Diffuser: Packet Spraying While Maintaining Order

Distributed Event Scheduler for Maintaining Packet Order while Packet Spraying in DPDK

VIGNESH PURUSHOTHAM SRINIVAS
Diffuser: Packet Spraying While Maintaining Order

Distributed Event Scheduler for Maintaining Packet Order while Packet Spraying in DPDK

VIGNESH PURUSHOTHAM SRINIVAS
Abstract

The demand for high-speed networking applications has made Network Processors (NPs) and Central Computing Units (CPUs) increasingly parallel and complex, containing numerous on-chip processing cores. This parallelism can only be exploited fully by the underlying packet scheduler by efficiently utilizing all the available cores. Classically, packets have been directed towards the processing cores at flow granularity, making them susceptible to traffic locality. Ensuring a good load balance among the processors improves the application’s throughput and packet loss characteristics. Hence, packet-level schedulers dispatch flows to the processing core at a packet granularity to improve the load balance. However, packet-level scheduling combined with advanced parallelism introduces out-of-order departure of the processed packets. Simultaneously optimizing both the load balance and packet order is challenging.

In this degree project, we micro-benchmark the DPDK’s (Dataplane Development Kit) event scheduler and identify many performance and scalability bottlenecks. We find the event scheduler consumes around 40% of the cycles on each participating core for event scheduling. Additionally, we find that DSW (Distributed Software Scheduler) cannot saturate all the workers with traffic because a single NIC (Network Interface Card) queue is polled for packets in our test setup. Then we propose Diffuser, an event scheduler for DPDK that combines the functional properties of both the flow and packet-level schedulers. The diffuser aims to achieve optimal load balance while minimizing out-of-order packet transmission. Diffuser uses stochastic flow assignments along with a load imbalance feedback mechanism to adaptively control the rate of flow migrations to optimize the scheduler’s load distribution. Diffuser reduces packet reordering by at least 65% with ten flows of 100 bytes at 25 MPPS (Million Packet Per Second) and at least 50% with one flow. While Diffuser improves the reordering performance, it slightly reduces throughput and increases latency due to flow migrations and reduced cache locality.

Keywords

Packet scheduling, Scheduling, Out of order, Data plane development kit, Parallel processing, Network processor
Sammanfattning


Nyckelord

Paketschemaläggning, Schemaläggning, oordning, Dataplansutvecklingskit, Parallell bearbetning, Nätverksprocessor
iv | Sammanfattning
Acknowledgments

It would be unfair and incomplete to introduce my thesis work without attributing my sincere gratitude to all the people who have relentlessly supported and encouraged me. It was only possible for me to finish my thesis with the help of these people.

First of all, I thank Ericsson for providing me with an opportunity and allowing me to work on the topic of my interest, especially Björn Kihlblom and Pär Pålhed, for selecting me for this project. Next, I thank my industrial supervisor - Zoltán Szandtner, for his constant guidance and unique investigation ideas throughout the project. Also, thank you for trusting me with my ability to steer the research. Last but not least, thanks to both the development teams - Team Matrix and Team Vector for supporting me in development and infrastructure needs. Special mentions to Heng Wang and Ravi Kumar Chennaparapu for technical consultation and Changwoo Kang and Gábor Fényes for helping me set up the testbed.

I want to thank KTH and the EECS school for their exciting curriculum, which inspires and motivates students like me to take up projects on novel and trending topics. I want to extend my special thanks to my examiner - Marco Chiesa, for his advice and instruction during the thesis. I appreciate your kindness and thank you for letting me extend my thesis duration.

Also, I don’t want to forget my friends and family who have supported me on a day-to-day basis. It would have been impossible for me to pursue the Master’s degree if not for my family. Thank you for allowing me to move to Sweden to start a new life. Thank you for believing in my decisions. Thanks to all my friends for fun, laughs, weekend vacations, and parties during stressful times. And finally, thank you, Sneha, for your endless love and always believing in me. Thank you for your continued support and your patience during the demanding spans of the thesis.

Stockholm, Sweden, September 2023
Vignesh Purushotham Srinivas
## Contents

1 Introduction  
  1.1 Background ................................................. 2  
  1.2 Problem Statement ........................................... 6  
    1.2.1 Research Questions ................................. 7  
  1.3 Goals ...................................................... 7  
  1.4 Purpose .................................................... 7  
  1.5 Sustainability and Ethical Reflection ...................... 8  
  1.6 Research Methodology ..................................... 9  
  1.7 Delimitations .............................................. 9  
  1.8 Contributions ............................................ 10  
  1.9 Structure of the thesis ................................... 11  

2 Background  
  2.1 Mobile Networks and Fastpath .............................. 12  
  2.2 Dataplane Development Kit (DPDK) .......................... 14  
    2.2.1 Poll Mode Drivers (PMDs) .......................... 15  
    2.2.2 Environment Abstraction Layer (EAL) ............... 15  
    2.2.3 Ring Library ....................................... 15  
    2.2.4 Reorder Library .................................. 17  
  2.3 Advantages of using DPDK instead of Linux Networking Stack 17  
  2.4 DPDK Execution Models ................................... 23  
  2.5 DPDK Event Library (Eventdev) ............................. 25  
    2.5.1 Event Model Architecture .......................... 26  
    2.5.2 Eventdev Queue Scheduling Types .................. 27  
  2.6 Eventdev Schedulers ..................................... 28  
    2.6.1 DSW Architecture ................................ 29  
    2.6.2 DSW Operation .................................. 30  
  2.7 Related Work ............................................. 32
2.7.1 RSS++: Load and State-aware Receive Side Scaling [16] ........................................... 32
2.7.2 Dyssect: Dynamic Scaling of Stateful Network Functions [15] .................................. 34
2.7.4 A Case for Spraying Packets in Software Middleboxes [50] ........................................... 36
2.7.5 Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets [52] .... 38

3 Research Methodology 39
3.1 Research Methods .................................................. 39
3.2 Data Collection and Validity ........................................ 40
3.3 Evaluation Framework .............................................. 40
  3.3.1 Traffic Generator ................................................ 41
  3.3.2 L1 Switch ....................................................... 41
  3.3.3 Device under Test (DUT) ..................................... 41
  3.3.4 DPDK Application ............................................. 42
    3.3.4.1 Eventdev Configuration ................................ 43
    3.3.4.2 L3fwd Application ..................................... 43
    3.3.4.3 Packet Reordering Setup ................................. 45
3.4 Planned Measurements and Experiments ........................ 46
  3.4.1 Micro-benchmarking DSW with perf ......................... 46
    3.4.1.1 Benchmarking with different TX rates ............... 46
    3.4.1.2 Benchmarking with different number of workers .......... 46
  3.4.2 Migrations Tests ............................................. 47
  3.4.3 Benchmarking Diffuser ..................................... 47

4 Implementation 49
4.1 Software Design of the Existing System ......................... 49
4.2 Software Design ................................................... 51
  4.2.1 Flow Assignment and Adjustment .......................... 52
  4.2.2 Load Imbalance Estimation ................................. 54
  4.2.3 Imbalance Threshold and Other Constants .................. 54
4.3 Implementation ..................................................... 55
  4.3.1 The Enqueue Function (The Scheduling Logic) ............. 55
## 5 Results and Analysis

5.1 Micro-benchmarking DSW with perf 
5.1.1 Benchmarking with Different TX Rates 
5.1.2 Benchmarking with Different Number of Workers 
5.2 Migrations Tests 
5.3 Benchmarking Diffuser 
5.3.1 Reordering and Loss Performance 
5.3.2 Throughput Performance 
5.3.3 Round Trip Time 
5.3.4 CPU Load Fairness

## 6 Discussion

## 7 Conclusions and Future work

7.1 Conclusions 
7.2 Limitations 
7.3 Future work

## References

A Callgraph for Functions in DSW
B Flamegraphs for DSW
C Benchmarking Diffuser with a Single Flow
List of Figures

1.2 CPU loads under atomic and parallel scheduling modes .... 5

2.1 The evolution of 3GPP CUPS in 4G and 5G core networks [23] 13
2.2 Fast path and slow path in a typical routing networking function 14
2.3 Ring buffer operation ................................................. 16
2.4 Packet transfer from NIC to userspace socket in NAPI and DPDK [38] ................................................................. 21
2.5 Run to completion model vs Pipeline model on a 3-core system. In the diagram, the lcore represents DPDK logical cores and Qx shows the hardware queues in the NIC .... 24
2.6 Operation of event model in DPDK ................................. 27
2.7 Shows the core assignment of the eventdev components with different event schedulers on a typical 6-core system. .... 29
2.8 Visualizes the DSW architecture and operation described above 31

3.1 Evaluation Testbed ...................................................... 41
3.2 Event processing pipeline for L3fwd .............................. 43
3.3 Application processing logic for L3 forwarding and packet reordering ................................................................. 45
3.4 Packet structure used for the experiments ......................... 48

5.1 Benchmarking DSW scheduler with different TX rates with DSW atomic mode and 100 flows ......................... 61
5.2 Benchmarking DSW scheduler with different TX rates with DSW parallel mode and 100 flows .......................... 65
5.3 Benchmarking DSW scheduler with different number of workers, for 5000 flows at 10 MPPS and default migration thresholds ..................................................... 66
5.4 Benchmarking DSW scheduler with different numbers of workers, for 5000 flows at 10 MPPS and modified migration thresholds ................................................................. 69
5.5 CPU utilization, Migrations, and Latency plots for DSW with different number of flows .......................................................... 71
5.6 The packet reorder and loss percentages with different DSW modes for 10 flows .............................................................. 72
5.7 Throughput with different DSW modes for 10 flows ................. 74
5.8 Round trip time with different DSW modes for 10 flows ........... 75
5.9 CPU loads under atomic, parallel, and diffuse ......................... 76
5.10 CPU loads under atomic, parallel, and diffuse when 1 flow is directed toward the application ................................................. 77

A.1 Call graph for RX adapter ................................................. 96
A.2 Call graph for TX adapter ................................................. 97
A.3 Call graph for worker core ............................................... 98
A.4 Call graph for reordering core ......................................... 99

B.1 Flamegraph showing different components of DSW under operation for 5 workers ......................................................... 102
B.2 Flamegraph showing different components of DSW under operation for 11 workers .................................................... 103

C.1 The packet reorder and loss percentages with different DSW modes for 1 flow ............................................................. 106
C.2 Throughput with different DSW modes for 1 flow ................. 107
C.3 Round trip time with different DSW modes for 1 flow .......... 107
List of Tables

3.1 Host and guest specification .................................. 42
3.2 Port and CPU core allocation .................................. 44
5.1 RX and TX Pipeline statistics collected from DPDK PMD .... 62
5.2 Total number of migrations with traffic patterns 1 and 2 .... 68
# Listings

1. Pseudocode for `dsw_queue_get_spray_port_id()` . . . . . 56
2. Code implementation for `dsw_port_consider_spray_bg()` . . . 58
3. Code implementation for `spray_compute_load_imbalance()` . . 59
# List of acronyms and abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3GPP</td>
<td>3rd Generation Partnership Project</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CaS</td>
<td>Compare-And-Swap</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off-the-Shelf</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CUPS</td>
<td>Control and User Plane Separation</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DPDK</td>
<td>Dataplane Development Kit</td>
</tr>
<tr>
<td>DSW</td>
<td>Distributed Software Scheduler</td>
</tr>
<tr>
<td>DSW1</td>
<td>Distributed Software Scheduler Version 1</td>
</tr>
<tr>
<td>DSW2</td>
<td>Distributed Software Scheduler Version 2</td>
</tr>
<tr>
<td>DUT</td>
<td>Device Under Test</td>
</tr>
<tr>
<td>EAL</td>
<td>Environment Abstraction Layer</td>
</tr>
<tr>
<td>EPC</td>
<td>Evolved Packet Core</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In-First-Out</td>
</tr>
<tr>
<td>HRW</td>
<td>High Random Weight</td>
</tr>
<tr>
<td>IPC</td>
<td>Inter Process Communication</td>
</tr>
<tr>
<td>IRQ</td>
<td>Interrupt Request</td>
</tr>
<tr>
<td>ISR</td>
<td>Interrupt Service Routine</td>
</tr>
<tr>
<td>L1</td>
<td>Layer 1</td>
</tr>
<tr>
<td>L3</td>
<td>Layer 3</td>
</tr>
<tr>
<td>LPM</td>
<td>Longest Prefix Match</td>
</tr>
<tr>
<td>LTE</td>
<td>Long Term Evolution</td>
</tr>
<tr>
<td>MPPS</td>
<td>Million Packets Per Second</td>
</tr>
<tr>
<td>NAPI</td>
<td>New API</td>
</tr>
<tr>
<td>NFV</td>
<td>Network Function Virtualization</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>NIC</td>
<td>Network Interface Card</td>
</tr>
<tr>
<td>NPU</td>
<td>Network Processing Unit</td>
</tr>
<tr>
<td>NUMA</td>
<td>Non-uniform Memory Access</td>
</tr>
<tr>
<td>OS</td>
<td>operating system</td>
</tr>
<tr>
<td>PE</td>
<td>Processing Element</td>
</tr>
<tr>
<td>PMD</td>
<td>Poll Mode Driver</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>RAN</td>
<td>Radio Access Network</td>
</tr>
<tr>
<td>RSS</td>
<td>Receive Side Scaling</td>
</tr>
<tr>
<td>RTT</td>
<td>Round Trip Time</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SDN</td>
<td>Software Defined Networking</td>
</tr>
<tr>
<td>SLF</td>
<td>Spatial Locality Factor</td>
</tr>
<tr>
<td>SSH</td>
<td>Secure Shell</td>
</tr>
<tr>
<td>SW</td>
<td>Software Scheduler</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TG</td>
<td>Traffic Generator</td>
</tr>
<tr>
<td>TLB</td>
<td>Translation Lookaside Buffer</td>
</tr>
<tr>
<td>TTL</td>
<td>Time-to-live</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>UPF</td>
<td>User Plane Function</td>
</tr>
</tbody>
</table>
List of Symbols Used

Following symbols are used within the body of the thesis:

- $Q$  Quantum of the round robin scheduler. The maximum number of bytes that can be scheduled to the Processing Element
- $Rx$  Reception
- $Tx$  Transmission
Chapter 1

Introduction

1.1 Background

OpenSignal reports that 5G smartphone users consume around 1.7 to 2.7 times more mobile data than 4G smartphone users [1]. The increase in the volume of mobile data, coupled with the explosive growth of 5G, will lead to a significant upsurge in traffic carried on the network infrastructure. Ericsson forecasts the global mobile data traffic to reach around 282EB (Exabytes) per month in 2027, which is a growth by a factor of 4.2 from 2021 [2]. This increased volume and speed will stress the underlying core network tremendously.

One potential solution to the increased load on the network infrastructure is to scale the underlying network resources to further exploit hardware parallelism. Thanks to the immense efforts invested into the R&D of Network Function Virtualization (NFV) and Software Defined Networking (SDN), it is now feasible to decouple network functions from specialized hardware. The virtualized instances of these functions can then be parallelized on many, cheap, commercially available platforms. However, with Moore’s law [3] constrained by the end of Dennard Scaling, the Central Processing Unit (CPU) clock frequencies have not increased lately rather thread-level parallelism has increased. Figure 1.1 shows that CPU operating frequencies grew exponentially from the early 1990s and stagnated around 2005 [4]. To address this concern, chip makers started packing more and more cores on the processor to continue supporting faster processing speeds. Inevitably, as the number of cores and threads-per-core increased, the processors’ parallelizability, energy efficiency, and thermal management posed new challenges. Effectively, the problem was moved from the hardware to the software domain and now required the application to be designed to scale
to multi-core architecture and utilize all the available cores. However, the increased parallelism is also ineffective primarily due to limits imposed by Amdahl’s law [5] and the lack of parallelizable applications and workloads [6].

This problem also manifests in packet processing hardware and software, where the system’s overall performance is directly dependent on the scalability of the underlying packet scheduler. That is, since the scheduler has the crucial role of dividing the processing workload across the available hardware resources, the application can exploit parallelism as long as the underlying scheduler can scale to multiple cores. Studying and developing packet schedulers that scale with the many-core architectures is paramount for maintaining performance that can keep up with the ever-increasing demands of network requirements.

Figure 1.1: Microprocessor trends. “Energy Efficient Computing Systems: Architectures, Abstractions and Modeling to Techniques and Standards” [6]

Hash-based packet schedulers are prevalent and often are bundled with data plane programming frameworks. A hash function divides the incoming flows into multiple sub-domains and maps each sub-domain to one/more CPU cores. The scheduler uses the packet header fields to identify the target worker for a given packet. These hash functions ensure that the packets belonging to the same flow are always routed to the same core. However, the dynamic nature of the internet traffic and the skewed flow sizes can cause
load imbalance across workers. The internet traffic exhibits a heavy-tailed distribution [7][8][9]. That is, most of the internet volume is carried by a small subset of flows (elephant flows), while a large fraction of flows (mice flows) are small in size and do not carry a substantial amount of data. As a result, some cores are overloaded while others are starved, leading to packet loss and degraded throughput [10].

To better understand the problem, we devised a simple experiment on our testbed. A basic Layer 3 (L3) router application [11] is used to understand the effects of skewed traffic on CPU load and packet scheduling. DPDK’s Distributed Software Scheduler (DSW) scheduler [12] is used to distribute the incoming packets to the application threads. This scheduler supports two modes of operation - atomic and parallel. In the atomic mode, the scheduler ensures that the packet from the same flow is always assigned to the same application thread using flow hashes. Contrarily, in the parallel mode, the packets are simply sprayed to all the application cores in a round-robin fashion; Detailed descriptions of DPDK, DSW and other related concepts are introduced in Section 2.

A simple elephant-mice flow pattern is set up according to the Pareto Principle [13][14]. That is, 80% of the traffic is carried by 20% of the flows, and 20% of the flows carry 80% of the traffic. We partition 15 flows into two flow groups such that the first group contains 12 flows with a cumulative packet rate of 4 Million Packets Per Second (MPPS), and the second group consists of 3 flows constituting a rate of 14 MPPS. Both flow groups carry packets of size 100 bytes.

The heatmap in Figure 1.2, illustrates the discrepancies in load distribution for atomic and parallel scheduling modes. In the atomic, while cores 2 and 4 only operate at 20%-50% of the maximal load, cores 0, 1, and 3 work under high loads with roughly 90% to 100% of the maximal load. To guarantee that a flow is always directed to the same processing CPU, DSW maintains a flow-to-core mapping. In this illustration, most likely the elephant flows saturate cores 0, 1, and 3, while the other flows use the remaining CPUs but still fall short of fully utilizing them.

Modern schedulers, therefore, incorporate adaptive load monitoring to achieve optimal load balance. The scheduler tries to move flows from an overloaded to an unloaded core when CPU usage exceeds a predetermined threshold. In Figure 1.2(atomic) the flow migration can be seen at around the 60-second mark. Flow(s) are transferred from core 4 (40%-60% load) to core 2, which is relatively less loaded (10%-30% load). Nevertheless, dynamically moving flows to achieve better load balance introduces its own
set of challenges. With constantly expanding Network Interface Card (NIC) speeds, tracking flow parameters to classify heavy hitters and small flows can be exceedingly challenging at line rates. Even if the flow sizes are known, choosing flows that need to be migrated to another core is not straightforward. For example, should one move a heavy hitter or mice to obtain the desired load balance? Once the flow is selected, identifying the target core that can satisfy the processing requirements of the new flow is also tricky.

As shown above, atomic schedulers must perform the complex act of balancing the system load by constantly tracking the system state and moving flows from one core to another. Any sub-optimal migrations can lead to oscillations and, as a result, affect the system’s stability and performance. Additionally, since schedulers are implemented for multicore platforms, the flow state has to be shared across many processors. In the case of the atomic scheduler, the ability to scale to a large number of CPUs is limited as a shared flow state leads to false-sharing. Hence, recent works propose using per-core flow states [15][16]. Some studies have also shown that migrating flows can lead to packet drop and affect packet ordering [17].

On the other hand, parallel scheduling exhibits a fair load distribution across all the cores in the system. Figure 1.1 shows all cores evenly occupied with an average load of 70%-90% at all times. The scheduler is able to achieve this optimal balance by splitting the incoming traffic equally across all the CPUs by employing a round-robin strategy (where packets are dispatched one by one to all the cores). Nonetheless, round-robin mechanisms often present
two primary drawbacks:

1. Since the round-robin distributes packets one by one to the workers, packets belonging to the same flow are forced to be processed on different cores, leading to out-of-order packet processing.

2. The cache utilization is sub-optimal. Packets of the same flow usually involve similar processing steps. Hence processors can benefit from cache locality when a flow’s packets are processed sequentially. However, in round-robin, the data locality is disturbed, because of the interleaved packet processing.

In this thesis, we aim to design a scheduler that incorporates the functional properties of both the atomic and parallel scheduling modes of DSW. Similar to parallel scheduling, the scheduler should be able to distribute packets belonging to the same flow to multiple workers to achieve optimal load balance. And, like the atomic scheduler, it should be able to maintain the packet order as much as possible within each flow.

1.2 Problem Statement

The packet scheduler is a critical component in packet processing applications, and the application’s overall performance is directly related to the scalability of the underlying packet scheduler. Hash and round-robin-based schedulers are the most commonly used and have different characteristics. Hash-based schedulers provide good cache efficiency and packet ordering but achieve poor load balance across the workers. On the other hand, the round-robin schedulers provide excellent load balance by spraying packets to all the available workers but introduce out-of-order packet processing. DPDK is a dataplane programming Software Development Kit (SDK) that offers many libraries to develop dataplane applications. DPDK includes an event scheduler called DSW. However, this scheduler lacks a scheduling mode that provides good load balance by spraying packets while still preserving packet order.
1.2.1 Research Questions

This thesis will answer the following questions:

1. **What are the bottlenecks affecting the atomic scheduling mode in DSW event scheduler.**

2. **Is it possible to design a DPDK packet scheduler that can distribute packets belonging to the same flows to multiple worker cores to achieve better load balance while still maintaining in-order packet processing?**

1.3 Goals

The main objectives of this thesis are listed below:

1. Identify performance and scalability bottlenecks in the DSW event scheduler by micro-benchmarking. Micro-benchmarking is a process of benchmarking an application at a functional level. Motivate packet spraying (scheduling packets of the same flow to different workers) to solve these bottlenecks.

2. Explore opportunities for using packet spraying as a viable solution for load balancing in high-speed network processing.

3. Propose an event scheduler design capable of spraying packets while minimizing packet reordering.

4. Benchmark and compare the performance of the new scheduler against other available options in DPDK.

1.4 Purpose

The supported data rate of the mobile network is directly proportional to the performance and the capacity of the interconnecting components between the radio interface and the core network. Fastpath is one such critical component in the Radio Access Network (RAN) infrastructure that is responsible for tunneling the user data between the air interface and the core network. In many cases, these components are deployed with constrained computing and memory power. Hence, it is paramount to design these applications such that they do not introduce any bottlenecks in the system. At the same time, even a
slight performance improvement through optimizations can hugely impact the network’s overall performance.

In this degree project, we study the requirements of the fastpath application in collaboration with Ericsson to identify areas that require further optimizations. One potential use case is developing a scheduler to achieve optimal load balance. The current industry trend of moving towards CPUs and Network Processing Units (NPUs) with many-core architecture requires a packet scheduler that can deliver high packet processing performance, good load balance, and linear scalability. This thesis aims to study and develop such a system. The scheduler designed in this thesis can potentially be deployed on Ericsson’s fastpath someday if it satisfies all the required characteristics.

From an academic standpoint, this project studies some of the crucial concepts in computer networks - packet schedulers, load balancers, software and hardware optimizations, advanced data plane development kits, etc. Any contributions of this project will play an important role in providing literature for future studies and degree projects. Current trends show that much work is being focused on increasing the packet processing capabilities[18] and supporting multi-gigabit speed on a single core. This work follows the same ideology and tries to contribute to the plethora of research already being accomplished in this domain.

1.5 Sustainability and Ethical Reflection

In 5G networks, the digital processing in base stations can increase by more than 300 times when compared to early Long Term Evolution (LTE) products. This increase is expected to grow even larger with 6G [19]. To mitigate any performance bottlenecks, many telecommunication companies overprovision their resources. However, neither horizontal nor vertical scaling will be environmentally sustainable in the long run, as power demands continue to rise along with every newly added hardware resource and machine. On the other hand, optimizations and effective usage of the existing hardware and software could considerably ease these demands.

By investigating the underlying packet scheduler, many hidden bottlenecks can be discovered, and, as a result, potential solutions can be proposed for hash-based schedulers. These solutions increase the utilization of the hardware. Additionally, this thesis work also studies the packet ordering problem associated with round-robin-like scheduling. The end-to-end goodput of the fastpath application can be improved by avoiding out-of-order packet processing, as they can induce re-transmissions in connection-oriented
protocols such as Transmission Control Protocol (TCP). To make things worse, re-transmissions can cut the size of the congestion window in half, further degrading the performance and leading to sub-optimal usage of the network infrastructure. This project minimizes unnecessary re-transmissions and maximizes the utilization of the underlying resources. Exaggeratedly, one can even argue that this can limit additional hardware purchases, which is directly linked to the reduction in power consumption and e-waste [20][21].

### 1.6 Research Methodology

This study will take a quantitative research approach, where the research questions are verified analytically by performing different experiments. Developing packet schedulers for NPUs is a trivial problem and has been extensively researched in the past. However, in this degree project, we intend to find a scheduler that is most suitable for fastpath applications; Hence, it is important to support our work with empirical evidence, data, and measurements. Additionally, performing experiments will yield gaugeable results that can be later compared with existing works to make architectural improvements to the future iterations of Ericsson’s fastpath. We take a positive philosophical attitude and assume that it is possible to design and implement a scheduler that can achieve optimal load balance and packet ordering. During the course of the project, the experiments are designed to prove this assumption to be valid.

All measurements and evaluations will be performed in an empirical and reproducible fashion. To eliminate any statistical errors, we embrace an iterative approach, where tests are run multiple times, and the results are presented with confidence intervals to improve the reliability and reproducibility of the presented data.

### 1.7 Delimitations

The project primarily focuses on developing solutions to reduce out-of-order packets in multicore processing environments. However, to minimize packet reordering, two classes of solutions are possible. A post-processing stage can be set up to reassemble packets and restore their original order. On the other hand, a different approach would require developing a scheduler that intelligently schedules packets so fewer jumbled packets are present at the egress stage. We limit our scope to only the second method, as setting up
an additional reordering stage incurs overhead in packet processing, making it unfavorable for fastpath applications.

While traditional polling techniques like - a single core polling a single NIC queue (with Receive Side Scaling (RSS)), or Software Event Scheduler (SW) in the ordered scheduling type, might offer better packet reordering performance, this project limits the scope only to the DSW scheduler.

Hardware solutions such as RSS or other improved variations of RSS, such as RSS++[16], can achieve higher throughput with minimal packet reordering. However, this project aims to achieve the best possible results in highly constrained environments. Public cloud platforms, containers, and legacy non-smart NICs without the support for RSS are some of the target operational environments for this project.

The testing of the proposed solution will be limited to only synthetic traffic generated by a Commercial Off-the-Shelf (COTS) traffic generator. However, the scheduler is tested with IP/UDP traffic, as this is the most dominant type of traffic on the fastpath.

1.8 Contributions

1. DPDK’s DSW event scheduler is micro-benchmarked with perf (a tool included with Linux that can be used for micro-benchmarking) to identify performance and scalability bottlenecks. The bottlenecks determined in these experiments can be extended to other packet schedulers.

2. The design, architecture, and operation of the DSW scheduler are studied and documented in a detailed manner.

3. We study the relevant literature to survey the different state-of-the-art packet schedulers and propose Diffuser. Diffuser is a distributed event spraying scheduler with improved packet ordering performance.

4. The performance of Diffuser is benchmarked and compared with other DSW modes.

5. The performance of DSW’s Reorder Buffer is ascertained.
1.9 Structure of the thesis

The thesis is structured as follows: Chapter 2 begins with a background review to introduce all the essential concepts required to follow this degree project. The first section of this chapter reviews all the concepts, while the second section summarizes some related works and relevant literature. Following that, Chapter 3 covers the research methodology and testing strategy used to answer the research questions. Next, Chapter 4 discusses the system implementation and, includes some examples of code snippets. Chapter 5 presents the results and evaluations. The first part of the results consists of the experiments carried out to profile DSW, and the second part contains the benchmarking results of Diffuser. Chapter 6 covers the discussion on the presented results and the work carried out. Finally, the thesis is concluded in Chapter 7, where the conclusions and future work are presented.
Chapter 2

Background

2.1 Mobile Networks and Fastpath

The SDN network design idea emphasizes the separation of the control plane and data plane of network elements. By disassociating the control plane, all complicated and time-consuming features may be conceptually segregated and implemented in centralized software units, while the more basic forwarding logic can be implemented in efficient, programmable, and distributed hardware units. Centralized management, signaling, automation, and policy enforcement are all possible with an isolated control plane. For example, user authentication, session management, billing, handovers, and so on may be controlled centrally without the complexity of signaling several dispersed control planes.

This principle of isolated planes is not new and has even been adopted in the mobile network’s core and is referred to as Control and User Plane Separation (CUPS) according to the 3rd Generation Partnership Project (3GPP) [22]. In the earlier versions of the 4G packet core, the control plane and the data plane were aggregated to form the core network, as shown in figure 2.1(a). However, in the later releases of the Evolved Packet Core (EPC), the control plane and the data plane were disaggregated to allow them to scale independently (Figure 2.1(b)). This separation was further embraced in the 5G architecture by introducing the User Plane Function (UPF) module, which handles all the data plane functions while isolating it from the rest of the system [23] (Figure 2.1(c)). The introduction of UPF also provided the opportunity to move the dataplane closer to the users to improve the quality of service. According to the 3GPP definition, the UPF is a packet processing node in mobile systems that links the RAN to the internet (or similar data networks),
forwarding packets to and from the internet [24].

The primary task of the user plane or data plane is to make forwarding decisions for packets arriving from the radio access network. When a data packet arrives, the data plane first identifies the forwarding rule by parsing the packet headers and matching the parsed fields against the available forwarding information. Based on the forwarding rule, the packet is either dispatched to one of the forwarding ports or is dropped at the interface. Since the forwarding decision does not involve any heavy parsing of the packet’s payload, processing in the data plane does not incur long overheads. Hence, the data plane is also termed the fast path. In the fast path, packet processing is usually accelerated with specialized hardware [25].

On the contrary, The control plane in the network function’s architecture is responsible for handling the control traffic, which primarily includes the signaling and synchronization information from other distributed control planes in the network topology. The control traffic is sporadic and requires complex computation and updates to routing tables, access control lists, forwarding tables, billing databases, etc. Since the packets arriving on the control plane require more processing time than the packets on the data plane, the control plane is also called the slow path. In most cases, general CPUs are used to execute the processing logic of control plane traffic [26]. The
management plane handles all the management traffic that is used to configure, monitor, and manage the network function.

![Network Function Diagram]

Figure 2.2: Fast path and slow path in a typical routing networking function

Figure 2.2 depicts the traffic flow in a network element’s slow and fast path. When traffic arrives at a network node, it is analyzed and classified on the data plane (fast path). If the packet is a control packet, it is sent to the control plane (slow path); otherwise, the packet is handled on the data plane. In many circumstances, the control plane may not be present in the same system; in such cases, the packet is transmitted to the disaggregated control plane. Once the packet has been processed, it is returned to the data plane and exits the system.

### 2.2 Dataplane Development Kit (DPDK)

The Dataplane Development Kit (DPDK) is an open-source framework for developing userspace data plane applications with accelerated packet processing [27]. DPDK provides many miscellaneous libraries for packet processing along with NIC polling drivers called Poll Mode Drivers (PMDs). DPDK was created in 2010 by Intel and later made available to the open-source community in 2017. Ever since the framework has continued to grow and now supports all major CPU architectures and NICs from multiple vendors. Today, DPDK has become the de facto standard for developing high-speed network applications. Many projects, such as FD.IO, Open vSwitch, Contrail vRouter, TRex, etc., rely on DPDK to achieve high throughput and low latency performance [28]. Some of the core libraries that are packaged with the DPDK
framework are:

### 2.2.1 Poll Mode Drivers (PMDs)

PMD provides Application Programming Interfaces (APIs) to access Tx and Rx descriptors of the NIC ports from the userspace application. The PMD interacts directly with the NIC, without any interrupts, to receive and transmit packets, bypassing the kernel’s networking stack to ensure high performance. DPDK supports hardware from several vendors. At the time of writing this document, DPDK contains PMDs for 18 different vendors, as well as a few virtualized NICs [29].

### 2.2.2 Environment Abstraction Layer (EAL)

DPDK adopts a clever strategy of allocating all the low-level resources required to run the application before launching the application itself. Therefore, reducing the allocation and deallocation overhead of fetching resources from the operating system at run time. The EAL provides this functionality by reserving and maintaining all the required memory pools, CPU cores, timers, and other specialized hardware.

Another responsibility of EAL is to assign CPU cores to the DPDK application. This is done by calling the pthread library to create and launch the application as user threads. The `pthread_setaffinity_np` function is used to pin each execution unit to a specific logical core. Setting the core affinity prevents the operating system’s scheduler from scheduling any non-DPDK tasks on cores. Additionally, the abstracted hardware provides a platform-agnostic interface for application development.

### 2.2.3 Ring Library

The ring library implements rings that can store and share pointers or actual data elements among multiple producers and consumers. This library provides a thread-safe and lock-free mechanism to share data. Rings are crucial in DPDK as it is used by many other libraries internally. For example, the eventdev library uses rings to move packets between event devices and CPU worker cores. Rings are implemented as First-In-First-Out (FIFO) buffers with two index pointers - `head` and `tail`. The head pointer keeps track of the data read from the ring while the tail pointer keeps track of the data written to the ring. When data is removed from the ring, the `tail` pointer is advanced; similarly, when the data is added to the ring, the `head` pointer is adjusted.
Figure 2.3 shows a ring buffer of size 16. When a three-element enqueue action is performed on the ring, the head pointer advances three places clockwise, and a four-element dequeue operation advances the tail pointer four locations clockwise. When the head and tail pointers collide, the ring buffer can either be full or empty. Rings have many advantages over linked lists. Firstly, they are faster than linked lists and use only a few operations to enqueue and dequeue data. In linked lists, many pointer operations are required to do the same. Secondly, rings provide efficient lock-free operations using atomic Compare-And-Swap (CaS) instructions. In high-speed processing, synchronization and locking are significant bottlenecks for performance. Finally, rings support bulk operation to eliminate per-packet function call overhead. Despite these advantages, rings require more memory than linked lists. Since Rings are statically allocated, even an empty ring of $n$ elements will still need to allocate $n$ block of memory.
2.2.4 Reorder Library

The reorder library is used to rearrange out-of-sequence mbufs. In many circumstances, the sequence of packets egressing from the DPDK application must be the same as the ingress order. For example, many audio and video internet calling applications require the packets to be least reordered on the receiving stage in order to provide satisfactory Quality of Service (QoS). However, it is possible that the mbufs are jumbled during processing within the DPDK application. The reorder library uses sequence numbers inserted at the time of ingress to restore the true order of mbufs at egress.

DPDK has implemented the reordering mechanism as a buffer, to which out-of-sequence mbufs can be enqueued while the re-sequenced mbufs can be dequeued from the buffer. However, internally two arrays - reorder and ready, and a sequence window-based mechanism is used to identify early, valid, and late mbufs. The reorder array always contains mbufs whose sequence numbers are always within the sequence window. The ready array holds the mbufs that are reassembled and waiting to be dequeued. When a new mbuf arrives at the reorder buffer, the sequence number of the mbuf is compared with the sequence window’s lower and upper limits. If it is within the limits of the window, it is considered a valid mbuf and is directly inserted into the indexed positions of the buffer. Any mbuf that is less than the window’s lower limit is a late mbuf and is dropped immediately; mbufs whose sequence is higher than that of the higher limit is an early mbuf. Over time, the sequence window moves and advances its limits to accommodate new sequence numbers. When the window slides, all the valid mbuf from the reorder buffer is transferred to the ready buffer for transmission.

A complete list of DPDK libraries can be found here [30].

2.3 Advantages of using DPDK instead of Linux Networking Stack

To understand the drawbacks of using the default networking stack, it is crucial to know how the network driver operates under the hood of the Linux subsystem. This section builds this background by describing the journey of a packet from its reception in the NIC to the userspace.

In the Linux system, all the network-related buffers and queues use a common data structure - sk_buff to store and propagate packets through the kernel [31]. The NIC’s device driver contains a ring buffer of sk_buffs
called the *packet descriptors*. Any free packet descriptors in the ring buffer is first memory mapped to the kernel’s IO space and then is initialized with an empty *sk_buff* structure for packet reception. The state of such pre-initialized packet descriptors are initially set to the ‘ready’ state. When a packet arrives at one of the NIC’s RX queue, it is first copied into one of the ‘ready’ descriptors, and then the state of the descriptor is changed to the ‘used’ state. Once the packet is completely copied into the kernel’s main memory using Direct Memory Access (DMA), an interrupt is raised by the device driver to copy the reference of the NIC’s device into the *poll_queue*. The *poll_queue* is a data structure that the interrupted CPU uses to fetch the information about the device that needs to be polled for incoming packets. The CPU fetches the device information from the *poll_queue* and then calls the *dev-* > *poll()* function of the device driver until all the packets from the ring buffer are processed. Once all the packets are processed, the used *sk_buffs* are reinitialized and refilled with empty *sk_buff* structures for the next batch of arriving packets [32].

In order to process the packets, the kernel calls the *ip_rcv()* function to process any IP packets. This function checks the integrity of the IP packet before determining the transport protocol and handing over the packet for transport protocol processing. Based on the protocol, *tcp_v4_rcv()* or *udp_rcv()* function is called to invoke the next stage of processing. Here, the protocol-specific processing is carried on, and finally, the packet is copied to the associated socket’s receive buffer. Applications can then copy the packet from the kernel space to the userspace using socket-related system calls.

Even though the Linux Kernel’s networking stack is a fully-featured general-purpose network stack for an operating system (OS), it does not fare well for specialized and high-performance packet processing [33], as it contains many inefficiencies. The primary drawback is the mismatch in operating speeds between fast hardware and slow kernel drivers for packet handling. The chain of packet processing in New API (NAPI)[34] has the following overheads [35]:

1. **System call overhead:** The Socket APIs rely on system calls (*read, recv, recvfrom*) to copy the packet from kernel space to userland and vice versa. Invoking such system calls forces the kernel to switch from unprivileged user mode to privileged kernel mode, resulting in costly performance overheads.

2. **Extra data copy:** Packets are copied between userspace buffer and kernel buffers (also vice versa), leading to performance degradations.

3. **Per packet processing:** The Linux network stack generates one
interrupt per incoming packet, which is unsuitable for high packet rates. Interrupt storms cause livelocks leading to inferior performance [36].

4. **Data structure management**: Dynamic memory allocation and freeing of the `sk_buff` structure causes additional overhead. Especially, handling incoming bursts of packets can quickly become a bottleneck.

DPDK mitigates the slow I/O problem by directly accessing the NICs from userspace. The Polling drivers bundled with DPDK directly poll the RX queues for incoming packets and copy them directly to pre-allocated memory in userspace. This mechanism avoids expensive context switches and multiple packet copies from kernel to userspace. DPDK employs many low-level and code-level optimizations to achieve high performance. The following list discusses the significant improvements over NAPI [37]:

1. DPDK applications bypass the Linux kernel networking stack’s heavy layers and talk directly to the network hardware.

2. Hugepages are used instead of regular pages to reduce the number of pages required for packet buffers. The reduced number of memory pages significantly reduces the Translation Lookaside Buffer (TLB) misses and increases the performance.

3. Pre-allocating packet buffers at the start of an application with no further allocation or deallocation during the execution reduces unnecessary overheads.

4. DPDK employs a zero-copy mechanism for packets, i.e., packets are directly DMA-ed by the NIC to the memory once, and the application uses this memory pointer for future processing.

5. Non-uniform Memory Access (NUMA) aware memory allocation ensures that the memory allocations happen on the closest NUMA node to the CPU core.

6. Polling is used to receive packets instead of interrupts.

7. Employs batch processing for API.

8. Code optimizations such as cache alignments of data structures, prefetching, etc., are used for optimal cache usage.
9. DPDK pins every pthread to one logical CPU core and avoids inter-thread scheduling overheads. This allows for significant performance gains.
Figure 2.4: Packet transfer from NIC to userspace socket in NAPI and DPDK [38]

The packet reception process with the NAPI networking stack is as follows
(Figure 2.4 (a)):

1. Packets arrive at the NIC

2. NIC employs RSS or a similar mechanism to distribute the incoming packets to different RX queues. If the NIC does not support multiple queues, then the packet is directly placed on the single available queue.

3. The packet is placed on the device drivers RX descriptor ring by invoking DMA. The DMA is invoked by the NIC.

4. After completing the DMA copy, a hardware interrupt is raised in the device driver to place NIC1’s device ID on the poll_queue. A software Interrupt Request (IRQ) is also raised to alert the kernel to start processing the buffered sk_buff in the descriptor ring.

5. The interrupted CPU then invokes the Interrupt Service Routine (ISR) associated with the interrupt and reads the poll_queue to fetch the device_id of NIC.

6. Once the device_id of the NIC is known, the kernel looks up for the NIC device and calls the dev->poll() function in the associated device driver to dequeue packets from the descriptor ring.

7. The dequeued packets are subjected to IP packet processing in the kernel. Necessary IP tables are updated.

8. Transport layer processing is carried out after IP processing.

9. The processed packets are then placed in the receive buffer of the destination socket.

10. The user application can then fetch the packets by reading the socket’s buffer.

Similarly, Figure 2.4 (b) shows the same process in a DPDK application. The sequence of steps are as follows:

1. Packets arrive at the NIC

2. NIC employs RSS or a similar mechanism to distribute the incoming packets to different RX queues.

3. The PMD drivers of DPDK take ownership of the NIC device and poll the RX queues to fetch the packets.
4. When a packet is available in the RX queues, DMA transfer is initialed
to copy the packets from the RX queues in the NIC to the mbuf ring
buffer in the userland.

5. The application in the userspace can directly access the packets by
dequeuing the mbuf ring in the userspace and applying any packet
processing, including the IP and transport layer processing.

2.4 DPDK Execution Models

There is no doubt that the industry is slowly but steadily transitioning from
using specialized network hardware to the modern NFV paradigm, where
the middleboxes are completely virtualized and run entirely on commodity
hardware [39]. However, the primary hindrance to this trend is that the
generalized and protocol-independent hardware cannot guarantee the same
performance as that of custom silicon. One potential solution to tackle
this problem is to pack more CPU cores on the chip and achieve high
levels of hardware parallelism. For these reasons, the commodity servers
available today are equipped with many CPU cores and require an efficient
multicore programming environment to exploit the parallelism. DPDK
provides multiple ways of configuring the application in multicore ecosystems.
These operational models provide design flexibility and make DPDK an
attractive offering to a wide range of platforms and architectures. For example:
if a system contains only a few processing cores, then the application can
employ the run-to-completion model (described below); if the application
requires many stages of dynamic packet processing, the programmer can
benefit from using the event-driven model of DPDK. The following models
of operations are available in DPDK:

1. Run-to-completion

In this model, each logical DPDK core is responsible for the end-to-end
packet processing. That is, each CPU thread is responsible for polling
one or more NIC RX queues, processing the packet, and placing the
processed packet back on the TX queue of the NIC (Figure 2.5 (a)).
In a multicore environment, all CPUs can be launched in the run-to-
completion mode to achieve the required parallelism. While this model
is advantageous in systems with fewer cores (where having a dedicated
RX and TX core is not affordable), this model is shown to be less
efficient in terms of throughput compared to the pipelined model [40].
2. **Pipeline model**

The pipelined architecture divides the packet processing into multiple functional stages. Each stage can be assigned to one or more logical cores. Packets must go through a number of these stages before the processing is complete, and then the packet is finally moved to the NIC’s TX Queue for transmission (Figure 2.5 (b)). Packets are moved from one stage to another through Ring buffers [41]. The available CPU threads are usually partitioned such that a few cores are dedicated for packet I/O while the others are used for general processing. This strategy avoids any I/O bottlenecks in the system and allows easy scaling.

3. **Event model**

The event model adds an event-based programming paradigm to DPDK. Contrary to the polling mechanism where the worker directly polls the NIC hardware, the event model uses a scheduler to distribute the events (in this case, packets) to the participating workers. The event-driven model has many applications:

- **Pipelining:** the scheduler can be extensively configured to support long chains of packet processing stages.

- **Multicore scaling:** Since the processing stages are not bound to specific cores, the system can dynamically scale without any changes to the underlying DPDK application.
• **Dynamic load balancing:** The scheduler ensures that all the worker cores are evenly loaded, and no worker is starved.

• **Hardware support:** Dedicated hardware event schedulers can be used for further acceleration. The processor from Marvel - OCTEON TX CN83XX SoC natively supports hardware scheduling [42]. Similarly, most hardware vendors (Intel, AMD, etc.) offer such native solutions [43].

• **Priority based processing:** DPDK allows differentiated processing of events. The scheduler maintains priorities for event queues and enforces that a higher-priority event is always processed before a low-priority one.

### 2.5 DPDK Event Library (Eventdev)

An event-driven programming model is a paradigm where events determine the processing flow within the program. An event is defined as an asynchronous notification from hardware or software. Typically, events consist of packets from the ethernet devices, expiry notifications from timers, notifications from the crypto devices, and inter-CPU notifications. In DPDK, an event is identified by the `rte_event` struct [44], which holds metadata of the event, scheduling information, and a pointer to the actual packet (pointer to `rte_mbuf` struct [45]). The essential fields of the `rte_event` struct are briefly described below:

- **flow_id:** ID computed by hashing the five tuples of network and transport layer identifiers. The flow-id and the scheduling type together are used to determine the destination stage where the packet will be processed.

- **sched_type:** The schedule type can be one of the following values - atomic, ordered, or parallel. Essentially, this field determines how packets belonging to the same flow are distributed across the different worker cores.

- **queue_id:** The queue id is a unique value that identifies an event queue in the system. This field determines the next stage of processing in the event pipeline.

- **priority:** This field marks the priority of an event. Event priorities are relative to other events in the event queue.
• mbuf: pointer of the actual packet content.

2.5.1 Event Model Architecture

The Eventdev library defines core components required to support the Event-Based programming model in DPDK. They include event queues, event ports, schedulers, and RX/TX adapters. An event queue is a buffer that holds events awaiting scheduling. An event queue is uniquely identified by a queue_id and used by the scheduler to disperse events across several queues. Event ports are the gateways to event queues; events can enter or exit queues through event ports. Multiple queues can be mapped to a single port. An eventdev device is a virtual device that is in charge of scheduling and load balancing.

Each event port in an eventdev pipeline typically acts as an enqueue or dequeue point from which worker cores can retrieve events for processing. The events are pushed back into the port again after processing. Ports have the ability to enqueue or dequeue a burst of events simultaneously. Burst operations enhance cache usage and cut down per-event IO overhead. Event queues serve as a queuing method while also enforcing the priority and order of packet consumption. The event queue identifies the stage of processing in the complete pipeline. Fundamentally, The eventdev scheduler moves events from one eventdev queue to another based on the pre-configured processing stages.

Figure 2.6 shows the operation of the event model. The step-by-step process is described below:

1. The RX adapter constantly polls the NICs for incoming packets and encapsulates them as events.

2. The RX adapter injects events into the system through an event port that is owned by the adapter. The newly encapsulated events are introduced into the scheduler through this event port.

3. The scheduler makes scheduling decisions for every event and forwards them to their destination event ports.

4. The worker cores poll their attached event ports, looking for events to process.

5. Once a worker processes the events, they are enqueued back to the scheduler for downstream processing.
6. Events that have completed processing exit the system when the scheduler pushes those events to the event port attached to the TX adapter.

7. The TX adapter strips the packets from the events and pushes them to the ethernet device for transmission.

8. The ethernet device then transmits all the packets to the NIC.

**2.5.2 Eventdev Queue Scheduling Types**

The event queue scheduling types supported by DPDK are:

1. **Atomic** - events from the atomic queue are distributed to the worker cores such that events from the same flow are always processed on the same core. The scheduler takes care to enforce that a single flow is not present on different CPU cores at the same time. This ensures that the flow state is consistent across the system and maintains the packet order during processing.
2. **Ordered** - events from the ordered queue are distributed to any available cores (the same flow can be processed on multiple cores simultaneously). However, the scheduler then reorders the egress events of the flow to match the ingress order when further enqueuing them to a downstream queue.

3. **Parallel** - parallel scheduling sprays events of all flows to all available cores and does not explicitly maintain packet order or flow states.

4. **Single link** - A single link queue is a FIFO queue that maintains packet order with a constraint that only a single event port can be attached to such queues.

In the event model, the application polls an event device port for receiving events instead of polling the ethdev ports for packets. As a result, an adapter between the ethdev and the event device is required. Hence DPDK introduced the RX/TX adapter library [46][47] along with the eventdev library. The RX/TX library translates ethdev packets into events that the eventdev device can interpret. The RX adapter essentially polls the NIC’s RX queues for packets, and on the reception of any packet, the library encapsulates the associated mbuf with a rte_event struct and injects it into the eventdev queues. Similarly, when the events are processed, the TX adapter converts the events back to packets and enqueues the mbufs pointers to the TX queues for transmission. In addition to employing software threads for adapters, DPDK can also use hardware adapters, which free up cores for executing the application logic.

### 2.6 Eventdev Schedulers

The eventdev scheduler is the component responsible for assigning events to worker cores. DPDK provides a flexible and extensible interface that allows various schedulers, both hardware and software, to be configured and used based on the requirements. The Software Scheduler (SW) is one such scheduler that is bundled with the eventdev library. The SW is completely implemented in software and uses ring buffers to move packets from one event queue to another. The scheduler is spawned as a software thread on a separate core, allowing the scheduler to run parallel and schedule events without needing software locks. SW supports all scheduling modes (atomic, ordered, and parallel).
The Distributed Software Scheduler Version 1 (DSW1) is a high-speed, high-performance scheduler added to DPDK in release 18.11 [48]. As the name suggests, the DSW1 distributes the scheduling functions to all the available worker cores in the system. Spreading the scheduling tasks across application cores frees up one additional core and allows the application to scale up to more logical cores. DSW1 supports only atomic and parallel scheduling modes[12]. In both SW and DSW1, the worker cores poll the event device instead of the ethernet device. Therefore, it is required to encapsulate/decapsulate the packet mbufs to/from rte_event structures, respectively. This packet IO is performed by the RX/TX adapters. Usually, both these adapters are launched on their own individual cores to avoid any IO bottlenecks. Distributed Software Scheduler Version 2 (DSW2) is an improved version of the DSW1 scheduler where the packet IO is also distributed across all worker cores.

Figure 2.7: Shows the core assignment of the eventdev components with different event schedulers on a typical 6-core system.

Sections 2.6.1 and 2.6.2 describe the architecture and operation of the DSW1 scheduler respectively. We skip the design details of the SW as this project focuses only on distributed scheduling. However, a similar implementation can be designed for the SW.

2.6.1 DSW Architecture

The DSW1 scheduler uses event ring buffers to switch events across different worker cores. An event port is connected to a worker core through a ring buffer structure called the in_ring. Each worker periodically polls its associated in_ring for events to process. Therefore, DSW1 maintains one ring per worker. The events stored in the in_ring are dequeued either as a single event or as a burst of events. The scheduler also maintains a buffer
called out_buffer. The out_buffer holds all the events that are yet to be scheduled. A worker will dequeue events from the in_ring of the event port. Once these events are processed, they are enqueued back into the out_buffer for subsequent scheduling.

When a new event (an event that is newly enqueued into the system or an event that has finished the upstream stage of processing and requires to be scheduled for the next stage) arrives, the scheduler will first compute the flow_id of the event. Then the flow_id, along with the other scheduling information such as the next stage queue_id and scheduling type is used to find out the destination event port and its associated out_ring. Finally, the event is enqueued to that out_ring. Whenever a worker core buffers processed events to the out_buffer, it also tries to flush out the old events awaiting scheduling in the out_buffer. After making a scheduling decision, the events in the out_buffer are moved to their destination ring_in. In this way, the scheduling task is distributed among all worker cores, and the actual scheduling happens when any of the workers enqueue events into an event port. The DSW operation is described using Figure 2.8:

1. The workers pull events scheduled for them from their associated in_ring.
2. Events are dequeued either as a batch or as a single event. These events are then processed.
3. The workers call the enqueue function of the port to return the processed events back to the scheduler.
4. During the enqueue operation, a section of the scheduler logic is executed, identifying the next processing stage and its next event port. Each event port maintains out_buffers for every other event port in the system. These buffers temporarily hold the events that are scheduled for them. Therefore, once the destination port is known, the events are buffered to that respective out_buffer.
5. Finally, all the out_buffers present in the event port that is trying to enqueue events are flushed. The flush operation will move the events from the out_buffer to the in_rings of the other ports.

### 2.6.2 DSW Operation

The actual scheduling of events in DSW occurs when a worker core enqueues packets to the scheduler. Each event port maintains a flow table called the
Figure 2.8: Visualizes the DSW architecture and operation described above

flow-hash-to-owning-port-table. This flow table keeps track of mappings of flow hashes to the event ports where the events belonging to a flow are currently being processed. The flow hashes are 15-bit hashes that are computed by hashing the eventdev event_flow_id [49]. DSW scheduler promises high performance because the events are directly moved from DSW’s out_buffer to the destination event port’s in_ring on the event enqueue by any port in the system. This ensures that events are scheduled as soon as possible and prevents worker cores from starving.

The DSW periodically computes the event ports’ load by measuring the number of events processed in given periods. This allows the scheduler to make scheduling decisions while ensuring that no core gets overloaded. As soon as the scheduler detects an overloaded port, it initiates a flow migration procedure to alleviate the load on the event port. The flow migration process identifies the smallest last-seen flow and changes the state of that flow to
paused on all worker cores by using an internal signaling mechanism. This flow interrupt signal will cause all the worker cores to temporarily stop scheduling the flow events and move them to a paused_flows buffer. The scheduler then identifies a suitable event port with sufficient load quanta to accommodate the paused flow. This computation is based on the load estimates maintained for every event port. Once a suitable candidate port is identified, the scheduler then updates all the flow-hash-to-owning-port-table to map the paused flow to the new event port. Finally, the state of the migrated flow is changed back to unpaused on all worker cores to resume scheduling, and the buffered events in paused_flows are drained.

2.7 Related Work

This section discusses several papers that are of interest to our thesis. Each sub-section summarizes an article, discussing the core ideas, results, and some design concepts. We draw inspiration for the design of our scheduler from these papers.

2.7.1 RSS++: Load and State-aware Receive Side Scaling [16]

The authors of this paper propose a new load-balancing technique for spreading flows to CPU cores using the NIC’s RSS indirection tables. The proposed system is called RSS++ and allows for high CPU utilization and dynamic scaling while reducing tail latency and packet drops. This paper identifies three critical factors to consider when designing a good load balancer. Firstly, The load-balancer must guarantee that an equal amount of work is distributed to all CPU cores. Secondly, The flow-to-core affinity is maintained, that is, packets from the same flow are processed on the same core. Finally, the number of flow migrations is minimized to reduce state transfers.

RSS++ achieves load balance by assigning different RSS buckets to shards. Each core tracks the number of packets received by RSS buckets using a table indexed by the low-order bits of the RSS hash. Periodically, the number of packets received on different buckets is collected, and the processing load of each bucket is estimated. This estimated load is then used to solve an optimization problem that computes the optimal load assignment, after which the RSS indirection table is updated with the new bucket-to-shard assignments using the NIC APIs.
Solving optimization problems can take a few tens of seconds and is not desirable for high-speed networking applications. Hence RSS++ proposes a greedy algorithm that does not look for the best solution but iteratively moves closer toward the best solution for the given optimization. RSS++ periodically selects the most overloaded bucket of the most overladed core and tries to fit this bucket into the most underloaded CPU core. This process continues until the load of all the underloaded cores is within 1% of the average core utilization of the system. This greedy algorithm takes less than 25us (plus 500us for queueing and preparing inputs for the conditions mentioned in the paper) to converge to a solution. Apart from presenting the algorithm, this paper also advocates for per-bucket flow tables. Per-bucket flow tables allow fast migrations of flow states when an RSS bucket is moved from one shard to another. Per-bucket flow states also improve the cache locality and hence the performance. The paper shows that shared flow states can degrade performance due to false sharing, while per-bucket flow tables can be implemented without locks.

RSS++ relies on the NIC APIs to update the RSS indirection tables after the load assignments computation. However, these APIs can take up to several milliseconds to update the NIC. While the simplified version of the algorithm only takes a few microseconds, the NIC APIs can take longer in order of milliseconds. Consequently, if the tables are updated frequently, hundreds of thousands of packets can be lost at high speeds. From the paper, Intel 82599 takes 20us while Mellanox ConnectX-4 and ConnectX-5 can take up to 20ms to update the table and additional needs to be rebooted.

This paper compares RSS++ against a packet spraying load balancer (Sprayer [50]). While packet spraying, packets of the same flow are dispatched and processed on different cores. This packet spreading reduces the cache locality of the shared flow tables and introduces out-of-order packet processing. Multiple cores accessing the same shared flow data leads to false sharing and frequent invalidation of cache lines. The efficiency of packet spraying can heavily depend on the type of flow processing required. The results show that if the workload is simple and only involves reading packets and not stateful actions (such as Deep packet inspection, etc.), Sprayer performs better than RSS++ in terms of cycles per packet. Additionally, as the number of flows increases, cache invalidation decreases and, as a result, slightly improves performance. Packet spraying introduces approximately 30% of out-of-order packets, which also reduces as the number of flows increases.
**2.7.2 Dyssect: Dynamic Scaling of Stateful Network Functions [15]**

Dyssect is a system that improves the performance of VNFs by proposing a shard migration technique to achieve optimal load balance. Additionally, Dyssect provides other functionalities such as traffic prioritization, resource minimization, and strict adherence to service-level objectives (SLOs). The authors of this work argue most NFs are stateful and require synchronized access to shared states. And, to preserve consistency and avoid race conditions in accessing such global states, it is necessary to implement locks and provide exclusion to the worker cores. However, locks degrade performance and prevent NFs from operating at line rates in multi-gigabit speeds. Sharding is another technique often used to avoid locks and involves partitioning the global state into multiple disjoint subsets and assigning each shard to one CPU core. Later, flows are assigned to shards based on the result of the hash function on the flow-id of the packets.

This work identifies some shortcomings of sharding. Firstly, static mapping of shards to CPU cores can cause load imbalance due to the skewed nature of the traffic and lead to increased latency and decreased throughput. On the contrary, dynamic shard assignment requires shard migrations, which at high frequencies can cause cache invalidations and consequently degrade performance. Secondly, increasing the number of shards can reduce the likelihood of multiple large-volume flows in the single shard, but however, a large number of shards reduces cache locality because CPU cores have to address a large number of different shards that may not fit into the core’s local cache. Finally, remapping the shards-to-cores mapping requires updating the NIC’s RSS indirection table, which can take a few hundred milliseconds.

Dyssect proposes two implementation concepts. Firstly, Dyssect disaggregates the global flow state from the NF by using a data structure that allows packets to carry the reference pointers to its associated flow state in its metadata. This mechanism avoids flow state lookups and enables any core to process packets from any flow, leading to faster migrations. Secondly, Dyssect sets up two optimization functions on long and short timescales to provide optimal load balance. The long-timescale optimization periodically computes the shard-to-core assignments and updates the NIC’s RSS indirection table to carry out shard migrations. Similarly, the short timescale optimization provides fine-grade load balance by moving flows to an offloading core. Dyssect divides the available CPU cores into working cores and offloading cores and migrates the low-priority flows from the working cores to offloading...
cores when any of the working cores gets overloaded. This fine-tuning allows Dyssect to prioritize flows and strictly follow SLAs. Dyssect can reduce the tail latency of VNFs by up to 32% and increase the throughput by approximately 19% compared to other state-of-the-art systems, such as RSS++.

2.7.3 An Adaptive Throughput-First Packet Scheduling Algorithm for DPDK-Based Packet Processing Systems [10]

Chuanhong Li et al. introduce an adaptive packet scheduler that prioritizes high-throughput and low packet loss. Programmable data plane technologies, like DPDK, Netmap[51], and others, are capable of providing line-rate packet processing[18] by leveraging parallelism in multicore systems. However, in such frameworks, the packet scheduler plays a critical role in assigning the incoming packets to the worker cores. This paper emphasizes that, in order to achieve good performance, schedulers must be able to equally distribute the load over all cores by ensuring that no worker is overloaded or underutilized.

Packet scheduling methods are divided into two types: packet-level and flow-level. Single packets are used as a unit of schedulable work in packet-level schedulers, which assign individual packets to different workers. Flow-level schedulers, on the other hand, assign entire flows (all packets with the same packet signature or headers) to cores for processing. While the packet-level strategy achieves excellent load balance, it has several drawbacks. Poor data locality and cache underutilization, maintaining and synchronizing flow state across multiple cores, and out-of-order packet processing are a few. Flow-level scheduling overcomes these limitations by processing all packets of the same flow on the same core; nonetheless, such schedulers fail to achieve optimal load distribution across the system. According to the authors, the fundamental cause of unbalanced load is the dynamic nature of internet traffic. Heavy flows typically overwhelm a few cores while starving the others. As a result, this research suggests a scheduling technique that uses subflow as a schedulable unit. This enables the proposed scheduler to retain the benefits of flow-level scheduling while still being flexible enough to distribute smaller batches of packets to avoid load imbalance.

To construct a subflow, this study suggests integrating DPDK’s burst-oriented packet reception with a hash function. A subflow is a collection of packets belonging to the same flow contained within a single burst operation. High Random Weight (HRW) is a hash function that computes the target core
for packets. HRW will produce a target CPU value for each packet in a burst, to which the packet will be dispatched. For the first packet of each flow, the HRW function is computed, and the scheduling decision is saved for the duration of the burst. For subsequent flow packets, the packet identifier is used as a key to look up the saved HRW decision. Each core’s load is estimated by monitoring the buffer occupancy of the worker queues (queues from where the worker cores dequeue packets). When the buffer occupancy exceeds a certain threshold or is zero, the algorithm initiates an HRW adjustment. When HRW is triggered with an adjustment factor (i.e., 1 - buffer occupancy), the target values for the packets are generated in proportion to the adjustment factor of each core. This guarantees fewer packets are sent to the overloaded cores while more packets are diverted towards underutilized cores to improve utilization. As a result, the system can adaptively tune the load balance from burst to burst.

This report offers several crucial lessons from which our work can benefit. For example, calculating CPU core usage by monitoring buffer occupancy is straightforward and platform-independent. Furthermore, establishing subflow as a schedulable unit is useful since they have the load-balancing properties of packet-level scheduling while still providing the cache efficiency of flow-level scheduling.

2.7.4 A Case for Spraying Packets in Software Middleboxes [50]

The authors of this study explore the use of finer granularity of flows for packet scheduling in software middleboxes. They show that spreading packets over numerous cores improves scheduling fairness and processor utilization, even when a single flow traverses the middlebox. The authors begin by looking into the shortcomings of flow-based scheduling methods. Flow-based scheduling techniques are inefficient since not all of the system’s cores can be engaged when just a small number of flows are incident. Furthermore, even when there are many flows, system utilization is still low due to the asymmetrical distribution of flow volumes on the internet. To prove this, the researchers examined real-world traffic data and discovered the "elephants and mice" flow pattern in Internet traffic [9]. As a result, flow distribution strategies (such as RSS) become inefficient. Firstly, because the number of active flows on the internet is still small, and minimizes the probability of using all of the Processing Elements (PEs) in the system, resulting in some cores being overloaded while others are idle. Secondly, the heavy-hitter flows suffocate the CPUs on which they are processed, leading to the same
unbalanced load distribution.

Packet Sprayer is created to spray packets to all of the system’s PEs. This, however, raises the issue of keeping multiple copies of flow-related data, resulting in frequent cache invalidation. Such invalidations are usually avoided by using centrally managed data structures and locking techniques. But, locking mechanisms have been demonstrated to significantly impair performance. Sprayer solves this issue by partitioning the flow table, allowing each worker core to maintain its own copy of flow states with enforced access limits. While it is permissible to write to local copies of the data, it is not permissible to write to non-local copies. Every worker, however, has complete read access to both local and non-local copies of the flow states.

Sprayer’s primary implementation is based on this method, as well as the fact that flow states are rarely updated, usually when a flow starts or finishes. Sprayer effectively sends packets that update the flow status to the same core to which the first flow packet was sent—this assists Sprayer in maintaining data writing consistency.

This paper presents compelling results for using Sprayer. The maximum improvement in performance while using Sprayer is seen when there is a single flow. With a single flow, Sprayer can outperform RSS. Additionally, the results show that while Sprayer provides a consistent throughput with any number of flows, RSS performs poorly with a small number of flows and improves when they are increased. RSS provides the best performance when the number of flows in the system is sufficiently large. This behavior is due to the Sprayer disaggregating the traffic locality by spraying packets and utilizing all the PEs. At the same time, Sprayer slightly improves the latency and completely eliminates the fairness problems caused by hash collisions. While the authors discuss the throughput gains of spraying packets, they do not publish any findings demonstrating the amount of out-of-order packets introduced by the system.

The key takeaway from this paper is to use partitioned flow tables to avoid locking and to improve the synchronization of shared data structures. Since we implement our solution on the DSW1 scheduler of DPDK, such partitioning can be immensely beneficial, especially because in DSW1, the scheduling is distributed across all the workers. The authors also present an idea where the packet spraying algorithm can be completely offloaded to the NIC’s hardware due to its simplicity.
2.7.5 Packet Order Matters! Improving Application Performance by Deliberately Delaying Packets [52]

The authors of this paper explore the effects of spatial and temporal locality of network traffic on packet processing. They systemically study the throughput, latency, cache miss, and average CPU cycles per packet for varying Spatial Locality Factor (SLF). SLF is a metric introduced in this paper that describes the number of packets of the same flow that are back-to-back in the packet stream; for example, for three flows - A, B, and C, with SLF=1, the input stream is similar to ABCABC..., and SLF=2 will correspond to AABBCCCAABBCC..., and so on.

The primary idea of this paper is to improve the performance of NFV by optimizing the utilization of memory caches. This is accomplished by reordering packets so that all packets requiring the same processing steps are next to one another and exhibit high spatial and temporal locality in the input stream of the NFV. The authors find that only a slight lack of spatial locality is sufficient to sharply degrade the performance by up to three times. The initial experiments show that increasing the SLF from 1 to 16 can deliver up to 54% reduction of L1 cache misses and reduce the CPU cycles per packet by up to 53%.

Another key result of this paper is that today’s realistic internet traffic demonstrates packet localization, which is detrimental to high-speed networking. The authors studied real traffic traces captured at their organization and concluded that more than 60% of a flow’s packets are interleaved with packets of other flows. The heavy deployment of congestion control algorithms on the internet is the primary cause of low spatial locality. Such algorithms spread the incoming packet of a flow as much as possible to avoid ‘bufferbloat.’ As a result, the performance of NFVs drops because the input stream cannot be processed in per-flow batches.

Reframer is a software system developed in this paper that briefly buffers, delays, and reorders packets at the beginning of an NFV or an NFV chain to increase the spatial locality of the incoming traffic.

Even though this paper is not directly related to the work presented in this thesis, many insights from this paper help design our proposed solution. This paper highlights that even though packets are deliberately delayed, it is still possible to achieve an overall low latency by optimizing the cache efficiency. However, the performance gains depend entirely on the balance between delaying packets and cache performance.
Chapter 3

Research Methodology

This chapter discusses the methodology and the strategy used to answer the research questions. The initial sections of this chapter, describe the research process and the paradigm adopted by the thesis. Next, the testbed details including the software, hardware, DPDK application, traffic profiles, etc. are provided. Finally, the metrics collected and their relevance to the project are discussed.

3.1 Research Methods

This work employs the quantitative research style described in Portal of Research Methods and Methodologies for Research Projects and Degree Projects [53]. The project first implements all the required software components. The scheduling logic is implemented as a DPDK driver and a high-level application that uses the scheduler under its hood for packet scheduling is developed. Next, these software components are integrated into a testbed containing a COTS Traffic Generator (TG). An experimental approach [53] is taken and various experiments are conducted to test the behavior of the scheduler. By varying the type and the amount of traffic, statistics of the scheduler’s performance is collected. The collected dataset is then subjected to statistical analysis to infer relationships between various test parameters and the results. Using the Deductive approach [53], it is then possible to draw conclusions to answer our research question.
3.2 Data Collection and Validity

Since this project follows the quantitative approach, extensive testing must be conducted to get reliable and usable data. Hence, we develop a simple automation script with Python that connects to a Server through Secure Shell (SSH) and launches the DPDK application. The same script is also used to configure the TG with the traffic profile, start/stop traffic, and collect the results. Automating the testing process has a twofold advantage; 1) the tests can be run multiple times to avoid statistical anomalies. 2) No human intervention is required to run tests, thereby, reducing human error.

In this project, most of the experiments are statistically averaged by executing each test 5 times and estimating the 95th percentile of the metric. This makes the results both reproducible and reliable. Eventually, several Gnuplot scripts are used to generate the desired graphs of the collected data for further analysis and visualization of the results.

3.3 Evaluation Framework

This section describes the testbed and the configuration used in this thesis to test and evaluate our proposed solution. The testbed consists of an IXIA TG and a Server connected in a back-to-back configuration. The Device Under Test (DUT) consists of the DPDK application deployed on the Server. The devices are interconnected such that the TG generates traffic according to the parameters supplied by the user. The traffic is then forwarded to the Server. The DUT receives these packets through one of its ports and starts processing them. After processing, it is sent back to the TG through a different port so that various performance parameters can be measured and logged. A Layer 1 (L1) switch is used between the Server and TG to ensure no interconnect switching (L1 and L2) takes place, and thus does not skew or disturb measurements. The L1 switch is a part of our testbed as we use the testbeds available in Ericsson’s Lab. All connections between the components are made using 25 Gbps connections to be able to saturate the application with high-speed traffic.

Figure 3.1 shows a generalized topology of the testbed. The following sections describe each component in more detail.

In order to test the proposed solution, two software components are developed. One is the test application, which is a modified version of the DPDK’s l3fwd example [11]. The second is the scheduler itself, which is implemented as a patch over the original DSW scheduler. The patch is applied
to the DPDK drivers at the time of compilation. The l3fwd application acts as a simple L3 router, receiving packets on one port, performing a dummy route lookup, and then reflecting the packets back to another port.

### 3.3.1 Traffic Generator

The packet generator used in this project is an Ixia IxNetwork 9.0 TG capable of generating packets at a peak rate of 25 MPPS for 100-byte packets. For all experiments, we use a packet size of 100 bytes to ensure that the TG has enough space in the packets to inject the packet tracking information. The packet tracking information allows IXIA to track each packet precisely and compute parameters such as packet reordering percentages, latency, etc.

### 3.3.2 L1 Switch

Calient S320 is used as a layer 1 switch in the testbed. This switch is an optical circuit switch with ultra-low latency of <30ns and can offer switching speeds of 200Gbps [54].

### 3.3.3 Device under Test (DUT)

The test application runs on a Server with a Linux operation system. Table 3.1 provides the specifications the Server used. A few critical optimizations on the testbed are listed below [38]:

![Evaluation Testbed Diagram](image-url)
1. The CPU cores on which the application runs are isolated. This optimization excludes the cores from the Linux kernel scheduler and prevents other tasks from being scheduled on the same cores. Therefore, preventing expensive CPU migrations and context switches.

2. The CPU frequency throttling feature is turned off to stop the CPU frequency from changing dynamically. Hence consistent and reproducible results can be produced.

3. The power-saving feature for the CPU cores is turned off, and the cores are set to performance mode.

4. Automatic NUMA balancing is disabling.

<table>
<thead>
<tr>
<th>Testbed Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Host OS</strong></td>
</tr>
<tr>
<td>Kernel release: 5.4.0-131-generic</td>
</tr>
<tr>
<td>Kernel version: 113-Ubuntu SMP Thu Feb 3 18:43:29 UTC 2022</td>
</tr>
<tr>
<td>Hardware platform: x86_64</td>
</tr>
<tr>
<td>OS: Ubuntu 20.04.4 LTS</td>
</tr>
<tr>
<td><strong>Host CPU</strong></td>
</tr>
<tr>
<td>Intel(R) Xeon(R) Gold 6132 CPU @ 2.60GHz</td>
</tr>
<tr>
<td><strong>Host NIC</strong></td>
</tr>
<tr>
<td>Hardware: Intel Corporation Ethernet Controller XXV710 for 25GbE SFP28 (rev 02)</td>
</tr>
<tr>
<td>Driver: 2.20.12</td>
</tr>
<tr>
<td>Firmware: 8.40 0x8000b1fb 20.5.13</td>
</tr>
<tr>
<td><strong>DPDK Version</strong></td>
</tr>
<tr>
<td>20.11.3</td>
</tr>
</tbody>
</table>

Table 3.1: Host and guest specification

3.3.4 DPDK Application

The DPDK eventdev library is used to set up an event-based packet processing pipeline for the l3fwd test application. The design consists of a pipeline with three event queues, where each queue represents a stage of packet processing. The incoming events are sequentially scheduled through all the queues, i.e., the events are first moved to $q=1$, then to $q=2$, and finally to $q=3$. The first stage ($q=1$) implements the IPv4 L3 routing logic. The scheduling type of this stage can be changed at runtime to select from "atomic," "parallel," or our custom
logic. Then the events are transferred to \( q=2 \), where a reordering buffer is used to implement a packet reordering logic. The third stage is introduced to collect all the events from the previous queues and prepare them for TX. The queue scheduling type of the third queue is set to "SINGLE_LINK," which means only a single dequeuing port can be attached to this queue. The Tx adapter is connected to the last event queue \( (q=3) \). The adapter receives all the processed events, extracts the mbuf objects from them, and forwards them to the ethernet device for transmission. Figure 3.2 depicts the eventdev configuration.

### 3.3.4.1 Eventdev Configuration

![Event processing pipeline for L3fwd](image)

Table 3.2 shows the event ports created in the \textit{l3fwd} application. An event port is attached to one of the free CPU cores. The worker cores can then dequeue events from the scheduler, process them, and enqueue them back to the scheduler so that the events can be scheduled to the next stage of processing. It is possible to launch the test program with a maximum of 11 logical CPU cores. We dedicate 2 cores for packet I/O; the TX and RX adapter threads are spawned on 1 core each. The packet reordering worker has access to another core. The primary L3 forwarding mechanism can then be executed on the remaining 8 cores.

### 3.3.4.2 L3fwd Application

This section describes the design of the \textit{l3fwd} application used in all the experiments conducted in this thesis. DPDK bundles this application as an example to showcase its many supported features. However, for the purpose
of our evaluation, we removed some functionalities to make the application simple and lightweight. The application supports only event-based packet scheduling. At the runtime, it is possible to select the scheduler and the scheduling type for the run using the command line arguments.

Two ethernet ports are configured in the DPDK application to receive and send packets. The application gets packets from port 0, processes them, and forwards them back to port 1. Both ports are configured with one hardware queue containing 1024 descriptors for each port. DPDK allows adding software callbacks to the RX/TX queues. When a packet arrives on one of the RX or TX queues, a registered callback function is launched with the packet pointer as an argument. We leverage this functionality to add a callback to preprocess the input packets. The callback adds a sequence number to the packet’s metadata. The reordering buffer later uses this sequence number to reorder the packets if required.

The Longest Prefix Match (LPM) is the lookup method supported by the application. The LPM object uses the destination address in the packet as a key for finding the best-suited route. It returns the ID of the next hop interface to which the packet must be sent. The table is preconfigured with 16 routes at the compile time. All the routes are intended to receive packets from port 0 and forward them to port 1. The LPM tables are created in a NUMA-aware fashion, where one LPM table is created for every CPU socket.

The L3 routing logic is implemented as defined in RFC 1812[55]. First, the packet’s layer 2 headers are stripped off. Then, the destination address is used to fetch the outgoing port ID. If no route is found, then the packet is silently dropped. All non-UDP packets are also dropped at this point. Then the Time-to-live (TTL) field is decremented and checks on the IP version, packet length, and TTL are performed. All nonconforming packets also dropped. Since the application only behaves as a forwarding loop, we swap the ethernet addresses.
Figure 3.3: Application processing logic for L3 forwarding and packet reordering

3.3.4.3 Packet Reordering Setup

The reordering stage uses the sequence numbers inserted into the packet metadata at ingress to restore the original order of the packets at egress. The command line argument can turn the reordering logic on or off. When the reordering is turned off, this stage behaves like a dummy stage with no processing. The events are simply received and forwarded to the next queue. However, when it’s turned on, the events are pushed into a DPDK reorder buffer. Ordered packets are then drained from the buffer and forwarded to the next stage. The reorder buffer is a standard library that is included in DPDK, and its operation is described in Section 2.2.4. The reorder buffer is created with a size of 8192 packets. The size of the reorder buffer affects...
the performance of the application. We did not find documentation that details information for choosing the size of the reorder buffer. Hence, we choose the value that is used by the DPDK example application bundled with DPDK.

3.4 Planned Measurements and Experiments

3.4.1 Micro-benchmarking DSW with perf

To completely understand the functioning of DSW and the underlying bottlenecks, we profile the scheduler. Linux perf [56] is used to collect CPU cycle information at a rate of 500Hz from all DPDK application CPUs. However, it’s not possible to capture deep call stack traces without compiling DPDK in debug mode. Hence, all the optimizations are turned off by using ‘-O0’ and ‘-g3’ C flags for these experiments. This drastically reduces the performance of the application.

We perform two variants of benchmarking as described below. However, for both variants, the first core is used as a reordering core, the second core as the TX adapter, the third as the RX adapter, and the remaining cores as workers. During the tests, it is observed that two workers are sufficient to process all the packets with 0% loss at 10 MPPS. The test considers and plots only the functions that are important and ignores the others that have small contributions/overhead to operation.

3.4.1.1 Benchmarking with different TX rates

The first varies the input TX traffic rate from 2 MPPS to 10 MPPS to identify the bottlenecks present in functions. This test identifies the parts of the scheduler that do not scale with higher speeds. This test uses 11 cores in total (2 for packet IO + 1 for reordering + 8 workers).

3.4.1.2 Benchmarking with different number of workers

The second test uses a constant TX rate and varies the number of workers. The number of workers is varied from 2 to 6. The goal of this test is to identify the overhead accumulated as more and more workers are connected to DSW.

The tests are executed with two traffic patterns. The patterns are configured to send a total of 5K flow at a cumulative rate of 10 MPPS. However, the first traffic group sends all 5K flows at a constant 10 MPPS rate while the second traffic group partitions flows into two subsets - one consisting of 4973
flows (99.45%) at 0.8 MPPS (8%) and the other with 27 flows (0.55%) with 9.2 MPPS (92%). The traffic is partitioned as described in [57] to simulate realistic traffic distribution.

This test’s objective is to identify bottlenecks caused by flow migrations. The second traffic pattern consists of skewed traffic distribution which creates load imbalance in DSW atomic mode and, therefore triggers flow migrations according to the DSW’s load balancing policies.

3.4.2 Migrations Tests

To study the flow migration behavior of DSW, the test application is compiled in release mode and deployed to the testbed. The traffic generator is configured to send 2000 to 16000 flows in increments of 2000 at a cumulative TX rate of 20MPPS. The flows are partitioned to send 99.45% of flows at an 8% rate and the remaining 0.55% of the flows at 92% of the total TX rate [57]. The choice for the maximum number of flows is made based on the code comments (dsw_evdev.h). The version of DPDK used in this thesis supports up to 32K. DPDK’s eventdev statistics are used to collect information about the total migrations and CPU utilization of all the workers. At the same time, we use the traffic generator to collect Round Trip Time (RTT). The migration threshold is changed by configuring DSW_MIN_SOURCE_LOAD_FOR_MIGRATION to 25% from the original value of 75%.

3.4.3 Benchmarking Diffuser

This test directly evaluates the performance characteristics of the proposed scheduler with the other scheduling mechanisms available in the DPDK suite. Additionally, these tests are carried out in two different configurations. One with the reordering logic turned off, and one when the reordering is turned on. These tests will ascertain the additional overhead and any possible degradation to the performance when using the DPDK’s reordering buffer. The test traffic consists of 100 byte User Datagram Protocol (UDP) packets as shown in Figure 3.4. The headers of the packet constitute 50 bytes, while the UDP payload is allocated 50 bytes to ensure sufficient space to accommodate IXIA packet tracking (Instrumentation) data. The traffic generator is configured to send 10 uni-directional UDP flows toward the DUT. The TX rate is changed from 10 MPPS to 25 MPPS in 1 MPPS increments every 60 seconds. On the Ixia TG the flows are configured with the settings: Tx mode interleaved, src/dest mesh OneToOne, route mesh OneToOne, and uni-directional. The latency
computation mode is set to average forwarding delay (FD).

![Packet structure](image)

**Figure 3.4:** Packet structure used for the experiments

The test captures and compares the throughput, latency, reorder percentage, and CPU cycles. To compute the percentage of the shuffled packets, we leverage the TG’s sequence-checking feature. The `advancedSequenceChecking` setting with `threshold=100` is configured on the TG. Essentially, this feature uses a buffer of size 100 for each packet stream and measures the number of packets that are out of order within this buffer. The TG refers to this metric as **Reordered Frames**. Any packets out of the buffer are considered late packet and is referred to as **Late Frames**. Since late packets are also out-of-sequence packets, we compute the number of out-of-order packets as the sum of both **Reordered Frames** and **Late Frames**.

The load distribution test evaluates the scheduling fairness of the proposed scheduler by periodically profiling the event ports. The DSW scheduler provides an important API `-rte_event_dev_xstats_by_name_get()` to fetch extended statistics for each event port. The `port_load` metric of the statistics is collected at a rate of 1Hz. This variable is the ratio of the number of CPU cycles spent on packet processing to CPU frequency for a given period of time. A heat map intuitively depicting the CPU load as a function of time for all the application cores is plotted. These heat maps can then be compared for all other scheduling modes supported by DPDK.
Chapter 4

Implementation

This chapter introduces the implementation of the proposed system. However, before diving straight into the software implementation, the first few sections present the existing DSW software architecture to better understand the requirements and the rationale behind its design. This is important because it allows us to follow a similar design pattern and reuse some existing components of the scheduler. Appendix A contains the detailed call graphs of DSW.

4.1 Software Design of the Existing System

The primary design goal of DSW is to eliminate the bottleneck introduced by the scheduling core, i.e., no single worker or CPU core is responsible for scheduling events in DSW, but all the participating worker cores together perform some scheduling tasks apart from the regular packet processing. This allows the application to scale to many cores. DSW accomplishes distributed scheduling by ensuring that the events are moved to the downstream worker’s buffers as soon as the upstream worker finishes processing and tries to enqueue events back to the scheduler. When the worker finishes processing a batch of events, it returns them to the scheduler using the `rte_event_enqueue_burst()` API. This function internally invokes the `dsw_event_enqueue_burst_generic()` function inside the DSW driver, which is the entry point to event scheduling in DSW. This function performs three primary operations:

1. Execute background maintenance tasks by calling the `dsw_port_bg_process()` function.
2. Compute and update event and port statistics.

3. Identify the next event queue in the processing chain, fetch the workers associated with this queue, and enqueue the processed events to the downstream workers’ port. The function `dsw_port_buffer_event()` is associated with this task.

We discuss each of these tasks in more detail below. At every enqueue operation, the `dsw_port_bg_process()` is invoked, but the function body is completely executed only after a fixed number of calls, ensuring that the housekeeping tasks are performed only at fixed time intervals. These tasks include checking the control ring for flow migration messages, flushing event buffers to ensure events don’t linger around buffers, updating the processing load of the worker, and considering flow migrations if the worker’s load exceeds a set threshold. Every DSW port maintains a control ring buffer called `ctl_in_ring` to implement a lightweight Inter Process Communication (IPC) mechanism. The ports can use `dsw_port_ctl_broadcast()` to broadcast messages to other workers and `dsw_port_ctl_process()` to receive notifications. The `dsw_port_consider_load_update()` function estimates the worker’s CPU load by tracking the number of cycles spent on packet processing for every worker core. This information, along with the CPU frequency, is used to compute the worker’s load. The `dsw_port_consider_emigration()` function initiates the flow migration process. When the worker’s processing load exceeds a set threshold, the port broadcasts a message to all the other worker cores, notifying them to pause the emigrating flow temporarily. Then a target worker that can accommodate the new flow is identified. Finally, the global flow state is updated and an unpause message is broadcast to the workers.

The `dsw_event_enqueue_burst_generic()` function then updates the event and port statistics. DSW uses a credit-based system to track the events that are still in the system. Additionally, DSW ports call `dsw_port_enqueue_stats()` and `dsw_port_dequeue_stats()` to maintain other statistics, such as the total number of events enqueued, dequeued, the number of new, old and released events, etc. The application can then query these statistics using the `rte_event_dev_xstats_get()` API. After finishing the maintenance and statistics tasks, the enqueue function finally schedules the incoming events downstream. To do this, the scheduling logic first fetches the `queue_id` from the event and uses it to get the `dsw_queue` structure which contains information about the next queue. This information includes the scheduling type, the worker ports attached to the
queue, and the routing table - flow_to_port_map. If the scheduling type is atomic, the enqueue function uses the flow_to_port_map table to look up the destination worker. This table stores the mapping between the flow hash and its currently scheduled worker port. The dsw_flow_id_hash() function is used to convert event flow_id to a 24-bit hash. This hash is then used to index the forwarding table to store the destination port_id. However, in the case of parallel scheduling, fetching the destination port_id is much simpler. The dsw_port_get_parallel_flow_id() is used to generate artificial flow_id's in a round-robin fashion. The parallel scheduling mode does not rely on the flow_to_port_map table but rather generates flow_id such that when these ids are hashed (using function dsw_flow_id_hash()), the hashes cycle through all the serving workers of the queue one-by-one. This ensures that packets are sprayed to all the worker's cores in random order. The function dsw_port_buffer_event() contains the above-mentioned scheduling logic. However, this function also checks to determine if a flow is paused before scheduling it downstream. The dsw_port_is_flow_paused() is used for this purpose. If the flow is halted, the function temporarily buffers the flow events by calling dsw_port_buffer_paused instead of forwarding them to the next queue. Flows are paused for a short interval during flow migration to avoid packet shuffling.

The code flow above explains the scheduling logic from a high-level functional view, however, is easy to interpret that the architects of DSW have tried to minimize the interaction between multiple threads as much as possible to avoid problems associated with multithreaded programs. Each event port or worker operates more like an independent unit with little or no inter-thread communication. A lightweight signaling mechanism and a few shared variables (forwarding table and port loads) are the only IPC mechanisms implemented. The shared variables are also designed to have partitioned read-write access, i.e., only one thread can write to the variable at any time. Also, all memory access to the shared variables uses atomic operations, making it a lock-free implementation.

### 4.2 Software Design

The atomic mode processes packets of the same flow on the same core to avoid packet reordering; this, however, leads to overloading some workers and starving others. As a result, the overall system’s throughput drops and leads to packet loss. Contrarily, the parallel mode sprays packets of all flows to all the
available cores, ensuring that the scheduler achieves a perfect load balance but
at the cost of packet reordering. In other words, the extent of packet reordering
and the load balance across the system constitute a classical tradeoff; that is,
when the system achieves good load distribution (through flow migrations or
spraying), there is a higher probability of packets being processed in an out-of-
order fashion and vice versa. The rate of flow migration also plays a significant
role in the system’s performance. From observations, migrating flows often
from one worker to another improves the load balance across the system but,
at the same time, also increases the number of out-of-order packets. We use
this observation as the main idea for our solution. We design a scheduler that
optimizes the rate of flow migration such that the system can achieve a good
tradeoff between load distribution and reordering performance. Also, we make
another important observation that packets are shuffled only when different
workers process the same flows simultaneously.

Based on these observations, we designed a scheduler that behaves both
like the atomic and parallel modes of DSW. A feedback-based mechanism is
employed to constantly monitor and measure the load distribution across all
the worker cores. When a load imbalance is detected, the scheduler changes
the flow to worker assignment to improve the load balance; however, until the
next adjustment of the flow assignment, the scheduler behaves as an atomic
scheduler, processing a packet of the same flow on the same worker. While
the scheduler operates in this mode, the system does not introduce any out-
of-order packets. But, as soon as the workers experience an imbalance in
the load, the flow assignments are immediately changed. This migration can
introduce reordering but is limited only until the buffers of the workers are
flushed and the next assignment comes into effect. The scheduler adaptively
controls the flow migration rate to manage the reordering performance of the
system. If the scheduler needs to improve the throughput of the system, the
flow migration rate is increased, and similarly, to improve the packet ordering
performance, the migration rate is decreased. The load imbalance feedback
decides if the migration rate should be increased or decreased to maintain
optimal performance during the run time.

The following subsections discuss these components in more detail.

4.2.1 Flow Assignment and Adjustment

Flow assignment dictates how flows are assigned to worker threads. DSW
in atomic modes relies on a forwarding table and flow hashes to assign
flows to worker cores. When DSW is spawned for the first time, the
forwarding table is initialized with a random flow assignment. As the system operates, the scheduler further optimizes the flow assignment based on the load requirements and thresholds. However, in Diffuser, we use the flow hashes to compute the destination worker using equation 4.1, where \( n \) is the number of workers.

\[
destination_{port\_id} = (flow\_hash \mod n) \quad (4.1)
\]

Flow assignment adjustment is a process in Diffuser when the scheduler tries to reassign all the flow to different worker cores randomly. During the adjustment, the scheduler does not check or verify any system parameters (such as load) but rather does it stochastically. Equation 4.1 is modified to 4.2 to include a random variable \( nonce \).

\[
destination_{port\_id} = ((flow\_hash \oplus nonce) \mod n) \quad (4.2)
\]

Equation 4.2 is used to determine the destination worker id at the time of flow adjustment. \( nonce \) is a random number generated and synched across all the worker threads. At any given time, the nonce value is the same across all the worker threads. This random number is XOR’ed with flow hashes to enable flow shuffling and ensure that the new flow assignment is different from the previous assignment. Many works in literature use different hash functions and random number generators, such as HRW, AHRW, etc. [58] [59], to determine the flow distribution. However, using the technique described above has many advantages. Firstly, since the Diffuser is designed as a distributed scheduler it is essential to maintain the consistency of the forwarding table across all the workers. Since hash functions are deterministic, and as long as all the workers have the same nonce value, every worker can independently derive the same forwarding table. Secondly, since only the nonce value needs to be synchronized across all workers, the scheduler can operate at high flow adjustment rates without degrading the performance. The current implementation uses a global variable for the nonce value as it is easier to synchronize; however, the ring IPC mechanism of DSW can also be leveraged to broadcast nonce updates to all the workers. This improvement can yield better results.
We name our schedule - **Diffuser**, because of the stochastic approach it takes to spread flows to all participating worker cores to break down the skewed nature of internet traffic. Even though the system follows an atomic-like scheduling between flow adjustments, in the long run, the system randomly sprays packets of the same flow to multiple workers, and hence the name.

### 4.2.2 Load Imbalance Estimation

Since the flow assignment in Diffuser is random and does not directly rely on the processing load of the workers, it is crucial to ensure that the incoming workload is fairly distributed across all the workers. This ensures that all the workers are evenly loaded. Hence, Diffuser consists of a feedback system that monitors the load of all the worker cores and computes a metric of load imbalance. Similar to the work presented in [60], we calculate the coefficient of variation of load $LB(l)$ as an indicator to load imbalance. Equation 4.3 is used to compute the $LB(l)$.

$$
LB(l) = 1 - \frac{(\sum_{i=0}^{m-1} l(i))^2}{m \sum_{i=0}^{m-1} l(i)^2}
$$

Where $l(i)$ is the processing load of the $i$ port and $m$ is the number of workers attached to the queue. When $LB(l)$ is close to 1, the load across the workers is highly imbalanced and requires an adjustment; similarly, when $LB(l)$ is close to 0, all the workers are more balanced and spend the same amount of CPU cycles on packet processing. A threshold is defined for the load imbalance. Whenever the estimated imbalance exceeds the threshold, the scheduler triggers a flow adjustment to rebalance the load across all the workers. On the other hand, if the imbalance is within the threshold, the worker cores are balanced, and hence the flow assignment is left undistributed. Using this strategy, the system can adaptively control flow migrations and, as a result, influence the system’s throughput and reordering performance.

### 4.2.3 Imbalance Threshold and Other Constants

This section discusses some of the thresholds and constants defined in the scheduler and their importance. The load imbalance threshold plays a crucial role in the system’s stability and performance. For instance, if the threshold value is too high, the scheduler waits until a significant load disparity is created before triggering an adjustment. While this helps the reordering performance, the load balance and throughput suffer. On the other hand, if the threshold is
too small, the adjustment is called frequently, leading to degraded reordering performance. For all the results presented in this thesis, a threshold of 5% is used. This value was derived by performing a grid search with different threshold values and constant rate traffic.

Estimating load imbalance requires a worker to fetch the processing load of all the other participating worker cores. In multithreaded programs, the cost of reading private variables from other threads increases with an increase in access frequency and the number of workers. Hence we design the imbalance computation logic to be triggered only periodically, and the period is based on a preconfigured value. Diffuser is designed as a distributed scheduler, some synchronization variables are required to protect the system’s stability and avoid performance oscillations. For instance, When a specific worker detects an imbalance, it initiates a flow adjustment. Most likely, the other workers will also detect the same imbalance and try to trigger subsequent adjustments. To protect against this case, we use a global variable to track when the previous adjustment is carried out. Using this, a worker core can ignore the imbalance if another worker recently adjusted the flows. Hence, we define a minimum period between two consecutive flow adjustments.

4.3 Implementation

This section provides the code implementation details of Diffuser. For the implementation, we use the DSW scheduler as a base and add Diffuser as an extension over the existing implementation. The complete source code of the implementation, testing framework, and plotting files are available at [61].

4.3.1 The Enqueue Function (The Scheduling Logic)

The enqueue function is called directly by the DSW scheduler when the Diffuser scheduling mode is activated. This function essentially executes all the scheduling logic and returns the destination port_id to where the given packet needs to be enqueued for processing. This function is implemented as dsw_queue_get_spray_port_id(). It takes the source_port, the downstream queue_id, and the flow_hash of the packet as input and returns the destination port_id. Diffuser uses one forwarding table per worker (forwarding_cache), as it’s easier to maintain consistency when a single thread writes to this cache. Also, it is possible to maintain individual caches in Diffuser, as a hash function is used to populate the forwarding table.
As long as all workers use the same nonce, the cache will remain consistent across all threads.

At the beginning of this function, `dsw_port_consider_spray_bg()` is called to execute all the diffuser-related maintenance tasks. Then, the forwarding cache is checked for the flow hash; if an entry exists, the `port_id` is returned as the destination port, else the macro `GET_DEST_PORT_FROM_FLOW_HASH()` is called to compute the new destination port. However, while calculating the `port_id`, the flow hash of the packet is XOR-ed with the current nonce value. The forwarding cache is updated with the new destination port. Listing 1 shows the pseudo code of this function.

---

Listing 1 Pseudocode for for `dsw_queue_get_spray_port_id()`

```
flow_hash: flow_hash of the packet
n: number of workers attached to queue

procedure GENERATE_DEST_PORT_ID(flow_hash, n)
    return (flow_hash mod n)
end procedure

queue: the queue to which the port belongs
port: port on which the maintenance is carried on
flow_hash: flow_hash of the packet

procedure DSW_QUEUE_GET_SPRAY_PORT_ID(queue, port, flow_hash)
    dsw_port_consider_spray_bg();
    forwarding_cache ← port's forwarding cache table
    if flow_hash in forwarding_cache then
        return forwarding_cache[flow_hash]
    else
        nonce ← port's nonce value
        new_flow_hash ← flow_hash ⊕ nonce
        n ← queue's number of workers
        dest_port_id ← GENERATE_DEST_PORT_ID(new_flow_hash, n)
        forwarding_cache[flow_hash] ← dest_port_id
        return dest_port_id
    end if
end procedure
```

---
4.3.2 Background Maintenance Tasks

The dsw_port_consider_spray_bg() implements all the background tasks associated with Diffuser. The background tasks consist primarily of two jobs.

Firstly, the function checks if any other worker has changed the global nonce. If the nonce has changed, the worker reads the global nonce using an atomic read instruction. In DPDK, this is accomplished by invoking rte_atomic16_read() to read a 2-byte unsigned integer value. The function then updates the worker’s private copy of the nonce and flushes the forwarding cache of the worker. The current implementation of Diffuser uses a global nonce value that all the workers atomically read.

Secondly, the function computes the load imbalance of the workers by calling spray_compute_load_imbalance() and checks if the imbalance is greater than the fixed threshold. If this condition is true, the function verifies if other workers recently performed the flow adjustment. If no other worker has adjusted the flows, the port performs the adjustment by resetting the global nonce value with a new random number. However, before updating the nonce, the worker first acquires a lock and then performs all updates atomically. Using locks in this scenario does not degrade the performance heavily because the worker tries to get the lock only once, and if not successful, it ignores the adjustment and proceeds. Failing to acquire the lock implies that another worker has already detected the imbalance, so the current worker can safely skip.

To prevent the background function from being invoked frequently and the global nonce from being updated consequently by different workers during the same imbalance, we use counters to track when the next update is eligible to be executed. During the initial version of Diffuser, we used a CPU cycles-based approach to control the execution frequency. However, reading the TSC register repeatedly to get the timer cycles (rte_get_timer_cycles()) degraded the performance of the scheduler significantly. Hence, we moved to a more straightforward counter-based approach. Listing 2 shows the code implementation of the maintenance function.
Listing 2 Code implementation for dsw_port_consider_spray_bg()

static __rte_always_inline void
update_port_nonce(struct dsw_port* port,
    uint16_t nonce)
{
    port->nonce = nonce;
    flush_port_lcore_map(port);
}

static void
dsw_port_consider_spray_bg(struct dsw_evdev* dsw,
    struct dsw_queue* queue,
    struct dsw_port* port)
{
    if (++port->next_spray_maintain < DSW_SPRAY_MAINTAIN_COUNT)
        return;
    port->next_spray_maintain = 0;

    // update the nonce (if some other port has updated the nonce)
    uint16_t queue_nonce = rte_atomic16_read(&queue->nonce);
    if (queue_nonce != port->nonce) {
        update_port_nonce(port, queue_nonce);
        return;
    }

    // Check for imbalance (if some other port has
    // not already check and change the nonce)
    uint16_t load_imbalance = spray_compute_load_imbalance(dsw, queue);
    if (load_imbalance > port->imbalance_threshold) {
        // to make sure that ports don't frequently change the nonce
        if (rte_atomic16_add_return(&queue->next_nonce_update, 1) < DSW_SPRAY_NONCE_UPDATE_COUNT)
            return;

        // If the port does not get the lock it is fine
        // because any other port is already reset nonce
        if (rte_spinlock_trylock(&queue->nonce_update_lock)) {
            rte_atomic16_clear(&queue->next_nonce_update);
            uint16_t nonce = generate_random_nonce();
            rte_atomic16_set(&queue->nonce, nonce);
            update_port_nonce(port, nonce);
            rte_spinlock_unlock(&queue->nonce_update_lock);
        }
    }
}
4.3.3 Load Imbalance Computation

The load imbalance computation function `spray_compute_load_imbalance()` has a straightforward implementation. The function consists of a for loop that goes through all the workers attached to the queue. For each port, `rte_atomic16_read(&port->load)` API is used to read the ports processing load, and `DSW_LOAD_TO_PERCENT()` macro to convert the load from cycles to percentage. The function implements equation 4.3, but instead of computing the value between 0-1, a value between 0-100 is calculated to avoid float datatype operations. Listing 3 shows the code implementation of the function.

**Listing 3 Code implementation for spray_compute_load_imbalance()**

```c
static __rte_always_inline uint16_t
spray_compute_load_imbalance(struct dsw_evdev *dsw, struct dsw_queue *queue)
{
    uint16_t num = 0;
    uint16_t den = 0;

    uint16_t num_ports = queue->num_serving_ports;
    for(int port_id = 0; port_id < num_ports; port_id++) {
        struct dsw_port *port = &dsw->ports[queue->serving_ports[port_id]];
        uint16_t load = DSW_LOAD_TO_PERCENT(rte_atomic16_read(&port->load));
        num += load;
        den += load*load;
    }
    return (den? (100-(num*num*100)/(num_ports*den)) : 0);
}
```
Chapter 5

Results and Analysis

This chapter discusses the evaluation and analysis of Diffuser. Sections 5.1 and 5.2 provides the results of micro-benchmarking the DSW scheduler, while Section 5.3 covers the performance evaluation of Diffuser. Appendix B contains flame graphs plotted with the perf data collected during the below experiments. While our results in the following sections discuss about the functions with dominant overheads, the flamegraphs give a complete picture. Appendix C show the benchmarking results of Diffuser for 1 flow.

5.1 Micro-benchmarking DSW with perf

5.1.1 Benchmarking with Different TX Rates

The plot in Figure 5.1 shows the percentage of cycles consumed by different components of DSW. The max throughput of the application is 7.2 MPPS when launched with 8 worker cores. The plot only considers the dominant functions and skips the others with small footprints. Hence, the cumulative CPU usage row-wise does not add up to 100%.

From the plot, it is straightforward to identify the bottleneck in the system. The RX adapter reaches about 95% for 8 MPPS and does not scale more than 95% for 10 MPPS. Similarly, the throughput in the system also does not improve after a TX rate of 8 MPPS (not shown in the plot). The NIC and RX/TX adapter statistics are used to verify that RX adapter is the point of packet drops.
Figure 5.1: Benchmarking DSW scheduler with different TX rates with DSW atomic mode and 100 flows
### Results and Analysis

<table>
<thead>
<tr>
<th>TX Rate</th>
<th>Ethernet Port 0</th>
<th>RX Adapter</th>
<th>Ethernet Port 1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>rx_good_pkts</td>
<td>rx_missed_errors</td>
<td>rx_packets</td>
</tr>
<tr>
<td>2 MPPS</td>
<td>131570167</td>
<td>0</td>
<td>131570167</td>
</tr>
<tr>
<td>4 MPPS</td>
<td>266399629</td>
<td>0</td>
<td>266399629</td>
</tr>
<tr>
<td>6 MPPS</td>
<td>394633282</td>
<td>0</td>
<td>394633282</td>
</tr>
<tr>
<td>8 MPPS</td>
<td>477421455</td>
<td>52452336</td>
<td>477421455</td>
</tr>
<tr>
<td>10 MPPS</td>
<td>478162542</td>
<td>183592230</td>
<td>478162542</td>
</tr>
</tbody>
</table>

Table 5.1: RX and TX Pipeline statistics collected from DPDK PMD

Table 5.1 shows the collected statistics. With a TX rate of 8 MPPS, ethernet port 0 (Port 0 is connected to the TX of the traffic generator) starts dropping packets at the NIC PMD. The number of dropped packets increases when the TX rate is 10 MPPS. However, once the packets are injected into the application pipeline, there are no other major bottlenecks after the RX adapter. The DSW scheduler features a credit based system that creates back-pressure when it detects slow moving events in the pipeline. This back-pressure prioritizes events that are already in the pipeline over new events and causes new events to be dropped at the NIC.

The plot shows dominant functions from three software components - RX/TX adapter, ethernet driver, and DSW scheduler. Taking a deeper look into the micro-benchmark of the RX adapter reveals that:

- `i40e_recv_pkts_vec_avx2` (from the ethernet driver),
- `dsw_event_enqueue_new_burst` (from the DSW scheduler), and
- `service_valid` (from the adapter) are among the functions that consume the most CPU cycles. Function `i40e_recv_pkts_vec_avx2` resides in the PMD and receives packets from the NIC. This function’s CPU usage grows from 19% to 31% when the TX rate is increased, mostly because the PMD has to poll for packets more frequently which results in a higher CPU consumption. However, it is surprising that...
DSW consumes more than 40% (dsw_event_enqueue_new_burst + dsw_port_flush_out_buffers + dsw_port_bg_process) of the CPU for scheduling at 8 MPPS and 10 MPPS. Out of the three functions, dsw_event_enqueue_new_burst has the largest footprint, because all the scheduling operations are carried out during this function call.

The RX adapter incorporates a spinlock to protect the ethernet polling function. The lock ensures that any NIC queue is polled only by one process at any given time. Even if multiple RX adapters, each running on a different core are attached to the same NIC queue, the lock ensures that only a single polling device can access the queue at any given time. This forms the primary bottleneck of the system and prevents the application from scaling to multiple workers. In our testbed, we have a single RX and TX adapter, therefore we see constant CPU utilization for the spinlock functions. This utilization can be higher if more than one adapter is attached to the same queue, as the new adapters have to wait to acquire the lock before polling the queue, resulting in the same throughput or even worst.

The adapters are launched as DPDK service functions on service cores [62], where the adapter callbacks are executed in an infinite loop until the state of the service core becomes inactive. The service_runner_func is the parent function spawned on the service core. This function performs atomic operations to set the state of the core before invoking the infinite while loop. The loop checks if the service contains a valid pointer to a function by calling the service_valid and then calls service_run with the function pointer. In our micro-benchmark, the service_valid function takes up close to 12% of the CPU (approx 1/3 of the RX driver function). The DPDK documentation suggests multiple ways of spawning event adapters, eg. adapters deployed on the hardware with some smart NICs, as service functions, etc. Employing such mechanisms can reduce this overhead and free up cycles for the driver function or the scheduler, resulting in slightly improved performance. Also, it is possible to optimize the code of service_runner_func to reduce the invocation of service_valid in the while loop.

From the analysis above, it can be concluded that the overhead added by the DSW scheduler and the service_runn_func are high and does not allow the ethernet driver function to scale across the CPU. Hence, at high speeds, the NIC descriptors get filled up and starts dropping packets. The NIC firmware records these drops as rx_missed_errors. The ironclad rule - an ethernet queue can be polled only by a single core at any given time, forms the basis of this bottleneck. The TX adapter shows similar benchmark trends as the RX
adapter with one distinction. In the experiments, the TX adapter does not form a bottleneck. However, it is possible that the bottleneck identified in the RX adapter can also cause performance degradation in the TX adapter. The TX adapter uses driver function \texttt{i40e\_xmit\_fixed\_burst\_vec\_avx2} to transmit packets to the ethernet device. \texttt{txa\_service\_tx} is the function responsible for moving the mbufs to different ethernet queues. This function checks the destination ethernet port and ethernet queue to which the packet must be delivered and moves them using the associated driver function. This function consumes around 22% of the CPU at 10 MPPS as it goes through each event one-by-one before transferring them to the NIC queues.

The plots for the reordering and worker cores are present in the bottom section of Figure 5.1. There are no traces for the reordering buffer in the graph because the test uses atomic scheduling. Since 8 workers cores are used, the \texttt{lpm\_process\_event\_pkt}, the function which performs the L3 routing, occupies only a max 5.2% of the CPU. This proves that the packet forwarding process does not introduce any bottlenecks. The atomic mode has different/random flow assignments in every test iteration (different speeds) and therefore results in different load assignment each time. Figure 5.2 uses the parallel scheduling mode and shows the load of \texttt{lpm\_process\_event\_pkt} increases with TX speed. For the worker cores, \texttt{dsw\_port\_bg\_process} consumes around 14% of the CPU, whereas the same function in the plots for the RX and TX adapter only consumes around 10%. The extra 4% could be due to the IPC mechanism of the DSW scheduler. This IPC mechanism is used for signaling and flow migrations. In the earlier case, the TX and the RX adapter are the only core, and IPC overhead is not present.

Figure 5.2 is a similar micro-benchmark plot executed with DSW parallel mode and active reorder buffer. The primary objective of this test is to ascertain the overhead of the reordering buffer. The plots show that the reorder buffer adds around 10% of total CPU overhead ($\texttt{rte\_reorder\_insert} + \texttt{rte\_reorder\_drain} + \texttt{rte\_pause}$). In this iteration, the reorder buffer does not form a bottleneck, but increases CPU utilization and leads to higher packet loss. In our setup, the RX adapter forms a bottleneck at the ingress of the pipeline and fails to overload the reorder buffer. The reorder buffer can still degrade performance at higher speeds in terms of packet loss.
Figure 5.2: Benchmarking DSW scheduler with different TX rates with DSW parallel mode and 100 flows
### 5.1.2 Benchmarking with Different Number of Workers

#### Worker with Traffic Pattern 1

<table>
<thead>
<tr>
<th>Workers</th>
<th>CPU Cycles Usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5.97 12.58 9.87 12.96 26.02</td>
</tr>
<tr>
<td>3</td>
<td>7.31 8.06 14.79 13.37 16.98</td>
</tr>
<tr>
<td>4</td>
<td>7.33 5.91 17.98 14.28 12.33</td>
</tr>
<tr>
<td>5</td>
<td>7    4.66 20.96 14.34 10.97</td>
</tr>
<tr>
<td>6</td>
<td>6.67 4.18 24    13.85 9.06</td>
</tr>
</tbody>
</table>

#### Worker with Traffic Pattern 2

<table>
<thead>
<tr>
<th>Workers</th>
<th>CPU Cycles Usage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7.04 11.68 10.81 11.42 25.12</td>
</tr>
<tr>
<td>3</td>
<td>7.31 4.73 16.97 15.31 10.7</td>
</tr>
<tr>
<td>4</td>
<td>7.16 4.61 19.61 14.85 10.31</td>
</tr>
<tr>
<td>5</td>
<td>7.27 4.63 20.95 14.29 10.31</td>
</tr>
<tr>
<td>6</td>
<td>6.88 3.08 25.17 15.02 6.47</td>
</tr>
</tbody>
</table>

Figure 5.3: Benchmarking DSW scheduler with different number of workers, for 5000 flows at 10 MPPS and default migration thresholds

The plots in Figure 5.3 show that for a given number of workers, the performance for both traffic patterns is almost similar. A deeper look at the scheduler’s statistics revealed that regardless of the traffic pattern the scheduler migrated only around 25 flows (0.005%) during the test. DSW identifies overloaded cores by comparing the estimated load with preconfigured thresholds. The lower threshold `DSW_MIN_SOURCE_LOAD_FOR_MIGRATION` is set
to 70% while the upper limit `DSW_MAX_TARGET_LOAD_FOR_MIGRATION` is set to 95%. The TX rate used in our experiments fails to saturate the worker cores. Additionally, when the number of workers is increased, the packet processing load decreases, making the worker cores ineligible for flow migrations. Hence we see a similar benchmark result for both traffic patterns.

As the number of workers is increased, the packet processing load experienced by `lpm_process_event_pkt` decreases. At the same time, the footprint of `dsw_port_bg_process` increases by about 2-3%. The rise in CPU usage can be attributed to the IPC mechanism. DSW uses ring buffers to implement IPC among workers, and with more workers, the background function has to fetch messages from all message rings. In worst cases, this function can occupy the CPU for longer periods if the traffic is unstable and requires many flow migrations. The `dsw_port_flush_out_buffers` is the other function that grows with the number of workers. For two cores this function only takes around 11%, while for six workers it consumes around 25%. This is because each DSW worker maintains two data structures essential for scheduling. The first structure is a ring buffer called `in_ring` and contains all the events that need to be processed by the worker. The second structure is the `out_buffer` (one `out_buffer` per port) (refer to the architecture diagram in Section 2.6.1) that contains events that are processed and needs to be moved to the other workers `in_rings`. The `dsw_port_flush_out_buffers` is responsible for iterating over all the `out_buffers` and flushing them. Therefore, as the number of workers increase CPU utilization of this function also increases. With two workers, each worker spends approximately 40% of the CPU on scheduling and has only about 60% remaining for actual packet processing. The overhead linearly grows and reaches 50% with six workers. Out of the total overhead, the scheduling decision (flow table lookup, etc.) consumes around 5%, while buffering, copying packet pointers, IPC, and housekeeping operations consume the remaining cycles.

In conclusion, DSW spends large amounts of CPU cycles on scheduling events, and the overhead linearly increases with the workers. The distributed nature of the scheduler (each worker schedules packets in addition to packet processing) is the primary reason for the additional overhead. Additionally, IPC techniques and multiple event buffers are required to perform distributed scheduling. While the scheduler can provide flexibility and scale to many cores, the overhead of having many workers can effectively reduce the available CPU capacity for packet processing. For example, with six workers almost half of the cycles are spent on scheduling, and this overhead can be
higher if DSW has to perform more flow migrations.

To force flow migration, we reduce DSW_MIN_SOURCE_LOAD\_FOR_MIGRATION from 70% to 25% to force DSW to recognize workers that have utilization of over 25% as overloaded and initiate flow migrations. Figure 5.4 shows the profiling data for the test with modified migration thresholds. The increased migration raises the footprint of dsw_port_bg_process. For traffic pattern one, the overhead increases from 13.55% to a maximum of 19.47%. Similarly, for traffic pattern two, the overhead increases from 11.44% to 14.44%. The results are surprising as traffic pattern one triggers more migrations than traffic pattern two even though the second pattern contains a more skewed traffic distribution. The reason for increased migrations in traffic pattern one is that all flows are of the same size and migrating such a flow to a target core will also introduce an imbalance on the target core causing the target core to get overloaded. DSW does not contain flows of different sizes to fine-tune load balance and hence enters a state of instability. To verify the results, we check the extended statistics of DSW to fetch the total number of migrations.

<table>
<thead>
<tr>
<th>Number of Workers</th>
<th>Migrations with Traffic Pattern 1</th>
<th>Migrations with Traffic Pattern 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>285090</td>
<td>96129</td>
</tr>
<tr>
<td>3</td>
<td>630250</td>
<td>2972</td>
</tr>
<tr>
<td>4</td>
<td>899993</td>
<td>166796</td>
</tr>
<tr>
<td>5</td>
<td>1117173</td>
<td>42995</td>
</tr>
<tr>
<td>6</td>
<td>1039676</td>
<td>49957</td>
</tr>
</tbody>
</table>

Table 5.2: Total number of migrations with traffic patterns 1 and 2

Table 5.2 shows the total number of migrations for both traffic items. The number of migration carried out for both traffic patterns are very high. However, migrations with pattern one are higher than with pattern two. From the statistics, it is clear that migrations cause the footprint of dsw_port_bg_process to grow. The experiments in the next section try to study the migrations in DSW.
5.2 Migrations Tests

Plots in Figure 5.5 show the data collected in this experiment. For all traffic flow configurations (2k - 16k flows), the average CPU utilization is only 20% when the application is launched with eight workers, and the load is above the minimum migration threshold for both the two and four-worker configurations. With six workers the utilization is exactly at the lower threshold. The number of migrations with eight workers is < 10,000 because a few workers can reach
the lower migration threshold (25%) and start migrating flows. The same is true for the six-worker configuration. For the four-worker configuration, the number of migrations slowly starts increasing from 4K flows (the migrations for 10K - 14K flows are much higher and are outside the upper limit of the y-axis). The two-worker configuration shows a similar trend, but migrations are more unstable than in the four-workers’ case. The large error bars indicate the migrations are very random and change from iteration to iteration. The primary reasons for migration instability with two workers are fewer workers are available for fine-grain load balancing and the CPU utilization is close to the upper limit of the migration threshold.

To conclude, the results show that regardless of the number of workers, DSW migrates too many flows to achieve optimal load balance. Even though the migration threshold was altered in these experiments, the migration strategy was unchanged. The unaltered DSW will have lower migrations but will still have many unnecessary ones. Under specific traffic conditions, the DSW scheduler can introduce instabilities and migration oscillations.

The latency for the eight workers’ configuration is the highest among the others. The results show that the latency does not show a strong relationship with the number of migrations. The DSW statistics show that the migration latency is a few hundred thousand CPU cycles and thus cannot be seen on the (us) Y-axis. Despite some discrepancies, a trend seen in the plots is as the number of workers increases the overall latency also increases (for all the flow configurations). This could be because of the overhead of \texttt{dsw\_port\_flush\_out\_buffers} function identified in section 5.1.2. As the number of workers increases the scheduler takes more time to flush out the event buffers leading to increased latency in the order of micro-seconds. In all cases, we see at least a 50us difference in latency between two and eight-worker configurations.
Figure 5.5: CPU utilization, Migrations, and Latency plots for DSW with different number of flows.
5.3 Benchmarking Diffuser

5.3.1 Reordering and Loss Performance

Figure 5.6: The packet reorder and loss percentages with different DSW modes for 10 flows

Figure 5.6 shows the reordering and the loss percentage for different DSW scheduling modes and Diffuser. The reordering plot shows that parallel scheduling always introduces around 70% of out-of-order packets. However, high reordering is expected as the parallel mode employs a packet-by-packet round-robin for scheduling. On the other hand, the Diffuser manages to keep the packet reordering rate below 10% for all TX Rates from 10 MPPS to 25 MPPS. Diffuser provides a robust packet reordering performance compared to the parallel mode. The reordering percent for the atomic mode is skipped in this plot to improve readability. The atomic mode does not produce any reordering, and the percentage is always 0%. From the plots above, it is certain
that DPDK’s reordering buffer can reassemble all the packets in the correct order. The figure shows that while using the reordering buffer, no out-of-order packets are transmitted from the workers.

The lower plot in Figure 5.6 shows the packet loss for the DSW scheduling modes. The scheduler does not drop any packets until 19 MPPS for all modes, but losses slowly creep in for TX rates above 19 MPPS. The atomic mode performs much better compared to Diffuser. The reason for better performance is because of the fewer scheduling operations executed by the atomic mode (when flow migrations are not involved). After 19 MPPS, all three modes, atomic, parallel, and diffuse, start dropping packets. This behavior may not be specific to the scheduling mode but rather is a bottleneck of the RX adapter that was identified in the earlier sections. A similar trend can also be observed in the other plots presented below. From this benchmarking, we conclude Diffuser has a good performance in terms of out-of-order packet processing but, also has higher packet loss. The reason for packet loss is the constant migrations of flows from core to core. The migration itself consumes some cycles, and additionally, after the migration, the cache locality is destroyed and hence the packet processing slows down. We observe that Diffuser with ROB drops fewer packets, this is because ROB provides a buffer where packets are accumulated for a short while before sending them out.
5.3.2 Throughput Performance

Figure 5.7 compares the throughput performance of the Diffuser with the different modes of DSW. From the plot, it is clear that the throughput linearly scales from 10 MPPS to 19 MPPS for all scheduling modes; after this, the throughput plateaus until 25 MPPS. In the linear section of the plot, it can be observed that all scheduling modes provide the same throughput performance. In the non-linear section, atomic mode performs better than other modes, especially Diffuser. This is the expected result as the atomic mode has the best cache efficiency. Additionally, in these tests, we used a 25 MPPS traffic generator with 8 application cores, and hence even in atomic mode, the workers were not completely saturated or even triggered any flow migrations. Therefore, the results presented in this section are ideal.
5.3.3 Round Trip Time

Round trip time (RTT) is the time taken for a packet to leave the TX port of the traffic generator, pass through the DPDK application, and return to the RX port of the traffic generator. Figure 5.8 plots the RTT for packets with different scheduling modes. With atomic mode, the packet takes slightly shorter to exit the application because the scheduler processes the packet belonging to the same flow on the same worker core. Hence, no frequent migration operations are carried out. Around 18 MPPS we see that the RTT of the packets increased by almost 3 times, this is due to the packets dropped at the RX adapter. The increase can be explained with two rationales. First, due to the packet drops, the TG miscalculates the RTT of the return packets. Second, we conclude that the RX adapter causes the packets to be buffered in the NIC’s RX descriptors for longer durations and, hence increases the latency.
5.3.4 CPU Load Fairness

Figures 5.9 and 5.10 plot the CPU load as a heatmap on a timeline to understand how different schedulers distribute incoming workloads to the available worker threads. The CPU load is queried every 1 second using the DSW’s xstats API. The traffic profile of the incident traffic is described in section 3.4. The atomic scheduler distributes the load unevenly. From the plot, it is evident that cores P1, P2, and P4 are more loaded than cores P0 and P3. The parallel mode can achieve a perfect load balance because it uses a packet-by-packet round-robin. In the case of Diffuser, it can be seen that the scheduler performs many flow migrations during the operation. As soon as the scheduler detects an imbalance, a flow adjustment is carried out. The imbalance can be visualized when looking at the core loads row-wise. The different colors indicate that the core loads are indeed imbalanced, and the change in the color after the adjustment can also be noticed.

An exaggerated version of this test can be seen in Figure 5.10, where instead of using many flows, a single flow is used to understand the scheduler load distribution. In the atomic mode, only one core (P0) processes packets while the others are idle, while parallel and spray have similar distributions.
Figure 5.10: CPU loads under atomic, parallel, and diffuse when 1 flow is directed toward the application
Chapter 6
Discussion

The work carried out in this thesis addresses two concerns. Firstly, the DSW scheduler is studied in depth to identify performance bottlenecks. Then, we propose a packet spraying scheduler-diffuser and compare its performance with other DSW modes.

DSW is a software scheduler that adds an event-based programming model to DPDK. The primary design goal of DSW is to eliminate the bottleneck presented by the single scheduling core and, hence DSW distributes the scheduling task among all the workers. The micro-benchmarking of the DSW scheduler reveals many bottlenecks in the system. The most prominent of them is the bottleneck introduced by the RX adapter. An RX adapter can poll a single NIC queue at any given time, forcing unserviced packets to be dropped at the NIC even before they are injected into DSW for scheduling. Hence, VNFs use technologies like RSS and RSS++ to distribute incoming traffic to multiple NIC queues. Both RSS and RSS++ require specialized hardware-programmable RSS indirection tables on the NIC. However, this hardware may not be available in all deployments, especially with the current trend of moving applications to the cloud. Even DPDK supports cloud-native applications running in containers[63][64]. Hence there is no guarantee that specific hardware is supported by all the cloud vendors. Therefore, DSW takes a software-based approach to scheduling.

The results also highlight some critical design factors to consider while designing a packet/event scheduler. Distributed scheduling might remove the constraints of a single scheduling core but will add overhead in terms of other synchronizations. In DSW, flow migrations require IPC, and therefore, when the number of migrations or workers increases, the CPU utilization also grows. Flow migrations are expensive and affect both the latency and throughput.
In most schedulers, when a flow is migrated from one processing core to another, the packet processing on the target core is not triggered until all the packets in the source core’s buffer are flushed. This temporary pause in the flow processing contributes to increased tail latencies. Additionally, migrating a flow from one worker to another will also require the flow’s state to be transferred. If this is not possible (using a shared flow state), then the throughput is harmed due to disrupted cache efficiency.

The second part of the thesis involves investigating the state-of-the-art schedulers and proposing Diffuser. Static load balancers such as RSS have many disadvantages. The load balance of the system entirely depends on the distribution of the hash function, and improper use of the hash function leads to a suboptimal load balance. RSS does not utilize the system efficiently when too few or too many flows are active. RSS has no feedback system to access the load across the system and balance the core utilization. Therefore, systems like RSS++ and Dyseect use an adaptive load-balancing mechanism. They constantly track the load across all the cores and optimize them by reassigning shards (logical groups of flows) to different cores. These systems also have an auto-scaling feature that automatically turns on and off the cores based on the required workload. While such systems provide very robust performance, we did not use a similar design for Diffuser for the following reasons:

1. Diffuser is completely implemented in software and does not rely on the hardware features implemented in NIC.

2. RSS++ and Dyssect require a control plane capable of collecting information from all the cores and solving complex optimization problems. Solving optimization problems can take up to 500us (including data aggregation and preprocessing) in the best case. Since DSW is a distributed scheduler, it would be very complex to implement and synchronize such control and data planes.

3. NIC APIs vastly vary from NIC to NIC and from different vendors. Using features of the NIC can tether the application to specific vendors and therefore prevent the application from being compatible with all NICs. For example, Mellanox needs the NICs to be rebooted after updating the RSS tables, while Intel NICs have no such requirement.

4. Performance of RSS++ and Dyseect depends on the number of shards in the RSS table. Both these mechanisms rely on migrating shards as opposed to individual flows. Hence, even with a heavily partitioned flow
table, it’s impossible to ensure, that two large flows do not overlap on a single shard.

Spraying packets comes with two major drawbacks - cache inefficiency and packet reordering. We argue that as long as the NF is not stateful, then packet spraying performs better than RSS++ because it can achieve perfect load balance. Similar results are stated in both Sparyer[50] and RSS++[16]. Hence, the primary focus of this thesis is the drawback of out-of-order packet processing. We believe that packet spraying (Diffuser) can make a successful packet scheduler for fastpath applications due to the following reasons:

1. The fastpath application behaves only as a simple router, where the workload consists of table lookups and packet forwarding. If any packets require additional processing they are forwarded to the Linux stack.

2. The fastpath application receives many flows. Hence, as the number of flows increases the cache efficiency also increases [15].
Chapter 7

Conclusions and Future work

This last chapter completes the degree project by discussing some of the conclusions, limitations, and future improvements drawn from this work.

7.1 Conclusions

The thesis aimed to design a new scheduling mode for the DPDK’s DSW scheduler. Specifically, the work started with two research questions: What are the primary bottlenecks in DSW? Is it possible to design a packet scheduler to achieve both good load balance and in-order packet processing? To answer these questions, we proposed Diffuser - a distributed scheduler that spreads the processing load across all the workers using stochastic flow assignment while still maintaining in-order packet processing as much as possible. The motivation for Diffuser is based on the observation that packet spraying leads to better load balance across the system and, as a result, improves the throughput and packet loss characteristics. Since the flow sizes of the realistic internet traffic are skewed, it is necessary for the scheduler to break traffic locality in order to deliver higher performance, and packet spraying is a potential solution to achieve this. However, we also conclude that packet spraying can only be used as long as the workload is stateless and does not depend heavily on the flow states. In this work, we justify that packet spraying could work well as a load-balancing mechanism for fastpath applications. However, to ascertain this claim it is still required to carry out more experiments by simulating workloads experienced by fastpath applications.

By profiling the DSW scheduler during operation, we uncover many bottlenecks in the system. We noticed DSW did not saturate the workers
because the RX adapter polled a single NIC queue. Hence enhancements are required to allow the DPDK’s RX adapter to poll multiple NIC queues simultaneously before injecting the packets into DSW for scheduling. DSW then needs to load balance across all the workers. Also, the distributed nature of the scheduler adds additional overhead due to IPC, synchronization, and background activities. The RRT of the traffic increased as more workers were attached to DSW. Additionally, the flow migration component moves far too many flows to balance the load. To conclude, with the l3fwd application, we noticed DSW consumes a lot of cycles only for scheduling, approximately 40% of the CPU when triggered with six workers.

The evaluation of Diffuser further bolsters the answers to these research questions. Compared to the parallel mode, Diffuser can reduce packet reordering by 70% at the peak rate of 25 MPPS with 10 flows. Similarly, it reduces the packet reordering by 50% in the single flow (25 MPPS) scenario; at the same time, Diffuser improves the throughput by around 30% when compared to the atomic mode by distributing the single flow to multiple cores. These improvements come at the cost of other performance characteristics such as throughput, loss, and latency. But to concretely comment on these observations, we need to benchmark the other modes of DSW at 100% load. Unfortunately, this was not possible in our testbed due to the availability of high-speed traffic generators and also because of the single RX adapter. Additionally, the reordering performance with the small number of flows needs further improvements. Some initial experiments showed that these improvements are possible but were not pursued due to the short time frame.

In conclusion, Diffuser shows promise as an event scheduler that provides good performance. Even though it’s not perfect, it is the first step toward designing a practical scheduler and has the potential to be integrated into the DPDK SDK for public use.

### 7.2 Limitations

Following are some of the limitations of the proposed system and evaluation:

1. In the testing framework, we deployed three event queues. The first queue is enabled with Diffuser. The Rx adapter is attached to the first queue, and test traffic is injected through the adapter. To thoroughly test the distributed feature of the scheduler, it is necessary to deploy another additional queue at the beginning of the pipeline with parallel mode enabled. Then multiple workers can be attached to this new queue.
to allow all the workers to enqueue packets to Diffuser simultaneously. This sort of testing was not carried out during the evaluation.

2. Diffuser was tested with two traffic profiles, ten flows of 100-byte packets, and one single flow with 100-byte packets. However, It is also critical to try the scheduler with real traffic traces (for example, CIADA traffic traces) to see how the scheduler performs with actual traffic. We skipped this test because it involved deploying the Trex traffic generator in the testing framework.

3. We use a simple NF - l3fwd with a small LPM table. L3fwd, therefore, performs only stateless operations and consumes only a few cycles. However, for future work, we propose Diffuser be benchmarked with a stateful NF or NF that consumes a few thousand CPU cycles.

4. The traffic generator used for testing can produce traffic at a peak rate of 25 MPPS for 100-byte packets, and the sample application was able to process approximately 14 MPPS per core. Hence this forced us to change the migration thresholds for the migration test. Better results can be achieved when the same test is run with a faster traffic generator, for example, 100 MPPS.

5. For micro-benchmarking, we compiled DPDK with the -O0 compiler option, which turned off all optimization in the application. This possibly could have made the application less efficient and biased the plots shown in this work. Future work can consider running these experiments with code optimization enabled.

### 7.3 Future work

Some areas for future work are listed below:

1. The diffuser can be improved to use the DSW’s ring IPC to broadcast nonce updates to the other workers. Diffuser currently relies on global shared variables and locks to facilitate nonce updates. Rings can help remove these locks and improve the scheduler’s performance as the global variables introduce memory inefficiencies.

2. Diffuser’s scheduling logic can be slightly updated to perform better with scheduling a small number of flows. For example, when a single flow traverses the application, the workers are always in load imbalance
because a flow is directed towards only one worker at any given time. Such imbalance introduces frequent adjustments, reducing the overall throughput.

3. A regressive grid search needs to be performed to understand how some constants in Diffuser affect the system’s performance. For example, the spray imbalance threshold is set to 5%, but a detailed test is required to understand how the threshold value can affect performance.

4. Study and replicate the traffic profile seen on Fastapth. This profile can then be used to find the best scheduling mode. This work speculates why packet spraying should be for fastpath traffic, but we do not conduct any experiments.

5. Bottlenecks identified during micro-benchmarking can be fixed and submitted to DPDK.

6. Adopt per-core flow tables and sharding techniques used in RSS++.
References


Appendix A

Callgraph for Functions in DSW

The figures below show the sequence of callgraphs plotted with the data collected with perf. It is difficult to fit a complete graph as an image on an A4-sized PDF document. The complete interactive figures are available in [61].
Figure A.1: Call graph for RX adapter
Figure A.2: Call graph for TX adapter
Figure A.3: Call graph for worker core
Figure A.4: Call graph for reordering core
Appendix B

Flamegraphs for DSW

The figures below show the flame graph plotted with the data collected during benchmarking. The flame graphs show the hottest functions in the whole call trace of the application. The complete interactive flame graph is available in [61].
Figure B.1: Flamegraph showing different components of DSW under operation for 5 workers
Figure B.2: Flamegraph showing different components of DSW under operation for 11 workers
Appendix C

Benchmarking Diffuser with a Single Flow

Figure C.1: The packet reorder and loss percentages with different DSW modes for 1 flow
Figure C.2: Throughput with different DSW modes for 1 flow

Figure C.3: Round trip time with different DSW modes for 1 flow

I detta examensprojekt utvärderar vi DPDKs (Dataplane Development Kit) händelseschemaläggare och hittar många flaskhalsar i prestanda och lastbalans och paketordning är en utmaning på paketnivå kombinerat med avancerad parallellism innebär dock att de behandlade paketen avgår i oordning. Att samtidigt optimera både lastbalans och paketordning är en utmaning.

---

**Keywords[eng]:** Packet scheduling, Scheduling, Out of order, Data plane development kit, Parallel processing, Network processor

---

Efterfrågan på höghastighets-nätverksapplikationer har gjort nätverksprocesorernas (NP:s) och centrala beräkningsenheter (CPU:s) parallellism, komplexa och innehållande många processorkärnor. Denna parallellitet kan endast utnyttjas fullt ut av den underrörande paketschemaläggaren genom att effektivt utnyttja all skillnad i kärnor. Varinvigt har paketschemaläggaren skickat paket till olika kärnor baserat på flödesgranularitet, vilket medför trafik-lokalitet. En bra belastningsbalans mellan processornas flödesmått och minskar förlorade paket. Detta skickar schemaläggaren på paketnivå istället flöde till kärnan med en paketgranularitet för att förbättra lastbalansen. Schemaläggning på paketnivå kombinerat med avancerad parallelism innebär dock att de behandlade paketen avgår i oordning. Att samtidigt optimera både lastbalans och paketordning är en utmaning.

"Keywords"[:sv] Paketschemaläggning, Schemaläggning, oordning, Dataplansutvecklingskit, Parallellbearbetning, Nätverksprocessor }