Preprints.org

Technical Note

Not peer-reviewed version

Implementation and Performance Optimization of a DPDK Packet Gateway on Manycore CPUs

Daisuke Sugisawa *

Posted Date: 21 October 2025

doi: 10.20944/preprints202510.1658.v1

Keywords: DPDK; manycore; cache coherence

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 October 2025 doi:10.20944/preprints202510.1658.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Implementation and Performance Optimization of a DPDK Packet Gateway on Manycore CPUs

Daisuke Sugisawa

Xander, LLC. Shibuya, Tokyo, Japan; daisuke.sugisawa.xander@gmail.com

Abstract

Since approximately 2005, major processor manufacturers have shifted their architectural focus from instruction-level parallelism (ILP) toward multicore and manycore parallelism to achieve higher performance. Rather than relying on deeper pipelines and speculative execution, performance gains have increasingly been realized through thread-level parallelism (TLP). Consequently, the responsibility for efficiently utilizing processor resources has transitioned from hardware mechanisms to software implementations. This technical note examines design strategies for achieving deterministic, high-throughput packet processing on manycore architectures using the Data Plane Development Kit (DPDK). It presents a simplified Packet Gateway (PGW) pipeline implementation, analyzing cache-coherence effects, NUMA-local memory allocation, and multicore scheduling patterns critical to maintaining per-packet processing budgets under nanosecond-level constraints.

Keywords: DPDK; manycore; cache coherence

1. Introduction

At modern data rates, the processing time available for a single packet is extremely limited. For instance, at 10 Gbps (14.88 Mpps), only 67.2 ns ($\approx$ 201CPU cycles @ 3 GHz) are available to process each packet. Under such constraints, even a single CPU cache miss (32 ns) consumes nearly half of the per-packet processing budget. Two consecutive cache misses would exceed the allowable processing delay per packet (64 ns).

In high-performance packet processing environments—such as those built using the Data Plane Development Kit (DPDK)[2]—the design must therefore minimize cache misses, system calls, and synchronization overheads.

On typical Intel Xeon architectures, socket buffers (s_buf) often reside in the L3 or L2 cache, and packet data can be placed directly into the cache through Data Direct I/O (DDIO) or Direct Cache Access (DCA) mechanisms.

The table below summarizes typical latency and synchronization costs observed on Intel Xeon systems [4].

The programmer is provided with a synchronous access interface to the memory hierarchy, while the underlying hardware employs complex mechanisms and architectural designs to ensure that these accesses appear consistent and coherent across cores and threads. However, while such abstractions simplify programming, application designers still bear the responsibility to understand hardware-level coherence mechanisms—such as cache coherence protocols—and to design memory access patterns that align with them.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 October 2025 doi:10.20944/preprints202510.1658.v1

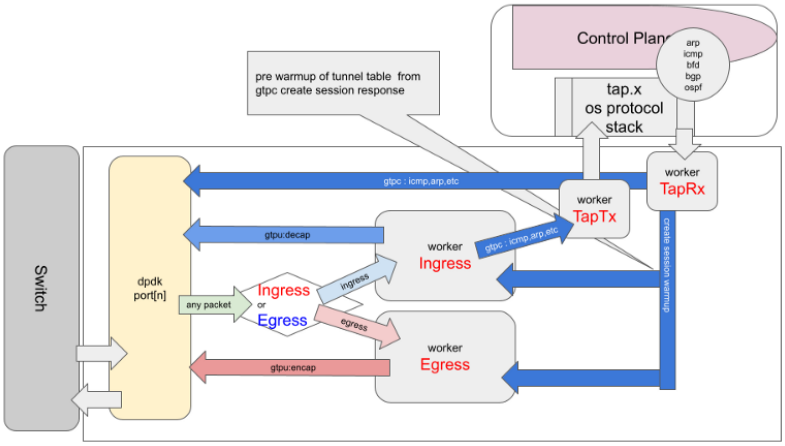

Figure 1. PGW Overview

Table 1. Internal Performance

Key	Value	Note
67.2 ns	201 cycles @3GHz	at 10Gbps / 14.8Mpps, time available for processing a single packet
CPU cache miss	32 ns	CPU cache miss time
CPU cache miss x 2	64 ns	Running out of available delay per packet
socket buffer: s_buf	fast	Hits L3/L2 cache in most cases
placed to L3 cache directly	packet cache	at Intel E5-xx, Data Direct I/O (DDIO) or DCA
L2 access cost	4.3 ns	lat_mem_rd 1024 128
L3 access cost	7.9 ns	lat_mem_rd 1024 128
atomic lock	8.2 ns	17–19 cycles
optimized spin lock	16.1 ns	34–39 cycles
system call overhead	too big enough	a few system call invocations consume over 67.2 ns
synchronized-cost		
spin_ [lock/unlock]	34 cycles 13.943 ns	simple
local_BH_ [disable/enable]	18 cycles 7.410 ns	SW interrupt
local_IRQ_ [disable/enable]	7 cycles 2.860 ns	HW interrupt
local_IRQ_ [save/restore]	37 cycles 14.837 ns	HW interrupt + status

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 October 2025 doi:10.20944/preprints202510.1658.v1

2. PGW

We implemented a Packet Gateway (PGW) system, [1] referred to as mixi-PGW, using the Data Plane Development Kit (DPDK) framework. The objective of this paper is to present and analyze several simplified pipeline scenarios of the PGW module built with DPDK, in order to illustrate key design considerations and performance characteristics of multicore packet-processing architectures.

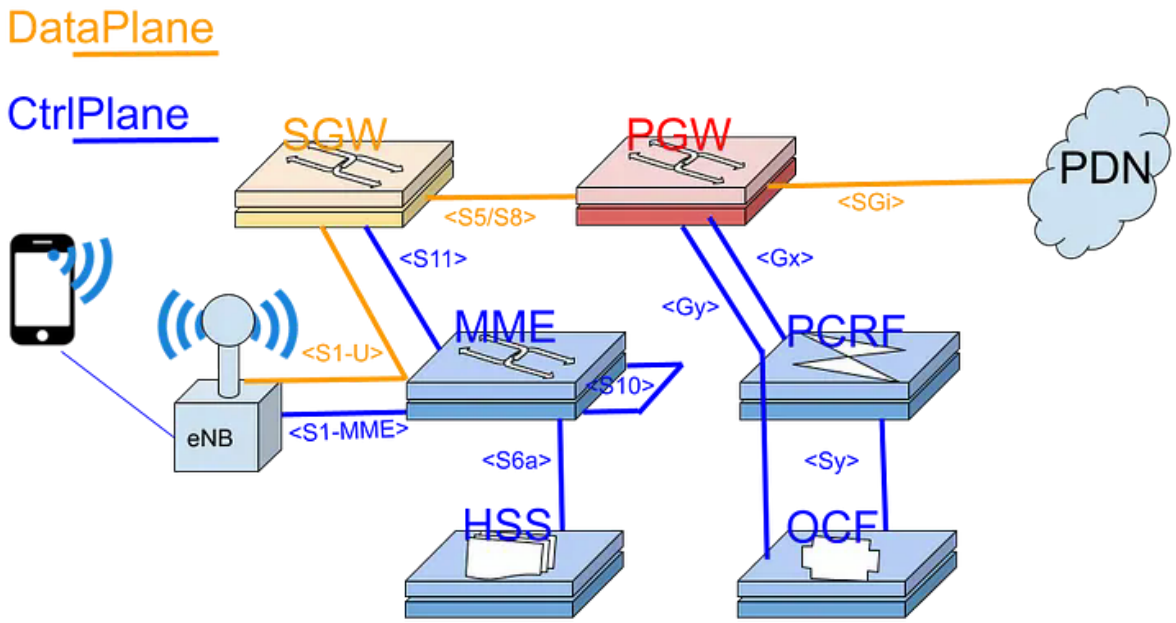

Figure 2. pgw placed in mobile-network

2.1. PipeLine Stage

To efficiently handle a large number of sessions within a single PGW data plane, it is essential to design a processing pipeline that achieves a well-balanced functional partitioning across CPU cores while maintaining low-latency operation Figure 3.

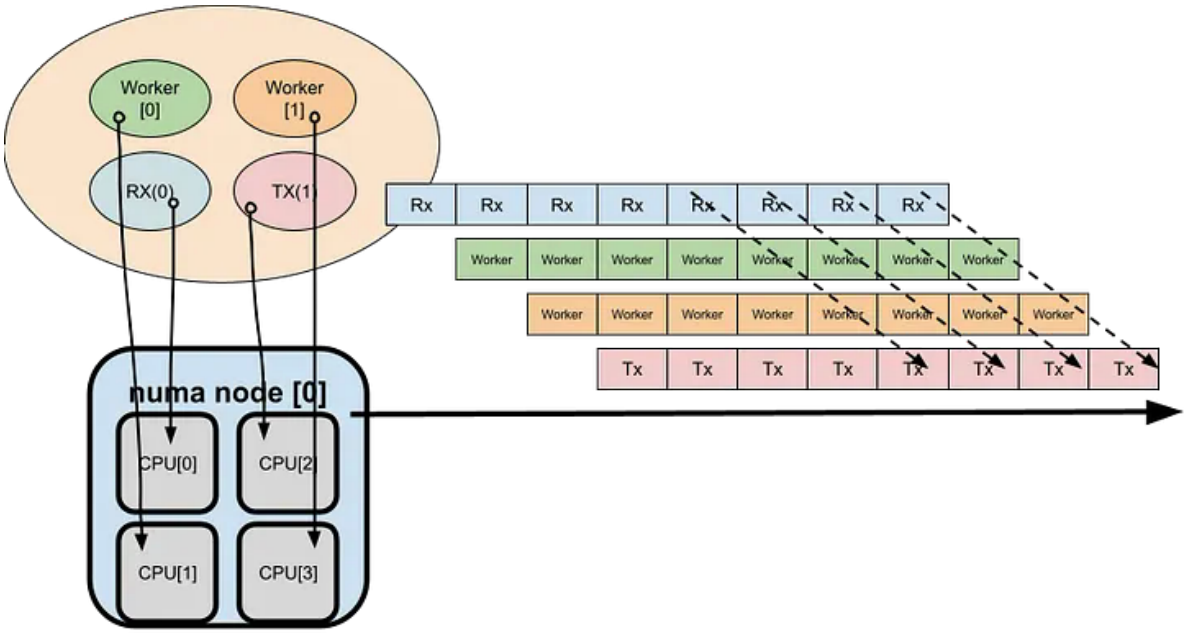

Figure 3. Overview of balanced pipeline stage

[1] https://github.com/mixigroup/mixi-pgw

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 October 2025 doi:10.20944/preprints202510.1658.v1

2.2. *mbuf Pools*

The receive (RX) functions cannot be invoked in parallel across multiple CPU cores, as packet reception for a given queue must remain serialized to preserve order and consistency. To achieve efficient utilization of the L3/L2 cache hierarchy, buffer pools are allocated on a per-NUMA-node and per-CPU-core basis, as illustrated below Figure 4.

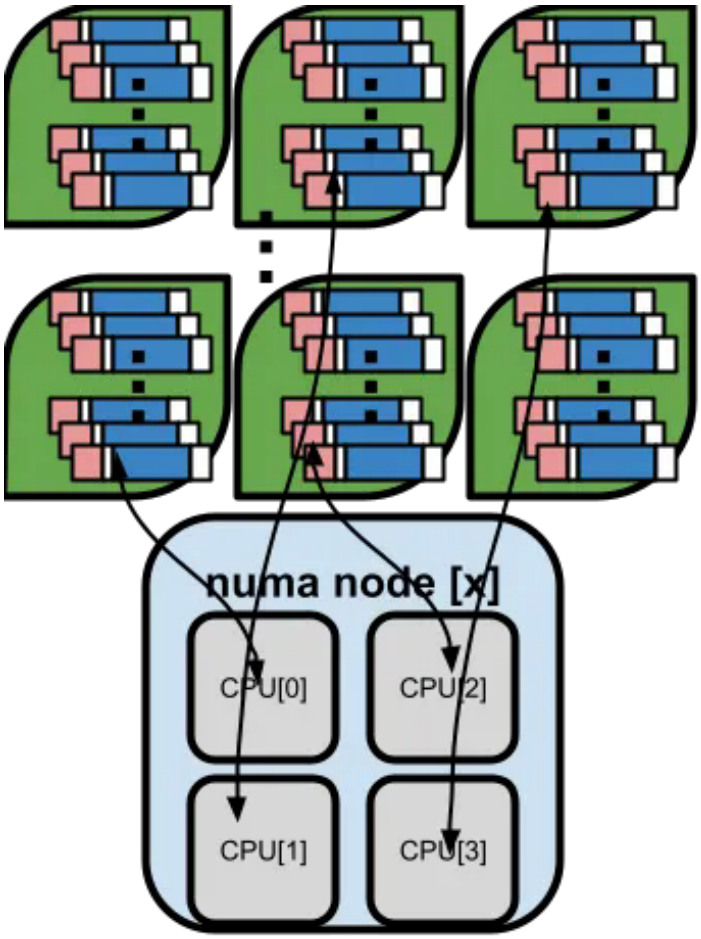

Figure 4. mbuf pools

2.3. *Pool Mode Driver*

Assign packet reception and Poll Mode Driver (PMD) processing to each NUMA node (or CPU-core group) to ensure locality of reference and minimize cross-node memory access latency.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 October 2025 doi:10.20944/preprints202510.1658.v1

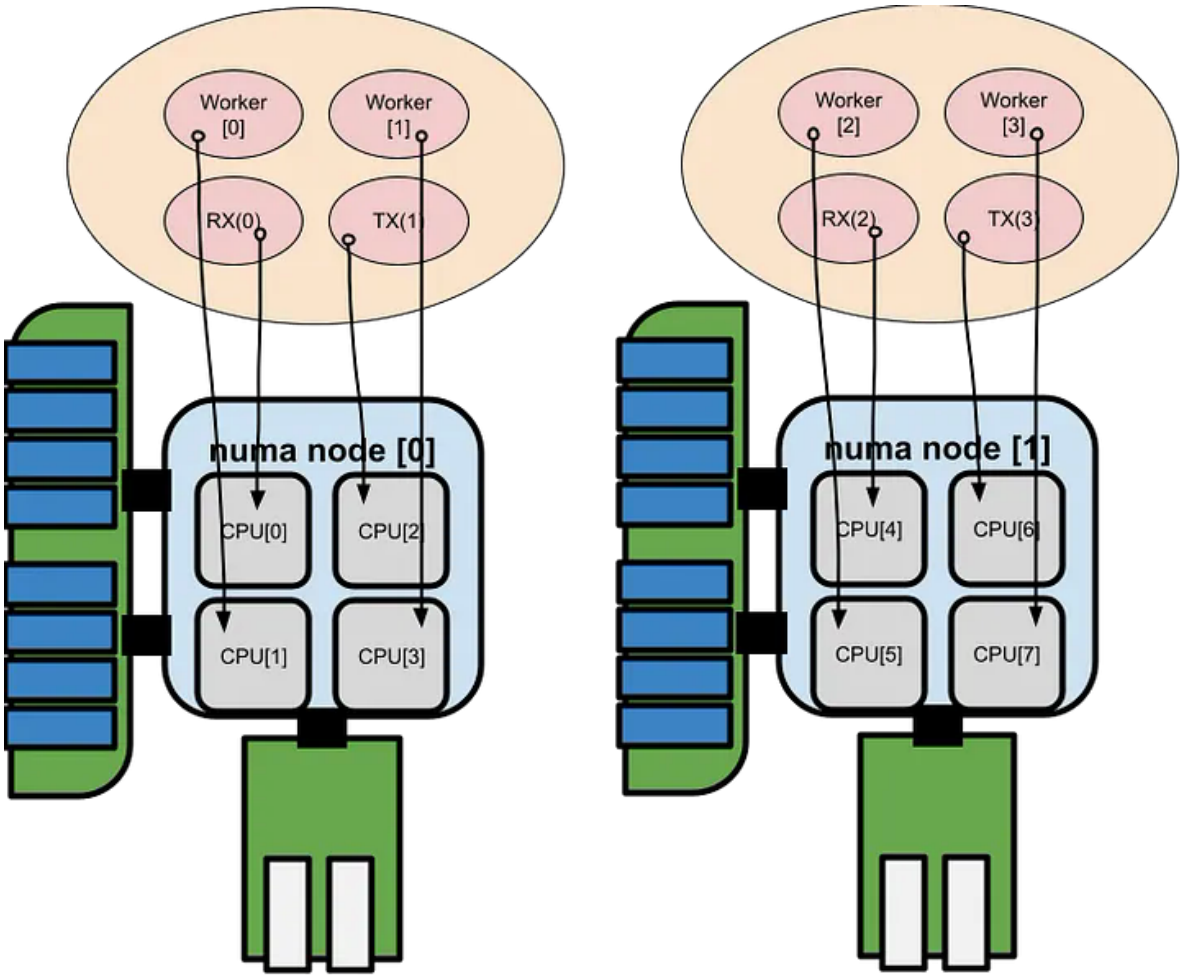

Figure 5. Pool Mode Driver

2.4. Mixed Latency Path

For scenarios that involve both low-latency and high-latency processing paths, a software ring is used to interconnect CPU cores, as illustrated below.

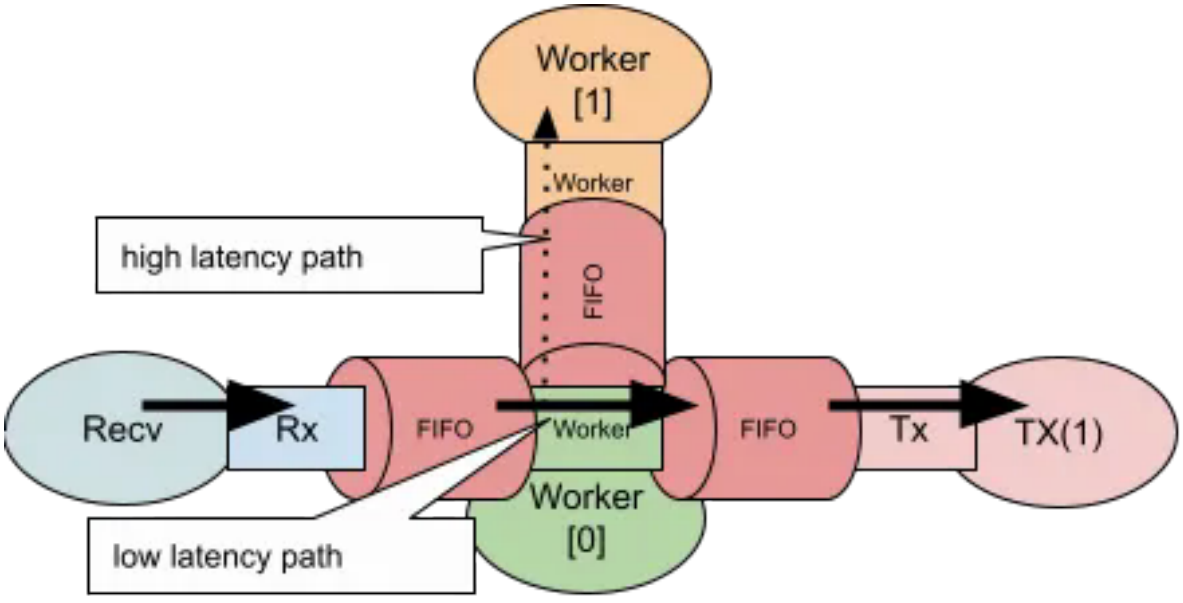

Figure 6. Mixed Latency Path

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 October 2025 doi:10.20944/preprints202510.1658.v1

2.5. mbuf Structure

A variable header start position offset enables encapsulation and decapsulation processing to be implemented with minimal memory copy overhead.

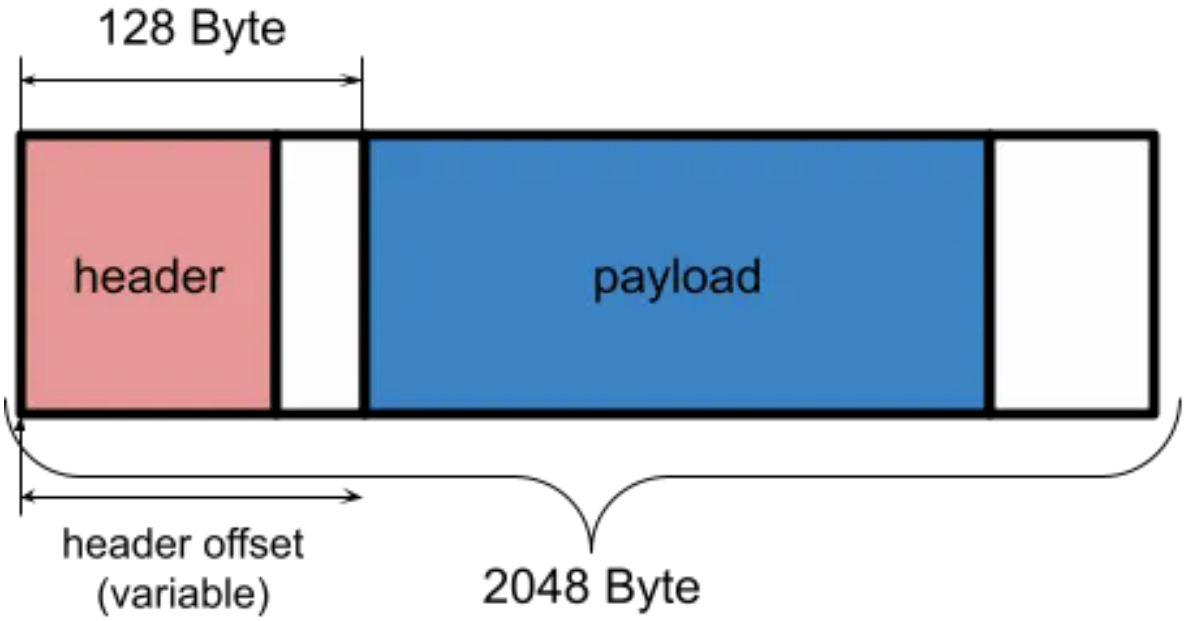

Figure 7. mbuf Strcture

We applied ManyCore - pipeline design to PGW encap/decap processing as follows. Figure 1

The mixi-PGW source code is released under the MIT License. Users should be aware of and comply with the respective licenses of any linked dependency libraries. It is our hope that this article serves as a practical reference for implementing custom user logic with DPDK, and provides insight into the architectural design and optimization of high-performance packet processing systems.

2.6. PGW-Dataplane

Figure 8: Distributor Ingress/Egress Architecture — Each circled number represents a logical CPU core. The Distributor dynamically assigns ingress/egress workers to handle 0.5–1.0 million user sessions, maintaining NUMA locality and cache affinity through lockless rings between threads.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 October 2025 doi:10.20944/preprints202510.1658.v1

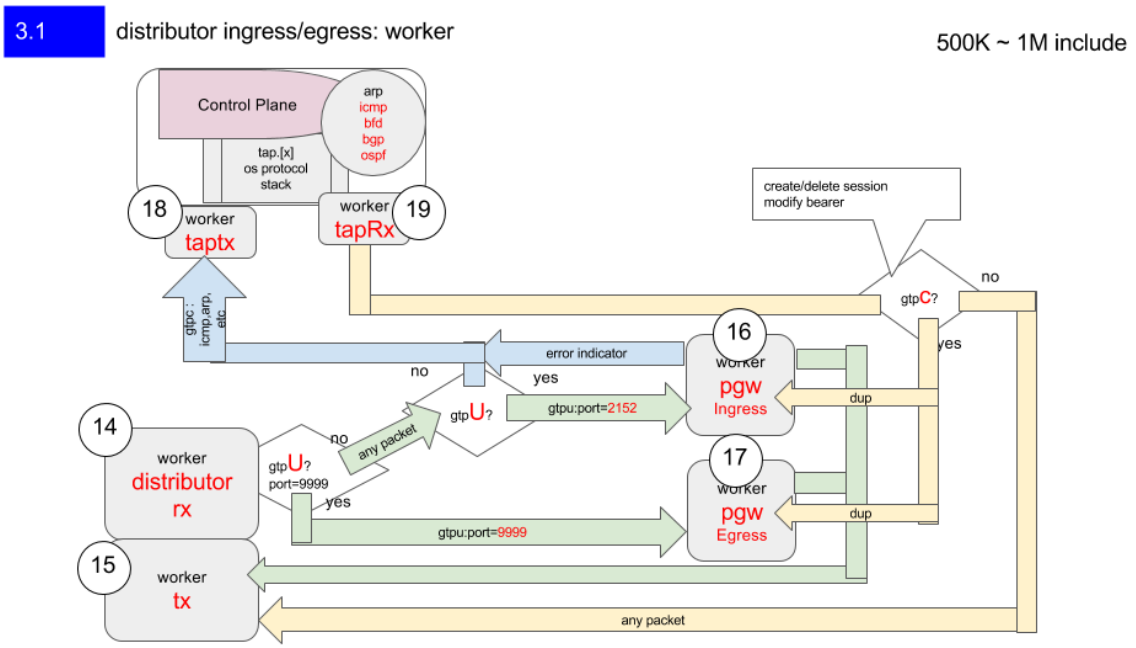

Figure 8. Distributor Ingress/Egress

2.7. User Fairness

Figure 9 illustrates the User Fairness Control mechanism implemented in the PGW dataplane. Each user session is measured by a two-bucket policer consisting of a Committed Burst Size (CBS) and an Excess Burst Size (EBS). The policer follows the Single Rate Three Color Marker (SRTCM) algorithm defined in RFC 2697 [1], classifying packets into green, yellow, and red states based on whether they conform to or exceed the committed rate.

Packets marked as exceeding the Committed Information Rate (CIR) are re-marked by adjusting the DSCP field in the IP header according to the Differentiated Services framework defined in RFC 2474 and RFC 2475. This allows higher-layer schedulers or routers to enforce per-hop behaviors (PHBs) such as AF (RFC 2597) or EF (RFC 3246), ensuring fair bandwidth allocation across 0.5–1.0 million concurrent user sessions.

By implementing this fairness mechanism entirely in user space through DPDK, the system maintains deterministic per-session control without relying on kernel-level QoS subsystems, preserving NUMA locality and cache efficiency within the manycore architecture.

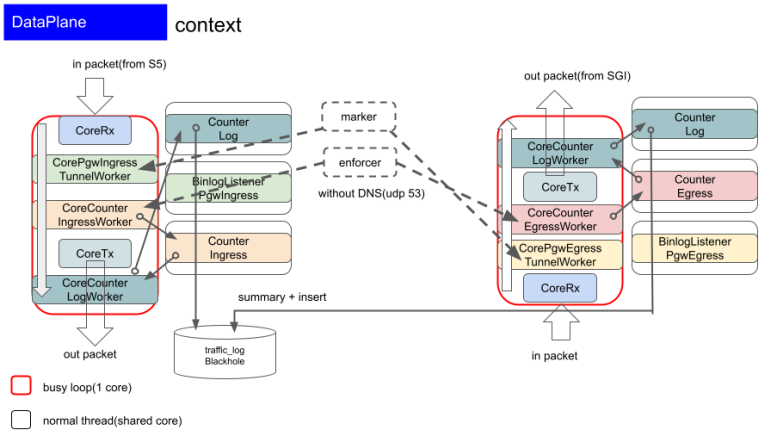

Figure 9. User Fairness Control

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 October 2025 doi:10.20944/preprints202510.1658.v1

3. Accelerated Network Application

3.1. Legacy Socket

Typical BSD-Socket packet sequence in network application can be visualized as follows, with time on the Y axis and packet data on the X axis.

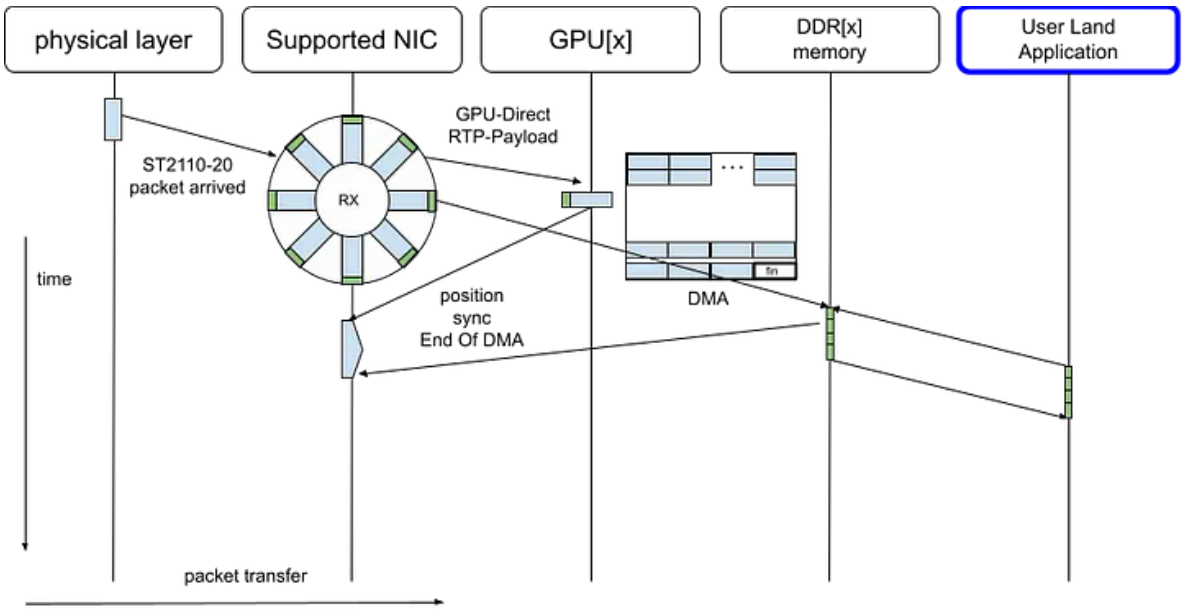

Figure 10. Legacy Socket

Even when abstracted through interfaces such as socket APIs, IOCP mechanisms, or various high-level frameworks, the underlying I/O mechanisms remain fundamentally identical. These abstractions differ primarily in their interface design, level of indirection, and implementation efficiency, but they ultimately rely on the same kernel-level primitives for asynchronous or event-driven communication.

For instance, the following libraries and functions represent conceptually equivalent approaches to I/O handling, despite variations in abstraction depth and runtime environment.

Table 2. Representative abstractions of I/O mechanisms across different environments

Name	Description
libevent	Event-driven I/O abstraction based on callback-oriented socket operations.
socket(2) + select(2)	Legacy synchronous socket API using `select(2)` for multiplexing.
fread(3)	Buffered binary stream I/O abstraction layered on top of system calls.
FUdpSocketReceiver	UDP socket wrapper within Unreal Engine 4's networking subsystem.
IOCP	Windows-specific asynchronous I/O mechanism based on I/O Completion Ports.
Any runtime socket wrapper	Language-level abstractions of socket primitives (e.g., Python, Go, Rust).
Netty	High-performance asynchronous network framework for the Java runtime.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 October 2025 doi:10.20944/preprints202510.1658.v1

In essence, these systems are different manifestations of a common architectural principle: the delegation of I/O events to an event loop or completion mechanism that bridges user-space abstractions and kernel-level event notification.

3.2. DPDK PMD

As illustrated in Figure 11, traditional socket-based applications suffer from increased memory access latency due to cache coherency protocols operating across multiple cores. These coherency mechanisms, while essential for maintaining consistency, introduce additional synchronization and invalidation traffic within the shared memory hierarchy.

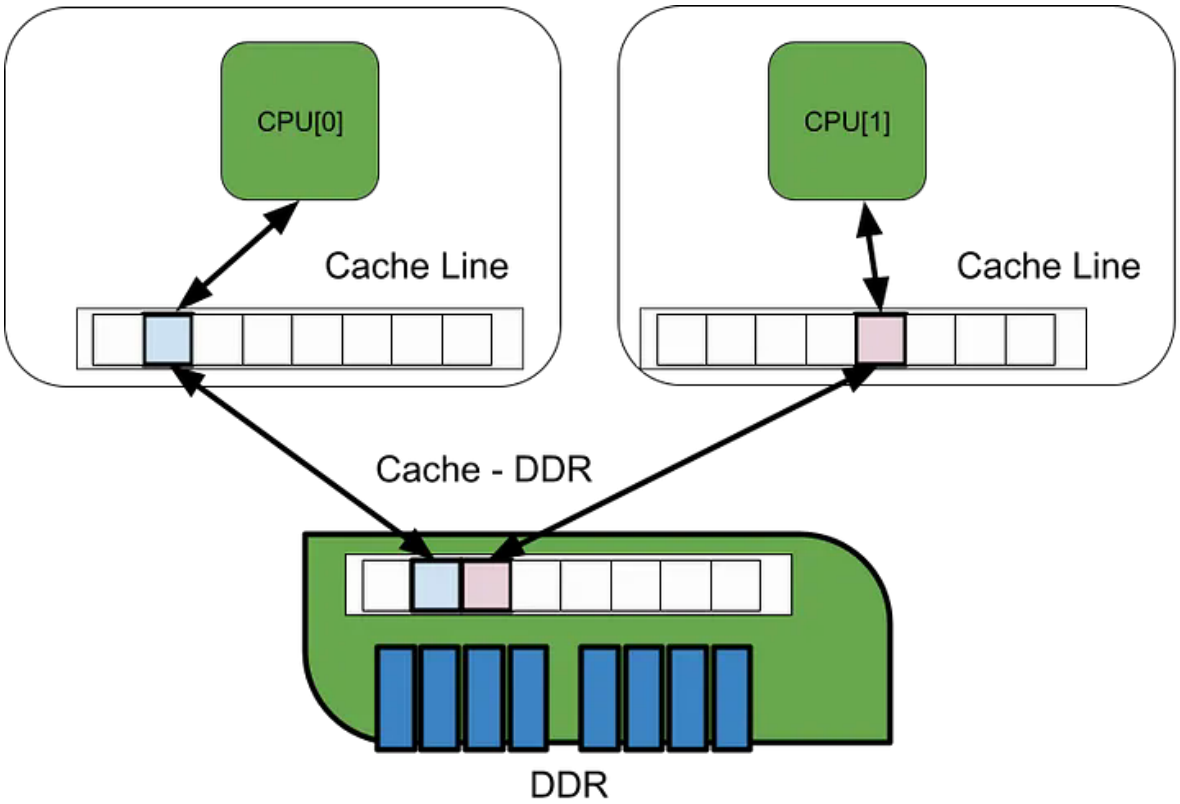

Figure 11. Coherency Architecture

In contrast, the use of **Poll Mode Drivers (PMDs)** and **Hugepages** mitigates such overhead by improving both spatial and temporal locality of reference. PMDs eliminate interrupt-driven I/O overhead by continuously polling network interfaces, thereby reducing context-switching latency. Hugepages, on the other hand, enlarge memory page sizes to minimize Translation Lookaside Buffer (TLB) misses and enhance cache line utilization.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 October 2025 doi:10.20944/preprints202510.1658.v1

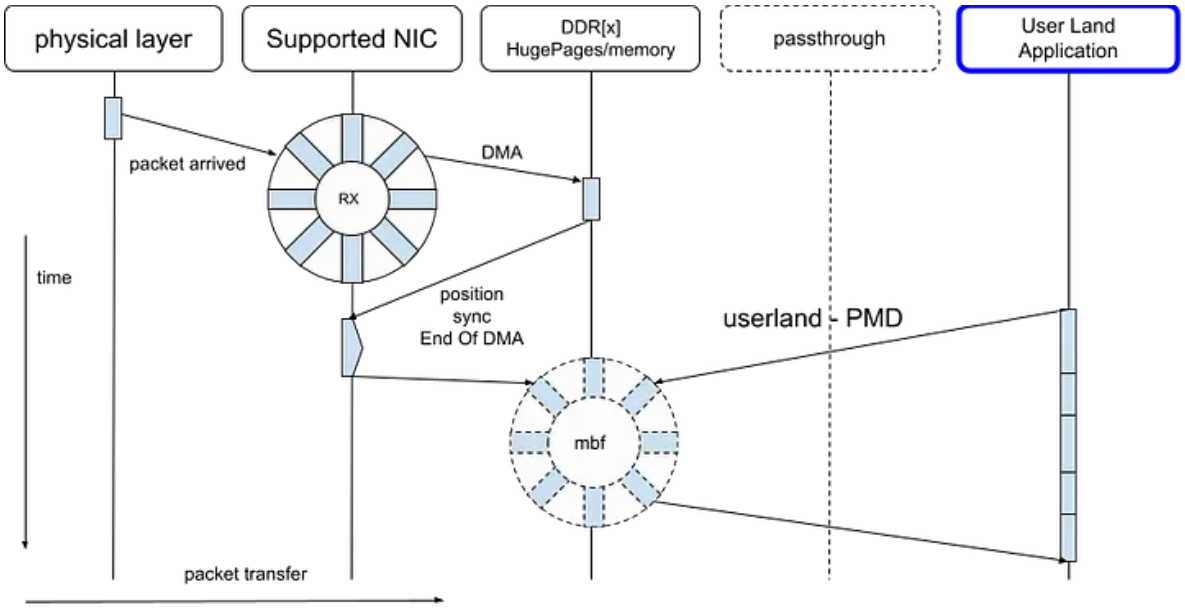

Figure 12. DPDK(PMD) strategy

Consequently, the improved cache locality not only benefits the PMD-based packet processing pipeline but also reduces the adverse effects of coherency protocols, leading to higher throughput and lower latency in data plane operations.

3.3. ConnectX GPU Direct

With the introduction of **ConnectX GPU Direct** and the **Rivermax SDK**, uncompressed ST 2110–20 video streams can be transferred directly from the network interface to the GPU via *Direct Memory Access* (DMA), completely bypassing the CPU memory subsystem [8].

In this configuration, the CPU is only responsible for lightweight control-plane operations, such as parsing the RTP headers to determine frame boundaries and initiating subsequent GPU-based processing tasks. These tasks include video composition, color space conversion, and format transformation.

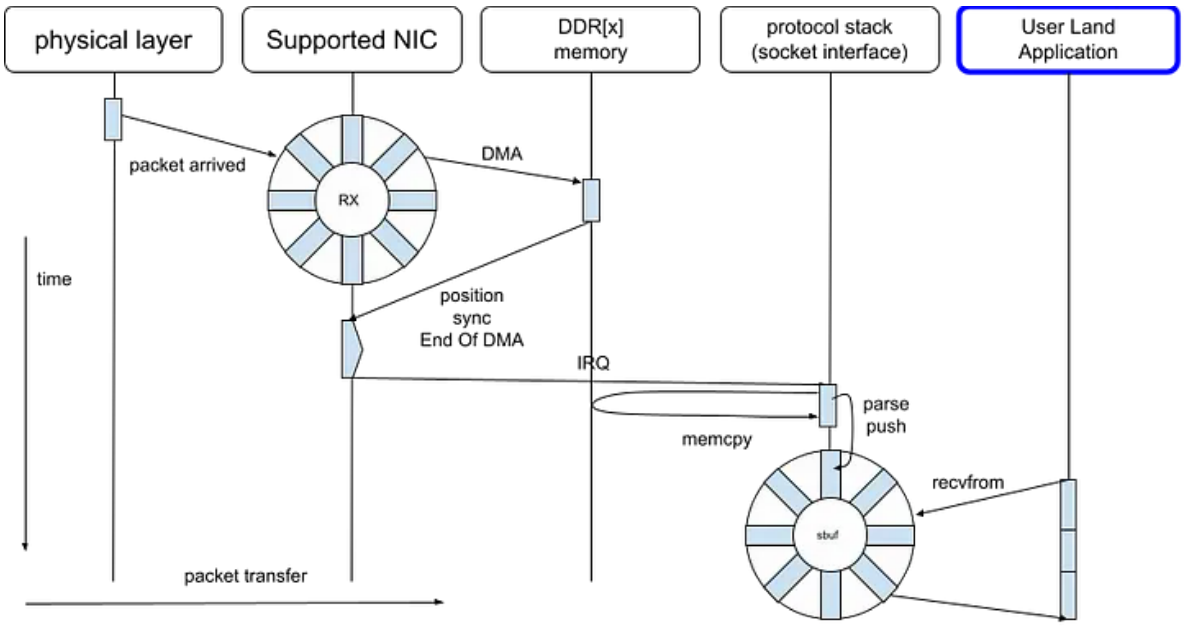

Figure 13. ConnectX GPU Direct

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 October 2025 doi:10.20944/preprints202510.1658.v1

By eliminating intermediate `memcpy` operations and avoiding CPU–GPU context switches, this approach minimizes latency and maximizes throughput, enabling real-time video pipeline execution with deterministic performance characteristics.

4. Related Work

4.1. netmap

Netmap [3] is a high-performance packet IO framework that extends the traditional kernel network stack by providing a shared memory interface between user space and the NIC driver. It enables applications to exchange packets through preallocated ring buffers with minimal system call overhead. However, its operation fundamentally depends on kernel-level support: the host system must include a netmap-enabled network driver. This dependency limits its portability and restricts fine-grained control over the IO path, particularly in environments where the kernel is not easily modified or where kernel-bypass is preferred.

In our early prototype, we initially employed a PMD (Poll Mode Driver) design built upon netmap, leveraging its low-latency buffer exchange to achieve zero-copy packet transfer. While this approach simplified kernel interaction, it constrained the implementation when scaling beyond a few cores, as the netmap kernel module imposed synchronization and buffer management overheads not easily optimized from user space.

In contrast, our current design adopts DPDK's fully user-space PMD architecture, which detaches the IO datapath from the kernel entirely. DPDK allows explicit mapping of RXTX queues to dedicated logical cores, NUMA-local memory pools, and lockless rings between worker threads. This model enables deterministic control over cache locality, prefetch timing, and flow-to-core affinity. These capabilities are essential for sustaining wire-speed throughput in a ManyCore environment where the processing budget per packet is limited to a few hundred CPU cycles.

Therefore, while netmap provides a pragmatic kernel-extended path suitable for prototyping or controlled environments, our DPDK-based design was selected to realize a fully kernel-independent data plane optimized for ManyCore scalability, NUMA locality, and fine-grained scheduling.

4.2. fd.io/VPP (Vector Packet Processing)

The PGW pipeline presented in this study adopts a design in which memory copies across the L2 cache boundary are explicitly controlled by an application-level FIFO. This enables deterministic management of inter-thread data movement and cache locality. In contrast, VPP employs a highly abstracted graph-node model [5], where the data paths and cache behaviors between nodes are encapsulated within the framework. As a result, it provides limited visibility and control over low-level optimizations.

Therefore, while FD.io/VPP offers a versatile and high-throughput packet-processing infrastructure, its abstraction layer makes it less compatible with multi-core pipeline designs—such as the PGW presented here—that rely on explicit data transfers and cache-level optimization at the L2 granularity.

4.3. NetVM

NetVM [6] proposed a high-performance and flexible framework that bridges user-space packet processing with virtualization by leveraging huge-page sharing between the hypervisor and guest VMs. It enables *true zero-copy* packet transfer and in-hypervisor switching, allowing flows to be dynamically steered between network functions with minimal overhead. Specifically, NetVM achieves this through

- (i) DMA from the NIC into shared huge pages
- (ii) lightweight descriptor rings between the hypervisor and each VM
- (iii) shared page references across trusted VMs

This design yields throughput improvements of up to 2.5× over SR-IOV, sustaining near line-rate performance for 64B packets on 10 GbE links (see Figures 2–5 and 10–12 in the original NetVM paper). The framework also introduces NUMA-aware thread assignment and adaptive VM load balancing.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 October 2025 doi:10.20944/preprints202510.1658.v1

While NetVM focuses primarily on *service chaining across virtual machines*, this work targets a fundamentally different design point: *intra-application manycore and NUMA optimization* for DPDK-based data-plane processing. Using a PGW (GTP-U encapsulation/decapsulation) pipeline as a case study, our design explicitly quantifies per-packet processing budgets (67.2 ns at 10 Gbps, 14.88 Mpps) and decomposes latency into cache, memory, and synchronization costs (Table 2). We then propose strategies that *avoid virtualization-induced overheads entirely*—such as NUMA-local mbuf pool allocation, fixed RX/TX queue–core mapping, explicit L2 boundary control, and hybrid low- and high-latency worker pipelines (Figures 4, 5, 6, 7).

Key Differences

- **System boundaries:** NetVM optimizes inter-VM communication through shared huge pages and hypervisor switching. Our work instead optimizes a single-process, multi-threaded data path, eliminating context-switch and IOTLB overheads entirely.
- **NUMA granularity:** Although NetVM considers NUMA locality, our design formalizes it at the mbuf pool and core level, detailing explicit placement and synchronization strategies for deterministic latency.
- **Pipeline composition:** NetVM supports flexible VM service chaining; our pipeline focuses on mixed-latency flow handling, offset-variable header processing, and copy-minimized encapsulation paths.
- **Target domain:** NetVM is suitable for NFV and multi-tenant environments, while our system addresses deterministic, carrier-grade data-plane workloads (e.g., PGW) where predictable per-packet latency is paramount.

Why This Work Still Matters Since NetVM, user-space I/O frameworks such as DPDK have become widespread. However, in environments operating at 100 GbE or with multi-level tunneling, the dominant bottlenecks have shifted: from inter-VM data transfer to cache hierarchy latency, NUMA cross-access, and synchronization contention within a single process. Our work complements NetVM by providing *quantitative insights (ns-level cost models)* and *implementation patterns* for such modern data-plane bottlenecks, offering a platform-agnostic foundation for deterministic performance on manycore systems.

4.4. Barrelfish Operating System

Barrelfish [7], developed by ETH Zürich and Microsoft Research, proposed a *multikernel* operating system architecture designed to scale across manycore processors. Its fundamental principle was to eliminate shared kernel state and instead let each CPU core run an independent kernel instance, communicating explicitly via message passing. This design sought to treat the hardware as a distributed system, rather than as a uniformly shared-memory machine, thereby improving scalability and predictability as core counts increased.

Our proposed PGW system can be regarded as a user-space realization and evolution of this concept. Specifically, the PGW architecture expresses the Barrelfish multikernel philosophy at the application level, forming a **distributed multi-user-kernel architecture** within the cloud environment. Each processing element (DPDK worker, NUMA-local pipeline, or virtual node) functions as an autonomous execution domain, coordinating through explicit, protocol-defined communication rather than shared kernel state. In this sense, our implementation demonstrates that the multikernel design principles of Barrelfish can be achieved and verified in user space, without requiring OS-level modification.

Furthermore, whereas Barrelfish validated its model within experimental kernel infrastructure, our PGW system adopts the same distributed coordination and cache-local design philosophy to practical packet data planes running atop existing Linux environments. Thus, it provides empirical evidence that Barrelfish's scalability-oriented architecture can be extended beyond kernel research into operational, cloud-native network functions.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 21 October 2025 doi:10.20944/preprints202510.1658.v1

5. Conclusion

This technical note presented a deterministic packet-processing architecture for manycore environments using the Data Plane Development Kit (DPDK). By quantifying per-packet processing budgets in nanosecond-level terms and analyzing cache, NUMA, and synchronization costs, we demonstrated design strategies capable of sustaining line-rate throughput under strict latency constraints.

Through the construction of a simplified Packet Gateway (PGW) pipeline, the study highlighted practical methods for minimizing cache misses, ensuring NUMA-local memory allocation, and maintaining per-session fairness entirely within user space. These techniques collectively enable scalable and predictable data-plane operation across hundreds of logical cores.

Compared with kernel-extended frameworks such as netmap or virtualized designs like NetVM, our user-space DPDK approach achieves superior determinism by eliminating kernel transitions and virtualization overhead. Furthermore, the architectural principles discussed—NUMA locality, lockless scheduling, and explicit L2-boundary control—extend beyond packet processing to GPU-direct and real-time multimedia pipelines.

Future work will focus on integrating adaptive load-balancing and telemetry feedback mechanisms to dynamically optimize core allocation under varying traffic patterns, advancing toward a fully autonomous, self-tuning manycore data-plane framework.

References

1. J. Heinanen and R. Guerin, "A Single Rate Three Color Marker (SRTCM)," RFC 2697, IETF Network Working Group, September 1999. Available at: https://datatracker.ietf.org/doc/html/rfc2697
2. Intel Corporation. DPDK: Data Plane Development Kit Programmers Guide. Intel Corporation, latest edition. https://www.dpdk.org/
3. Rizzo, L., Netmap: A Novel Framework for Fast Packet I/O. In USENIX Annual Technical Conference (USENIX ATC 2012), 2012. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo
4. Jesper Dangaard Brouer, "Network Stack Challenges at Increasing Speeds: The 100 Gbit/s Challenge," *Linux-Con North America*, August 2015. Available at: http://events17.linuxfoundation.org/sites/events/files/slides/net_stack_challenges_100G_1.pdf
5. Cisco Systems, "FD.io / VPP (Vector Packet Processing)," 2016. Available: https://fd.io/
6. J. Hwang, K. K. Ramakrishnan, and T. Wood, "NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms," in *Proc. USENIX NSDI*, 2014. https://www.usenix.org/system/files/conference/nsdi14/nsdi14-paper-hwang.pdf
7. "Barrelfish: Exploring a Multicore OS," Microsoft Research Blog, July 7, 2011. URL: https://www.microsoft.com/en-us/research/blog/barrelfish-exploring-multicore-os/
8. NVIDIA Corporation, NVIDIA Rivermax SDK Documentation, Available: https://developer.nvidia.com/networking/rivermax

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.