs using a hybrid scheduler, combining (FCFS – First Come First Serve) and (DRR – Deficit Round Robin)-based processor sharing. ClickNP [70] exposes a modular programming abstraction to accelerate NFs with FPGA programmable devices.

3.3 Summary

This chapter reviews research efforts aimed at scaling network services across various environments, including intra- and inter-server setups, as well as both stateless and stateful service architectures. We discuss the challenges inherent to stateful applications, particularly the need to manage both application and TCP protocol states. These challenges underscore the critical importance of worker allocation strategies, as choosing which worker executes application tasks and network stack operations can influence performance. Similarly, stateful network functions require load-balancing mechanisms to achieve high throughput and reduced latencies, especially when handling long-lived and large-volume flows (*i.e.,* elephant flows).

The related work reviewed in this chapter highlights the importance of overcoming these challenges to optimize performance in stateful systems. We discuss how foundational studies on worker assignment and load balancing directly inform and support this thesis's goals, providing the foundation for the proposed approaches and their significance in advancing the field.

Chapter 4

Multicore Scheduling in TCP Applications

In this chapter, we examine how the scheduling of the network stack affects the performance of applications with service times on the order of a few microseconds. Specifically, we outline the design space, discuss various trade-offs, evaluate different architectures, and offer guidelines for selecting a multicore architecture based on key factors, including application service time, offered load, and queueing model.

4.1 Design Space of Multicore Scheduling

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

4.1.1 System Model

We assume a kernel-bypass system that runs at microsecond-scale using commonly-available datacenter hardware [4, 77, 109, 117, 127], similar to many recent systems [29, 37, 55, 60, 82, 108]. The hardware includes a high-performance network of at least 100 Gbps and a Non-Uniform Memory Access (NUMA) server with multicore CPUs of 2 GHz or higher with common features such as multi-level caching and hyperthreading. The TCP stack is fully featured and takes about 1.81 μs to receive and 0.19 μs to send a packet (*i.e.,* $\sim$2 μs for total network processing), including background work for retransmission and congestion and flow control. These times are based on the *Demieagle* TCP stack but are representative of other TCP stacks [108]. However, for our evaluation, the ratio between the application service time and the processing time of the TCP stack is more important than its absolute time.

We also assume an (RPC – Remote Procedure Call)-processing application where service times follow different distributions. Each application request goes through four stages of work: NIC packet delivery, inbound TCP processing, application processing, and outbound TCP processing. These four work units must be scheduled in sequence, but there are no other limitations to how or when they are scheduled.

4.1.2 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 (*i.e.,* cores and hyperthreads).

Table 4.1: Classification of existing systems in the design space.

System	Worker Assignment		Queueing Model			Spatial Scheduling		
	Inline	Dispatcher	Centralized (cFCFS)	Distributed (dFCFS)	Work-stealing (WS)	HT	SN	MN
Demikernel [138]	✓		✓			✗	–	✗
Arrakis [108]	✓			✓		–	–	✗
IX [9]	✓			✓		✓	–	✗
ZygOS [111]	✓				✓	✓	✓	✗
Shenango [99]	✓				✓	✓	✓	✗
Caladan [37]	✓				✓	✓	✓	✗
Shinjuku [57]		✓	✓			✓	✓	✗
RamCloud [100]		✓	✓			–	–	✗
mTCP [55]		✓		✓		✓	–	✗
TAS [65]		✓		✓		✓	–	✗
Perséphone [25]		✓	✓			–	–	✗
Demieagle	✓	✓	✓	✓	✓	✓	✓	✓

4.1.2.1 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. Additionally, a stage can be combined with the previous stage to run *inline* on the same worker. For example, NIC polling and packet processing may be combined into a single stage in the same worker. We focus primarily on two predominant models used by many existing systems [9, 25, 37, 57, 99, 138]: 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.

4.1.2.2 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. Depending on the allocation of stages to workers, there can be more than one queueing model simultaneously in use. We only consider FCFS-based processing as recent work has shown that it provides the best tail latency across workloads [83]. Although other queueing disciplines are possible, they would significantly expand the design space,

so we limit our investigation to FCFS for simplicity.

4.1.2.3 Spatial Scheduling

The final dimension of the design space is spatial scheduling, which maps workers to cores. Here we consider worker placement when different cores have non-uniform memory accesses (NUMA) or when cores share underlying physical execution units (hyperthreads). These decisions can have large impacts on the performance of the overall system, particularly at microsecond-scale.

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 4.1 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.

4.2 Benchmark Setup

To evaluate multicore architectures within the design space defined in Section 4.1, we define a benchmark suite designed to test points in the design space using different workloads and application types. This section describes workloads, experimental setup, and implementation details of our benchmark.

4.2.1 Workload Characterization

The first workload characteristic represents application types by their service times. To mimic different applications [5, 24, 93], we consider the following service-time distributions:

- **Constant:** applications that execute the same number of instructions, regardless of the request type, in order to represent the specified service time.
- **Bimodal:** applications with different processing times depending on the request type. We use two commonly studied bimodal distributions in which a fraction of the requests complete in 1 μs while the other requests complete in 100 μs [25, 57]. These service times mimic read and write-heavy workloads [135]. We model these workloads exhibiting different service times after examples found in academic and industry references [2, 5, 18, 23, 88].

Table 4.2 summarizes the service times and distributions in our benchmark. The application service times of 1, 10, and 100 μs represent ratios of application-to-network processing times of about 0.5, 5, and 50, respectively, as the network stack runs in about 2 μs.

Table 4.2: Application service times and distributions in our benchmark suite. The network-stack processing time is $\sim$2 μs for all scenarios.

Distribution	Service Time (μs)	
Constant	1, 10, 100	
Bimodal1	Group A (99.5%)	Group B (0.5%)
	$\mu_A = 1$	$\mu_B = 100$
Bimodal2	Group A (50%)	Group B (50%)
	$\mu_A = 1$	$\mu_B = 100$

The second workload characteristic is the application memory footprint. The benchmark includes tests for the following types of applications:

- **Small memory footprint:** an application that repeatedly calculates the square root of floating-point numbers. We call this application `CPU-bound`.
- **Large memory footprint:** applications that read memory buffers in different ways, with buffer sizes ranging from 20% to 80% of the LLC size. The `MEM-bound` application reads the buffer in strides, while $\texttt{MEM-bound}_{\texttt{R}}$ reads it randomly.

These applications have computational and memory access patterns that cause the cache to load and unload data differently and expose performance bottlenecks in the system, such as cache inefficiency, excessive memory access, and contention on CPU execution units.

4.2.1.1 Calibration

We calibrate each application to match target service times using a single worker and adjusting the number of instructions (*e.g.,* square root computations or memory reads) the worker executes. At the start, the worker executes a single instruction and measures the difference between the two timestamps taken before and after the instruction. If the measured time is less than the target service time, we double the number of instructions and repeat the measurement. This process continues until the time difference exceeds the target. At that point, we reduce the number of instructions by one unit. We repeat this process until the measured time is sufficiently close to the target. For example, the `CPU-bound` application requires 248 instructions to achieve a service time of 1 μs on our server. As we use a single worker to calibrate the service times, actual execution times may differ for multiple workers due to aggregated high-memory reading access on the worker buffers or execution-unit contention. We ensure that compiler optimizations do not change the application in ways that affect our setup, for example, by using `std::hint::black_box` from Rust.

4.2.2 Experimental Setup and Implementation

We evaluate the multicore architectures on a testbed consisting of two servers connected in a classic traffic generator and device-under-test configuration. Each server has two Intel(R) Xeon(R) Silver 4114 (10-cores @2.20 GHz), 128 GB of RAM, and a dual-port NVIDIA ConnectX-6 100 Gbps NIC. We disable Turbo Boost,

isolate nine physical cores from the Linux scheduler per processor (18 hyperthreads), and reserve 64 GB of hugepages for DPDK. We use Linux's `perf` stat command for collecting processor statistics.

The client is an open-loop generator that generates requests using the exponential distribution to model bursty traffic. It establishes TCP connections with the server, with the number of connections determined by the number of application workers on the server. The client sends the requests in 128-byte packets and processes the replies to measure the performance metrics, avoiding, therefore, instrumentation on the server that would interfere with the performance measurements. The client is implemented in C with DPDK, using multithreading to handle RX and TX operations separately. We run ten independent experiment rounds for 20 seconds each, with the first half serving as a warm-up period, and report the 99.9th percentile round-trip latencies [44, 72, 87, 123, 139] of applications with a confidence interval of 95%.

The server uses the Demikernel datapath OS [138] to implement the Inline and Dispatcher models described in Section 4.1. We use one spinlock for each TCP session state (*i.e.,* TCP control block) to prevent race conditions in models that allow multiple cores to update the control block, while also minimizing lock contention. Additionally, we use RSS with DPDK's `rte_flow` to deliver the packets across multiple NIC queues and ensure an equal distribution of TCP sessions between the cores. This distribution prevents imbalances in the number of TCP sessions per worker that could affect the performance evaluation of the different architectures.

4.3 Worker Assignment and Queueing Model

This section examines various worker assignments and queueing models for scheduling network and application processing, highlighting how each configuration impacts performance differently. Despite these differences, existing systems rarely

account for these factors in their design. Typically, systems aim to minimize latency and maximize throughput, often involving multiple cores to handle high request rates and achieve efficient utilization. In queueing theory, centralized scheduling tends to offer the best utilization and lowest latency by routing work to the most available worker. However, in practice, centralized scheduling faces scaling bottlenecks on multicore architectures at microsecond timescales. These limitations have led kernel-bypass systems to adopt two distinct approaches for network processing on multicore architectures.

The first approach, used in many kernel-bypass systems, is the *Inline* model, where each worker core handles both network and application processing. The set of workers can either share a single NIC queue or configure the NIC to use Receive Side Scaling (RSS) to distribute packets across per-worker NIC queues. We explore each approach in Section 4.3.1.

In the *Dispatcher* model, a dedicated worker handles incoming network traffic and dispatches each request to an available application worker. This model allows greater control over CPU time allocation between network and application tasks, flexibility in selecting CPU resources (such as scheduling on different hyperthreads or NUMA nodes), and the ability to run the application and networking stack in separate security domains. However, the dispatcher itself can become a bottleneck, capping the application's total throughput. We next delve into each approach and examine different design choices within each one.

4.3.1 Inline Model

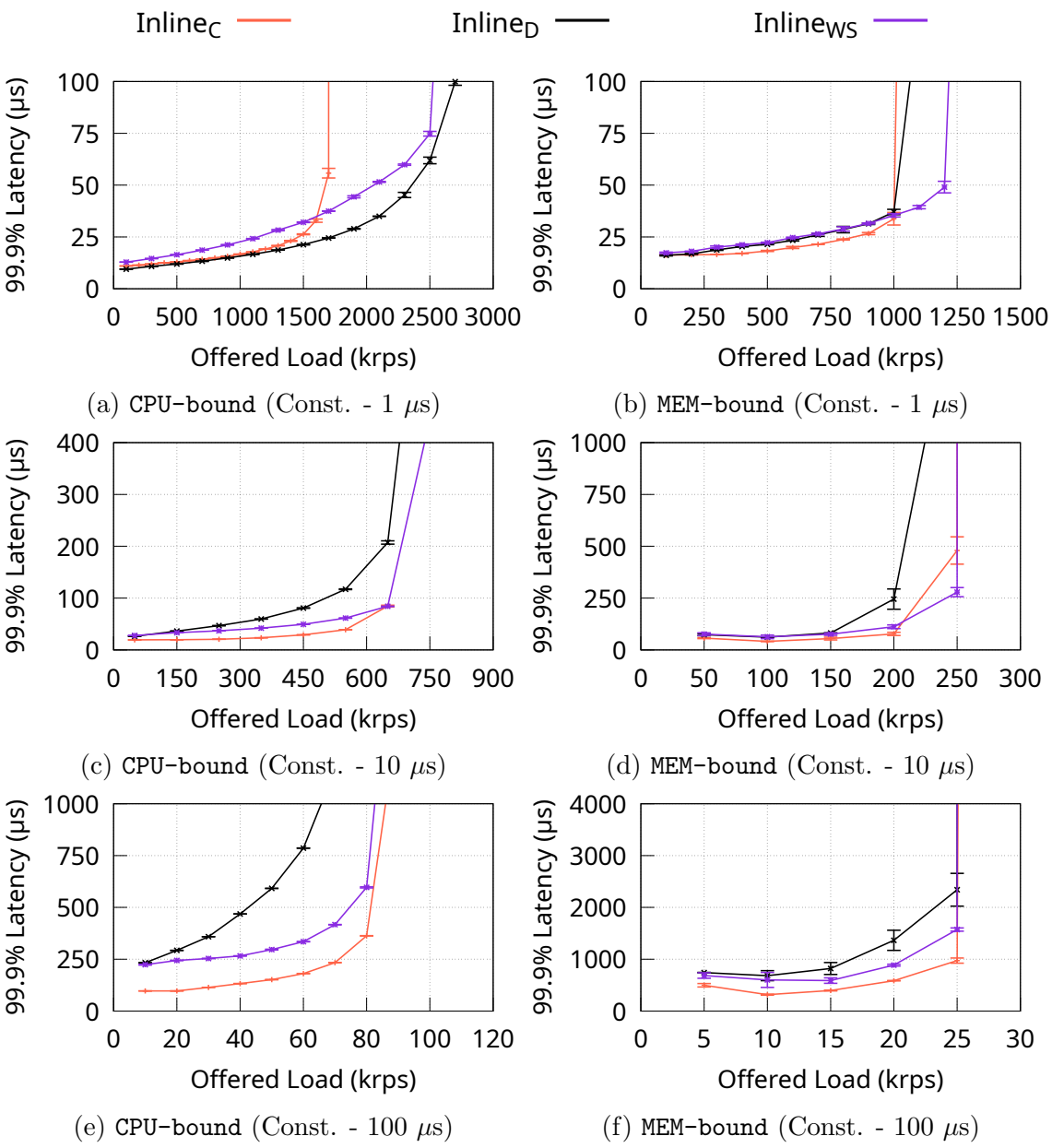
In the Inline model, the network stack and application logic run on the same worker. Packet delivery and request processing can follow different queueing models—dFCFS, cFCFS, or work-stealing (WS)—each introducing specific challenges and trade-offs that impact performance.

To scale the Inline model across many workers, systems require efficient traffic distribution methods. We explore two approaches, each supporting different queueing policies. The first uses a centralized single packet queue, polled by all workers, allowing requests to be processed in a first-come-first-serve manner (cFCFS). A spinlock protects the queue, ensuring each packet is delivered to a single core. However, since packets from the same TCP connection might be processed on different cores, the TCP connection state also requires protection via spinlocks.

Spinlocks, however, do not scale well across cores [15]. To reduce cross-core coherence traffic, the second model employs per-worker NIC queues, eliminating the need for locks. Traffic is distributed across these queues using RSS, which guarantees that packets from the same flow always reach the same queue, avoiding the need for per-connection locking. Systems like IX and Arrakis use this shared-nothing approach, leading to a distributed FCFS (dFCFS) policy. While this approach maximizes per-core throughput and efficiency, it is susceptible to head-of-line (HOL) blocking, where requests that could run on an otherwise idle worker are forced to queue while their assigned core processes a different request.

Work-stealing offers a middle ground between cFCFS and dFCFS. In this model, each worker has its own network queue but can steal work from others when idle. This alleviates load imbalances and helps reduce tail latency. To enable stealing and allow multiple cores to process the same flow, spinlocks protect each queue and individual connection states. Unlike the centralized model, these spinlocks are typically acquired by the same core, making the operation significantly cheaper than acquiring a spinlock from a different core's cache. As a result, work-stealing scales more effectively than cFCFS.

4.3.1.1 Inline Model Evaluation



(a) `CPU-bound` (Const. - 1 μs) (b) `MEM-bound` (Const. - 1 μs)

(c) `CPU-bound` (Const. - 10 μs) (d) `MEM-bound` (Const. - 10 μs)

(e) `CPU-bound` (Const. - 100 μs) (f) `MEM-bound` (Const. - 100 μs)

Figure 4.1: The Inline model results running `CPU-bound` and `MEM-bound` applications using the constant service times of 1, 10, and 100 μs, where light red, black, and purple lines represent cFCFS, dFCFS, WS, respectively.

We start with a simple and well studied experiment where we measure the tail latency as a function of the offered load to an application performing synthetic work for each request. We vary the service times used to show how the ideal queueing

policy varies by workload.

We evaluated the Inline model with eight workers using two synthetic applications: `CPU-bound` and `MEM-bound`. The `MEM-bound` workers access private buffer regions, sized to 80% of the LLC, which generates DRAM traffic when multiple workers are active. We tested three queueing models: cFCFS, dFCFS, and work-stealing (WS), with service times of 1, 10, and 100 μs for both applications.

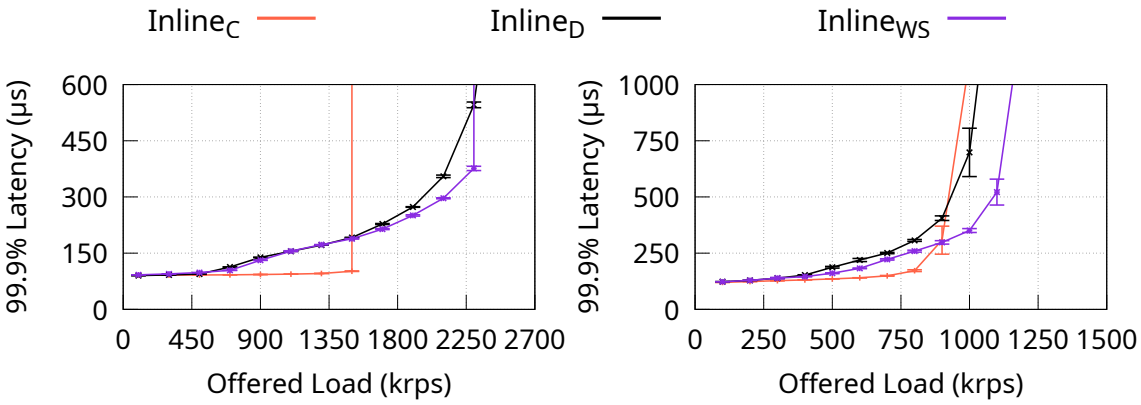
Figure 4.1 shows that, as expected, cFCFS consistently achieves the lowest tail latency of each approach. However, with service times of 1 μs and 10 μs the spinlock prevents the application from achieving full throughput. At 1 μs, dFCFS is able to scale to full throughput, and because of the small service times, it does not suffer badly from HOL blocking. WS also scales to full throughput, but with increased latency due to synchronization overheads. However, at 10 μs service times, dFCFS increasingly suffers from HOL blocking and its tail latency worsens, leading to WS achieving lower tail latency than dFCFS and higher throughput than cFCFS. At 100 μs service times, the cost of spin locking the centralized queue is small enough that cFCFS can achieve full throughput, with significantly lower latency than the other two approaches.

Finding 1: *cFCFS is an optimal scheduling policy for tail latency [72, 83], but the overhead of inter-core coordination (i.e., locking) dominates when service times are low. At lower service times, approaches that sacrifice perfect fairness (dFCFS and WS) achieve higher throughputs.*

Finding 2: *WS effectively mitigates HOL blocking without creating new bottlenecks, but incurs non-trivial overheads that can be avoided when service times are particularly low or high.*

Despite the initial calibration for the `MEM-bound` application, multiple buffers, the strided access, and the buffer size result in higher service times due to increased cache misses and memory accesses. Figures 4.1b and 4.1f show that the actual

service times differ significantly from the 1 and 100 μs that were initially calibrated. In both cases, however, cFCFS consistently outperforms the distributed models. As the service time increases to 100 μs, the performance gap between the centralized and distributed models becomes more pronounced, with cFCFS outperforming the others (see Figure 4.1f), similar to the `CPU-bound` application.



(a) `CPU-bound` *Bimodal1* (99.5 – 1, 0.5 – 100)

(b) `MEM-bound` *Bimodal1* (99.5 – 1, 0.5 – 100)

(c) `CPU-bound` *Bimodal2* (50 – 1, 50 – 100)

(d) `MEM-bound` *Bimodal2* (50 – 1, 50 – 100)

Figure 4.2: The Inline model results running `CPU-bound` and `MEM-bound` applications using the bimodal distributions, respectively, where light red, black, and purple lines represent cFCFS, dFCFS, WS, respectively.

We also evaluate the performance of the Inline model using the bimodal distributions in Table 4.2, using the same `CPU-bound` and `MEM-bound` applications. These distributions exhibit a dual-peaked pattern, representing two types of

requests: short (1 μs) and long (100 μs) service times, which are common in datacenter applications.

Figure 4.2a shows the latency results for the *Bimodal1* distribution (99.5% short requests, 0.5% long requests) applied to the `CPU-bound` application. In this case, cFCFS outperforms the distributed queueing models until it reaches its capacity limit, where the single queue can no longer handle all incoming packets. With the significant separation between the short (99.5% at 1 μs) and long (0.5% at 100 μs) requests, WS mitigates HOL blocking, resulting in better tail latencies. With the *Bimodal2* distribution (50% short, 50% long requests) for `CPU-bound`, the single queue of cFCFS does not become a bottleneck, allowing cFCFS to outperform both distributed queueing models, as shown in Figure 4.2c.

For the `MEM-bound` application, which has a large memory footprint, the cFCFS model does not experience the single-queue bottleneck seen with `CPU-bound` (Figure 4.2a). Consequently, cFCFS achieves the best tail latencies up to 800 krps for the *Bimodal1* distribution and across all evaluated offered loads for *Bimodal2*, as shown in Figures 4.2b and 4.2d.

Similarly to `CPU-bound`, WS effectively handles higher loads in the *Bimodal1* scenario by mitigating HOL blocking.

Finding 3: *High variability in service times favors centralized queueing models which reduce HOL blocking, leading to lower latencies.*

The Inline model presents some drawbacks, such as inefficient utilization of multicore systems, as both the network stack and application are bound to a single worker, limiting load distribution. It reduces flexibility in resource allocation, making it impossible to prioritize network stack or application during changing workloads. For example, when the application's service time is high, the network stack runs less frequently, which can lead to packet drops and performance degradation as TCP must buffer out-of-order packets before delivering them to the

application. Also, running the network stack and the application sequentially can reduce cache efficiency, as one task may replace the cached data of the other one.

4.3.2 Dispatcher Model

In the Dispatcher model, the network stack and application logic run on separate workers, enabling independent scaling of both tasks based on workload demands. This separation allows flexible allocation of resources—for example, dedicating more workers to network processing in cases where TCP handling takes longer than application processing, which is common in microservice-based systems. By isolating network and application tasks on different workers, cache locality is improved, boosting overall performance.

The number of dispatchers in this model can vary from 1 to N, where each dispatcher handles both network packet reception and the dispatching of requests to application workers. A dispatcher polls its NIC queue, runs the network stack, and forwards application requests to an idle worker in its pool of workers. After completing the request, the application workers send the response back to the same dispatcher, which transmits the reply over the network.

To scale the system as the load increases, additional dispatchers can be introduced, each with its own NIC queue and set of application workers. Traffic is partitioned among dispatchers using RSS, allowing each dispatcher to be responsible for maintaining the TCP state for its own flows without synchronizing with others.

The Dispatcher model allows for flexible configurations of CPU cores and hyperthreads. Figure 4.3 illustrates several possible setups using eight physical cores with two hyperthreads each. With one hyperthread left idle, we consider three core allocation options: *(i)* assign one core to the network stack and the remaining seven to the application (Figure 4.3a), *(ii)* assign two cores to the network stack and six to the application (Figure 4.3b), or *(iii)* allocate four cores to both the network stack

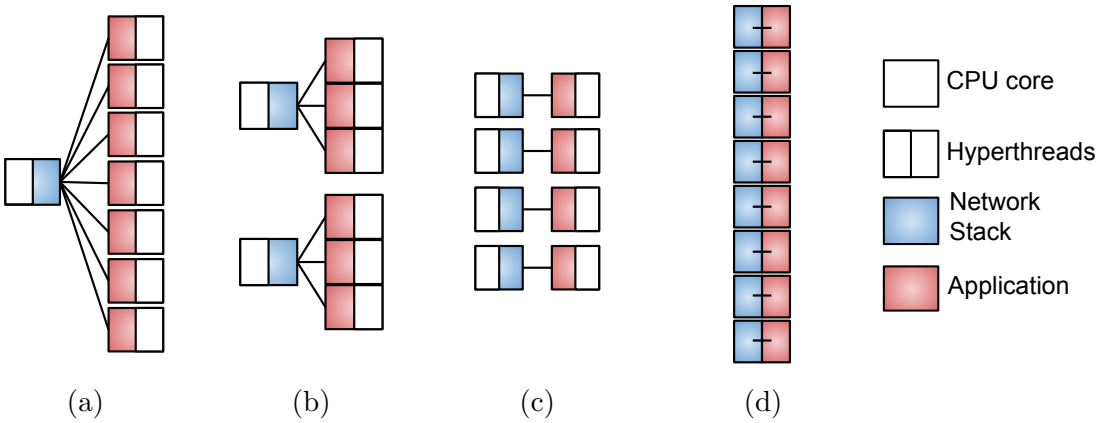
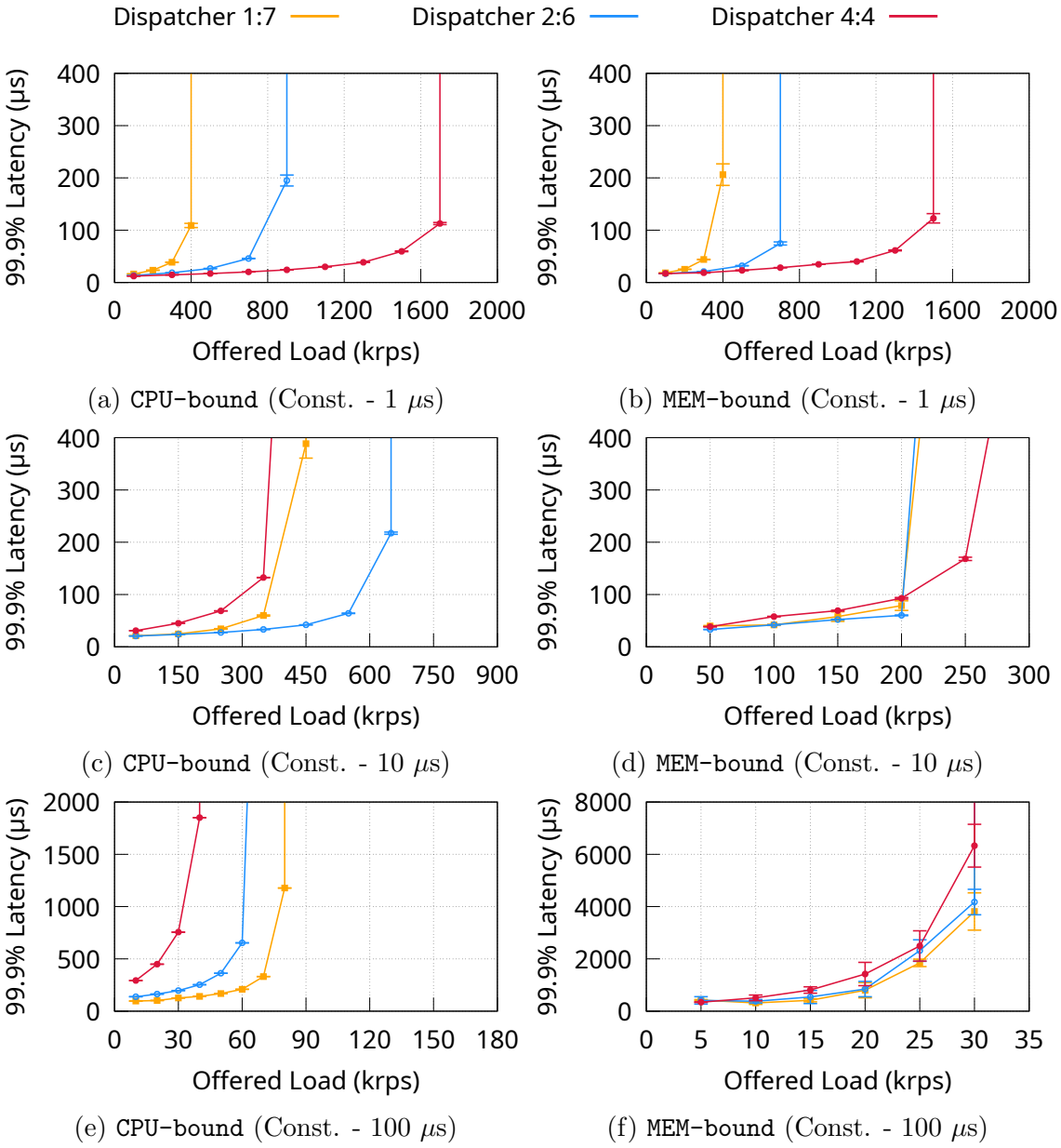


Figure 4.3: Different arrangements for the Dispatcher model with eight cores. Using a single hyperthread per core, the configurations are: (a) 1+7, (b) 2+6, and (c) 4+4, where the first and second numbers represent the quantity of network-stack and application workers, respectively. With both hyperthreads active on a core, we have (d) one runs the network stack while the other runs the application.

and application (Figure 4.3c). These configurations are labeled "1+7," "2+6," and "4+4," respectively. Additional configurations are possible when both hyperthreads are used. Figure 4.3d illustrates a setup where one hyperthread runs the network stack and the other runs the application on the same core.

4.3.2.1 Dispatcher Model Evaluation

In this section, we evaluate the Dispatcher model using the `CPU-bound` and `MEM-bound` applications with eight workers and the service times in Table 4.2, as in Section 4.3.1.1. We evaluate the performance of the model using 1, 2, and 4 workers for network-stack processing, corresponding to Figures 4.3a, 4.3b, and 4.3c, respectively. We label these configurations as Dispatcher 1:7, Dispatcher 2:6, Dispatcher 4:4. This evaluation presents tail latencies for constant service times of 1, 10, and 100 μs and the bimodal distributions. In all scenarios, we use a single hyperthread from each core and leave its sibling idle. We investigate the performance impacts of hyperthreading in Section 4.4.



(a) `CPU-bound` (Const. - 1 μs)

(b) `MEM-bound` (Const. - 1 μs)

(c) `CPU-bound` (Const. - 10 μs)

(d) `MEM-bound` (Const. - 10 μs)

(e) `CPU-bound` (Const. - 100 μs)

(f) `MEM-bound` (Const. - 100 μs)

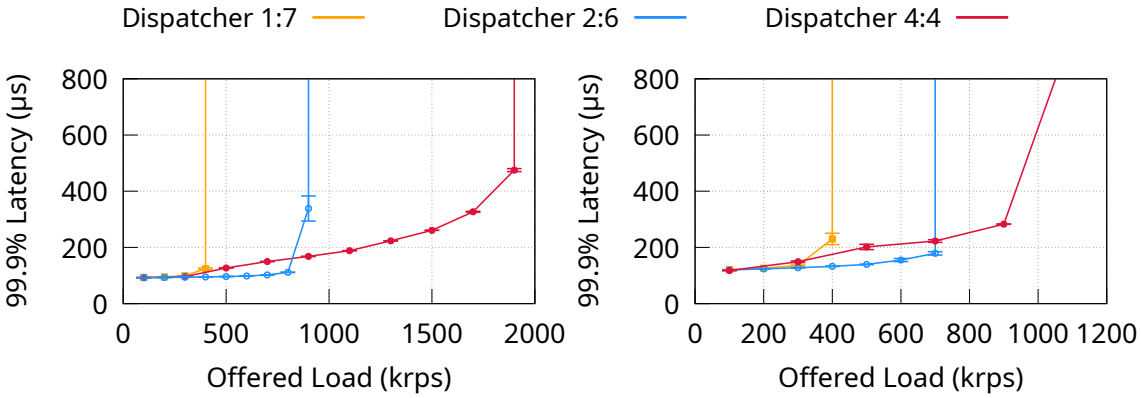
Figure 4.4: The Dispatcher model results running `CPU-bound` and `MEM-bound` applications using the constant service times of 1, 10, and 100 μs, where orange, blue, and red lines represent Dispatcher 1:7, Dispatcher 2:6, and Dispatcher 4:4, respectively.

We can balance the allocation of workers between the network stack and application depending on the workload, emphasizing the network stack's critical role in performance. Figure 4.4a shows the tail latencies for the different configurations when the `CPU-bound` application runs with a service time of 1 μs. In this experiment,

we observe that the network stack becomes the bottleneck as the offered load increases. For example, the Dispatcher 1:7 configuration reaches its processing limit at 400 krps because a single network-stack worker cannot handle the incoming packet rate, leaving the application workers idle.

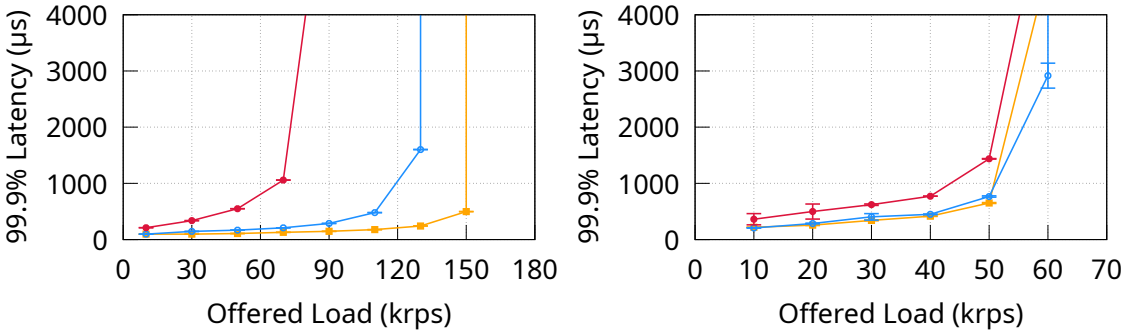
Finding 4: *In the Dispatcher model, the network stack becomes the bottleneck at high packet rates when the application service time is low.*

Conversely, when the service time increases, the bottleneck shifts to the application workers. Compared to the 1 μs service time in Figure 4.4a, the bottleneck shifts from the network stack to the application as service times increase. Figure 4.4c shows the tail latency with a 10 μs service time, where Dispatcher 2:6 outperforms the other two configurations, as the single network-stack worker in Dispatcher 1:7 becomes a bottleneck. In contrast, for a 100 μs service time, Dispatcher 1:7 outperforms the other two configurations, as shown in Figure 4.4e. By decoupling the network-stack processing from the application, the Dispatcher model allows the allocation of more workers to process the application requests without interference from the network-stack processing.



(a) `CPU-bound` *Bimodal1* (99.5 – 1, 0.5 – 100)

(b) `MEM-bound` *Bimodal1* (99.5 – 1, 0.5 – 100)



(c) `CPU-bound` *Bimodal2* (50 – 1, 50 – 100)

(d) `MEM-bound` *Bimodal2* (50 – 1, 50 – 100)

Figure 4.5: The Dispatcher model results running `CPU-bound` and `MEM-bound` applications using the bimodal distributions, respectively, where orange, blue, and red lines represent Dispatcher 1:7, Dispatcher 2:6, and Dispatcher 4:4, respectively.

We also evaluate the Dispatcher model using the `MEM-bound` application, which introduces significant cache pressure due to its large memory footprint (80% of LLC size) and 63-byte stride iterations. In this case, memory accesses span different cache lines at a 63/64 rate, resulting in 63 cache misses and 1 cache hit for every 64 memory accesses. This evaluation shows that under high cache pressure, system performance is primarily constrained by memory access patterns rather than the ratio of application to network-stack workers. Figure 4.4b presents the tail latency with a 1 μs service time and shows that the results are similar to those from the `CPU-bound` application. While the different configurations perform similarly at low

service times, higher service times of 10 and 100 μs reveal the impact of the large memory footprint, causing more memory accesses and capping throughput at 250 krps and 30 krps, respectively, as shown in Figures 4.4d and 4.4f. These findings suggest that in memory-bound applications with high service times, the Dispatcher model's performance is primarily constrained by cache pressure, indicating that optimizing cache and memory access is more critical than adjusting the ratio of application-to-network workers.

Finding 5: *In the Dispatcher model, optimizing cache and memory access is more important than adjusting the ratio of application-to-network workers for memory-intensive applications with high service times.*

To account for different request types and access distributions, we evaluate the Dispatcher model using the bimodal distributions in Table 4.2 for both `CPU-bound` and `MEM-bound`. In the *Bimodal1* distribution (99.5% short requests), the number of network-stack workers significantly impacts tail latency for both applications, as shown in Figures 4.5a and 4.5b. In the *Bimodal2* distribution (50% short, 50% long requests), the number of application workers plays a more critical role, particularly for the `CPU-bound` application (Figure 4.5c). However, for the memory-intensive `MEM-bound` application, as shown in Figure 4.5d, the 1+7 and 2+6 configurations exhibit similar performance, as the number of application workers is the primary factor. The 4+4 configuration, with fewer application workers, results in the highest tail latencies, as the network-stack workers are underutilized while the application workers are insufficient to handle the workload.

Finding 6: *In the Dispatcher model, as service time increases, the number of application workers becomes the bottleneck, and increasing their number while keeping enough network-stack workers reduces latency.*

While the Dispatcher model allows independent scaling of network-stack and application workers, it also presents some shortcomings. First, inter-worker

communication, which involves different cores or hyperthreads, is required to route requests to application workers and receive their replies. Second, it is essential to minimizing delays in the handoff process both from the moment a request is ready after TCP stack processing to when it reaches the application worker, and from when the application worker completes the reply to when the network-stack worker prepares the outgoing packets. This optimization is especially important for latency-sensitive applications, where even small transition delays can have a significant impact on performance.

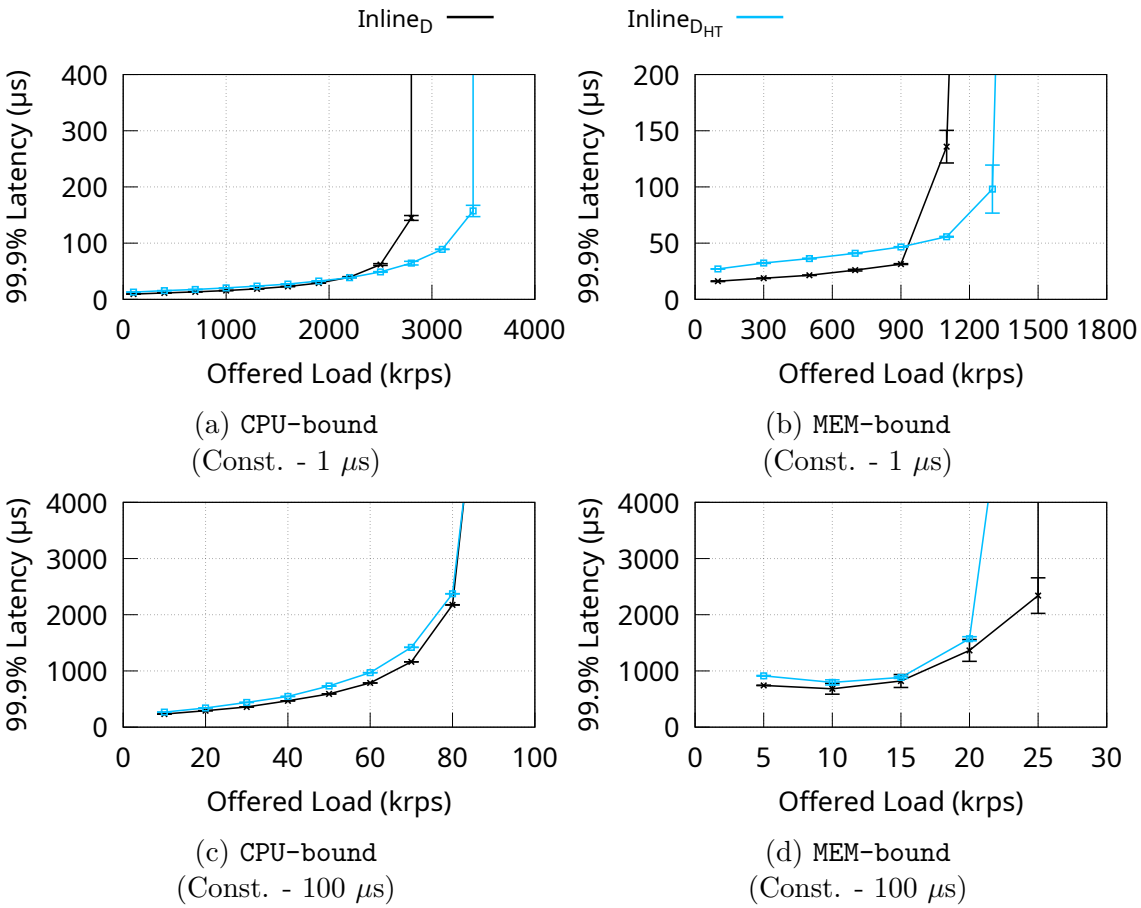
4.4 Spatial Scheduling and Co-location

Hardware elements like caches, hyperthreads, and NUMA nodes can significantly influence the performance of network applications, particularly when the response time must be in the order of a few microseconds. Cache misses, for instance, can introduce delays, revealing the trade-offs between cache size, data access patterns, and processing efficiency. Hyperthreading adds another complexity dimension, as sibling hyperthreads share the core's resources (*e.g.,* execution engine and caches), which may negatively impact the processing latency, especially at the tail. NUMA nodes have private resources (*e.g.,* L1, L2, and L3) and can access local memory much faster than the remote ones [80]. Striking a balance on where to place the different components of a network application is challenging. For instance, using the additional resources (*e.g.,* caches) of a remote node may pay off the price of crossing the nodes' interconnect.

4.4.1 Scheduling with Hyperthreads

With hyperthreading, the operating system treats each physical core with two hyperthreads as two logical processors, allowing two tasks to be scheduled on the same core, one per hyperthread. In the Inline model, where both the network

stack and the application run on the same worker, the hyperthreads can either operate independently as two workers or one hyperthread can be idle, allowing its sibling to use the core's full resources. Since memory access patterns also influence hyperthreading performance, we evaluate both placements for the Inline model using the `CPU-bound` and `MEM-bound` applications.



(a) `CPU-bound` (Const. - 1 μs)

(b) `MEM-bound` (Const. - 1 μs)

(c) `CPU-bound` (Const. - 100 μs)

(d) `MEM-bound` (Const. - 100 μs)

Figure 4.6: The Inline model (dFCFS) results using both hyperthreads on each CPU core running `CPU-bound` and `MEM-bound` applications, where black and blue lines represent making the sibling hyperthread idle and using both hyperthreads on each CPU core, respectively.

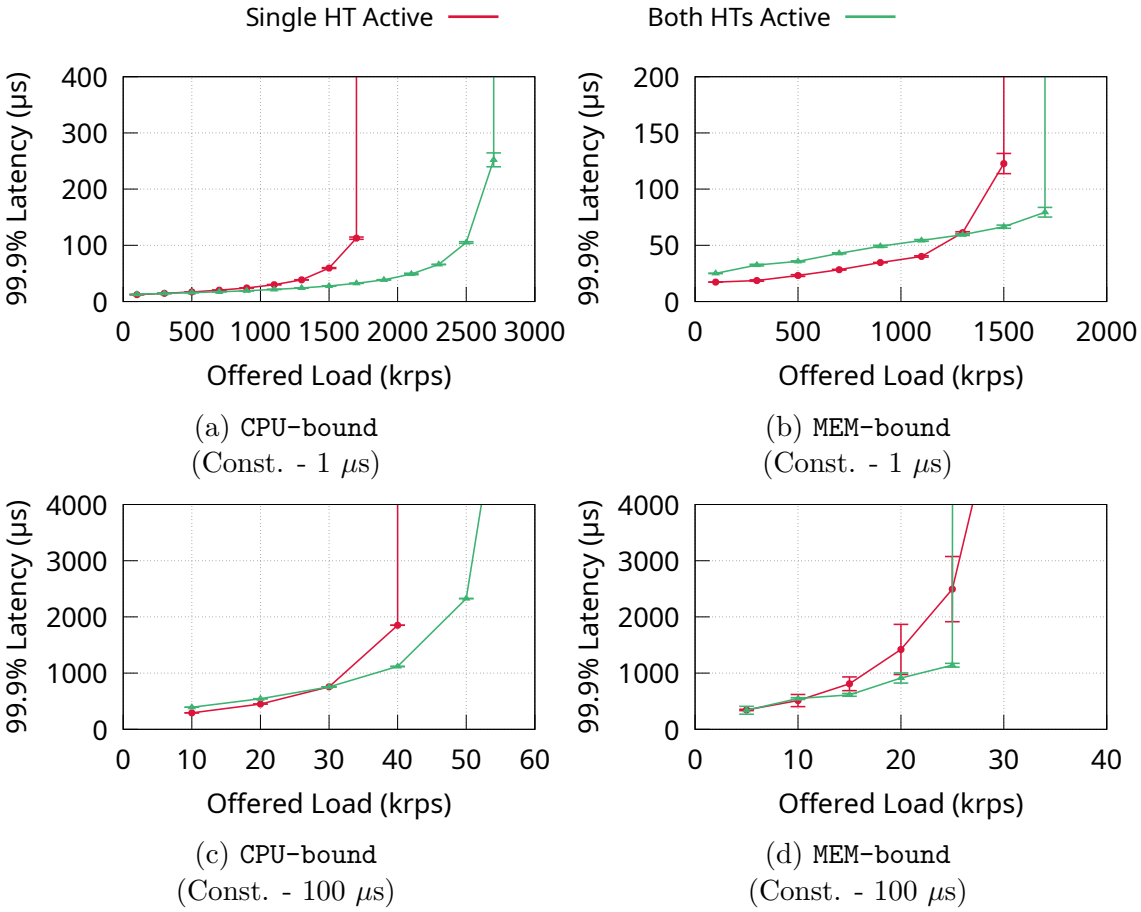
We evaluate the Inline model with dFCFS using one or both hyperthreads on the same core across eight CPU cores (*i.e.,* 8 or 16 workers). For the `CPU-bound` application with a 1 μs service time, Figure 4.6a shows that running the workers on both hyperthreads (Inline$_{D_{HT}}$) results in slightly higher tail latencies at low loads

but about 25% higher throughput than using a single worker on one hyperthread (Inline_D), allowing the system to handle a higher offered load. When the service time increases to 100 μs, the tail latency remains higher with both hyperthreads active and there is no significant difference in throughput as seen in Figure 4.6c.

For the `MEM-bound` application, similar behavior is observed with the 1 μs service time: two hyperthreads increase tail latency at low loads but also increase throughput, as depicted in Figure 4.6b. However, when service time increases to 100 μs, having the sibling hyperthread idle improves both latency and throughput (Figure 4.6d). Leaving a hyperthread idle improves performance by reducing competition for shared resources within the CPU core (*e.g.,* cache), allowing the active hyperthread to fully use these resources without interference.

For the memory-intensive applications `MEM-bound`, additional hyperthreads exacerbate memory accesses and cache misses because each worker uses a buffer that occupies 80% of the LLC. These frequent memory accesses, resulting from inefficient cache usage, reduce the system's capacity to handle higher loads. Consequently, the combined effects of execution unit contention, reduced cache efficiency, and increased memory accesses lead to lower throughput for the `MEM-bound` application with 16 hyperthreads compared to using just eight.

Finding 7: *In the Inline model with dFCFS, using both hyperthreads of a CPU core increases the latency when the offered load is low for both types of applications.*



(a) `CPU-bound` (Const. - 1 μs)

(b) `MEM-bound` (Const. - 1 μs)

(c) `CPU-bound` (Const. - 100 μs)

(d) `MEM-bound` (Const. - 100 μs)

Figure 4.7: The Dispatcher model results using both hyperthreads on each CPU core running `CPU-bound` and `MEM-bound` applications, where red and green lines represent making the sibling hyperthread idle and using both hyperthreads on each CPU core, respectively.

To evaluate the Dispatcher model with hyperthreading, we use the configurations shown in Figure 4.3: *(i)* Figure 4.3c, where the sibling hyperthread is idle, with four workers dedicated to the network stack and four to the application and *(ii)* Figure 4.3d, where each core runs both the network stack and application on separate hyperthreads.

Figures 4.7a and 4.7c show tail latencies for the `CPU-bound` application with 1 and 100 μs service times, respectively. For the 1 μs case, using both hyperthreads (green line) results in higher throughput than leaving the sibling hyperthread idle (red line), as shown in Figure 4.7a. Even with a 100 μs service time, running both

network stack and application on each core yields the highest throughput. However, using only a single hyperthread per core (red line) at lower offered loads results in the lowest tail latency, as depicted in Figure 4.7c.

For the `MEM-bound` application with a 1 μs service time, using both hyperthreads increases throughput, but the difference between running the network stack on both hyperthreads or not (red and green lines) is considerable at low offered load, as shown in Figure 4.7b. At 100 μs service time, the large memory footprint severely limits the throughput, regardless of hyperthreading or the Dispatcher model configuration, capping it at around 25 krps. However, running both the network stack and the application on the same core (green line) produces the better latency results, as seen in Figure 4.7d.

Finding 8: *Running CPU-intensive applications with lower service times on both hyperthreads improves throughput in both Inline and Dispatcher models but increases tail latency at low offered loads.*

4.5 Scheduling with NUMA Nodes

We evaluate the impact of scheduling application workers across local and remote NUMA nodes on throughput and latency using two memory-intensive applications. The first is $\texttt{MEM-bound}_{\texttt{R}}$, where each worker accesses its buffer randomly, and the second is `MEM-bound`, where each worker iterates over its buffer with a 63-byte stride.

The purpose of using random and strided memory access patterns is to stress 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:

- All application workers run on the local NUMA node where the NIC is attached (labeled as “L / L”);

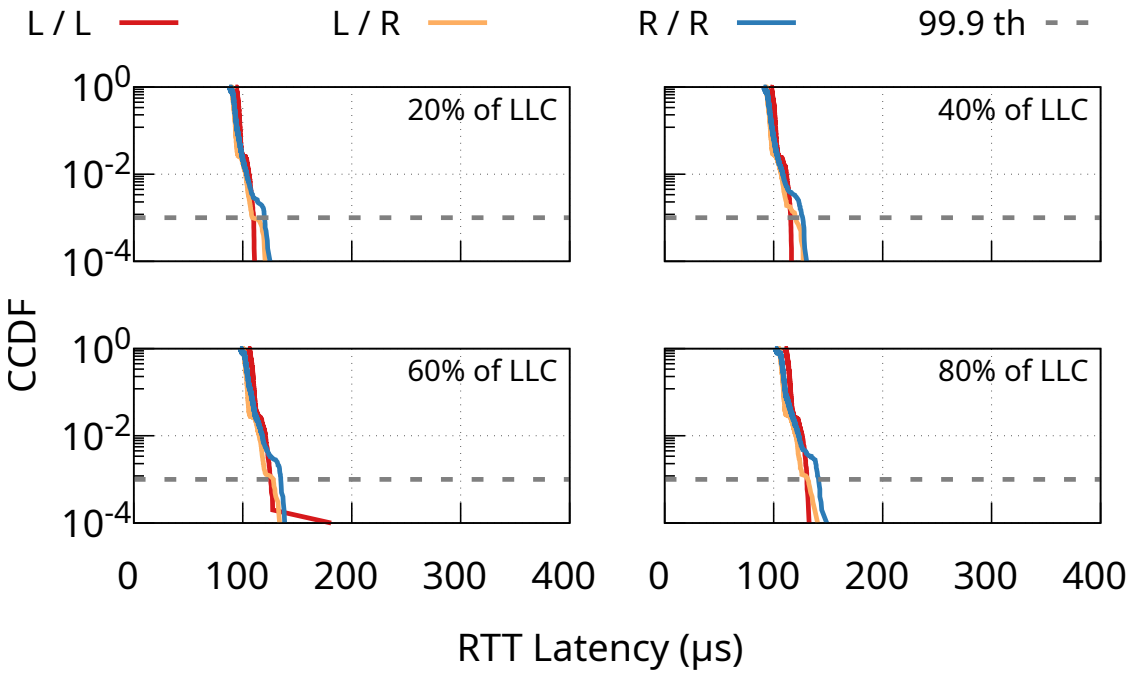


Figure 4.8: The Dispatcher model results using both NUMA nodes running `MEM-bound`$_R$ application, where the red, orange, and blue lines represent using only the local NUMA node, both local and remote NUMA nodes, and only the remote NUMA node, respectively.

- Half of the application workers run on the local NUMA node, and the other half in the remote NUMA node (labeled as "L / R");
- All application workers run on the remote NUMA node, where the NIC is not attached (labeled "R / R")

We evaluate both applications with the buffer size of each worker varying as a function of the LLC size. The workload is fixed at 10 krps, with a constant number of iterations over the buffer calibrated to take 100 μs. Note that the actual service time may be longer depending on the memory-access pattern and cache pressure.

Figure 4.8 shows the CCDF (Complementary Cumulative Distribution Function) latency for `MEM-bound`$_R$ with eight application workers and buffer sizes of 20%, 40%, 60%, and 80% of the LLC size. The results indicate no significant differences in performance across different NUMA node configurations. Additionally, there is minimal LLC contention, even when each buffer occupies 80% of the LLC (Figure

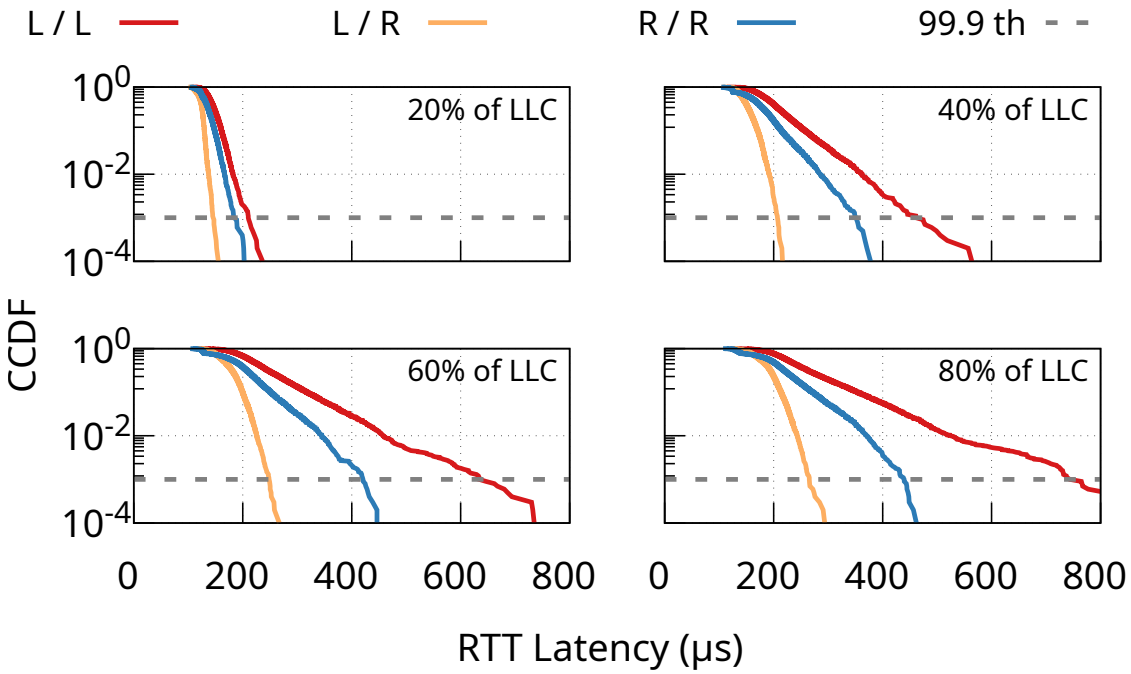
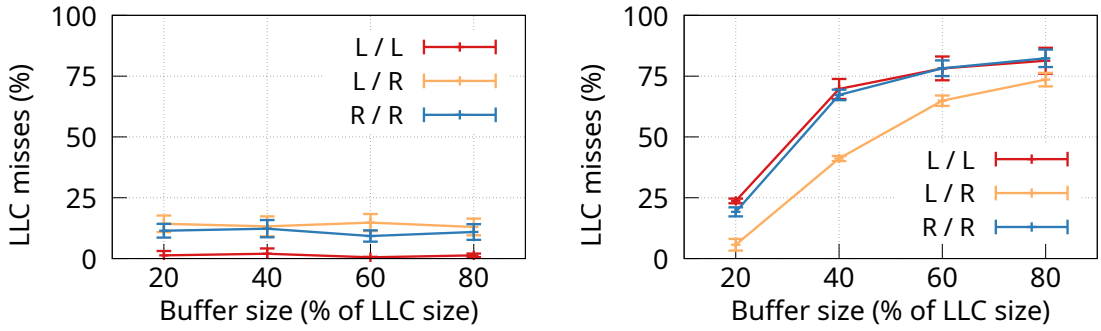


Figure 4.9: The Dispatcher model results using both NUMA nodes running `MEM-bound` application, where the red, orange, and blue lines represent using only the local NUMA node, both local and remote NUMA nodes, and only the remote NUMA node, respectively.

4.10a).

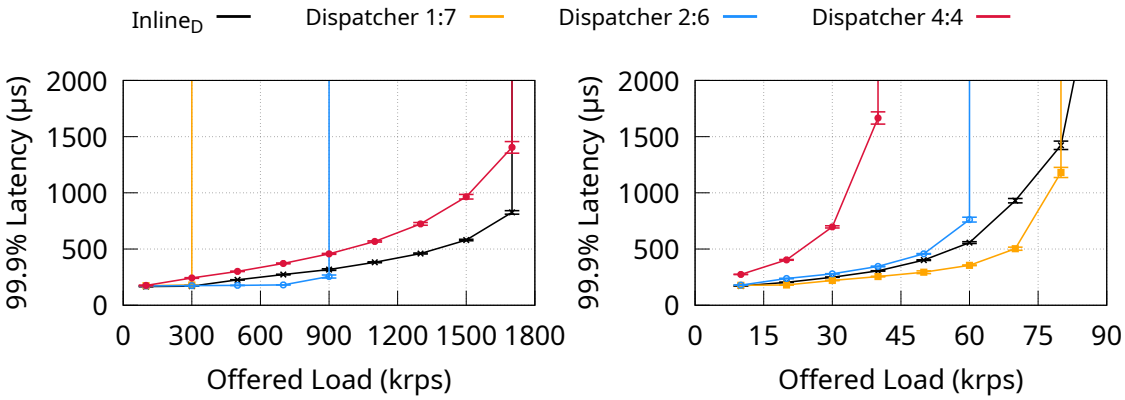
Finding 9: *Random iteration over the memory buffers makes no significant difference in latency, regardless of application buffer size or NUMA node configuration.*

In contrast, the `MEM-bound` application, which was tested using the same parameters, yields different results. Figure 4.9 shows the latency for the same buffer sizes used in the previous experiment. 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 (Figure 4.10b). 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.



(a) `MEM-bound`$_R$ (b) `MEM-bound`

Figure 4.10: LLC misses for `MEM-bound`$_R$ and `MEM-bound` applications, where we vary the buffer size and red, orange, and blue lines represent scheduling application workers in local NUMA node, both NUMA nodes, and remote NUMA node, respectively.



(a) (99.5 – GET, 0.5 – SCAN) (b) (50 – GET, 50 – SCAN)

Figure 4.11: RocksDB results comparing the Inline and Dispatcher models for different GET/SCAN request proportions: (a) 99.5% / 0.5% and (b) 50% / 50%.

Finding 10: *Strided memory access with large strides shows significant latency improvements when utilizing both NUMA nodes, particularly due to reduced cache misses.*

4.6 Real Application Evaluation

In Sections 4.3 and 4.4, we evaluate the performance of the Inline and Dispatcher models in controlled settings where the behavior of the applications is well-defined.

These settings allow us to simulate resource contention and memory access patterns to identify architectural bottlenecks. To ensure that our results are applicable to real-world scenarios, we integrate RocksDB [28], an embedded high-performance key value database commonly used in storage systems and real-time data processing, into *Demieagle* and evaluate its performance under various workloads.

In our experiments, we send two types of requests to RocksDB: **GET** and **SCAN**. GET requests, which we classify as "short," typically complete in $\sim$1 μs. SCAN requests, classified as "long," take $\sim$200 μs to finish. The database contains 10,000 keys. A GET request randomly accesses one of the existing keys. SCAN operations, on the other hand, sequentially traverse all keys, resulting in longer service times.

We evaluate the Inline model with dFCFS and various configurations of the Dispatcher model with two distributions: 99.5% short and 0.5% long requests (as in *Bimodal1*) and an even split of 50% short and 50% long requests (as in *Bimodal2*).

Figure 4.11a illustrates the tail latency for Inline_C, Dispatcher 1:7, Dispatcher 2:6, and Dispatcher 4:4 models under a 99.5% short and 0.5% long request distribution. Similar to our synthetic experiments, the number of network-stack workers is the primary bottleneck in the Dispatcher model. However, until the protocol stack limit is reached, the Dispatcher model (2+6 configuration) achieves lower tail latencies than the Inline model. The 4+4 configuration, despite exhibiting the worst tail latency, reaches the highest load, matching the performance of the Inline model. Notably, dFCFS achieves the lowest tail latency at higher loads during this experiment.

In contrast, Figure 4.11b shows the results for an even distribution of GET and SCAN requests. Here, the number of application workers becomes the dominant factor affecting performance due to the high volume of SCAN requests. Consequently, the bottleneck shifts from the network stack to the application workers. In this scenario, the Dispatcher model (1+7 configuration) delivers better

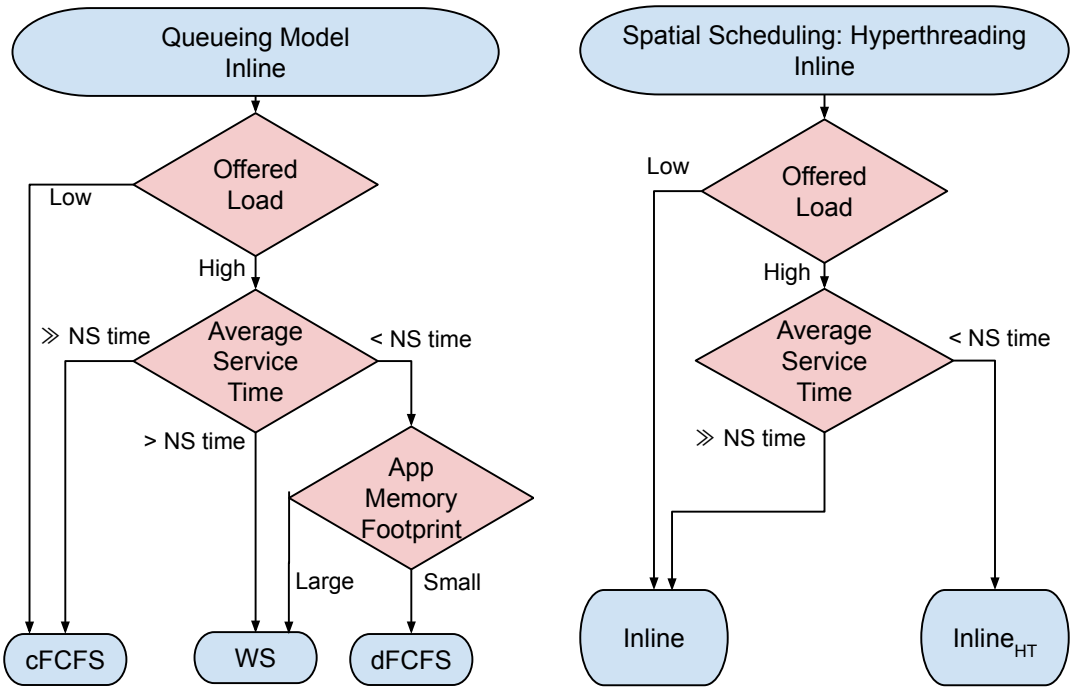
tail latencies than the Inline model across the board.

Evaluating real-world applications like RocksDB introduces uncontrolled factors, such as application locks and memory-access patterns. However, the findings obtained with the synthetic applications and reported in Sections 4.3.1 and 4.3.2 match the results observed with RocksDB, attesting to the representativeness of our synthetic applications.

4.7 Guidelines

Figures 4.12a and 4.12b provide flowcharts to guide the selection of the models discussed in Section 4.3.1. Figure 4.12a guides the selection of the queueing model (cFCFS, dFCFS, or WS) based on three key factors: Offered Load (low or high), Average Service Time (relative to the network stack's time), and Application Memory Footprint (small or large). Figure 4.12b adds hyperthreading to these two factors to decide when both hyperthreads should be used in a core. The figures referenced in the labels of each flowchart show the experimental results that support each decision.

Figures 4.13a, 4.13b, and 4.13c cover the models from Section 4.3.2, summarizing key findings for the Dispatcher model. Figure 4.13a provides guidelines for spatial scheduling with NUMA based on the Memory Access Pattern (strided or random). Figure 4.13b offers hyperthreading guidelines focused on the Offered Load (low or high). Figure 4.13c illustrates the decisions based on the number of network-stack and applications worker, considering Average Service Time (relative to the network stack's time), Time Variability (constant or variable), and Offered Load.



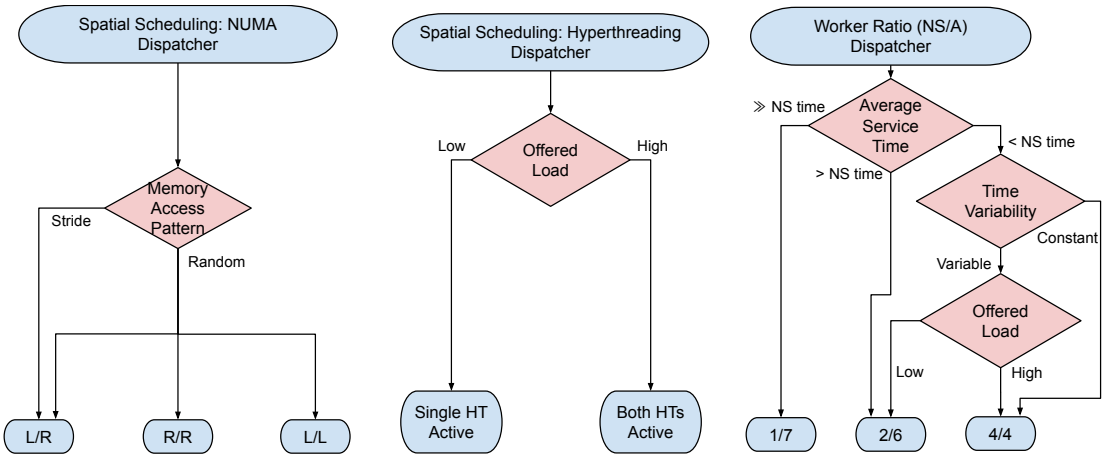
(a) Based on Figures 4.1 and 4.2. (b) Based on Figure 4.6.

Figure 4.12: Guidelines for selecting a model for the Inline model based on the tail latencies for the cases evaluated in Sections 4.3 and 4.4, where *NS* means Network Stack and *A* means Application.

4.8 Summary

In this chapter, we analyze the trade-offs associated with different multicore architectures tailored for microsecond-scale network applications. We classify these architectures and introduce a new benchmark suite that explores various points in the design space, offering a structured way to evaluate each approach. The benchmark framework, *Demieagle*, is designed to support and test these classified architectures, facilitating direct comparisons between different multicore architectures and scheduling policies across various synthetic and real-world workloads.

Using *Demieagle*, we extensively evaluate all classified architectures, including synthetic workloads and a real-world application, highlighting key performance characteristics and limitations. Our three main contributions are *(i)* a classification



(a) Based on Figures 4.8 and 4.9.

(b) Based on Figure 4.7.

(c) Based on Figures 4.4 and 4.5.

Figure 4.13: Guidelines for selecting a model for the Dispatcher model based on the tail latencies for the cases evaluated in Sections 4.3 and 4.4, where *NS* means Network Stack and *A* means Application.

of multicore architectures, *(ii)* the design and implementation of *Demieagle*, and *(iii)* a detailed evaluation to provide valuable insights for system developers. The findings from our evaluation enable better decision-making when optimizing network stacks and applications for microsecond-scale environments, addressing the need for efficient resource utilization, scalability, and low latency.

Chapter 5

Dynamic Scaling of Stateful Network Functions

In this chapter, we apply the ideas from the previous chapter regarding multicore architectures for fast packet processing and present the design, correctness analysis, and performance evaluation 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.

5.1 *Dyssect*

Dyssect manages the workload of stateful VNF (Virtual Network Function) 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 (see the results we show in Figure 1.4), 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 (Section 5.1.1). 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 (Section 5.1.2). 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 (Section 5.1.3).

Certain states in network functions, such as the NAT pool of addresses and ports, have a global scope and cannot be partitioned per flow through sharding. Since *Dyssect* depends on sharding, it only supports network functions with *per-flow state*. In this case, 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 for communication and synchronization across instances.However, some systems [39, 40, 56, 112, 113, 121, 132] deal with network functions (*e.g.,* Bro [106], PRADS [110], Snort [122], Suricata [126]) in which multiple instances share the network function state. In this case, these systems require synchronization and coordination mechanisms to ensure consistency and avoid conflicts to provide scaling of stateful network functions. *Dyssect* supports network functions where 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 and load balancers [132, 140]. *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 [7, 63, 68, 116].

5.1.1 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 $S \leq E$, 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 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.

5.1.1.1 State Initialization

A recent study shows that state lookup represents a significant cost for packet processing in software [130]. 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 when calling other *Dyssect* functions.

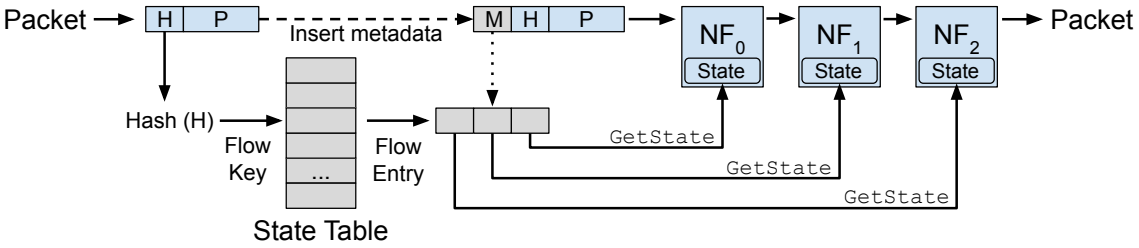


Figure 5.1: State management in *Dyssect*.

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.

5.1.1.2 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 an 8-byte pointer to each NF's state. 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 a reference of the network function's state by dereferencing the packet metadata pointer without doing new lookups. Figure 5.1 shows an overview of this process.

5.1.1.3 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 state, allocating the necessary amount of memory for its state and releasing memory when a flow ends; *Dyssect* only manages access to state via (opaque) pointers.

5.1.2 Flow Assignment

Dyssect combines two mechanisms for assigning flows to cores (see Figure 5.2). 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. *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 Section 5.1.3.

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

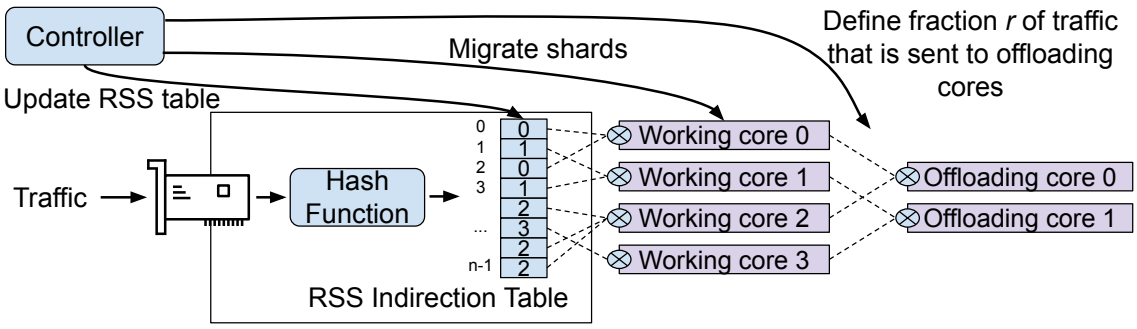


Figure 5.2: *Dyssect* overview.

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 (Section 5.1.1). 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 Section 5.1.2.2). 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.

5.1.2.1 *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 5.1 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 4). On each iteration, the controller builds a set of signals specifying autoscaling operations to be executed by each core (initialization in Line 5 and notification in Line 19). Over long timescales (Line 6), the controller runs an optimization that computes the number

Algorithm 5.1 Central controller.

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

of working and offloading cores.

On every autoscaling operation, the controller runs a short-timescale solver. The short-timescale solver receives as input 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 8).[2] For each offload ratio, assignment or mapping changes (Lines 9, 11, and 17), the controller updates signal accordingly (Lines 10, 16, and 18).

For each shard s that will migrate to another working core, the controller

[2]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 (Section 5.1.2.2), and there is no significant impact for signaling when there is no change in offload ratios.

configures the working core w receiving the shard to hold packets from s in a queue for later processing (shards[s].hold, Line 13). 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 14), which will cause packets from s to go into w's queue.

5.1.2.2 *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.[3]

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. The main idea is that the reading core has to process all the enqueued packets on a queue $\mathcal{Q}$ when the migration starts before $\mathcal{Q}$

Algorithm 5.2 Packet processing at working cores.

```
1: procedure ProcessPackets
2:     for each shard s assigned to this core do
3:         if |self.Q_s^held| > 1 and not shard[s].hold then
4:             RunSFC(self.Q_s^held)
5:     for each packet p in self.Q^NIC do
6:         s ← getShard(p)
7:         if shard[s].hold then
8:             self.Q_s^held ≪ p
9:         else if isOffloaded(p, r_s) then
10:            self.Q^offloading ≪ p
11:        else if self.offloadRatioChange and
12:              isOffloaded(p, r_s^old) then
13:            self.Q_s^pending ≪ p
14:        else
15:            RunSFC(p)
```

is reassigned to another core to process new arriving packets.

Algorithm 5.2 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

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

Algorithm 5.3 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 ← ∅
```

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 5.2 processes one packet at a time for clarity, *Dyssect*'s implementation operates over batches of packets for improved performance.

Algorithm 5.3 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 5.1, Line 17). When updating offload ratios, the working core replaces $\mathsf{self}.\mathcal{Q}^{\text{offloading}}$ 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 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 $\mathsf{self}.\mathcal{Q}^{\text{offloading}}$ (see Algorithm 5.2, Line 10). After the offloading core has finished clearing $\mathcal{Q}^{\text{old}}$, the working core processes any pending packets in $\mathsf{self}.\mathcal{Q}^{\text{pending}}$ (Lines 6–7, also Lines 11–13 in Algorithm 5.2).

Changing offloading cores is equivalent, with two differences: self.offloadRatioChange is false and $\mathsf{self}.\mathcal{Q}^{\text{pending}} = \emptyset$, 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 5.1, 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 5.2, 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.

5.1.2.3 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 $\mathsf{RunSFC}(p)$, 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 5.1.1 (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.

Deadlock freedom: There are five locations that can block the processing of packets: Line 3 in Algorithm 5.2 plus Lines 4, 11, 16 and 17 in Algorithm 5.3.

Because the processing of packets can be disabled by the controller (Line 11, Algorithm 5.1), 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 5.3). Further, only the controller can disable the processing of packets, preventing any cyclic dependencies among working cores.

The *waits* in Algorithm 5.3 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 5.2 never blocks.

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

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 because both packets are processed by Line 15 in Algorithm 5.2.
- If p and q are processed by the offloading core, $e_R^p \prec e_R^q$ holds because both packets are queued for processing in Line 10 of Algorithm 5.2 and processed in order by the offloading core.
- If p and q are processed by the offloading and working cores respectively (*i.e.,* packet p is queued in Line 10 and packet q is queued in Line 13 of Algorithm 2), $e_R^p \prec e_R^q$ holds because of Lines 4 and 7 in Algorithm 5.3.
- If p and q are processed by the working and offloading cores respectively, then $e_R^p \prec e_R^q$ holds because p is immediately processed by Line 15 of Algorithm 5.2 before packet q is queued for processing at Line 10 of Algorithm 5.2.

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 5.3 which happens only after the old offloading core is emptied at Line 11.

Table 5.1: 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
$n^{\mathrm{w}}, n^{\mathrm{o}}$	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
$u_c^{\mathrm{w}}, u_k^{\mathrm{o}}$	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
$u_{\max}^{\mathrm{w}}, u_{\max}^{\mathrm{o}}$	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
$\mathrm{CA}^{\mathrm{p}}, \mathrm{CS}^{\mathrm{p}}$	CV of priority packet interarrival times and proc. times
$\mathrm{CA}^{\mathrm{r}}, \mathrm{CS}^{\mathrm{r}}$	As above, for regular flow packets
$T^{\mathrm{p}}, T^{\mathrm{r}}$	Mean processing time for priority and regular packets
$\mathrm{SLO}^{\mathrm{p}}, \mathrm{SLO}^{\mathrm{r}}$	Max. queueing delay for priority and regular packets
$\boldsymbol{A}^{\mathrm{old}}$	Previous assignment of shards to working cores
$\boldsymbol{O}^{\mathrm{old}}$	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 5.1), but are not yet processed as the controller disables their processing at Line 11 of Algorithm 5.1. Processing is enabled only at Line 19 in Algorithm 5.3 after the packets in $\mathcal{Q}^{\mathrm{offloading}}$ and $\mathcal{Q}^{\mathrm{NIC}}$ are cycled in Lines 16–17.

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

5.1.3 Flow Assignment Optimization

5.1.3.1 Long-timescale Optimization

The long-timescale optimization problem is shown in Table 5.2. The objective function (Eq. 5.1) minimizes resource utilization, expressed as the number of active working and offloading cores. Equation 5.2 ensures that there is at least one active working core and limits the maximum to all available cores. Equation 5.3 enforces that each shard s is assigned one working core c. Table 5.1 describes the notation used in the models.

Equation 5.4 ensures no shards are assigned to inactive working cores. The first half of Equation 5.5 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 5.6 enforces that an active working core has at least one shard assigned to it. Equation 5.7 ensures that no working core offloads traffic to an inactive offloading core. Equation 5.8 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 5.9 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 5.10 and 5.11 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.

We model each core as a queue and limit core utilization such that packet wait (queueing) times in the system satisfy a target SLO. We make no assumptions on

Table 5.2: Long-timescale optimization problem.

$$\text{minimize} \quad \sum_{i \in C} (w_i + f_i), \quad \text{subject to} \tag{5.1}$$

$$1 \leq \sum_{i \in C} (w_i + f_i) \leq |C| \tag{5.2}$$

$$\sum_{c \in C} \boldsymbol{A}_{s,c} = 1 \qquad \forall s \in S \tag{5.3}$$

$$\boldsymbol{A}_{s,c} \leq w_c \qquad \forall s \in S, c \in C \tag{5.4}$$

$$r_s \boldsymbol{A}_{s,c} \leq \sum_{k \in C} \boldsymbol{O}_{c,k} \leq w_c \qquad \forall s \in S, c \in C \tag{5.5}$$

$$\sum_{s \in S} \boldsymbol{A}_{s,c} \geq 1 \qquad \forall c \in C \text{ with } w_c = 1 \tag{5.6}$$

$$\boldsymbol{O}_{c,k} \leq f_k \qquad \forall c \in C, k \in C \tag{5.7}$$

$$\sum_{c \in C} \boldsymbol{O}_{c,k} \geq 2 \qquad \forall k \in C \text{ with } f_k = 1 \tag{5.8}$$

$$0 \leq r_s \leq r_s^{\max} \leq 1 \qquad \forall s \in S \tag{5.9}$$

$$u_c^{\mathrm{w}} = w_c \sum_{s \in S} L_s \boldsymbol{A}_{s,c} (1 - r_s) \qquad \forall c \in C \tag{5.10}$$

$$u_k^{\mathrm{o}} = f_k \sum_{c \in C} \left(\boldsymbol{O}_{c,k} \sum_{s \in S} L_s \boldsymbol{A}_{s,c} r_s \right) \qquad \forall k \in C \tag{5.11}$$

$$\left(\frac{u_c^{\mathrm{w}}}{1 - u_c^{\mathrm{w}}} \right) \frac{(\mathrm{CA}^{\mathrm{p}})^2 + (\mathrm{CS}^{\mathrm{p}})^2}{2} T^{\mathrm{p}} \leq \mathrm{SLO}^{\mathrm{p}} \qquad \forall c \in C \tag{5.12}$$

$$\left(\frac{u_k^{\mathrm{o}}}{1 - u_k^{\mathrm{o}}} \right) \frac{(\mathrm{CA}^{\mathrm{r}})^2 + (\mathrm{CS}^{\mathrm{r}})^2}{2} T^{\mathrm{r}} \leq \mathrm{SLO}^{\mathrm{r}} \qquad \forall k \in C \tag{5.13}$$

$$u_c^{\mathrm{w}} \leq u_{\max}^{\mathrm{w}} \qquad \forall c \in C \tag{5.14}$$

$$u_k^{\mathrm{o}} \leq u_{\max}^{\mathrm{o}} \qquad \forall k \in C \tag{5.15}$$

the packet arrival process or the NFV processing times, and model each core as a G/G/1 queue.[4] We compute the mean packet wait time as a function of core utilization using Kingman's formula [67]. Equations 5.12 and 5.13 capture the SLO requirement on the mean packet wait times.[5] 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

Table 5.3: Short-timescale optimization problem.

$$\text{minimize} \quad F(\boldsymbol{A} - \boldsymbol{A}^{\text{old}}) + F(\boldsymbol{O} - \boldsymbol{O}^{\text{old}}), \tag{5.16}$$

subject to Equations 5.2–5.13, and

$$u_c^{\text{w}} < 1 \qquad \forall c \in C \tag{5.17}$$

$$u_k^{\text{o}} < 1 \qquad \forall k \in C \tag{5.18}$$

$$\sum_{c \in C} w_c = n^{\text{w}} \tag{5.19}$$

$$\sum_{k \in C} f_k = n^{\text{o}} \tag{5.20}$$

have low variation and can be modeled by an M/M/1 queue, then Equations 5.12 and 5.13 can be changed to use the exact distribution of packet wait times instead of Kingman's formula for more precise SLO bounds.

5.1.3.2 Short-timescale Optimization

Core utilizations are bounded by Equations 5.14 and 5.15, which limit utilization to the thresholds $u_{\text{max}}^{\text{w}}$ and $u_{\text{max}}^{\text{o}}$. These thresholds can be used to prevent solutions from running cores too close to 100% utilization. We limit utilization as a safety measure to allow some spare processing capacity in the case of traffic bursts, providing flexibility to the short-timescale optimization and mitigating the risk of performance degradation or packet loss during unexpected traffic bursts.

The short-timescale optimization, shown in Table 5.3, 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. 5.19 and 5.20). It computes $\boldsymbol{A}$, $\boldsymbol{O}$, and r_s to minimize the number of shard migrations and offloading core reassociations

[4] 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.

(Eq. 5.16) 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 5.14 and 5.15. These constraints are replaced by Equations 5.17 and 5.18, 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 5.12 and 5.13.

5.2 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 evaluate tail latency and packet loss rates of *Dyssect* and RSS++ operating under high load. In the fourth use case, we show that state disaggregation allows offloading some functionalities to a SmartNIC.[6] We show the performance impacts in the function offloaded to the NIC and processor metrics, such as L1 and L2 cache misses and instructions per cycle.

5.2.1 General Setup

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

[5]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 5.12 and 5.13.

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 (Center of Applied Internet Data Analysis) [17] 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.

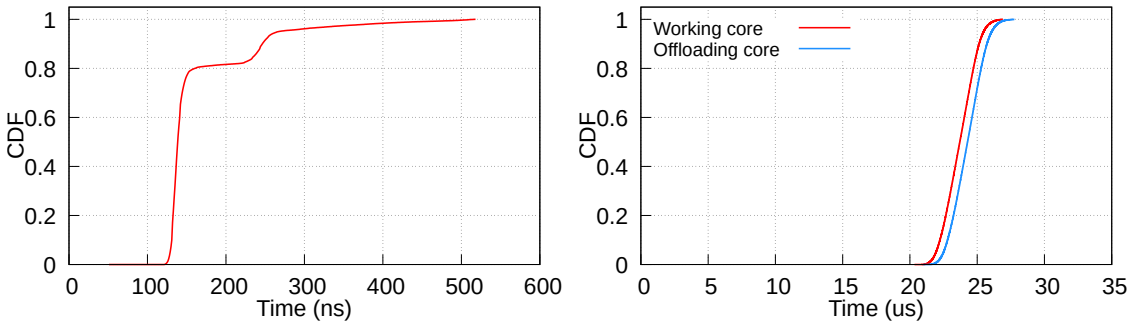
5.2.2 Offloading Costs

Dyssect uses offloading cores to redistribute traffic and prioritize certain types of traffic based on operator-defined SLOs (as discussed in Section 5.1.3). Each offloading core receives packets from an exclusive queue, where packets are enqueued by working cores based on the controller-calculated shard ratio (as discussed in Section 5.1.2.1). To demonstrate the cost and impact of offloading, we ran an experiment with one working core and one offloading core receiving packets from a single flow at a low packet rate of 100 kpps. During the experiment, the working core only performs a state lookup and does not run any network function to focus on the offload overhead. This experiment has two phases. In the first phase, we set $r = 0$ for all shards to ensure that packets are not enqueued to the offloading core and measure the RTT (Round-Trip Time) latency. In the second phase, we set $r = 1$ for all shards to ensure that the working core enqueues all the packets to the offloading cores and measure the queue delay and the RTT.

Figure 5.3a shows the CDF (Cumulative Distribution Function) of the offloading

[6]In this thesis, we use "SmartNICs" and "Programmable NICs" interchangeably.

latency considering 99.5% of the samples. The median, mean, and maximum latencies are 139 ns, 166.93 ns, and 518 ns, respectively. Figure 5.3b shows the offloading impact on the RTT. We observe that the overhead is insignificant even when the RTT is in the order of a few microseconds, which opens possibilities and brings benefits when prioritizing certain types of traffic or meeting SLOs defined by the operator, as we show in Sections 5.2.3, 5.2.4, 5.3, and 5.2.5.



(a) Latency distribution.

(b) RTT Latency.

Figure 5.3: Offloading latency (Figure 5.3a) and its impact on the RTT (Figure 5.3b).

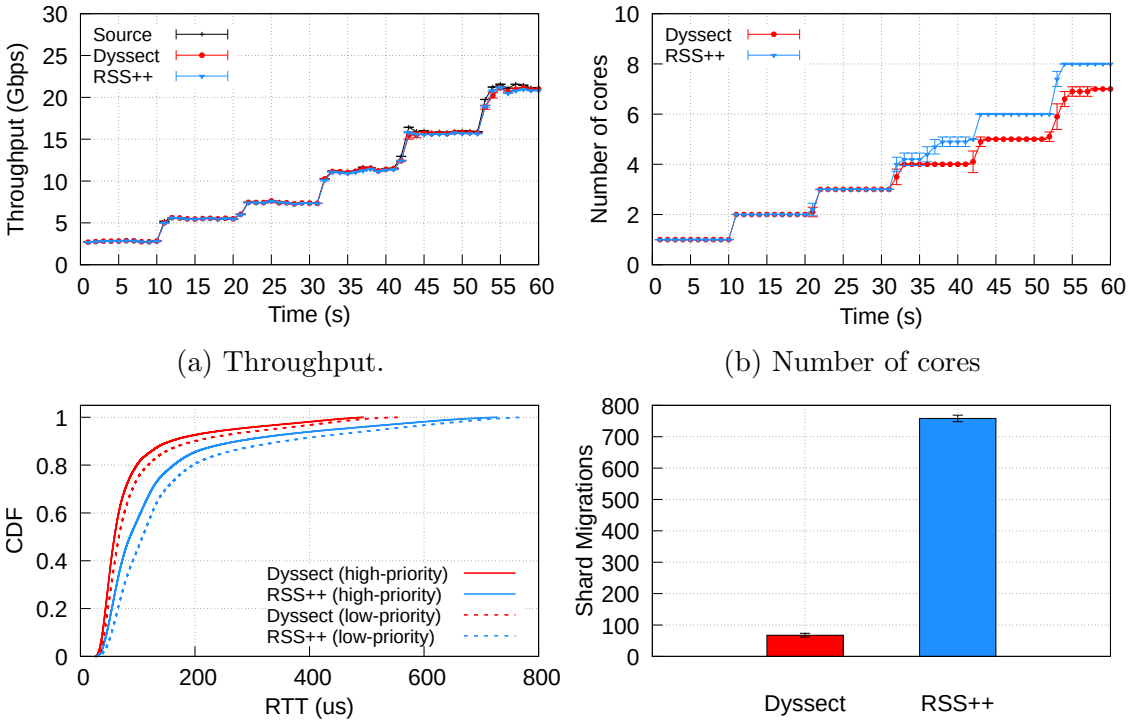
5.2.3 Use Case I: Traffic Class Prioritization

In this use case, *Dyssect* runs the optimization models (Tables 5.2 and 5.3), 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 5.12 and 5.13. 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 executes an automaton to search patterns on the TCP payload to represent CPU-intensive functions. Every 10s, 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 5.4 compares the performance results of *Dyssect* with the results of RSS++. *Dyssect* can process the same or higher traffic load as RSS++ (Figure 5.4a) using the same amount or fewer cores (Figure 5.4b), without causing traffic disruptions while migrating shards and flows (steps).



(a) Throughput.

(b) Number of cores

(c) Latency.

(d) Shard Migrations.

Figure 5.4: Performance results for 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 5.4c 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 32.04% reduction), without compromising the low priority traffic (dashed lines). Figure 5.4d shows that

Dyssect achieves its results performing less than 9% the number of shard migrations performed by 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.

5.2.4 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 5.4 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 5.16 scaled by a (small) factor α to reduce the number of shard migrations and offloading core reassociations when the load is balanced.

Table 5.4: Load balance optimization problem.

$$\text{minimize} \quad \sum_{c \in C} (u_c^{\text{w}} - T)^2 + \sum_{k \in C} (u_k^{\text{o}} - T)^2 + \alpha(\text{Eq. 5.16}), \tag{5.21}$$

subject to Equations 5.2 – 5.11 and Equations 5.19 – 5.20

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 (Section 5.2.3) 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 5.2 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++[7] 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 $(1, 4, 16, 32)$.

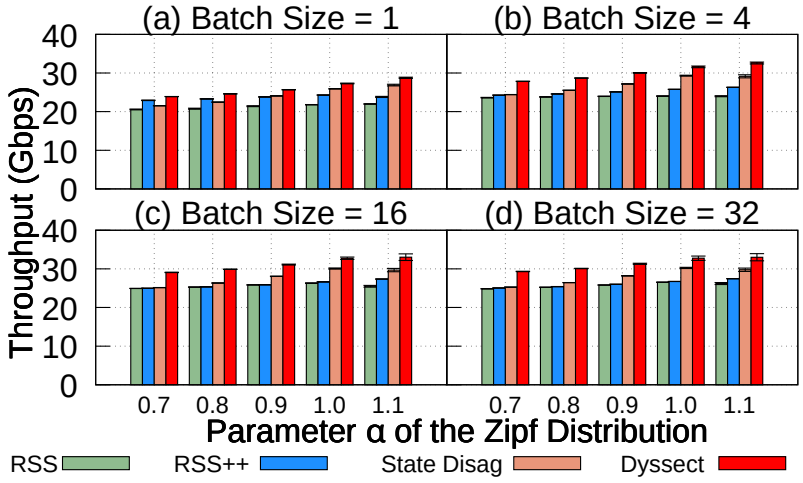


Figure 5.5: Throughput of *Dyssect* compared to RSS and RSS++ for different batch sizes and α parameters of the Zipf distribution.

Figure 5.5 shows the throughput of the four approaches under different batch

[7]This one-to-one mapping is not a fundamental limitation of RSS++, as it could use fewer shards at the cost of decreasing its allocation granularity and reducing its ability to balance the load across cores.

Table 5.5: 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

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 5.5 shows the absolute number of shard migrations performed by *Dyssect* and RSS++ averaged over the 10 experiment runs. *Dyssect* achieves higher throughput with fewer shard migrations in all scenarios.

5.2.5 Use Case III: High Packet Rates

To assess how *Dyssect* compares to RSS++ under high packet rates, we run an experiment with a single NAT processing 128-byte packets with interarrival times drawn from a Zipf distribution ($\alpha = 1.1$) and 1M flows, 0.1% of which are high priority. *Dyssect* uses seven working cores and one offloading core, while RSS++ uses eight cores. To keep the same number of cores throughout the experiment, we disable the long-timescale optimization model for *Dyssect* and the auto scale for RSS++. However, both systems still dynamically migrate shards and flows between

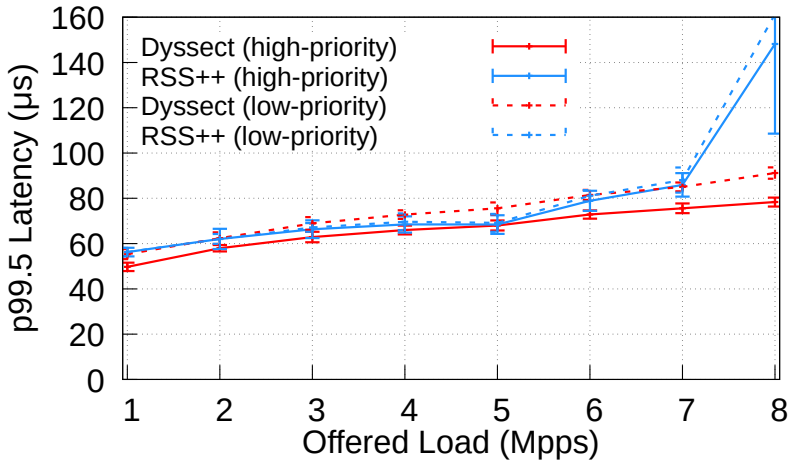
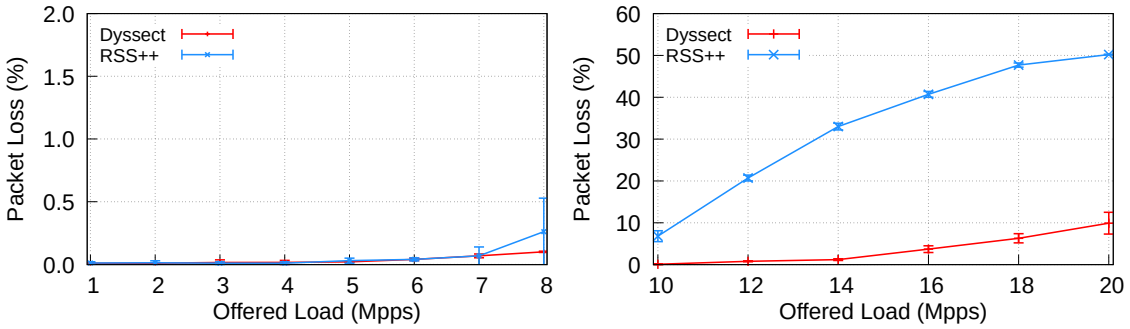


Figure 5.6: Latency results for NAT processing under increasing offered load.



(a) Packet loss.

(b) Packet loss.

Figure 5.7: Packet loss results for NAT processing under increasing offered load.

cores.

Figure 5.6 shows the tail latency (99.5 percentile) for high and low-priority flows. *Dyssect* exhibits lower latency for high-priority traffic due to prioritization, while low-priority flow latencies are comparable between the two systems. However, there is a significant difference when the packet rate increases. Figure 5.7a displays the packet loss observed during the same experiment. Although both systems start dropping packets when we increase the offered load, we also evaluate how *Dyssect* and RSS++ behave when they receive even higher packet rates. Figure 5.7b shows the packet drop for both systems up to 20 Mpps offered load. In this case, we can see a significantly better performance from *Dyssect*.

5.3 SmartNIC Offloading

In this section, we show how *Dyssect* can take advantage of state disaggregation by evaluating trade-offs of entirely or partially offloading the processing of a service chain to a SmartNIC. First, we assess the impacts of offloading to the SmartNIC the lookup function for a subset of flows defined by the controller. Second, we evaluate the hardware-software trade-offs of executing a SFC on the SmartNIC (Hardware), on the server CPU (Software), or on both (Hybrid). Finally, we show that the state disaggregation provided by *Dyssect* allows online migration of flow processing of stateful network functions from and to the NIC without traffic disruptions.

5.3.1 Packet Steering and Memory Management

The Netronome NFP-4000 features five memory types: LMEM (Local Memory), CLS (Cluster Local Scratch), CTM (Cluster Target Memory), IMEM (Internal Memory), and EMEM (External Memory), which vary in their memory latency, with LMEM having the lowest latency (1-3 cycles) and EMEM having the highest (150-500 cycles) [90]. Despite the preference for LMEM, the SmartNIC allocates packets and flow states to slower memory areas. Additionally, the NFP-4000 has 60 Flow Processing Cores (FPCs) that can concurrently access flow states to process incoming packets, regardless of their traffic class or flow-affinity limitations.

To prevent race conditions, minimize contention, and meet SmartNIC's memory limits, we implement one semaphore for each hash table entry, which contains a few buckets with flow states. Using one semaphore per flow is not feasible due to memory constraints, although it would provide lower latency. The FPC computes the hash of the incoming packet's 5-tuple to determine its corresponding table entry.

Since different FPCs can process packets from the same flow (*i.e.,* the FPCs do not respect flow affinity), we enable the Global Reordering (GRO) module of the

NFP-4000 [91] to prevent packet reordering by the NIC.

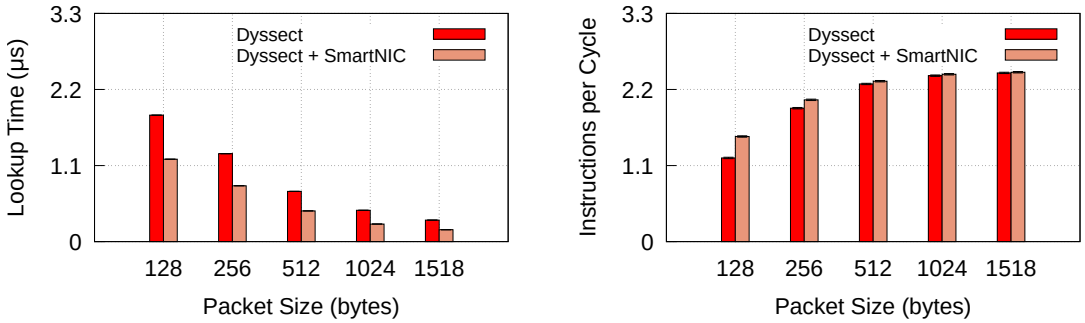
5.3.2 Lookup Function Offloading

We offload to the SmartNIC the flow entry lookup function for a subset of flows defined by the controller.[8] When receiving the first packet of a flow that should 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 [11,31,62,73,78], we can decrease packet processing times and free CPU cycles.

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.

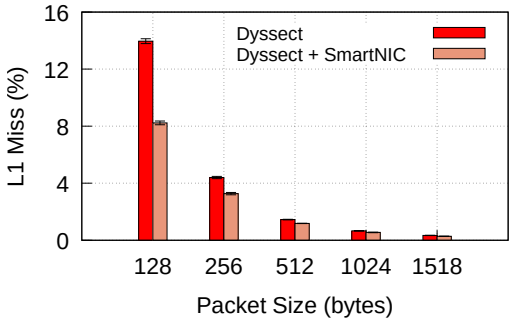
Figure 5.8a shows that average lookup times decrease significantly when *Dyssect* uses a SmartNIC, as expected. However, the takeaway of this experiment is that offloading reduces not only the lookup time but also improves the CPU performance metrics, such as instructions per cycle (Figure 5.8b) and miss rates for L1 (Figure 5.8c) and L2 (Figure 5.8d) 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.

[8]Handling each flow requires one P4 rule in the Netronome NFP-4000 NIC, which supports about 12K rules in a P4 table.

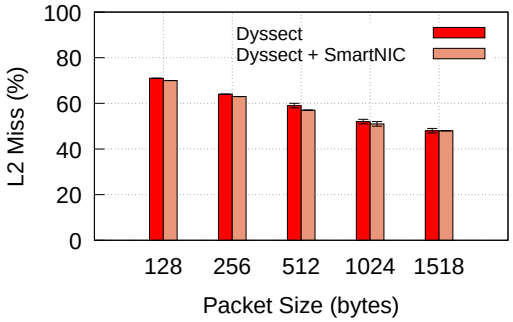


(a) Average Lookup Time.

(b) Instructions per Cycle.



(c) L1 Cache Miss.

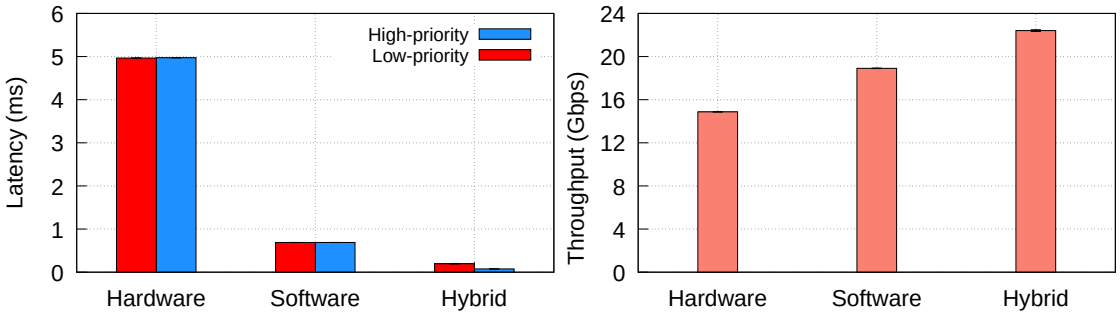


(d) L2 Cache Miss.

Figure 5.8: Average lookup times, instructions per cycle, L1 and L2 cache-miss rates of *Dyssect* with and without offloading to the SmartNIC.

5.3.3 Hardware-Software Trade-offs

The general belief is that offloading network processing to the NIC always pays off [3, 36, 45, 48, 50, 77, 84, 109, 124]. However, depending on the characteristics of a network function and the traffic workload, this belief might not be accurate, as the NICs have limitations that can slow down the processing, such as low clock and memory speeds. In this section, we investigate performance gains and overheads of offloading processing to a programmable NIC. We start by investigating the performance of three models of execution. First, we run the entire service chain, comprising one NAT and one IDS network functions, in the programmable NIC (Hardware). Second, we run the same service chain on the server using a single core and the same configuration defined in Section 5.2.1 (Software). Finally, we use a hybrid model in which the NIC processes the packets of high-priority flows and

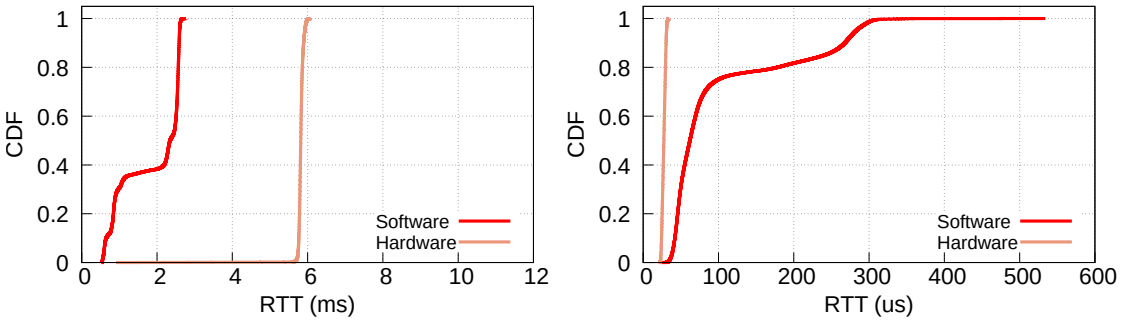


(a) Latency. (b) Throughput.

Figure 5.9: Latency (5.9a) and throughput (5.9b) results for processing a service chain on the NIC (Hardware), on the CPU of the host (Software), or on both (Hybrid).

forwards the other packets to the server CPU (Hybrid). We use the same parameters as the ones in Use Case II in Section 5.2.4 (*i.e.,* 1M flows, α=1.1, Batch=32, CPU clock fixed at 2.2 GHz, and 1500-byte packets.

Figures 5.9a and 5.9b show latency and throughput, respectively, for the three models of execution for the service chain processing 1500-byte packets. As we can see, running the entire service chain on the NIC yields the worst result. We can explain this result by observing two main characteristics of the NIC and the traffic load. First, although the Netronome NFP-4000 has 60 Flow Processing Cores (FPCs), each core has a clock of only 800 MHz. Therefore, iterating over the payload of a large packet takes significantly more time than executing the same task at the host CPU, which runs at 2.2 GHz. Second, the service chain requires state update for each packet. As different FPCs can process packets of the same flow simultaneously, we need to synchronize access to the states stored at the NIC. Even though we use one semaphore for each hash table entry, where each entry has a small set of buckets containing the flow states, the skewed Zipf distribution increases contention at a few semaphores and slows the processing. For the hybrid execution model, semaphore contention is very low, as the NIC processes the full service chain only for a small number of high-priority flows with low bit rate. As a takeaway of this experiment,



(a) Latency for 1500-byte packets.

(b) Latency for 128-byte packets.

Figure 5.10: Latency results for processing a service chain on the NIC (Hardware) or on the host (Software) for different packet sizes.

we argue that we must offload processing to the NIC with parsimony, as some tasks are better executed at the host CPU.

Processing large packets also stresses the memory of the NIC. For a large packet, the NIC cannot store the entire packet on its fastest memory (*e.g.,* Local Memory, LMEM) and has to split and store it into slower memory regions, such as Cluster Local Scratch (CLS) or Cluster Target Memory (CTM) [90]. To investigate the combined effect of clock and memory speeds, we executed another experiment to collect packet latencies when executing a service chain with 1500-byte and 128-byte packets. Figure 5.10a shows the latency CDF observing the 99.9% lowest values (*i.e.,* 99.9th percentile) for 1500-byte packets. As we can see, the latencies are significantly higher for large packets when the service chain is processed entirely on the NIC. On the other hand, Figure 5.10b shows that when the packet size is small, the NIC can process the service chain much faster than if the packets were processed on the host. Finally, Figure 5.11 shows that the throughput is slightly higher (4.23 *vs.* 4.07 Gbps) when the service chain is processed on the NIC with smaller packets and significantly higher (23.6 *vs.* 14.8 Gbps) when the service chain is processed on the host with large packets.

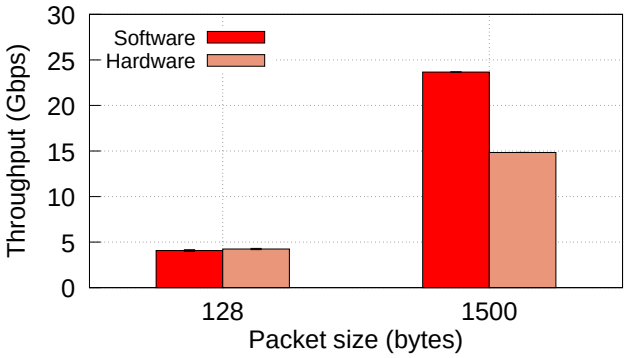


Figure 5.11: Throughput for processing a service chain on the NIC (Hardware) or on the host (Software) for different packet sizes.

5.3.4 Seamless Migration

Dyssect lets a packet carry a reference to a memory region that stores the states of the network functions that process it. This mechanism brings a lot of flexibility to the placement of a network function, as *Dyssect* can move its state and change the location where a packet is processed to achieve different performance objectives. In particular, this flexibility allows the *Dyssect* Controller to offload a network function to a programmable NIC and control if and when the NIC processes a specific set of flows. To facilitate the interaction with the NIC, the Netronome NFP-4000 programmable NIC offers a mechanism to map its internal memory into the address space of a user process, which allows the *Dyssect* Controller to move memory regions seamlessly from the host to the NIC and vice-versa.

To evaluate *Dyssect*'s ability to seamlessly migrate flow processing to and from the NIC on the fly, we measure the RTT of two flows of different types (high and low priorities) during 60 seconds. Initially, the NIC processes the packets only from the high-priority flows, and a single core of the CPU processes the low-priority ones. In this experiment, we use a single network function running a NAT, and we use 128-byte packets to carry timestamps in their payloads for computing the RTT of the traffic going through the NAT. Since the NAT does not iterate over the packet

payload, the packet size is unimportant.

Figure 5.12 shows the latencies for the two flows. At 15 seconds, the *Dyssect* Controller signals the NIC to start processing the packets from the low-priority flow. Then, after 10 seconds, the *Dyssect* Controller moves the state and processing of the low-priority flow back to the CPU memory. Similarly, these transfers repeat at 35, 40, and 50 seconds. As we can see, when the NIC processes the packets from the low-priority flow, their latencies decrease to similar values of the ones from the high-priority flow. Also, processing transfer between the NIC and the host CPU does not cause traffic disruption or packet losses.

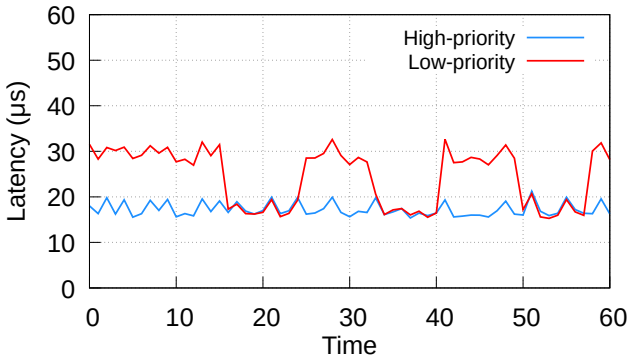


Figure 5.12: Packet latencies for high- and low-priority flows when the processing of the low-priority flow moves to and from the NIC in different moments.

5.4 Summary

In this chapter, we present *Dyssect*, a system that dynamically scales stateful NFs by disaggregating the states of network functions and seamlessly migrating them between cores while still preventing race conditions and packet reordering. To dispatch the incoming packets to cores, *Dyssect* uses a coarse-grained mechanism that maps the NIC's indirection table entries, and consequently shards, to cores and a fine-grained mechanism that assigns a subset of flows to offloading cores. *Dyssect*'s state management moves fractions of traffic between cores with zero-copy

operations and no unnecessary lookups to meet service-level objectives, such as latency constraints and load balancing. *Dyssect* can also offload the processing of a service chain (or part of it) to a programmable NIC to alleviate the CPU load or satisfy latency guarantees of selected flows.

Our experimental evaluations with real and synthetic traffic show that *Dyssect* increases throughput up to 19.36% and reduces tail latency up to 32.04% compared with RSS++, the state-of-the-art for load-balancing within a server. We also show the benefits and pitfalls of offloading the processing of a service chain to a programmable NIC.

Dyssect aligns with the NFV-MANO [92] principles, which were designed to dictate how to manage and orchestrate Virtual Network Functions (VNF) within scalable infrastructures. *Dyssect* creates VNF instances dynamically based on load, balances the workload efficiently across cores executing the network function chain, and maximizes resource utilization. *Dyssect* directly addresses the scalability and dynamic-management goals that are central to NFV-MANO by ensuring flexible and controlled scaling.

Although *Dyssect*'s primary goal is to balance loads within a server by distributing packets across cores, it can also be adapted for inter-server load balancing because of its state disaggregation. With *Dyssect*'s state management and the existence of the Controller, packets from specific flows can be forwarded to different servers at various phases: before processing a network function chain, after completing a function within the chain, or even at the end of the chain. This approach shows how inter-server load balancing, combined with *Dyssect*'s state management and the state disaggregation of network functions, can significantly enhance the load distribution.

Chapter 6

Conclusion

This thesis presents our research on scaling stateful network services on multicore architectures. We explore two network services, TCP network stacks and network functions, that need to maintain consistent state during scaling and workload redistribution across the workers. In Chapters 4 and 5, we address the primary challenges of this thesis: efficiently updating state on a per-packet basis while dynamically balancing the workload across cores in stateful network services on multicore architectures.

In Chapter 4, we systematically evaluate, with synthetic and real-world workloads, multicore architectures for microsecond-scale network applications using TCP. We evaluate the impact of worker configurations using local and remote NUMA nodes on performance, revealing limitations and opportunities for optimization. Evaluated strategies dedicate more workers to the protocol stack or to the application based on specific workload demands. We evaluate the architectures and strategies with *Demieagle*, a benchmark framework that we developed to evaluate the multicore architectures under the same conditions, including a full-featured TCP stack and a flexible scheduler based on Demikernel [138].

In Chapter 5, we present *Dyssect*, a system designed to enhance the performance of stateful network functions through dynamic core scaling and state disaggregation.

Dyssect facilitates seamless state migration between cores by providing exclusive access to the network-function states. It employs a coarse-grained approach to distribute incoming packets across cores by mapping NIC indirection table entries to shards, along with a fine-grained method that assigns specific flows within a shard to an offloading core. This state management enables *Dyssect* to redirect traffic between workers using zero-copy operations, which minimizes latency and facilitates effective load balancing.

Exploring state disaggregation and extending state management in *Dyssect* allow us to offload stateful network functions to programmable NICs. This offloading enables us to examine tailored hardware-software trade-offs for service chains (*i.e.,* a sequence of interconnected network functions). While offloading network functions to programmable NICs typically promises performance gains, our findings reveal that this strategy does not always yield the expected results, primarily due to the diverse multicore architectures of programmable NICs and the specific processing requirements of network functions.

6.1 Future Work

This thesis contributes to a better understanding of the interactions between network stack and application and the trade-offs for dynamically scaling stateful network functions, but some challenges and open problems still remain. This section lists some of these research opportunities for future work on TCP applications and stateful network functions.

Our exploration of the design space for investigating joint scheduling of network stack and application considers only FCFS-based processing the workers. Although recent research has shown that this approach provides the best tail latency across workloads [83], future research may extend this design space by exploring various queueing disciplines and considering the simultaneous use of multiple queueing

models.

Building upon the insights presented in Chapter 4, we plan to integrate our findings into a kernel-bypass system to enable the dynamic selection of different models from the design space at runtime depending on the workload. Given that requests from applications using the TCP protocol may exhibit varying service times, mainly due to load fluctuations, dynamically adapting the models to better reflect the proposed guidelines would be a valuable extension of this work. Specifically, this future approach will focus on understanding application behavior and investigating how modifying runtime models could optimize performance. For instance, one potential strategy involves temporarily leaving a hyperthread idle to minimize resource contention and activating both hyperthreads to process the protocol stack or application when the workload increases, ensuring more efficient utilization of available resources. Another possibility is to use machine learning to understand system behavior [54] and design control algorithms to dynamically adapt the system or predict which model to use during certain periods.

This thesis presents guidelines for developing scalable stateful network functions by addressing resource management challenges, including synchronization, state migration, and load balancing. We focus on overcoming these obstacles by leveraging programmable NICs and improving load distribution across cores. Our findings demonstrate that, contrary to common assumptions, offloading packet processing to programmable NICs does not always yield performance benefits. Although *Dyssect* leverages these NICs for improved performance, further opportunities remain for future exploration, such as allowing the NIC to manage load balancing by dynamically selecting optimal workers on a server or even routing packets across servers to achieve inter-server load balancing. These challenges highlight promising directions for future research.

Bibliography

[1] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *Proc. of ACM SIGCOMM*, 2010.

[2] G. Amvrosiadis, J. W. Park, G. R. Ganger, G. A. Gibson, E. Baseman, and N. DeBardeleben. On the diversity of cluster workloads and its impact on research results. In *Proc. of USENIX ATC*, 2018.

[3] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff. Enabling Programmable Transport Protocols in High-Speed NICs. In *Proc. of USENIX NSDI*, 2020.

[4] S. Arslan, S. Ibanez, A. Mallery, C. Kim, and N. McKeown. NanoTransport: A Low-Latency, Programmable Transport Layer for NICs. In *Proc. of ACM SOSR*, 2021.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proc. of ACM SIGMETRICS*, 2012.

[6] 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.

[7] T. Barbette, C. Soldani, and L. Mathy. Fast Userspace Packet Processing. In *Proc. of ACM/IEEE ANCS*, 2015.

[8] 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.

[9] A. Belay, G. Prekas, M. Primorac, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Transactions of Computer Systems*, 2016.

[10] J. Bellardo and S. Savage. Measuring Packet Reordering. In *Proc. of ACM SIGCOMM*, 2002.

[11] 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 Transactions on Networking*, 2019.

[12] J. C. R. Bennett, C. Partridge, and N. Shectman. Packet Reordering is Not Pathological Network Behavior. *IEEE/ACM Transactions on Networking*, Dec. 1999.

[13] M. Berghetti, F. B. Carvalho, and R. Ferreira. AFP: Um Escalonador de Requisições de Microsserviços Guiado por Feedback. In *Anais do XLII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, Porto Alegre, RS, Brasil, 2024. SBC.

[14] E. Blanton and M. Allman. On Making TCP More Robust to Packet Reordering. *ACM SIGCOMM Computer Communication Review*, 2002.

[15] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-Scalable Locks Are Dangerous. In *Proc. of the Linux Symposium*, 2012.

[16] 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.

[17] CAIDA Data Monitors. Available at `https://www.caida.org/catalog/datasets/monitors/` [Online. Accessed on 2024/09/22].

[18] Z. Cao, V. Tarasov, H. P. Raman, D. Hildebrand, and E. Zadok. On the Performance Variation in Modern Storage Stacks. In *Proc. of USENIX FAST*, 2017.

[19] F. B. Carvalho, R. A. Ferreira, I. Cunha, M. A. M. Vieira, and M. K. Ramanathan. Dyssect: Dynamic Scaling of Stateful Network Functions. In *Proc. of IEEE INFOCOM*, 2022.

[20] F. B. Carvalho, R. A. Ferreira, I. Cunha, M. A. M. Vieira, and M. K. Ramanathan. State Disaggregation for Dynamic Scaling of Network Functions. *IEEE/ACM Transactions on Networking*, 2024.

[21] S. Chen, C. Delimitrou, and J. F. Martínez. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proc. of ACM ASPLOS*, 2019.

[22] M. Claypool and K. Claypool. Latency and player actions in online games. *Communications of the ACM*, 2006.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proc. of ACM SoCC*, 2010.

[24] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proc. of ACM SOSP*, 2007.

[25] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *Proc. of ACM SOSP*, 2021.

[26] M. Dick, O. Wellnitz, and L. Wolf. Analysis of factors affecting players' performance and perception in multiplayer games. In *Proc. of ACM SIGCOMM Workshop on Network and System Support for Games*, 2005.

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

[28] S. Dong, A. Kryczka, Y. Jin, and M. Stumm. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Transactions on Storage*, 2021.

[29] A. Dragojevic, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast Remote Memory. In *Proc. of USENIX NSDI*, 2014.

[30] 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.

[31] C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting. In *Proc. of ACM SIGCOMM*, 2002.

[32] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: compact and concurrent MemCache with dumber caching and smarter hashing. In *Proc. of USENIX NSDI*, 2013.

[33] 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.

[34] 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.

[35] A. Farshin, A. Roozbeh, G. Q. Maguire, and D. Kostić. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-Hundred-Gigabit Networks. In *Proc. of USENIX ATC*, 2020.

[36] D. Firestone et al. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proc. of USENIX NSDI*, 2018.

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

[38] V. Gavrielatos, A. Katsarakis, A. Joshi, N. Oswald, B. Grot, and V. Nagarajan. Scale-out ccNUMA: exploiting skew with strongly consistent caching. In *Proc. of ACM EuroSys*, 2018.

[39] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, V. Sekar, and A. Akella. Stratos: A Network-Aware Orchestration Layer for Virtual Middleboxes in Clouds, 2014.

[40] 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.

[41] H. Ghasemirahni, T. Barbette, G. P. Katsikas, A. Farshin, A. Roozbeh, M. Girondi, M. Chiesa, G. Q. M. Jr., and D. Kostić. Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets. In *Proc. of USENIX NSDI*, 2022.

[42] H. Ghasemirahni, A. Farshin, M. Scazzariello, G. Q. Maguire, D. Kostić, and M. Chiesa. FAJITA: Stateful Packet Processing at 100 Million pps. *Proc. ACM Netw.*, 2024.

[43] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A Software NIC to Augment Hardware. Technical report, EECS Department, University of California, Berkeley, 2015.

[44] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *Proc. of IEEE/ACM MICRO*, 2017.

[45] H. Harkous, M. Jarschel, M. He, R. Priest, and W. Kellerer. Towards Understanding the Performance of P4 Programmable Hardware. In *Proc. of ACM/IEEE ANCS*, 2019.

[46] T. Herbert. *The Linux TCP/IP Stack: Networking for Embedded Systems (Networking Series)*. Charles River Media, Inc., USA, 2004.

[47] B. R. Huaytalla, A. S. Jacobs, M. V. B. Silva, F. B. Carvalho, R. A. Ferreira, W. Willinger, and L. Z. Granville. DWT in P4: Periodicity Detection in the Data Plane. In *Prof. of IEEE GLOBECOM*, 2022.

[48] J. T. Humphries, K. Kaffes, D. Mazières, and C. Kozyrakis. Mind the Gap: A Case for Informed Request Scheduling at the NIC. In *Proc. of ACM HotNets*, 2019.

[49] J. Hwang, K. K. Ramakrishnan, and T. Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. of USENIX NSDI*, 2014.

[50] S. Ibanez, M. Shahbaz, and N. McKeown. The Case for a Network Fast Path to the CPU. In *Proc. of ACM HotNets*, 2019.

[51] Intel. Improving Network Performance in Multi-Core Systems. Available at `http://www.intel.com/content/dam/support/us/en/documents/network/sb/318483001us2.pdf` [Online. Accessed on 2024/09/22].

[52] Intel. Intel® Data Direct I/O Technology. Available at `https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html` [Online. Accessed on 2024/09/22].

[53] Intel. Introduction to Intel Ethernet Flow Director and Memcached Performance. Available at `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf` [Online. Accessed on 2024/09/22].

[54] A. S. Jacobs, R. Beltiukov, W. Willinger, R. A. Ferreira, A. Gupta, and L. Granville. AI/ML and Cybersecurity: The Emperor has no Clothes. In *ACM Conference on Computer and Communications Security (ACM CCS'22)*, Los Angeles, CA, USA, November 2022.

[55] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. MTCP: A Highly Scalable User-Level TCP Stack for Multicore Systems. In *Proc. of USENIX NSDI*, 2014.

[56] 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.

[57] 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.

[58] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA Efficiently for Key-Value Services. In *Proc. of ACM SIGCOMM*, 2014.

[59] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance RDMA systems. In *Proc. of USENIX ATC*, 2016.

[60] A. Kalia, M. Kaminsky, and D. G. Andersen. Datacenter rpcs can be general and fast. In *Proc. of USENIX NSDI*, 2019.

[61] 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.

[62] R. M. Karp, S. Shenker, and C. H. Papadimitriou. A Simple Algorithm for Finding Frequent Elements in Streams and Bags. *ACM Transactions on Database Systems*, 2003.

[63] 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.

[64] A. Kaufmann, S. Peter, N. K. Sharma, T. Anderson, and A. Krishnamurthy. High Performance Packet Processing with FlexNIC. In *Proc. of ACM ASPLOS*, 2016.

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

[66] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout. SLIK: scalable low-latency indexes for a key-value store. In *Proc. of USENIX ATC*, 2016.

[67] J. F. C. Kingman. The Single Server Queue in Heavy Traffic. *Mathematical Proceedings of the Cambridge Philosophical Society*, 1961.

[68] 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.

[69] K.-c. Leung, V. O. Li, and D. Yang. An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges. *IEEE Transactions on Parallel and Distributed Systems*, 2007.

[70] B. Li et al. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *Proc. of ACM SIGCOMM*, 2016.

[71] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proc. of ACM SOSP*, 2017.

[72] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proc. of ACM SoCC*, 2014.

[73] T. Li, S. Chen, and Y. Ling. Per-Flow Traffic Measurement through Randomized Counter Sharing. *IEEE/ACM Transactions on Networking*, 2012.

[74] X. Li, D. G. Andersen, M. Kaminsky, and M. J. Freedman. Algorithmic improvements for fast concurrent Cuckoo hashing. In *Proc. of ACM EuroSys*, 2014.

[75] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: a memory-efficient, high-performance key-value store. In *Proc. of ACM SOSP*, 2011.

[76] 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.

[77] 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.

[78] 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.

[79] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proc. of ACM/IEEE ISCA*, 2015.

[80] Z. Majo and T. R. Gross. (Mis)understanding the NUMA memory system performance of multithreaded workloads. In *Proc. of IEEE IISWC*, 2013.

[81] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the Art of Network Function Virtualization. In *Proc. of USENIX NSDI*, 2014.

[82] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W. C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, and A. Vahdat. Snap: A Microkernel Approach to Host Networking. In *Proc. of ACM SOSP*, 2019.

[83] S. McClure, A. Ousterhout, S. Shenker, and S. Ratnasamy. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *Prof. of USENIX NSDI*, Apr. 2022.

[84] W. M. Mellette, R. Das, Y. Guo, R. McGuinness, A. C. Snoeren, and G. Porter. Expanding across Time to Deliver Bandwidth Efficiency and Low Latency. In *Proc. of USENIX NSDI*, 2020.

[85] Memcached. Available at `https://memcached.org` [Online. Accessed on 2024/09/22].

[86] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. CPHASH: A Cache-Partitioned Hash Table. In *Proc. of ACM SIGPLAN*, 2012.

[87] A. Mirhosseini, A. Sriraman, and T. F. Wenisch. Enhancing Server Efficiency in the Face of Killer Microseconds. In *Proc. of IEEE HPCA*, 2019.

[88] P. A. Misra, M. F. Borge, I. Goiri, A. R. Lebeck, W. Zwaenepoel, and R. Bianchini. Managing Tail Latency in Datacenter-Scale File Systems Under Production Constraints. In *Proc. of ACM EuroSys*, 2019.

[89] Y. Moon, S. Lee, M. A. Jamshed, and K. Park. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *Proc. of USENIX NSDI*, 2020.

[90] Netronome Systems, Inc. Netronome: The Joy of Micro-C.

[91] Netronome Systems, Inc. NFP-4000 Theory of Operation. White paper, Netronome, 2016.

[92] NFV-MANO. Available at `https://www.etsi.org/deliver/etsi_gs/nfv-man/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf` [Online. Accessed on 2024/11/10].

[93] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proc. of USENIX NSDI*, 2013.

[94] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. Scale-out NUMA. In *Proc. of ACM ASPLOS*, 2014.

[95] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot. The Case for RackOut: Scalable Data Serving Using Rack-Scale Systems. In *Proc. of ACM SoCC*, 2016.

[96] NVIDIA. ConnectX 5 EN. Available at `https://www.mellanox.com/products/ethernet-adapters/connectx-5-en` [Online. Accessed on 2024/09/22].

[97] 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.

[98] G. Optimization. Available at `https://www.gurobi.com` [Online. Accessed on 2024/09/22].

[99] 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.

[100] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The ramcloud storage system. *ACM Transactions of Computer Systems*, 2015.

[101] C. Paasch and O. Bonaventure. Multipath TCP. *Communications of the ACM*, 2014.

[102] R. D. G. Pacífico, L. F. S. Duarte, M. S. Castanho, L. F. M. Vieira, J. A. Nacif, and M. A. M. Vieira. eBPFlow: A Hardware/Software Platform to Seamlessly Offload Network Functions Leveraging eBPF. *IEEE/ACM Transactions on Networking*, pages 1319–1332, 2024.

[103] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A Framework for NFV Applications. In *Proc. of ACM SOSP*, 2015.

[104] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. NetBricks: Taking the V out of NFV. In *Proc. of USENIX OSDI*, 2016.

[105] V. Paxson. End-to-End Internet Packet Dynamics. In *Proc. of ACM SIGCOMM*, 1997.

[106] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Prof. of USENIX Security*, Jan. 1998.

[107] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proc. of ACM EuroSys*, 2012.

[108] 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.

[109] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *Proc. of USENIX NSDI*, 2018.

[110] PRADS. Available at `http://manpages.ubuntu.com/manpages/wily/man1/prads.1.html` [Online. Accessed on 2024/09/22].

[111] G. Prekas, M. Kogias, and E. Bugnion. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proc. of ACM SOSP*, 2017.

[112] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico Replication: A High Availability Framework for Middleboxes. In *Proc. of ACM SOCC*, 2013.

[113] 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.

[114] R. J. Recio, P. R. Culley, D. Garcia, B. Metzler, and J. Hilland. RFC 5040: A Remote Direct Memory Access Protocol Specification, 2007.

[115] Redis. Available at `https://redis.io` [Online. Accessed on 2024/09/22].

[116] L. Rizzo and G. Lettieri. VALE, a Switched Ethernet for Virtual Machines. In *Proc. of ACM CoNEXT*, 2012.

[117] A. Rucker, M. Shahbaz, T. Swamy, and K. Olukotun. Elastic RSS: Co-Scheduling Packets and Cores Using Programmable NICs. In *Proc. of ACM APNet*, 2019.

[118] 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.

[119] L. Saino, I. Psaras, and G. Pavlou. Understanding Sharded Caching Systems. In *Proc. of IEEE INFOCOM*, 2016.

[120] R. Shashidhara, T. Stamler, A. Kaufmann, and S. Peter. FlexTOE: Flexible TCP Offload with Fine-Grained Parallelism. In *Proc. of USENIX NSDI*, 2022.

[121] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback-Recovery for Middleboxes. In *Proc. of ACM SIGCOMM*, 2015.

[122] Snort. Available at `https://www.snort.org/snort3` [Online. Accessed on 2024/09/22].

[123] A. Sriraman, A. Dhanotia, and T. F. Wenisch. SoftSKU: Optimizing Server Architectures for Microservice Diversity @scale. In *Proc. of ACM/IEEE ISCA*, 2019.

[124] B. Stephens, A. Akella, and M. M. Swift. Your Programmable NIC Should Be a Programmable Switch. In *Proc. of ACM HotNets*, 2018.

[125] W. Stevens. RFC2001: TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, 1997.

[126] Suricata. Available at `https://suricata-ids.org/l` [Online. Accessed on 2024/09/22].

[127] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis. The NeBuLa RPC-Optimized Architecture. In *Proc. of ACM/IEEE ISCA*, 2020.

[128] V. Vasudevan, M. Kaminsky, and D. G. Andersen. Using vector interfaces to deliver millions of IOPS from a networked key-value storage server. In *Proc. of ACM SoCC*, 2012.

[129] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira. Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications. *ACM Computing Surveys*, 2020.

[130] 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.

[131] Y. Wang, G. Lu, and X. Li. A Study of Internet Packet Reordering. In *Information Networking. Networking Technologies for Broadband and Mobile Networks*, 2004.

[132] 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.

[133] W. Wu, P. DeMar, and M. Crawford. Why Can Some Advanced Ethernet NICs Cause Packet Reordering? *IEEE Communications Letters*, 2011.

[134] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM SIGARCH Computer Architecture News*, 1995.

[135] J. Yang, Y. Yue, and K. V. Rashmi. A Large Scale Analysis of Hundreds of In-Memory Cache Clusters at Twitter. In *Proc. of USENIX OSDI*, 2020.

[136] P. Zave, F. B. Carvalho, R. A. Ferreira, J. Rexford, M. Morimoto, and X. K. Zou. A Verified Session Protocol for Dynamic Service Chaining. *IEEE/ACM Transactions on Networking*, 2021.

[137] P. Zave, R. A. Ferreira, X. K. Zou, M. Morimoto, and J. Rexford. Dynamic Service Chaining with Dysco. In *Proc. of ACM SIGCOMM*, 2017.

[138] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam. The Demikernel Datapath OS Architecture for Microsecond-Scale Datacenter Systems. In *Proc. of ACM SOSP*, 2021.

[139] Y. Zhang, D. Meisner, J. Mars, and L. Tang. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *Proc. of ACM/IEEE ISCA*, 2016.

[140] 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.

[141] P. Zheng, A. Narayanan, and Z.-L. Zhang. A Closer Look at NFV Execution Models. In *Proc. of ACM APNet*, 2019.

Appendix A

Publications

We submitted the results of Chapter 4 to publication and presented the results of Chapter 5 at IEEE INFOCOM 2022. An extended version of the INFOCOM paper was published in the IEEE/ACM Transactions on Networking in 2024. Additionally, we contributed to publications in other conferences, such as IEEE GLOBECOM and the Brazilian Symposium on Computer Networks and Distributed Systems (SBRC). We produced the following manuscripts during the development of this thesis:

- **CARVALHO, F. B.**; FERREIRA, R. A.; CUNHA, Í.; VIEIRA, M. A. M.; RAMANATHAN, M. K. **Dyssect: Dynamic Scaling of Stateful Network Functions**. In IEEE INFOCOM 2022 IEEE Conference on Computer Communications, p. 1529-1538, 2022 [19].

- **CARVALHO, F. B.**; FERREIRA, R. A.; CUNHA, Í.; VIEIRA, M. A. M.; RAMANATHAN, M. K. **State Disaggregation for Dynamic Scaling of Network Functions**. In IEEE/ACM Transactions on Networking, v. 32, p. 81-95, 2024 [20].

- **CARVALHO, F. B.**; FRIED, J. STOLET, M.; SUNEJA, P.; PENNA, P. H.; BONDE, A.; ZHANG, I.; KAUFMANN, A.; FERREIRA, R. A. **A Principled Approach to Multicore Scheduling in the Microsecond**

Era. Submitted to publication.

- ZAVE, P.; **CARVALHO, F. B.**; FERREIRA, R. A.; REXFORD, J.; MORIMOTO, M.; ZOU, X. K. **A Verified Session Protocol for Dynamic Service Chaining**. In IEEE/ACM Transactions on Networking, v. 28, p. 423-437, 2021 [136].

- HUAYTALLA, B. R.; JACOBS, A. S.; SILVA, M. V. B.; **CARVALHO, F. B.**; FERREIRA, R. A.; WILLINGER, W.; GRANVILLE, L. Z. **DWT in P4: Periodicity Detection in the Data Plane.** In GLOBECOM 2022 - 2022 IEEE Global Communications Conference, Rio de Janeiro, RJ, Brazil, p. 6343-6348, 2022 [47].

- BERGHETTI, M. S.; **CARVALHO, F. B.**; FERREIRA, R. A. **AFP: Um Escalonador de Requisições de Microsserviços Guiado por Feedback**. In Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC), Niterói, RJ, Brazil, p. 1134-1147, 2024 [13] (In Portuguese).

Although not directly related to the core topic of this thesis, our previous work on dynamic service chaining [136] provided valuable experience in implementation and performance evaluation, which was instrumental in acquiring the knowledge of fast packet processing that we later applied to the development of *Dyssect* (Chapter 5). Furthermore, our work in [47] gave us essential experience with the data plane in programmable NICs, where we implemented a DWT method to develop an energy function-based technique to detect periodic patterns in network traffic in real-time and at line rate. Furthermore, the research in [13] helped me initiate the mentoring of master's students, an invaluable skill for any researcher. In summary, these publications contributed significantly to both the development of this thesis and my broader education, increasing my knowledge and shaping my research.