Adaptive Interrupt Handling for High-Performance Network I/O in Linux and Virtualized Environments

by

Mohammadamin Shafiei

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Mohammadamin Shafiei 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Modern high-performance computing environments face increasing demands from data-intensive applications, making efficient network I/O a critical performance factor. The Linux network stack, traditionally reliant on interrupt-driven models, employs mechanisms, such as NAPI and SoftIRQ, to balance responsiveness and throughput. However, frequent context switches, cache pollution, and lack of alignment between application execution and interrupt handling can degrade performance, especially under heavy network loads. Virtualized environments further complicate this landscape, as software-emulated devices and layered interrupt delivery introduce additional overhead. Recent advances, such as busy polling and hardware interrupt coalescing, offer improvements but often trade off CPU efficiency for lower latency and higher throughput.

This thesis investigates the design and evaluation of a hybrid interrupt handling mechanism, `IRQ Suspend`, within the Linux network stack. The approach dynamically switches between interrupt-driven and busy polling modes based on application activity and traffic load, aiming to reduce context switches and better synchronize packet processing with application execution. Through comprehensive experiments on both bare-metal and virtualized systems, including scenarios with VirtIO-net, SR-IOV, and PCI-passthrough, the study demonstrates that `IRQ Suspend` consistently improves throughput and tail latency compared to traditional interrupt-driven and static busy polling configurations. The results show that `IRQ Suspend` achieves performance close to that of pure busy polling under high load, while significantly reducing CPU utilization during idle periods. Additionally, examining the cache behavior of the `IRQ Suspend` approach reveals more efficient cache usage compared to interrupt-driven methods, with particularly noticeable improvements in L1 instruction cache utilization. These findings highlight the importance of adaptive, application-aware interrupt management for modern network stacks, offering a practical path to high efficiency and predictable performance in both physical and virtualized environments.

Acknowledgements

I would like to thank Professor Martin Karsten, my supervisor, whose support and willingness to guide me made this thesis possible.

I would like to express my appreciation to all my labmates throughout the program, Gan Wang, Peter Cai, Armin Afsharian, Alireza Salamat, Ali Abbasi, and Qishen Wu. Thank you for your support, friendship, and the memorable experiences we shared.

Dedication

This thesis is dedicated to my family. To my brothers, who introduced me to the world of Computer Science and set me on this path. And to my parents, whose unwavering emotional support carried me through the most challenging moments of my studies. Their guidance, love, and sacrifices have made this achievement possible.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Dedication v

List of Figures ix

List of Tables x

List of Abbreviations xi

1 Introduction 1

1.1 Contributions 2

2 Background and Related Work 3

2.1 Busy Polling vs. Interrupt-Driven I/O 3

2.2 Interrupt-Driven Handling in the Network Stack 4

2.2.1 Interrupt-Driven approach in Linux Kernel 4

2.2.2 Adaptive Polling in Linux: NAPI 6

2.2.3 SoftIRQ Coalescing (Timer-Based NAPI) in Linux 7

2.2.4 Hardware Interrupt Coalescing 8

2.3 Polling in the Network Stack 8

2.3.1 Busy Polling in Linux 9

2.4 Virtualization 10
2.4.1 Traditional Network Virtualization 11
2.4.2 VirtIO-net: Optimizing Network Virtualization 12
2.4.3 Journey of a Packet through VirtIO-net Driver 13
2.4.4 Vhost-net: Optimizing the VirtIO-net Driver 14
2.4.5 Device Passthrough 16
2.4.6 SR-IOV 17

3 Network Stack Alignment **19**
3.1 IRQ Routing 19
3.1.1 Dynamic Routing 20
3.1.2 Static Routing 20
3.1.3 IRQ Routing in Virtualized Environment 21
3.2 Polling 22
3.2.1 Busy Wait 23
3.2.2 IRQ Suspend 24
3.2.3 Polling in Virtualized Environment 26

4 Evaluation **27**
4.1 Experimental Setup 27
4.1.1 Hardware 27
4.1.2 Software 27
4.1.3 Benchmark 28
4.2 Data Collection 29
4.3 Performance Metrics 31
4.4 IRQ Suspend Effectiveness 31
4.4.1 Closed-loop Maximum Throughput 32
4.4.2 Open-loop Latency 33
4.4.3 CPU Utilization 34
4.5 VirtIO-net Tests 36
4.5.1 Closed-loop Maximum Throughput 37
4.5.2 Open-loop Latency 38
4.5.3 CPU Utilization 39
4.6 Full Passthrough and Partial Passthrough Tests 42

5 Performance Breakdown 46
5.1 Cache Hierarchy 46
5.1.1 Journey of a Cache Line 47
5.2 Cache metrics 47
5.3 Performance Assessment 48
5.3.1 Cache Behavior of Test Cases 49
6 Conclusion 52
References 54
APPENDICES 61
A 62
A.1 No Coalescing 62
A.2 Adaptive Coalscing 63
A.3 Busy Wait 64
A.4 IRQ Suspend 65
B 66
B.1 Aligned, CPU and NIC Configuration 66
B.2 4 + 2 Cores, CPU and NIC Configuration 67

List of Figures

2.1 VirtIO-net Architecture, Reprinted From [47] With Permission 14
2.2 Vhost-net Architecture Reprinted From [2] With Permission 15
2.3 PCI-passthrough and SR-IOV Architecture for a NIC 18

3.1 IRQ Suspend (Pseudo-code With Highlighted Modifications) 25

4.1 Latency and CPU Utilization vs Throughput, Memcached 8 Cores (Lower is Better)[1] 35
4.2 Latency and CPU Utilization vs Throughput, Memcached 4 Cores Aligned in VirtIO-net Scenario (Lower is Better)[2] 40
4.3 Latency and CPU Utilization vs Throughput, Memcached 4 + 2 Cores in VirtIO-net Scenario (Lower is Better)[3] 41
4.4 Latency and CPU Utilization vs Throughput, Memcached 8 cores in Full Passthrough Scenario (Lower is Better)[4] 44
4.5 Latency and CPU Utilization vs Throughput, Memcached 8 Cores in Partial Passthrough Scenario (Lower is Better)[5] 45

List of Tables

4.1 Throughput Metrics on Physical Machine, Memcached 8 Cores 32
4.2 Throughput Metrics in VirtIO-net, Memcached 4 Cores Aligned 37
4.3 Throughput Metrics in VirtIO-net, Memcached 4 + 2 Cores [2] 37
4.4 Throughput Metrics in Full Passthrough Scenario, Memcached 8 Cores . . 42
4.5 Throughput Metrics in Partial Passthrough Scenario, Memcached 8 Cores . 42

5.1 Cache and Performance Comparison: No Coalescing 50
5.2 Cache and Performance Comparison: Adaptive Coalescing 50

List of Abbreviations

AMAT Average Memory Access Time 49, 51

CPQ Cycles per Query 37, 48, 49

CPS Cycles per Second 31, 48, 51

DIM Dynamic Interrupt Moderation 8

DMA Direct Memory Access 4, 5, 16–18, 43

DPDK Intel's Data Plane Development Kit 22

HV Hypervisor 10–13, 15–17, 21, 22, 26, 28, 38, 39, 42, 43

IOMMU Input-Output Memory Management Unit 16–18, 43

IPC Instructions per Cycle 31, 32, 37, 38, 46, 48, 49, 51, 52

IPQ Instructions per Query 31

IRQ interrupt request 2, 4, 7, 19–22, 27, 28, 33, 34, 36–39, 53, 66, 67

ISR Interrupt Service Routine 3, 4, 19, 43

L1 Level-1-cache 46–51

L1d Level-1-data-cache 27, 46, 51

L1i Level-1-instruction-cache 27, 46, 51

L2 Level-2-cache 27, 46–49, 51

L3 Level-3-cache 46

LLC Last-Level-Cache 27, 46–49, 51

NAPI New API 1, 2, 4, 6–10, 25

NIC network interface card 1, 3–9, 11–13, 16, 17, 19, 21–24, 26–28, 31–34, 36–39, 42, 43, 51

NUMA Non-Uniform Memory Access 19–21, 27, 28, 36

OS Operating Systems 10, 11

PF Physical Function 17

PMU Performance Monitoring Unit 30, 47, 48

QPS Queries per Second 29, 31, 48, 49

RSS Receive Side Scaling 1, 5, 21

SAR System Activity Reporter 29, 30

SoftIRQ software interrupt request 1, 5–8, 10, 19, 23–25, 32–34, 38

SR-IOV Single Root Input/Output Virtualization 11, 17, 26, 42

TLB Translation Lookaside Buffer 16, 51

VF Virtual Function 17, 18

VFIO Virtual Function I/O 16, 17

VM Virtual Machine 10–13, 16–18, 21, 22, 26, 28, 36, 38, 66

Vring Virtual Ring 12–14

Chapter 1

Introduction

Modern high-performance computing environments support increasingly demanding workloads, such as large-scale key-value stores, web services, and cloud platforms, that depend heavily on fast and reliable network I/O. As network bandwidth and CPU core counts continue to grow, the task of efficiently delivering and processing network packets becomes more challenging. In these systems, the operating system's method of handling incoming network interrupts plays a key role in determining overall performance, influencing both throughput and latency under various traffic conditions.

Traditionally, Linux has employed an interrupt-driven model for network packet delivery. In this approach, the arrival of a packet at the network interface card (NIC) triggers a hardware interrupt, which is serviced by a high-priority handler within the kernel. To avoid excessive overhead from frequent interrupts, especially under high packet rates, Linux defers most packet processing to a lower-priority context known as software interrupt request (SoftIRQ) [34]. Components such as Receive Side Scaling (RSS) [59] and New API (NAPI) [66] further refine this model by distributing interrupt handling across multiple cores and adaptively switching between interrupt-driven and polling-based processing, respectively.

Despite these optimizations, interrupt-driven processing can still introduce significant overhead and unpredictability. Frequent context switches, cache pollution, and the lack of coordination with application execution can degrade both throughput and tail latency, especially under high or bursty traffic conditions. To address these challenges, Linux has introduced busy polling. Busy polling allows application threads to directly poll the NIC's receive queues, bypassing the interrupt path and enabling more predictable, low-latency packet processing. However, this approach can lead to high CPU usage, as polling threads remain active even during idle periods.

The complexity of interrupt handling is further magnified in virtualized environments, where network devices are emulated and interrupt delivery is managed entirely in software. Technologies such as VirtIO-net [49, 65] and Vhost-net [2] have been developed to reduce the overhead of virtualized network I/O, but the fundamental trade-offs between interrupt-driven and polling-based processing remain.

Given these challenges, there is an opportunity for application-aware interrupt management strategies that dynamically adapt to workload characteristics and system state. This thesis focuses on the evaluation of a hybrid interrupt request (IRQ) handling mechanism, known as `IRQ Suspend` [15], within the Linux network stack, which dynamically switches between interrupt-driven and busy polling modes based on application activity and traffic load.

The remainder of this thesis is organized as follows. Chapter 2 reviews the background and related work in Linux network stack interrupt handling, including detailed discussions of interrupt-driven I/O, NAPI, busy polling, and virtualization-specific considerations. Chapter 3 presents the design of the proposed hybrid IRQ handling method and explores various network stack alignment strategies. Chapter 4 details the experimental setup and evaluation methodology, while Chapter 5 analyzes performance results and provides insights into the underlying hardware behavior. Finally, Chapter 6 concludes with a summary of findings and directions for future research.

1.1 Contributions

The core contributions of this thesis include:

- Assessed the effectiveness of the `IRQ Suspend` in virtualized environments.
- Identified underlying factors contributing to performance improvements.
- Built upon the initial proposal of `IRQ suspend` from prior work [10, 11].
- Contributed to the implementation of the mechanism [75].
- Performed tests on early versions to validate both functionality and performance.

Chapter 2

Background and Related Work

2.1 Busy Polling vs. Interrupt-Driven I/O

In device I/O operations, the CPU requires a mechanism to determine when a peripheral device, such as a disk or a NIC, has data available or has completed an operation. Two principal methods are employed for this interaction: busy polling (synchronous) and interrupt-driven (asynchronous) signaling. In busy polling, the CPU continuously inspects the device's status registers in a tight loop to check if servicing is needed. This approach ensures immediate awareness of device readiness by actively polling the hardware. Alternatively, interrupt-driven I/O allows the device to notify the CPU when it requires attention by raising an interrupt. Upon receiving such a signal, the CPU temporarily halts its current task and invokes the appropriate Interrupt Service Routine (ISR) within the operating system to handle the event.

From a performance standpoint, busy polling avoids the overhead of context switching and keeps execution within the same thread, which may reduce latency in some scenarios. However, it leads to inefficient CPU usage during idle device periods, as the processor cannot sleep or perform other useful work. It also prevents entry into low-power states, resulting in higher energy consumption.

In contrast, interrupt-driven I/O provides more efficient CPU utilization by allowing the processor to execute other tasks or enter power-saving modes, responding to devices only when needed. Despite this efficiency, it introduces performance overheads due to the context switch required to service each interrupt. Under high interrupt load, this can escalate into `interrupt livelock` [48], where excessive time spent in interrupt handling starves user-level processes. Additionally, interrupts introduce inherent latency between the device's signal and the CPU's handling, caused by factors such as scheduling delays, interrupt masking, or preemption by higher-priority events.

In modern systems the network stack is one of the most demanding device I/O subsystems, as NIC link speeds have grown from gigabit to tens or even hundreds of gigabits per

second. Every incoming packet must traverse multiple layers of the network stack, from the NIC's driver up through protocol layers such as TCP or UDP. Inefficient I/O handling at any stage can lead to packet drops, CPU saturation, and degraded application performance. Relying solely on interrupts at these speeds can generate millions of events per second, overwhelming both the CPU and the interrupt subsystem. By contrast, continuous polling can waste vast numbers of CPU cycles when the load is below a certain point. This trade-off has driven the adoption of hybrid solutions, most notably NAPI, and further optimizations, such as multiqueue processing, IRQ affinity, and deferred polling [16] to sustain increasing traffic volumes without sacrificing responsiveness or unduly taxing CPU resources.

2.2 Interrupt-Driven Handling in the Network Stack

In traditional UNIX-like [71] operating system kernels (such as Linux), the network stack relied exclusively on hardware interrupts to deliver incoming packets to the protocol layers. When a packet arrives, the NIC raises an interrupt that invokes its ISR. The ISR then disables further NIC interrupts, fetches the packet from the Direct Memory Access (DMA) ring buffer, and hands it off directly to the network stack. After traversing the IP, transport, and socket layers, any user-space process waiting for that packet is awakened. Because this delivery occurs independently of the application's context, it is referred to as asynchronous execution.

Early applications typically used blocking I/O calls (e.g., `read`, `write`) under a one-thread-per-connection model, which led to substantial scheduling overhead as connection counts increased. To mitigate this, operating systems introduced I/O-multiplexing APIs, first `select` and `poll` [5, 22], and later `epoll` [46] and `kqueue` [38], enabling a single thread to efficiently monitor and service many simultaneous connections.

Under light traffic, this interrupt-driven approach incurs minimal overhead and delivers packets to the stack almost instantaneously. However, as packet rates increase, its efficiency collapses: the ISR often runs for extended periods with interrupts disabled, incurring costly context switches, and can even suppress lower-priority interrupts entirely. This leads to erratic latency spikes and risks dropped events when the CPU remains tied up in the ISR. To address these shortcomings, the modern network stacks incorporate adaptive mechanisms, such as IRQ deferral. This thesis examines these Linux-specific interrupt-handling strategies in depth.

2.2.1 Interrupt-Driven approach in Linux Kernel

Linux systems adopt an interrupt-driven I/O model for handling high-speed NICs, but with key modifications to manage the high frequency of events these devices can generate. When a packet arrives at a NIC (`RX` path), the standard processing sequence begins with

the device transferring the packet into system memory using DMA, placing it into a pre-allocated ring buffer, known as an `RX` queue. Once the transfer is complete, the NIC raises a hardware interrupt to notify the CPU that a new packet is available.

The interrupt is initially serviced by the NIC device driver through a short interrupt handler, commonly referred to as the `top half` in Linux terminology [77]. This handler runs at high priority, often with interrupts on the same line temporarily disabled to avoid nested handling. Its role is intentionally limited in scope to ensure minimal execution time. Typically, the handler acknowledges the interrupt and flags the packet for further processing. To prevent interrupt flooding, the handler also disables further interrupts from the same device and defers the main processing to a lower-priority execution context. This deferral strategy is essential because complete packet processing, such as passing it through the network stack, can be computationally expensive. Performing such work directly within the interrupt handler would extend the duration during which the CPU is unavailable for other tasks and may delay the handling of subsequent interrupts. By offloading this work to a deferred context, Linux ensures more balanced CPU utilization and reduced latency across system operations.

Linux handles deferred interrupt processing using a mechanism known as SoftIRQ [34, 78]. A SoftIRQ serves as the `bottom half` handler, a routine that is scheduled to execute outside of the immediate hardware interrupt context, allowing the system to complete tasks initiated by an interrupt without holding up the CPU. The Linux networking subsystem relies extensively on SoftIRQ. Specifically, it defines dedicated SoftIRQ types such as `NET_RX_SOFTIRQ` for packet reception and `NET_TX_SOFTIRQ` for transmission (`TX` path).

The execution of a SoftIRQ generally occurs shortly after it is raised, either as the CPU exits the hardware interrupt context or at the next suitable opportunity within the kernel's scheduling cycle. The Linux kernel checks for any pending SoftIRQs after completing hardware interrupt handling, as well as during system call returns or when performing a reschedule. For incoming network traffic, the SoftIRQ handler is the `net_rx_action` function, which is registered during system initialization as the handler for `NET_RX_SOFTIRQ`. The `net_rx_action` function carries out the majority of the work associated with processing received packets, making it a central component of the Linux network stack's receive path.

SoftIRQs in Linux offer significant scalability benefits but also introduce potential limitations. One of their key strengths is the ability to execute concurrently across multiple CPUs, enabling high-throughput workloads, such as handling large volumes of incoming network packets, to be effectively distributed among CPUs. On systems with multi-queue NIC, techniques, such as RSS [59] allow each queue's interrupt to be directed to a separate CPU core. Each core can then independently raise and process its own instance of the network SoftIRQ, which supports parallel packet processing and improves performance on multi-core systems. However, this design also carries the risk of SoftIRQ induced CPU starvation. If packet arrivals are continuous and excessive, the system may spend an inordinate amount of time handling SoftIRQs, delaying other kernel- or user-space tasks.

To address this, Linux introduces heuristics and control mechanisms, such as capping the number of packets or the time spent per SoftIRQ execution (e.g., limiting processing to a 2 ms window). When these thresholds are exceeded, the remaining work is either rescheduled or offloaded to a dedicated per-CPU kernel thread known as `ksoftirqd`, ensuring a balance between responsiveness and fairness in CPU time allocation.

2.2.2 Adaptive Polling in Linux: NAPI

While SoftIRQ improves efficiency by deferring and batching work, very high rates of incoming network packets can still strain system resources by triggering many interrupts. To mitigate this, the Linux networking subsystem employs NAPI [74, 66], a mechanism designed to reduce interrupt overhead through adaptive polling. NAPI provides a hybrid approach that blends interrupt-driven and polling-based processing. When a packet arrives on an idle network interface, the NIC initially generates an interrupt, triggering the driver's interrupt handler. Rather than processing just a single packet, the handler disables further interrupts for that NIC and schedules a NAPI poll routine, which is executed in SoftIRQ context via `net_rx_action`. This routine enters polling mode temporarily, fetching and processing packets directly from the NIC's `RX` queues in batches. The poll function continues either until it reaches a configured packet limit (the budget) or the `RX` queue is empty. By processing multiple packets per interrupt, NAPI significantly reduces the frequency of interrupts and spreads their overhead across many packets, improving overall system efficiency in high-traffic scenarios.

During the NAPI polling cycle, if the poll function processes fewer packets than the defined budget, this suggests that the system has caught up with the incoming traffic and the NIC's receive buffer is empty. In such cases, the polling routine concludes by invoking `napi_complete_done`, which re-enables hardware interrupts for that NIC. This allows the system to revert to interrupt-driven processing for subsequent packets, maintaining low-latency responsiveness under light traffic conditions. Conversely, if the polling cycle reaches the budget limit, indicating that additional packets may still be pending or arriving faster than they can be processed, the system does not immediately re-enable interrupts. Instead, it keeps interrupts disabled and typically schedules another polling round to handle the remaining load. This adaptive behavior enables the system to remain in polling mode during periods of heavy traffic, effectively reducing interrupt overhead, while still transitioning back to interrupt mode during lighter traffic to optimize for responsiveness.

NAPI offers a balanced and adaptive mechanism for handling network packet reception, combining the strengths of both interrupt-driven and polling-based processing. Under high traffic conditions, NAPI significantly improves throughput and reduces CPU usage by minimizing interrupt overhead and context switching. It allows packet processing to scale across multiple CPUs by associating each `RX` queue with a dedicated NAPI context, which can be scheduled concurrently via SoftIRQ. This design improves cache locality, supports efficient packet drops at the NIC buffer under overload, and helps preserve packet

ordering. Modern multi-queue NICs benefit greatly from this architecture, as each queue and its corresponding interrupt can be mapped to separate CPU cores, enabling parallel and efficient packet handling.

However, in low-load scenarios, the interrupt-driven nature of NAPI causes application threads to relinquish the CPU when no packets are available, transitioning cores into sleep states. This introduces additional latency due to CPU wake-up time, thread scheduling delays, and reduced processing speeds caused by power-saving states.

2.2.3 SoftIRQ Coalescing (Timer-Based NAPI) in Linux

The modern Linux kernel[1] introduces a mechanism called SoftIRQ Coalescing (also known as timer-based NAPI) [74], which further suppresses both hardware and software interrupts when a NAPI polling cycle finishes without processing any packets. This approach aims to lower the overhead caused by repeated SoftIRQ activations and frequent re-enabling of hardware interrupts, especially under bursty traffic conditions.

Section 2.2.2 describes that when a NIC interrupt triggers NAPI polling, the driver disables additional `RX` interrupts, and the NAPI poll function begins processing packets in a loop. Once the polling cycle finishes, the NAPI logic re-enables hardware interrupts so the NIC can signal the arrival of future packets. When SoftIRQ coalescing is enabled, this behavior changes. Rather than immediately re-enabling the NIC interrupt at the end of each poll, the kernel sets a timer to initiate the next NAPI poll after a configured delay. This delayed activation reduces the frequency of interrupts and lowers processing overhead, particularly under bursty or variable traffic. In effect, NAPI operates in a timer-driven polling mode, postponing the re-enabling of interrupts for a short time. If packets arrive during this interval, they do not trigger new interrupts; instead, the next scheduled NAPI poll picks them up.

To implement this mechanism, two flags have been introduced: `gro_flush_timeout` [19] and `napi_defer_hard_irqs` [21]. It is important to note that the `gro_flush_timeout` flag originally existed in the kernel to give the Generic Receive Offload engine its high-resolution timer for flushing partially-aggregated packets. Later, this flag was repurposed to be used as part of the SoftIRQ Coalescing. The `gro_flush_timeout` parameter defines the delay interval in nanoseconds for the timer. This value determines how long the system defers re-enabling the NIC interrupt while waiting for additional packets to arrive. The `napi_defer_hard_irqs` parameter specifies the number of consecutive empty NAPI polls the system allows before giving up and re-enabling interrupts. Together, these settings control how long the system stays in polling mode before returning to interrupt-driven processing. These flags can be configured globally per NIC device using `sysfs`. However, recent Linux kernels[2] offer granular control through per-NAPI configuration, allowing system administrators and developers to fine-tune software IRQ coalescing on a per-instance

[1] Linux version 5.0

[2] Linux version 6.13

basis rather than globally [75]. This granular configurability is especially valuable in systems with multiqueue NIC or heterogeneous traffic, where different coalescing strategies can be applied to optimize the trade-off between CPU utilization and packet-processing latency.

2.2.4 Hardware Interrupt Coalescing

Although the kernel's interrupt mitigation strategies, from pure interrupt-driven delivery through NAPI's adaptive polling to timer-based SoftIRQ coalescing, have steadily reduced CPU overhead, they still incur processing costs on the host. Modern NICs, therefore, offload much of this work into hardware, using both static and adaptive coalescing to group packet arrivals before raising interrupts.

Static hardware interrupt coalescing relies on two primary mechanisms to reduce interrupt frequency. In the timer-based configuration, the NIC defers raising an interrupt for a specified duration (controlled via parameters `rx_usecs` and `tx_usecs` in `ethtool`), allowing multiple packets to be grouped and processed together. Alternatively, in the threshold-based mode, the NIC generates an interrupt once a predetermined number of packets have arrived, using settings such as `rx_frames` and `tx_frames` [29, 62].

These configurations introduce some latency during periods of low network activity, as packets may remain queued in the hardware until the timer expires or the packet threshold is met. However, under heavier traffic conditions, this batching strategy significantly reduces CPU overhead by minimizing the interrupt rate.

Static parameters cannot simultaneously optimize for latency in quiet periods and few interrupts under heavy load. To address this trade-off, modern NICs support Dynamic Interrupt Moderation (DIM). DIM periodically samples per-queue statistics, such as packets received, bytes processed, and interrupts fired, and adjusts the interrupt rate registers on the fly. When sustained high traffic is detected, DIM lengthens the coalescing timer or raises the frame threshold to group more packets and reduce overhead; in quieter intervals, it shortens timers or lowers thresholds to ensure prompt packet delivery [23, 31].

2.3 Polling in the Network Stack

Section 2.2 shows that interrupt-driven I/O operates independently of the application's execution state. In this model, packet delivery is initiated by the hardware and proceeds regardless of whether the receiving application is currently active. This lack of coordination can distort application execution by causing frequent context switches between the application and the interrupt handler.

This issue becomes particularly pronounced in scenarios involving a dominant application. A networking workload that consumes the majority of system traffic and runs on

a designated subset of NIC queues and CPU cores. When interrupts intended for this application are not coordinated with its execution, performance suffers due to application execution distortion. Additionally, packets may arrive for other, less active applications, which further amplifies context switching (between the application and the interrupt handler context) and degrades CPU efficiency, even when those applications are not actively scheduled.

In contrast, a pure polling approach places the responsibility of packet reception on the application, which continuously loops to check the NIC's `RX` queues for incoming packets. Since this polling occurs within the application's thread context, polling is often referred to as a synchronous processing model. A key advantage of busy polling over interrupt-driven methods is that the application determines when polling should begin. However, this control comes at the cost of increased CPU usage, as the application must poll continuously regardless of whether packets are arriving. This thesis focuses on polling mechanisms implemented within the Linux kernel.

2.3.1 Busy Polling in Linux

Linux supports busy polling by allowing the socket layer to directly access the NIC's receive queue, bypassing the traditional interrupt-driven model for packet notification. When busy polling is enabled, the kernel probes the NIC's `RX` queue for new packets within the calling application's thread context, continuing until either a predefined threshold is met or a timeout occurs. This avoids putting the thread to sleep and eliminates reliance on interrupt wake-ups. The mechanism is built on top of the NAPI infrastructure, where the thread actively invokes the NIC driver's NAPI poll function to fetch packets. Busy polling has undergone a notable evolution since its inception. Initially, it required dedicated driver support through the `ndo_busy_poll` callback. With the release of Linux 4.12, busy polling was further extended to work with the `epoll` interface, broadening its applicability to more I/O-driven applications [79, 20].

Busy polling in Linux can be enabled system-wide through the `net.core.busy_poll` parameter, or configured on an `epoll` instance for finer control starting with kernel version 6.13 [18]. When enabled, applications can initiate busy polling during I/O multiplexing calls such as `epoll_wait`. If no packet arrives during the polling window, the application thread transitions to a sleep state and waits for new interrupts. However, if a packet is received during this period, it is processed synchronously in the same execution context as the application.

This model allows applications to actively notify the kernel when they are idle and waiting for incoming packets by invoking `epoll_wait`. It establishes a cooperative interaction between the application and the kernel, aligning packet processing with the application's execution state. Similarly, the NAPI mechanism (refer to Section 2.2.2) performs a form of speculative polling when no packets are initially detected. In such cases, it runs a short optimistic loop to check for new packets before re-enabling interrupts. However, this NAPI

polling loop fundamentally differs from busy polling. While busy polling occurs within the context of the application thread, the NAPI loop runs independently in the SoftIRQ context and has no awareness of the application's workload.

Despite its advantages, busy polling is not without limitations. Interrupts can still interfere with application execution. Although they are temporarily suppressed during the busy polling phase, the kernel re-enables them afterward. As a result, while the application is processing previously received data, it may still experience disruptions to its execution flow. In addition, continuous `RX` polling leads to wasting CPU cycles.

2.4 Virtualization

Virtualization is a foundational technology in modern computing that enables multiple Operating Systems (OS) to run on a single physical machine. A Virtual Machine (VM), or a *guest*, executes an OS image on a physical machine. This physical machine that executes a VM, is termed a Hypervisor (HV) [2]. Virtualization abstracts the underlying hardware and provides each VM with an isolated environment, such as virtualized CPUs, memory, disks, and network interfaces. This approach maximizes hardware utilization and enhances flexibility for deploying, managing, and maintaining software systems. Several technologies and tools work together to enable virtualization. Among the most prominent for Linux are:

- **Kernel-based Virtual Machine (KVM) [35]:** KVM is a virtualization solution that allows the Linux kernel to act as a HV. It enables multiple VMs to run in isolation on a single physical host. Each VM operates within its own isolated environment, complete with virtualized components such as CPU, memory, storage, and network interfaces. KVM also leverages hardware-assisted virtualization technologies, such as Intel VT-x [30] and AMD-V [55], to reduce overhead associated with caching, I/O, and memory operations, allowing virtualized workloads to execute with performance close to that of native hardware.
- **Quick Emulator (QEMU) [7]:** QEMU is an open-source machine emulator capable of replicating various hardware devices and CPU architectures. It can emulate full systems, including CPUs and device peripherals. When used in combination with hardware acceleration (e.g., Intel VT-d) or a virtualization solution like KVM, QEMU can run virtual machines with near-native performance. Virtual machines can be launched using QEMU's command-line interface.
- **Libvirt [72]:** Libvirt is an open-source API designed for managing virtualization platforms. It offers a standardized interface for creating, monitoring, and controlling virtual machines. Libvirt translates XML-based configurations into QEMU command-line instructions. It also abstracts the complexity of managing different

HV technologies, including KVM, Xen [6], and VMware [76], making it easier to manage virtualized environments.

Each virtual machine operates with its own user and kernel space contained within a QEMU process. A separate QEMU process exists for each virtual machine. Meanwhile, KVM operates within the kernel space of the host virtualization system [61].

The discussion in the previous sections is focused on real interrupts generated by physical NICs. However, in virtualized environments, the scenario is different, in that the underlying physical machine emulates interrupts. These virtual interrupts play a crucial role in the performance of virtual NICs. It is important to understand how these emulated interrupts are generated and how they affect the overall packet processing efficiency in a virtualized setup. This section explores the behavior of virtual NICs and examines how they handle packet processing within such environments. This section also discusses alternative virtualization networking techniques used in virtualized environments, including PCI-passthrough and Single Root Input/Output Virtualization (SR-IOV), in which direct access to the physical NIC is achievable.

2.4.1 Traditional Network Virtualization

In traditional network virtualization, the VM is provided with a fully emulated NIC, such as an Intel E1000 or Realtek RTL8139 Ethernet card [9]. When the guest OS boots, it detects a virtual PCI NIC that mimics a real device and loads the appropriate driver (e.g., the e1000 driver [73]). The guest driver operates with this virtual NIC as if it were actual hardware, writing to control registers and configuring descriptor rings for sending and receiving packets. All of these interactions are intercepted and handled by the HV's virtual device emulation layer.

When the guest driver attempts to transmit a packet by writing to a transmit descriptor register, it invokes a `VM_EXIT` hypercall—which saves the guest CPU's state and switches execution into the HV, performing a context switch from the guest to the HV. At this point, QEMU's emulated NIC reads the packet from the guest's memory and forwards it to the host's networking subsystem, typically via a TAP device [37], which then routes the packet through the physical NIC or a virtual switch. For incoming packets, the process is reversed: the host receives the packet (for example, on a TAP device), passes it to the HV's emulated NIC, which copies the data into the guest's receive buffer and raises a virtual interrupt to inform the guest OS of the arrival. In this setup, QEMU handles all aspects of device operation, including register emulation and networking logic, serving as the intermediary between the guest and the host network.

This setup offers both benefits and drawbacks. Its main advantage lies in broad compatibility, any HV that supports the emulated device driver can use it without requiring changes to the guest OS. However, because the HV manages all packet transmission and reception while the guest remains unaware of the underlying virtualization, each packet

incurs multiple context switches and data transfers between the guest and host. This overhead results in a significant performance penalty.

2.4.2 VirtIO-net: Optimizing Network Virtualization

The VirtIO-net driver is a paravirtualized network driver designed to enhance the performance of network I/O in a virtualized environment. Conventional network virtualization techniques depend on full emulation of the NIC [65, 70], causing high context switching (`VM_EXIT`) overhead. VirtIO-net, on the other hand, is an example of a paravirtualization method by which the guest operating system knows it exists in a virtual environment and makes use of specialized drivers to communicate with the HV.

VirtIO-net offers several notable advantages over traditional virtualization methods. It reduces the redundant context switches caused by `VM_EXIT` calls, which significantly reduces CPU overhead during network operations. Rather than emulating hardware interfaces, VirtIO-net leverages shared memory for data exchange between the guest and host systems, creating a more efficient communication path. Additionally, it provides a versatile solution that adapts well to various virtualized environments, making it an optimal choice for modern virtualization deployments.

The VirtIO-net architecture consists of two main components: the frontend driver in the guest operating system and the backend driver in the HV [2]. These components communicate with each other through shared memory. The frontend driver in VirtIO-net is a network device in the guest operating system. It handles network operations and communicates with the backend driver through Virtqueues. Virtqueue is a shared memory structure between the guest and HV that manages the I/O operations.

Within the HV environment, the backend driver (typically integrated into systems like QEMU for KVM) handles the critical task of transferring data between Virtqueues and the host's network infrastructure. When a guest transmits packets, the backend driver forwards them to a TAP device [37], which functions as a virtual network interface operating on the host system. A TAP device is a software-emulated Ethernet adapter that delivers raw Ethernet frames to user-space programs via a file descriptor, allowing the QEMU to read or inject packets into the physical NIC. Conversely, when packets arrive from external networks, they first reach the TAP device, then pass through the backend driver, which subsequently places them into the Virtqueue's receive queue for guest processing.

The Virtqueue is described as a shared memory structure between the guest and the HV, used to manage I/O operations in a virtualized environment. However, Virtual Ring (Vring) is the ring buffer structure that is responsible for the actual data transportation between HV and VM. The Vring mechanism is the core of VirtIO-net's performance and functionality. A Vring consists of three parts:

- **Descriptor Table**: Holds metadata of data buffers, including their memory location and size.

- **Available Ring**: The guest uses this ring to tell which buffers wait for processing.
- **Used Ring**: The HV updates the used ring to indicate it processed some buffers.

This architecture enables efficient asynchronous communication between the frontend and backend drivers with less synchronization overhead.

One of the key optimization features of Vring are the interrupt suppression flags. These interrupt suppression flags allow batch operation, which results in fewer host-guest notifications. For instance, multiple buffers can be added to the available ring before signaling the host, enabling efficient handling of bursts of I/O requests. Similarly, the HV can suppress interrupts when updating the used ring, reducing interrupt overhead during high-throughput operations. This communication mechanism between HV and the guest reduces the latency compared to the traditional methods. Benchmarks show that VirtIO-net's paravirtualized transmission mechanism achieves 25% lower latency than fully emulated devices [49].

2.4.3 Journey of a Packet through VirtIO-net Driver

This subsection shows how VirtIO-net moves packets between the backend and frontend sides in both directions for `TX` and `RX` paths.

When the frontend driver writes a packet into the `Available Ring` of the Vring, it notifies the backend by writing to a shared memory region, which triggers a `VM_EXIT`[3], pausing the VM and handing control to the KVM. It is important to note that when the interrupt suppression flag is active, the interrupt is postponed until a certain number of packets have been enqueued. The KVM's exit-processing path in the kernel catches the `VM_EXIT`, inspects its cause (a write to a specific memory region), packages up the exit reason, and returns control to the QEMU in userspace. The QEMU identifies the event as originating from the VirtIO-net device and, using the details provided by the KVM, triggers its backend emulation routine. It then interacts with the TAP interface to simulate a physical NIC and updates the Vring descriptor to indicate that the buffer has been used. After completing this processing, the QEMU notifies the KVM, which restores the virtual CPU's saved state and resumes execution within the guest.

Once the guest is running again, the receive side of the VirtIO-net path takes over: within the VirtIO-net backend, the QEMU spawns dedicated I/O threads tasked with monitoring the TAP interface's file descriptor for incoming traffic. These threads wake as soon as the descriptor becomes readable, indicating that packets have arrived on the physical NIC. At that point, the QEMU invokes the registered callback function, which

[3] A `VM_EXIT` occurs when the CPU halts execution within the VM and transitions into the HV mode to manage operations that the guest is not permitted to handle directly, such as I/O, privileged instructions, or access to restricted memory. During this process, the CPU preserves the guest's state, then transfers control to the HV.

reads the packets from the TAP device, scans the shared Vrings to locate an available RX descriptor, copies the packet data into the RX Vring, advances the Used Ring pointer, and marks the descriptor as used. At this point, the QEMU checks for the interrupt suppression flag. If it is enabled, the QEMU waits for more packets until a threshold is met before triggering an interrupt. After placing the packets into the shared queue, the QEMU signals the KVM to run the guest and inject an interrupt into the virtual CPU. If the guest is idle and waiting for packets, the KVM immediately resumes its execution; otherwise, the HV records the pending notification and waits for the next VM_EXIT, such as a timer or another I/O event, to inject the signal [69]. The guest's VirtIO-net frontend driver then handles the interrupt by entering its polling routine, processing any received packets, and, if no additional packets remain, returning to an interrupt-waiting state. Figure 2.1 illustrates the packet processing workflow as defined by the VirtIO-net architecture.

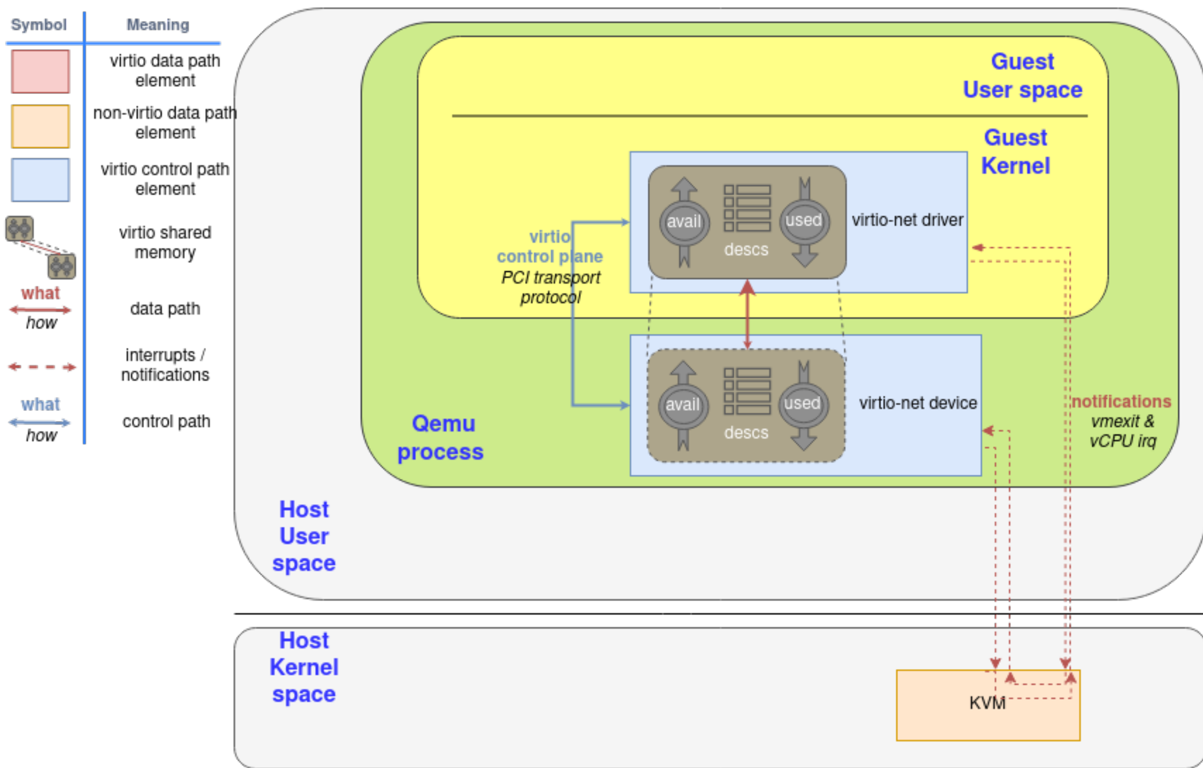


Figure 2.1: VirtIO-net Architecture, Reprinted From [47] With Permission

2.4.4 Vhost-net: Optimizing the VirtIO-net Driver

The Vhost-net architecture enhances the traditional VirtIO-net design by shifting the data path from the QEMU to a specialized kernel-space driver. With the Vhost-net driver operating within the host kernel, tasks such as Virtqueue management and packet forwarding are performed entirely in kernel space, eliminating the need to route packets through

the QEMU userspace process [2]. This design significantly reduces system calls, context switches, and unnecessary data copying, which would otherwise be incurred during packet transfers between the HV and the guest.

In the standard QEMU-based VirtIO-net setup (refer to Section 2.4.2), each packet transmission involves the guest placing a buffer into a Virtqueue and notifying the QEMU through KVM. The QEMU then retrieves the buffer, transmits it using the TAP interface, and notifies the guest, again via the KVM, that the buffer has been processed. A similar pattern occurs on the receive path, with interactions between the QEMU and the host kernel (via the KVM). These exchanges between the userspace and kernel space of the host introduce overhead due to frequent user-kernel transitions, memory copies, and context switches. The Vhost-net approach avoids this overhead by allowing the kernel to manage both the Virtqueues and the TAP interface directly. This enables packet transfers to bypass the QEMU entirely, allowing data to move directly between the guest and the host network stack.

Figure 2.2 illustrates how the packet path differs within the Vhost-net setup in comparison to VirtIO-net architecture. In this configuration, the Virtqueues are moved from the QEMU process in the host userspace to the host kernel space.

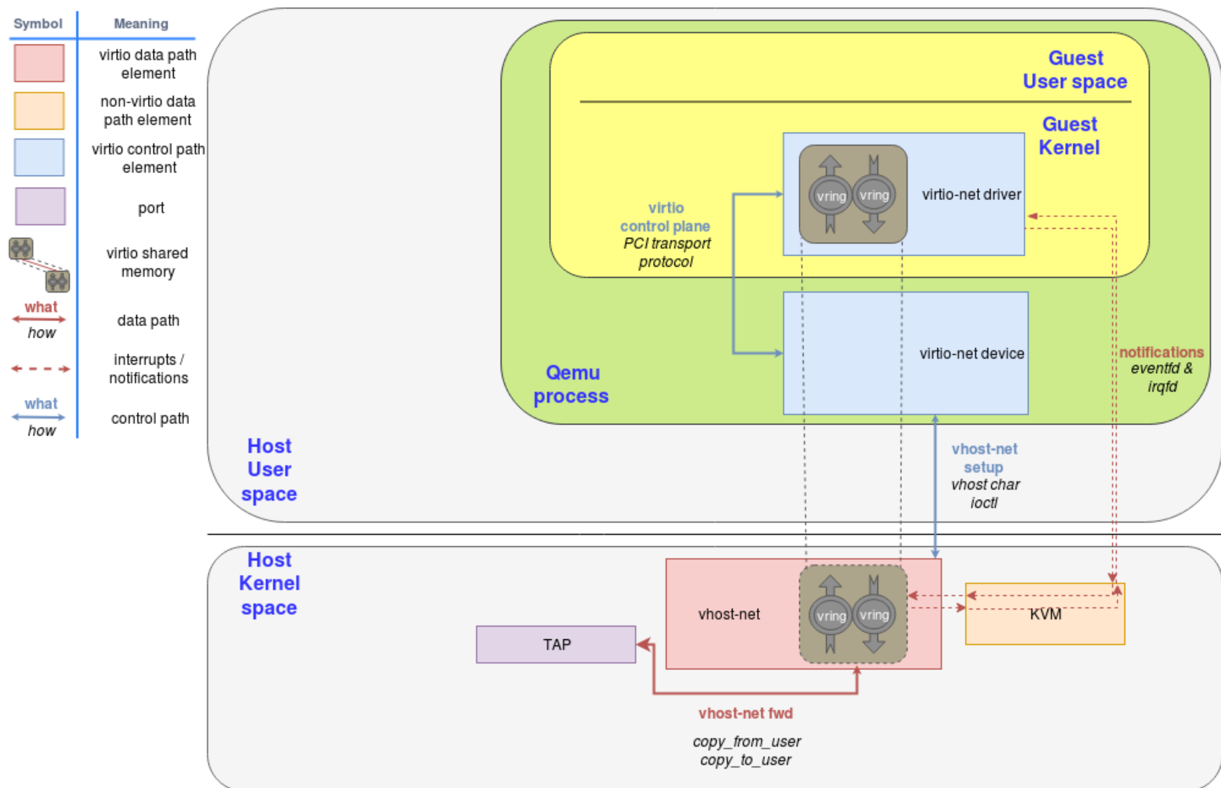


Figure 2.2: Vhost-net Architecture Reprinted From [2] With Permission

2.4.5 Device Passthrough

The VirtIO-net driver provides a paravirtualized network interface to virtual machines, presenting a standardized virtual NIC to VMs rather than emulating specific hardware. By offloading much of the virtualization work to the HV, VirtIO-net achieves good performance and portability, but it still relies on software interception of every packet and interrupt. In workloads demanding high throughput or low latency, such as large-scale web services, or real-time data applications, this overhead can become a bottleneck. To address these limitations, modern server-class NICs offer capabilities for direct device assignment, allowing a physical NIC to be passed through to a virtual machine. In this setup, the guest interacts with the NIC as if it were directly attached, just like the host would. This approach bypasses the host's network stack entirely, and hardware-generated interrupts are delivered directly to the guest rather than being emulated. The Virtual Function I/O (VFIO) kernel driver is responsible for enabling and managing this direct assignment.

The Linux kernel's VFIO framework enables secure userspace access to hardware devices by utilizing modern Input-Output Memory Management Unit (IOMMU) [1] features. Developed to replace older methods such as UIO [36] and KVM-specific device assignment, VFIO delivers a unified environment protected by IOMMU, enabling direct device access while preserving overall system integrity. This capability proves essential in areas like virtualization, high-performance computing, and low-latency I/O scenarios [41].

The security foundation of VFIO relies on IOMMU groups, which represent the smallest units of hardware isolation enforced by the system's IOMMU. Each group contains devices that share physical DMA routes, and the Linux kernel mandates that all devices in an IOMMU group must bind to VFIO drivers simultaneously. This rule ensures no HV driver retains access to any group member, effectively closing off potential security risks by preventing unauthorized DMA access. Within these groups, page tables manage address translation for device memory accesses; although each group provides the minimum isolation required for secure operation, VFIO supports page table sharing between groups through its container abstraction, thereby reducing both platform and userspace overhead.

To coordinate multiple IOMMU groups and streamline their management, the VFIO framework introduces containers as a higher-level abstraction. A container aggregates several IOMMU groups under a single security boundary, manages their shared DMA mapping policies and page tables, and enforces strict isolation of device DMA behavior. By linking groups via a container, userspace applications can share page tables across those groups, substantially decreasing Translation Lookaside Buffer (TLB) thrashing and enhancing performance by reducing address translation overhead [41].

VFIO-PCI acts as a bridge between physical PCI devices and virtual machines, granting VMs direct device access with near-native performance while preserving security through IOMMU protection [58]. When a VM requests access to a physical device, the host first binds the PCI device to the VFIO-PCI driver within the HV. QEMU then communicates with the driver to present the device to the guest, allowing nearly transparent hardware

access with minimal intervention from the HV. VFIO-PCI handles this device exposure by mapping the device's memory regions into the VM and managing its interrupts on behalf of the guest. Under the covers, the VFIO-PCI driver uses the IOMMU to map each Base Address Register into the VM's address space, yielding an IOMMU-protected region that the guest can safely use. When the VM issues DMA requests, VFIO-PCI collaborates with the IOMMU to establish I/O virtual-to-physical address mappings, ensuring that device-initiated DMA operations remain confined to the VM's assigned memory and cannot breach isolation guarantees.

Building on the VFIO-PCI functionality, PCI-passthrough grants a VM exclusive, near-native access to the NIC by binding it to the VFIO-PCI driver on the host. To set this up, the host must first enable and configure the IOMMU (e.g., Intel VT-d or AMD Vi) in BIOS. Once the IOMMU is active, the VFIO framework programs the DMA remapping tables so that all NIC-initiated memory accesses are confined to the VM's physical address space. This remapping is achieved through the IOMMU. The IOMMU translates the NIC-generated addresses to the VMs physical address space. In addition, VFIO enables the interrupt remapping, ensuring that the NIC's IRQs are routed directly to the guest rather than handled by the host. At this point, the administrator binds the NIC to VFIO-PCI (replacing any in-kernel host driver), and QEMU is instructed to attach that VFIO device to the VM. From then on, the guest sees the NIC as if it were its own: NIC's memory regions appear in guest's address space, DMA operations flow securely through the IOMMU, and interrupts fire directly inside the VM, all without additional context switches through userspace. However, because PCI-passthrough allocates the entire physical function exclusively to a single VM, it prevents sharing of that device among multiple guests.

2.4.6 SR-IOV

PCI-passthrough implements a `Full Passthrough` model, granting a VM exclusive control of the physical NIC and preventing any other guest from accessing it. An alternative method exists to enable multiple VMs to share the same NIC hardware. The option is called SR-IOV, referred to as `Partial Passthrough` in this thesis.

SR-IOV extends the PCI-passthrough model by subdividing one Physical Function (PF) into multiple Virtual Function (VF), each of which can be passed through independently. The PF is a fully featured PCIe function that manages SR-IOV capabilities (e.g., enabling or disabling virtualization), whereas each VF is a lightweight PCIe endpoint that shares the PF's underlying hardware resources, such as queues, buffers, and DMA engines, but presents its own base address registers, interrupt lines, and DMA streams. Virtual functions cannot perform administrative tasks, such as resetting the device or changing shared configuration; those remain the PF's responsibility, ensuring safe separation between control and data planes.

To enable SR-IOV, the PF driver on the host writes the desired number of VFs to the

NIC's PCI configuration space (by setting the /sys/class/net/NIC_PF/sriov_numvfs). The hardware then exposes that many new VF PCIe functions, each appearing as a separate PCI device with its own DMA regions and interrupts. Administrators bind each VF to VFIO-PCI just as with full passthrough. Underneath, the IOMMU is configured to maintain isolated DMA mappings for each VF, and interrupt remapping directs each VF's interrupts straight to its assigned VM. Finally, QEMU attaches the VFIO-bound VFs to individual guests, giving each VM a hardware-enforced, high-performance interface while sharing the same underlying physical device.

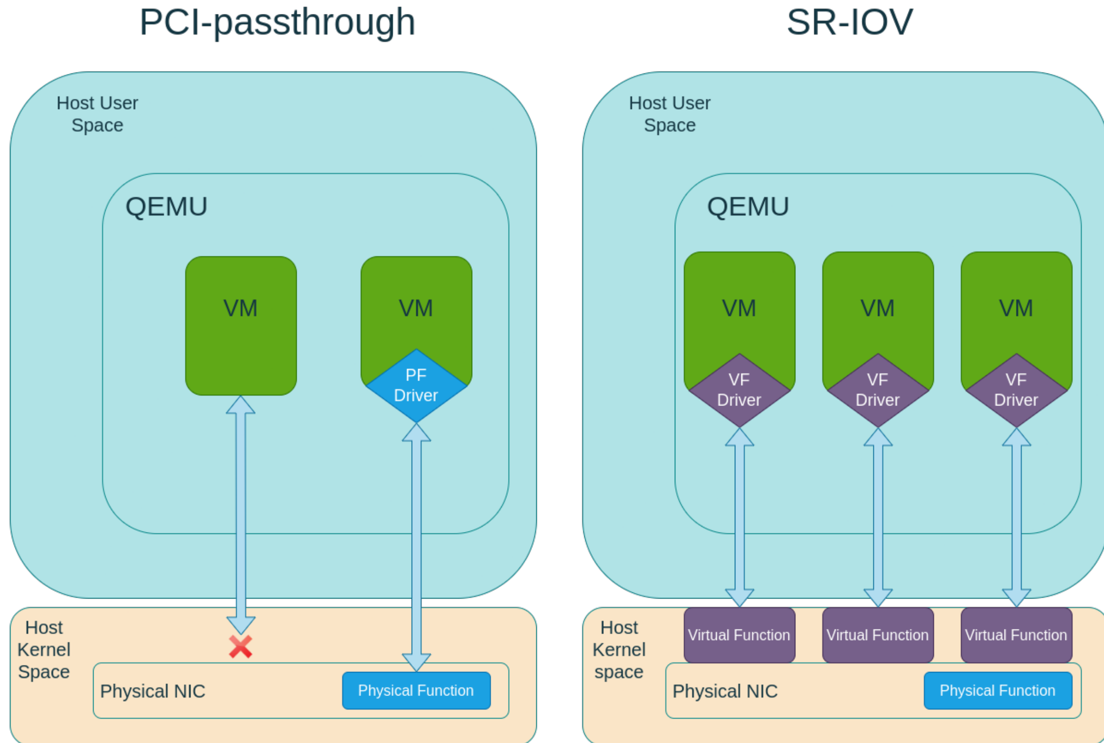


Figure 2.3: PCI-passthrough and SR-IOV Architecture for a NIC

Chapter 3

Network Stack Alignment

Chapter 2 explains that modern network stacks rely on two fundamental models for processing incoming packets. The first is the asynchronous, interrupt-driven model, where packet arrival triggers a hardware interrupt that invokes the NIC driver's ISR, which in turn defers further processing to the SoftIRQ context. The second is the synchronous, polling-based model, where either the application or a kernel-bypass framework continuously scans the NIC's descriptor rings in a tight loop, enabling immediate packet handling without relying on interrupts. The Linux networking subsystem accommodates both models. However, each introduces its own set of limitations that can affect performance under different workloads.

This chapter introduces a set of configurations for organizing IRQ handling that explore various combinations of spatial and temporal alignment within the Linux network stack. Spatial alignment refers to the mapping of interrupt handling and application execution to the set of physical CPU cores (core locality), whereas temporal alignment captures the coordination of interrupt handling with the application's execution path.

It is also important to emphasize that all configurations focus exclusively on optimizing the `RX` path of the NIC. Since `TX` interrupts occur much less frequently and are typically triggered only to reclaim transmit descriptors, they contribute minimally to the overall overhead and are therefore not the focus of optimization in this work.

3.1 IRQ Routing

Efficient interrupt routing is a key aspect of modern network stack performance, particularly in systems with multiple CPU cores and Non-Uniform Memory Access (NUMA) configurations. Affinity mappings define which CPU core is responsible for handling each hardware interrupt. Each queue on the NIC is linked to a unique IRQ number, which can be explicitly mapped to a set of CPU cores. This mapping influences cache locality, wake-up latency, and the cost of inter-core communication. The impact is especially critical in

multi-socket systems, as NICs are physically connected to a specific CPU socket. When interrupts and packet processing are handled on cores or memory outside that socket, the traffic must traverse inter-socket links, resulting in additional latency and performance penalties.

The method used to handle each IRQ is important in shaping network stack performance, especially in interrupt-driven setups where `RX` queues produce frequent interrupts. While the optimal strategy may vary based on workload characteristics, two prevalent approaches are generally adopted: one involves dynamically distributing IRQs across available CPU cores, whereas the other uses static assignments that bind specific IRQs to dedicated cores.

3.1.1 Dynamic Routing

In dynamic IRQ routing, a user-space daemon is typically responsible for monitoring and then reprogramming the interrupt handling across multiple CPU cores, based on the monitored data, to prevent bottlenecks on any single core. A common implementation of this strategy is IRQ balancing, which distributes the load of hardware interrupts based on observed traffic patterns. Since different IRQs may experience varying levels of activity depending on the workload, a daemon (called `irqbalance` daemon in Linux) monitors IRQ usage and periodically updates CPU affinity settings to maintain an even distribution of interrupt processing across available cores [27, 50, 57].

However, this approach has notable drawbacks. It does not account for the placement and scheduling of the application, which can lead to poor performance or sub-optimal locality (e.g., interrupts are assigned to CPUs on different NUMA nodes). In addition, different assignments for IRQ placement can produce performance variations. This variability is problematic for high-performance networking applications that demand a stable and predictable throughput. Finally, for a controlled performance evaluation, this approach is inadequate, as it makes reproducing consistent results more difficult. As a result, disabling the `irqbalance` daemon and utilizing a static method for IRQ affinity is usually preferred in such scenarios.

3.1.2 Static Routing

An alternative is static routing, where each IRQ line is permanently bound to specific CPU cores. There are two common strategies for static assignment. The first approach maps IRQs to the same cores used by the application, allowing both packet processing and application execution to occur on the same set of CPUs. The second approach, known as IRQ packing, allocates a small, dedicated set of cores exclusively for handling interrupts.

The first approach, often referred to as one-to-one mapping, assigns each `RX` queue to a dedicated CPU core. In a standard configuration, an application utilizing N cores

(with N threads) is paired with N `RX` and N `TX` queues, where each core handles one `RX` and one `TX` interrupt. It is typically assumed that the NIC evenly distributes traffic across all `RX` and `TX` queues, resulting in a near-balanced interrupt load across cores. This simple static setup enables efficient IRQ load balancing without relying on dynamic routing solutions [10]. When used alongside features such as RSS and the `SO_INCOMING_NAPI_ID` [42] socket option, the configuration ensures that packets arriving on a specific NIC queue are consistently handled by the corresponding CPU.

In contrast, the IRQ packing strategy reserves one or more CPU cores specifically for handling hardware interrupts, while application logic runs on the remaining cores. This approach sets the IRQ affinities to a small group of isolated CPUs, effectively concentrating all interrupt handling on those cores. By doing so, it reduces context switching and prevents cache pollution on cores running application workloads. In this setup, the number of NIC queues is configured to match the number of dedicated IRQ-handling cores. These dedicated cores perform both the interrupt processing and initial packet handling, then forward the processed packets to application threads running on separate cores.

The literature [10] has shown that IRQ packing can deliver excellent performance and competitive tail latency, highlighting the performance cost associated with hardware interrupt handling. This supports the idea that separating application execution from interrupt processing can lead to significant performance benefits. However, this method introduces several challenges:

- **Packing Configuration:** Determining the optimal number of dedicated IRQ cores depends heavily on the specific workload. Dynamically adjusting this allocation at runtime is complex and not easily supported. Furthermore, in systems with a NUMA setup, configuring IRQ packing becomes significantly more challenging, as it requires careful coordination to avoid spreading resources across NUMA nodes. When this separation is not managed properly, the system incurs additional latency and performance penalties due to non-local memory access.
- **Resource Overhead:** In certain scenarios, more cores may be allocated to IRQ handling than are actually needed. Since CPU cores can only be assigned in whole units, it is not possible to precisely match the required processing capacity, which may lead to underutilized resources.

3.1.3 IRQ Routing in Virtualized Environment

Section 2.4.3 explains that interrupt processing in a virtualized environment (specifically VirtIO-net) involves two distinct stages. The first stage occurs on the HV, where the physical NIC generates a hardware interrupt upon receiving a packet to alert the host CPU. The second stage takes place inside the VM, after the I/O threads of QEMU (or Vhost) are woken up by the hardware interrupts and begin polling the received packets, and forwarding

them to the guest. As a result, the way hardware IRQs are configured on the host can directly impact the performance experienced within the VM. It is important to note that the IRQ routing discussed in this section specifically pertains to the VirtIO-net driver. The impact of IRQ routing on `Full Passthrough` and `Partial Passthrough` remains consistent with the analysis in the previous section, as both methods expose hardware interrupts directly to the guest.

This thesis looks at two distinct ways to handle the hardware IRQs for a VM. The first approach consists of the one-to-one static configuration discussed in section 3.1.2. The second one is to use IRQ packing on the HV to separate the hardware interrupt from the VM CPU cores.

In the first approach, every physical core that runs a VM application thread is also responsible for handling a NIC queue. As a result, each `RX` queue of the NIC is explicitly mapped to a dedicated physical CPU, and the corresponding virtual CPU of the VM is scheduled on that same core.

This setup achieves a near-perfect alignment between the VM and the IRQ handler, as both operate on the same set of cores. This alignment maintains a hot path for the data between the HV and the VM. However, this configuration raises the possibility of hardware interrupts interfering with the VM's execution (distorting the application processing inside the VM). Since the physical cores are shared, incoming hardware interrupts trigger context switches from the VM application to the IRQ handler. Once the I/O threads complete packet polling, the system restores the VM context.

Alternatively, the IRQ packing strategy on the HV addresses this issue by assigning hardware interrupts to a dedicated set of physical cores that do not overlap with the virtual CPUs of the VM. As described in Section 3.1.2, this setup reserves specific cores on the HV solely for handling hardware interrupts, while the virtual CPUs run on separate cores.

This approach helps reduce context-switching overhead by preventing virtual CPU interruption during application execution, an advantage particularly relevant for latency-sensitive workloads. However, it introduces trade-offs: dedicating a core exclusively for interrupt handling may lead to underutilized CPU resources, and it forfeits the tight alignment between interrupt and application processing seen in the previous configuration.

3.2 Polling

Interrupt-driven methods alone are not suitable as a general-purpose solution. Approaches such as IRQ packing may result in inefficient use of system resources, while others can compromise memory locality, leading to degraded performance. These limitations arise because interrupt handling operates independently of the application's execution context.

User-level networking frameworks often rely on polling to avoid the overhead associated with interrupts and better synchronize with application logic. For example, Intel's Data

Plane Development Kit (DPDK) [8, 40] uses Poll Mode Drivers that expose the NIC's RX/TX descriptor rings directly to user space, allowing the application to continuously poll and process packets in batches. In this design, interrupts are completely disabled, and the application is fully responsible for both retrieving and handling packets. Packet polling begins when the application has no other tasks to perform and pauses when it is actively processing packets.

Linux offers a form of busy polling for applications (through I/O multiplexing calls such as `epoll_wait`), though it comes with certain limitations. As outlined in Section 2.3.1, the kernel disables hardware interrupts during the busy polling phase; however, it re-enables them immediately afterward. As a result, while the application continues to process previously received data, incoming interrupts may disrupt its execution. These interrupts allow the application to detect newly arrived packets even before it enters the polling cycle. This behavior limits the efficiency of busy polling. To address this issue and better emulate the behavior seen in user-level networking frameworks, `Busy Wait` and the `IRQ Suspend` mechanisms are introduced.

3.2.1 Busy Wait

The previous section shows that the current busy polling implementation re-enables interrupts without considering whether the application is still handling data from a previous polling cycle. A straightforward solution to this issue is to keep interrupts disabled throughout the application's execution. This allows the application to start polling for packets only when it has completed processing and is ready for more work. This method is referred to as `Busy Wait` in this thesis work, as the application continuously polls the `RX` queues without relying on interrupts.

The `Busy Wait` mode is activated by enabling both the SoftIRQ coalescing mechanism (see Section 2.2.3) and `epoll`'s busy polling support. Key parameters that control `epoll` busy polling, such as `prefer_busy_poll`, `busy_poll_budget`, and `busy_poll_usecs`, can be configured using the `EPIOCSPARAMS` ioctl interface [74]. Additionally, the `epoll` instance must be set to non-blocking mode, meaning its timeout value must be less than or equal to the busy polling timeout to ensure proper operation. Configuring the `epoll` timeout in this manner prevents the application from entering the sleep state, even when a busy polling cycle completes without receiving any packets.

When an application invokes the `epoll_wait` system call, the kernel's busy polling logic, via the `napi_busy_loop` function, executes until its polling budget is exhausted or the timeout expires. At the end of the polling cycle, the kernel checks whether `prefer_busy_poll` is enabled. If this flag is set, the system then consults the `gro_flush_timeout` and `napi_defer_hard_irqs` values. If enabled, the SoftIRQ timer is deferred by the value of `gro_flush_timeout`. Setting a sufficiently large value effectively postpones SoftIRQ execution indefinitely, thereby keeping the system in busy polling mode.

The described execution flow proceeds independently of whether events are present. If `epoll_wait` returns with events, the application begins processing the received packets. A high value for `gro_flush_timeout` ensures that the SoftIRQ mechanism does not interrupt the application during this processing. Once the processing is complete, the application repeats the same steps. If no events are available, it continues invoking `epoll_wait` until an event occurs. This way, the packet polling path completely shifts from the SoftIRQ side to the application side.

To fully leverage this method, a static one-to-one mapping between application threads and the NIC's `RX` queues (see Section 3.1.2) is strongly recommended. This one-to-one mapping requires that every application thread handle one specific `RX` queue. If this configuration is not applied, it might lead to performance degradation. For instance, if a single core polls multiple queues, that core can become a bottleneck due to excessive polling load.

The `Busy Wait` approach shifts full control of packet polling to the application, ensuring that packet handling remains highly synchronous. However, it has a significant drawback: the mechanism continuously polls the `RX` queues, even under light traffic conditions. As a result, CPU cores remain fully utilized regardless of workload, leading to inefficient use of system resources.

3.2.2 IRQ Suspend

The `Busy Wait` approach delegates the responsibility of packet polling to the application. However, it shows a limitation: when no packets are found during the polling loop, the application continues to poll the NIC queues unnecessarily, resulting in wasted CPU cycles. This highlights the need for a mechanism that allows the application to pause and switch to an interrupt-driven mode during low traffic periods. The `IRQ Suspend` mechanism addresses this issue by extending the `Busy Wait` design to enable dynamic switching between polling and interrupt modes based on traffic load.

Section 2.2.3 shows that the kernel uses two configuration flags to defer both hardware and software interrupts by rearming the SoftIRQ timer. During the polling loop in `epoll_wait`, the busy polling logic checks these flags, and if set, it re-arms the SoftIRQ timer. While this defers software interrupt handling, it does so without considering the current state of the application workload. As a result, a low timeout value may interrupt the application prematurely, while a high value can delay packet processing. To improve coordination, a new mechanism is needed to address these shortcomings by making interrupt control more responsive to application behavior.

The modification is implemented directly within the `epoll_wait` system call and leverages per-NAPI configuration flags to simplify integration. As part of the enhancement, a new parameter called `irq_suspend_timeout` is introduced to further delay interrupt handling when the busy polling loop exits with available events. As previously discussed, using a small timeout value can prematurely interrupt the application, potentially disrupting its

processing. The new parameter addresses this by allowing the application to extend the interrupt deferral period. Specifically, as shown in Figure 3.1, this parameter is used to rearm the SoftIRQ timer with a configurable timeout. The timeout should be set high enough to ensure that the application can complete processing without interference. The logic for this rearming behavior is implemented in the `napi_suspend_irqs` function.

On the other hand, when the application finishes processing its workload and a polling cycle returns without any new events, the packet processing needs to migrate from busy polling to interrupt-driven before the application thread enters the sleep state. This transition is triggered by scheduling the associated NAPI instance, which performs an additional polling round. If no packets are available during this cycle and the `gro_flush_timeout` parameter is set, the SoftIRQ timer is restarted using this timeout value. If the parameter is not set, interrupts are re-enabled immediately. This step is essential because, during the earlier busy polling phases, the application masked interrupts by rearming the SoftIRQ timer. To restore normal interrupt handling, the system should proactively re-enable interrupts or reset the SoftIRQ timer to a smaller value (`gro_flush_timeout`) rather than waiting for the timer to expire. This logic is implemented inside the `napi_resume_irqs` function.

```
Procedure EP_POLL(ep, events)                          // epoll_wait() implementation

1.  eavail ← ep_events_available(ep)
2.  loop
3.      if eavail then
4.          res ← ep_send_events(ep, events, maxevents)   // Send events to application
5.          if res > 0 then
6.              napi_suspend_irqs(ep.napi_id)
7.          end if
8.          if res > 0 then
9.              return res
10.         end if
11.     end if
12.     eavail ← do_busy_poll(ep)
13.     if eavail then
14.         continue
15.     end if
16.     napi_resume_irqs(ep.napi_id)
17.     sleep_until_notified(ep)
18.     eavail ← ep_events_available(ep)
19. end loop
```

Figure 3.1: IRQ Suspend (Pseudo-code With Highlighted Modifications)

The `IRQ Suspend` mechanism can be activated by configuring SoftIRQ coalescing (optional to set, but assigning a small value to `gro_flush_timeout` is recommended), assigning

a large value to irq_suspend_timeout, and enabling the SO_PREFER_BUSY_POLLING flag [1]. Unlike Busy Wait, the application's associated epoll instance should be blocking. Blocking epoll refers to configuring its timeout so that the application enters a sleep state when no events are detected. This behavior can be achieved by assigning a sufficiently large finite timeout value or by setting the timeout to -1, which causes the epoll_wait to block indefinitely until an event occurs.

3.2.3 Polling in Virtualized Environment

Polling plays a critical role in virtualized environments because it helps reduce the frequency of context switches between the HV and the guest. Section 2.4.3 shows that when the HV injects an interrupt into the guest, a VM_EXIT is triggered. In contrast, during busy polling, the guest can access packet data directly from the shared memory structures without relying on interrupts. By masking interrupts, the guest avoids unnecessary exits, allowing the HV (under VirtIO-net scenario) or the NIC (under SR-IOV and PCI-passthrough scenario) to place data into shared buffers without disrupting the virtual CPU [64]. This approach significantly reduces context switching. This is especially beneficial under high load. Therefore, polling inside the VM can lead to a better performance than the interrupt-driven scenario.

The HV treats virtual CPUs as regular threads that must compete with other threads for scheduling. Under a constant polling scenario, the virtual CPU remains active and avoids voluntarily yielding, which effectively pressures the scheduler to keep it running. Therefore, to fully utilize polling within a VM, it is recommended to pin each virtual CPU to a dedicated physical core [32]. This approach minimizes contention and reduces interference between virtual CPUs, leading to more consistent performance.

However, this behavior keeps the physical core underlying the virtual CPU fully occupied, resulting in continuous CPU usage. If the core is shared with other threads, it may raise fairness concerns. Hence, a more flexible approach, such as IRQ Suspend (refer to Section 3.2.2), which dynamically switches between interrupt-driven and polling modes, offers a better balance between performance and resource sharing.

[1] IRQ Suspend is added to the kernel version 6.13 or later

Chapter 4

Evaluation

In this chapter, the IRQ Suspend mechanism is evaluated in comparison to other representative configurations to analyze its effects on the network stack under controlled conditions. The selected test cases reflect the extremes of the configuration spectrum.

4.1 Experimental Setup

4.1.1 Hardware

The evaluation is conducted on a system equipped with two Intel Xeon E5-2670 CPUs, each with eight cores, resulting in a total of 16 cores within a dual-socket NUMA architecture. The system features 32 GiB of memory, split evenly between the two NUMA nodes, with 16 GiB allocated to each. A Mellanox ConnectX-4 Lx 25 Gigabit Ethernet adapter provides network connectivity. The base frequency of the CPU is 2.6 GHz, but during high load the CPU runs at a constant turbo frequency of 3 GHz. Hyperthreading is avoided by binding threads exclusively to the first hardware thread of each physical core. Cache configuration for each core on the server includes 32KB for both Level-1-data-cache (L1d) and Level-1-instruction-cache (L1i), 256KB of Level-2-cache (L2) cache, and 20MB of Last-Level-Cache (LLC). Additionally, six client machines generate workload for the server.

4.1.2 Software

All servers used in the experiments run Ubuntu 24.04. Because the experiments require per-NAPI configuration support (refer to Section 2.2.3), a Linux kernel version higher than 6.13 is necessary. All experiments were performed using Linux kernel version 6.15.0, compiled with the generic configuration [43].

In each test configuration, IRQs are statically assigned by mapping every CPU core to a unique pair of RX and TX queues on the NIC. This setup ensures that each core is

exclusively responsible for one `RX` and one `TX` queue, thereby eliminating contention or sharing across interrupt contexts. To maintain consistency, dynamic IRQ-assignment service (`irqbalance`) is disabled, preventing interrupt migration during execution. Furthermore, all workloads are confined to a single NUMA node.

For the VM experiments, a bridge-based networking setup is used to route traffic between the HV and the VM. In general, when relying solely on software to connect a guest VM to the outside world (or to the host), the choice is between port forwarding (NAT) and bridging. With port forwarding, the Linux network stack intercepts packets destined for the VM by matching them against `iptable` rules, rewriting addresses as needed, and then forwarding the traffic to the VM's virtual interface. The routing overhead introduced by `iptable` does not scale well with increasing traffic. As the packet rate increases, each packet must pass through the `iptable` before reaching the VM, which can create a performance bottleneck under heavy network load [60].

By contrast, bridge networking relies on a "bridge device" in the host kernel, which essentially functions as a software-based switch at the Ethernet layer. In Linux, a bridge is created (for example, using `brctl`) and then associated with one or more physical and virtual interfaces. In this setup, the bridge device is attached directly to the physical NIC (refer to Section 4.1.1), and the VM's virtual network interface (typically exposed as a TAP) is bonded to that same bridge. Once configured, the bridge learns MAC addresses dynamically by inspecting the source addresses of incoming frames on each port. When the VM sends a packet, it is injected into the bridge; the bridge then forwards the frame either to the physical NIC (if the destination MAC resides on the external network) or directly to another enslaved interface. Likewise, packets arriving from the external network reach the bridge through the physical NIC, and the bridge forwards them to the VM's interface if the destination MAC matches. Because the bridge operates at the Ethernet layer, the VM appears on the same Ethernet segment as the host. This structure eliminates the overhead of IP-level translations and allows frames to flow more directly [56]. Additionally, to enhance networking performance, Vhost-net is enabled (refer to Section 2.4.4).

To minimize the scheduling overhead of virtual CPUs onto physical CPUs, each virtual CPU is explicitly pinned to a designated physical core. These physical cores are all located within the same NUMA node, ensuring consistent experiment results.

4.1.3 Benchmark

Memcached serves as an ideal application for benchmarking network stacks and systems software because it is both widely used in production environments and lightweight enough to reveal the underlying runtime system's performance and efficiency characteristics. All experiments in this study use Memcached version 1.6.32 running on the Linux kernel's network stack.

Mutilate [39], a widely recognized benchmarking tool for Memcached, generates the workload for these experiments. Each of the 6 clients runs 8 threads (one per core), with

every client thread establishing 16 connections to the server and operating with a pipeline depth of 1. The experiments simulate Mutilate's synthetic version of Facebook's "ETC" workload, as documented in the literature [4], using a dataset of 1 million records [10].

4.2 Data Collection

Outputs from various tools are used to comprehensively analyze the behavior and effectiveness of each test case. The primary tools used for data collection include Mutilate, Perf [51, 25], and System Activity Reporter (SAR) [24].

Section 4.1.3 outlines that Mutilate serves as the load generator for the experiments. After each test run, it produces several key metrics:

- Queries per Second (QPS): Measures the number of client requests processed by the `Memcached` server per second, providing an overall indication of system throughput.
- **Latency**: Captures the time taken to fetch data from memory, measured from the moment a `GET` request is issued to the point the response is received by the client. Mutilate reports several statistics for this metric, including the average, minimum, maximum, and the 95th and 99th percentiles. This study focuses on the 95th and 99th percentile values.

To measure the server's peak throughput, a *closed-loop* experiment is used. In this setup, each Mutilate agent sends pipelined requests up to a specified depth, then waits for the responses before issuing new ones. Since the depth is set to 1 in this study, each agent sends a single request and blocks until a response is received. By using a sufficient number of parallel connections, the server can be driven to full capacity, with CPU usage reaching 100%. Under these conditions, the measured QPS represents the server's peak request-handling rate.

Latency is measured using an *open-loop* setup, where the load generator controls the request rate to allow precise monitoring of response times. In this configuration, the generator sends requests at a fixed rate below the maximum capacity, and the reported latency reflects the server's response time for each request. For latency analysis, researchers typically focus on tail latency rather than average latency. Consequently, this thesis examines both the 99th and 95th percentiles, which indicate that 99% and 95% of the observed latencies fall below a specific thresholds.

The SAR utility, included in the `sysstat` package, offers a detailed method for monitoring and analyzing system performance on Linux platforms. It collects data on various metrics, such as CPU usage, memory consumption, I/O activity, and network performance by accessing the `/proc` filesystem or reading from daily binary logs created by the `sadc` daemon [24]. In this study, SAR is used to track CPU utilization of each application core at

1-second intervals. After running for 10 seconds, the SAR process is terminated, yielding 10 data points that reflect the CPU usage of each application core.

`Perf` is a versatile performance analysis utility that interfaces directly with the CPU's Performance Monitoring Unit (PMU), allowing for unified tracking of both hardware and software events. It supports a broad range of performance metrics with minimal overhead on the application or system. In this study, the focus is primarily on events that reflect CPU cycle consumption and cache behavior:

- `cycles`: total number of CPU core clock cycles consumed during program execution.
- `instructions`: total number of instructions executed during the program runtime.
- `L1-icache-load-misses`: number of instruction fetch misses in the L1 instruction cache, highlighting stalls due to cold or capacity misses.
- `L1-dcache-loads`: total data load operations targeting the L1 data cache, representing both cache hits and misses.
- `L1-dcache-load-misses`: count of data load operations that missed in the L1 data cache, reflecting inefficiencies at the first cache level.
- `LLC-loads`: total number of instruction and data loads accessing the last-level cache (L3 cache).
- `LLC-load-misses`: number of loads that missed in the last-level cache, resulting in accesses to main memory and contributing to memory-bound stalls.

Modern CPUs support only a limited number of hardware performance counters, making it impractical to monitor a large set of events simultaneously without incurring tradeoffs. When the number of events exceeds the available counters, the system must employ counter multiplexing, sharing counters across events. Although `Perf` can sample the additional events, each sample introduces an interrupt and triggers kernel-side bookkeeping, adding nontrivial overhead and perturbing the measured workload. Moreover, since each event receives a fraction of the sampling window, the reported values may not accurately reflect actual system behavior. To mitigate these issues, a more reliable approach involves grouping related events [44], such as cache metrics, instruction counts, and CPU cycles, and collecting them in separate runs. Following this method, the evaluation in this study is performed in two stages: the first captures overall CPU cycles and instruction counts, and the second focuses on cache-level behavior.

4.3 Performance Metrics

To evaluate the performance differences across various test cases, it is essential to examine changes in the total number of CPU cycles and executed instructions. By analyzing how the instruction count and cycle usage vary between configurations, the specific factors contributing to performance improvements are identified.

As noted in [10, 68], the Instructions per Cycle (IPC) metric serves as an indicator of how effectively the CPU's execution pipeline handles a given workload. In this context, IPC correlates with overall performance (QPS). According to [10], the relationship is defined by the following formula:

$$\mathrm{IPC} = \frac{instructions}{cycles} \quad \mathrm{IPQ} = \frac{instructions}{queries} \quad \mathrm{CPS} = \frac{cycles}{unit\ time(1s)} \tag{4.1}$$

$$\mathrm{QPS} = \frac{CPS \times IPC}{IPQ} \Leftrightarrow \frac{QPS}{CPS} = \frac{IPC}{IPQ} \tag{4.2}$$

The formula shows that performance improves, as indicated by higher QPS, when Instructions per Query (IPQ) is lower or IPC is higher. A lower IPQ means the CPU needs fewer instructions to handle each query, pointing to more efficient execution. Similarly, a higher IPC typically reflects reduced pipeline stalls, allowing the processor to complete more instructions per cycle. Additionally, a lower Cycles per Second (CPS) indicates that the CPU consumed fewer cycles to execute the same number of instructions. However, this metric is meaningful only in a *closed-loop* setup, where the CPU remains consistently active. In contrast, during *open-loop* experiments, CPS may not provide useful insights, as a lower value could simply reflect underutilization of the CPU rather than improved efficiency. A low CPS and IPQ under *closed-loop* scenario shows a more efficient CPU execution path, resulting in a better IPC and QPS.

4.4 IRQ Suspend Effectiveness

This section replicates prior experiments from [10, 17] and reports the results to establish a performance and efficiency baseline. A comparison is made between the performance characteristics of the following test cases:

- `No Coalescing`: This configuration adopts a fully interrupt-driven model with no interrupt coalescing, waking up the CPU in response to incoming interrupts. As a result, it yields the lowest throughput, since it depends entirely on interrupts for packet processing. Analyzing this setup is valuable because it highlights the overhead associated with frequent interrupt handling. To implement this configuration, adaptive interrupt coalescing is disabled on the NIC, and the static coalescing threshold is

set to its minimum, causing an interrupt to be generated for each individual packet (see Appendix A.1).

- `Adaptive Coalescing`: This setting is similar to `No Coalescing`, but operates with the NIC's default hardware coalescing settings enabled (adaptive coalescing of the NIC is enabled). Therefore, the NIC decides when to issue an interrupt, resulting in a variable batch of packets before issuing an interrupt. This scenario is important to evaluate because it reflects the standard behavior of interrupt handling with default settings applied on both the kernel and the NIC (see Appendix A.2).

- `Busy Wait`: This setup reflects a fully busy waiting approach (discussed in Section 3.2.1), where the application continuously polls the NIC queues and interrupts are completely disabled. By doing so, it maximizes throughput and keeps the CPU fully occupied at all times, independent of the incoming traffic rate. Evaluating this approach is essential because it demonstrates the highest level of performance the application can potentially reach (see Appendix A.3).

- `IRQ Suspend`: This method dynamically alternates between polling and interrupt-driven modes for packet processing based on the traffic load, offering a more adaptive strategy (see Appendix A.4).

4.4.1 Closed-loop Maximum Throughput

Table 4.1: Throughput Metrics on Physical Machine, Memcached 8 Cores

Testcase	**QPS**	**IPC**	**CPQ**	**CPU Breakdown (%)**		
				User	**System**	**SoftIRQ**
No Coalescing	668,071	0.657	35,841	17	49	34
Adaptive Coalescing	905,219	0.829	26,451	21	53	26
IRQ Suspend	958,737	0.892	24,975	23	77	0
Busy Wait	963,527	0.895	24,850	22	78	0

Table 4.1 shows the performance metrics discussed in Section 4.3 for each test case. It also presents the proportion of time each configuration spends in the SoftIRQ context.

The improvements observed in the `IRQ Suspend` and `Busy Wait` test cases arise from shifting the packet processing path away from the traditional interrupt and SoftIRQ pipeline. Instead, these mechanisms rely on the `napi_busy_poll` function, which enables the kernel to poll the NIC queues directly from the context of the application thread. This design change moves the "hot path" of packet processing into the system context, reducing context switches and simplifying the execution flow.

A key outcome of this transition is the enhancement of the IPC metric. Specifically, `IRQ Suspend` and `Busy Wait` demonstrate a 36% improvement in IPC compared to the

No Coalescing setup, and achieve a 7% increase relative to the Adaptive Coalescing configuration. Since the packet handling now occurs in the same context as the application, instruction paths remain more consistent and predictable. This visibility of instruction flow, without interruption by asynchronous events, allows the CPU pipeline to execute instructions more efficiently.

Table 4.1 confirms this behavior, showing that in both IRQ Suspend and Busy Wait, the time spent in the SoftIRQ context is zero. This shift in execution highlights the performance advantage of handling packets directly in the system path. Complementing this, software-based IRQ coalescing helps delay the re-enabling of interrupts, which further reduces interrupt frequency. Together, these mechanisms lower processing overhead, streamline the packet handling pipeline, and contribute to improved throughput and responsiveness under high traffic conditions.

Table 4.1 indicates that No Coalescing achieves the lowest throughput among all configurations. This outcome is primarily driven by the high volume of interrupts it generates. Since No Coalescing relies exclusively on interrupt-based processing, a significant portion of CPU time is consumed by the SoftIRQ context (34% of the CPU time), reducing the time available for the application thread. The elevated SoftIRQ utilization in Table 4.1 reflects this behavior. The Adaptive Coalescing setting delivers a notable performance boost over the No Coalescing configuration by allowing the NIC to batch more packets before generating an interrupt at the hardware level, resulting in a lower SoftIRQ percentage. However, despite this gain, it still underperforms compared to the IRQ Suspend and Busy Wait mechanism. This gap arises because adaptive coalescing operates solely at the hardware level and does not take the application's state into account. In contrast, IRQ Suspend and Busy Wait align packet processing with the application's execution, ensuring that interrupts do not disrupt its flow. As a result, they achieve greater performance improvements.

4.4.2 Open-loop Latency

In addition to throughput, tail latency is a critical metric for comparing the IRQ Suspend mechanism against alternatives. To measure tail latency, the load generators (refer to Section 4.1.1) operate in an *open-loop* mode. The results presented in Figures 4.1a and 4.1b reflect the observed 95th and the 99th percentile latency in relation to the achieved throughput during the experiment.

Figures 4.1a and 4.1b clearly demonstrate that both the IRQ Suspend and Busy Wait configurations outperform the baselines (No Coalescing and Adaptive Coalescing) and exhibit comparable tail latency characteristics. In the case of busy polling, the CPU actively spins in a tight loop, continuously monitoring the NIC's receive queue. As soon as a packet arrives, it is processed immediately, eliminating the need for interrupt handling, context switching, or scheduler coordination. Since the system never waits on an interrupt, it avoids the latency introduced by deferred processing in components like SoftIRQ. The IRQ

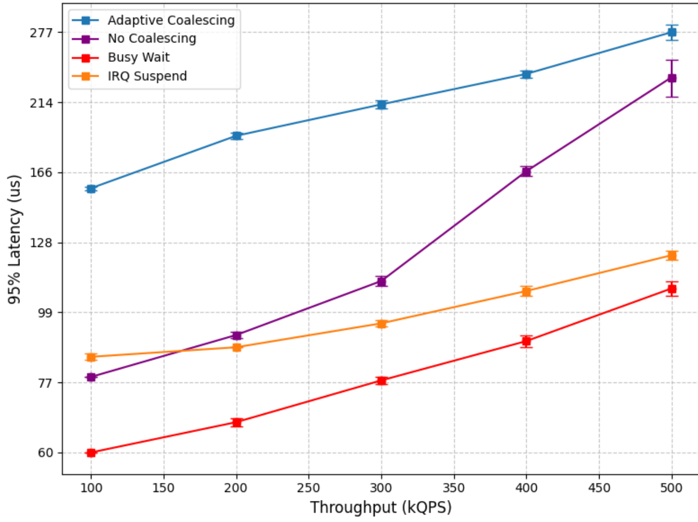
`Suspend` approach behaves similarly by remaining in polling mode while the application has work to perform, but it gracefully falls back to interrupt mode once the workload subsides. This design avoids delays associated with purely interrupt-driven processing and creates a hybrid model that effectively bridges the behavior of the high-performance `Busy Wait` and the interrupt-driven `Adaptive Coalescing` configuration. As a result, the latency curves for `IRQ Suspend` and `Busy Wait` closely resemble each other, particularly under high traffic conditions, as shown in Figures 4.1a and 4.1b. However, under light traffic, the behavior of `IRQ Suspend` aligns more closely with that of the `No Coalescing` configuration. Another notable observation is that the error bars become larger at higher throughput levels. This behavior is expected, as increasing traffic causes packets to accumulate in various stages of the packet processing path, such as NIC hardware queues and application socket buffers.

Conversely, the `No Coalescing` and `Adaptive Coalescing` setups trigger an interrupt for each or a batch of incoming packets. Each packet must first pass through the IRQ handler, then defer to the SoftIRQ context, and eventually wake the application thread. This multi-stage process introduces additional overhead due to context switches and thread wake-up delays, ultimately increasing the time required to process each packet. Since adaptive hardware coalescing is disabled in the `No Coalescing` setup, the NIC triggers an interrupt after receiving a packet. As discussed in Section 2.2.4, this configuration enables the host CPU to react promptly to packet arrival. Consequently, packet polling starts immediately after the interrupt, resulting in lower latency compared to the `Adaptive Coalescing` configuration, where processing is deferred until a packet threshold is reached or a timer expires.

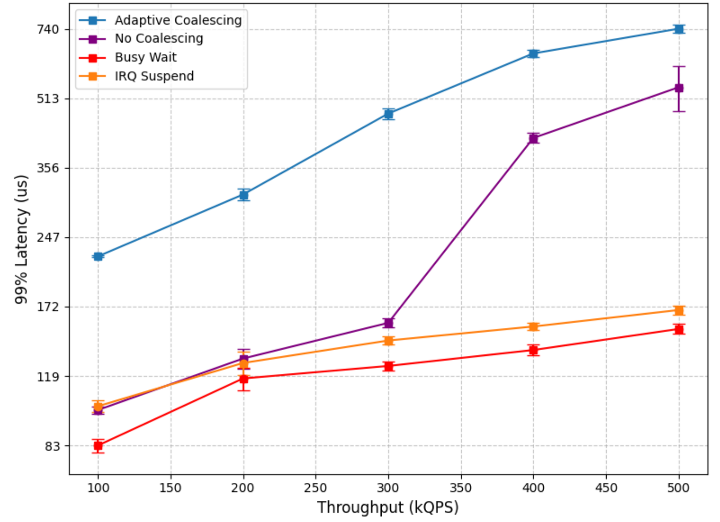
4.4.3 CPU Utilization

Up to this point, the results show that both the `IRQ Suspend` and `Busy Wait` configurations deliver comparable throughput and tail latency. However, the `Busy Wait` approach keeps the CPU continuously active throughout the entire experiment, leading to constant CPU usage and potential inefficiencies due to idle spinning and wasted hardware cycles. In contrast, the `Adaptive Coalescing` and `No Coalescing` cases activate the CPU only when packets arrive at the NIC, conserving CPU resources during idle periods.

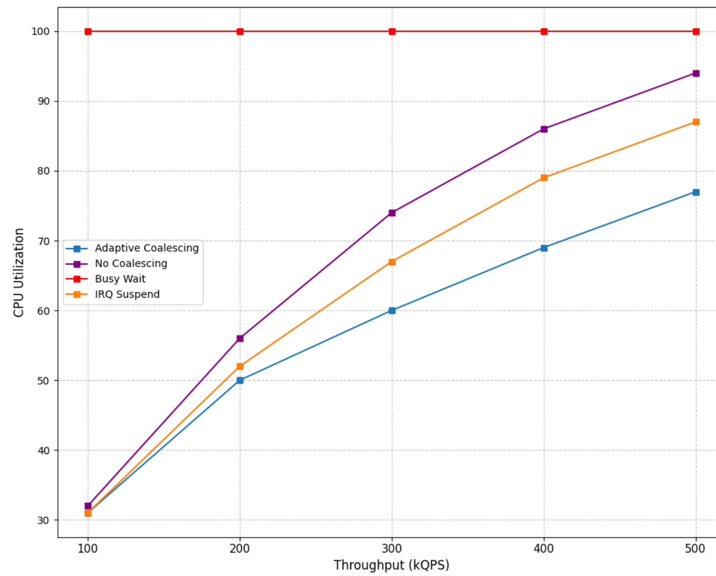
The `IRQ Suspend` mechanism, acting as a hybrid approach, dynamically shifts between `Adaptive Coalescing` and `Busy Wait` behavior depending on the network load. Under low traffic, it behaves like the `Adaptive Coalescing` case by minimizing CPU usage, while under high load, it ramps up utilization to match the performance of `Busy Wait`. In other words, the `IRQ Suspend` mechanism re-enables interrupts only when the application has no remaining workload to process. This adaptive behavior is shown in Figure 4.1c. The `Adaptive Coalescing` configuration shows more efficient CPU utilization compared to `No Coalescing`, as it benefits from packet batching. In contrast, `No Coalescing` generates interrupts more frequently, causing the host CPU to become more active in handling a smaller batch of incoming packets.



(a) 95th Percentile Latency



(b) 99th Percentile Latency



(c) CPU Utilization

Figure 4.1: Latency and CPU Utilization vs Throughput, Memcached 8 Cores (Lower is Better)[1]

[1] The X-axis is restricted to a QPS range of 100K to 500K because the `NO Coalescing` configuration reaches its maximum throughput at approximately 660K. Including higher QPS values would distort the latency comparison by disproportionately affecting the plot for this scenario. Limiting the range helps maintain consistent visual comparison across all configurations.

4.5 VirtIO-net Tests

While the experiments in Section 4.4 are conducted on a physical machine equipped with a real NIC (refer to Section 4.1.1), this section demonstrates that the `IRQ Suspend` method also performs effectively in a virtualized environment using an emulated NIC. These virtual machine experiments are valuable because they highlight the effectiveness of the `IRQ Suspend` mechanism even in scenarios where only software-generated IRQs are used.

In addition, evaluating a virtualized environment under various test scenarios is particularly important due to the increasing dependence on virtualization within cloud computing. VMs form the foundation of today's cloud infrastructure, enabling scalable services and multi-tenant workloads on shared physical resources [3]. Showing that the `IRQ Suspend` and `Busy Wait` mechanisms remain effective even when only software-generated interrupts are present reinforces their reliability and practical value for real-world cloud applications.

To configure the VM on the hypervisor, 8 virtual CPUs are provisioned and pinned to a single NUMA node to avoid performance inconsistencies from cross-node memory access. Since the guest system is unaware of the underlying hardware interrupts (see Section 2.4.3), IRQ routing does not directly influence packet handling within the VM. However, to demonstrate the effectiveness of the `IRQ Suspend` mechanism under different IRQ routing strategies at the hypervisor level, two distinct configurations are evaluated:

- **Aligned 4 cores**: This setup ensures that the number of hardware interrupt-handling cores on the hypervisor matches the number of VM application cores. Four cores are allocated inside the VM for application processing, and the same four cores on the hypervisor are assigned to handle hardware interrupts (see Appendix B.1).
- **IRQ packing 4 + 2**: In this configuration, two hypervisor cores are dedicated to interrupt handling, while four cores inside the VM are used for application processing. Section 3.1.2 outlines that determining the optimal number of the interrupt-handling cores in IRQ packing is challenging. However, this setup ensures that the two IRQ handling cores are utilized nearly to their full capacity, which is essential to get the full advantages of the IRQ packing (see Appendix B.2).

Each configuration is evaluated under three execution modes: `No Coalescing`, `Busy Wait`, and `IRQ Suspend`. The `Adaptive Coalescing` configuration is excluded because the virtual NIC lacks support for adaptive coalescing and only allows static coalescing. In this setup, the `No Coalescing` mode is configured to trigger an interrupt for every individual packet.

It is important to note that in the VM experiments presented in this section, only 4 cores are assigned to the application within the VM. This choice stems from the observation that allocating all available VM cores to the application can result in under-utilization due to the limitations of bridged networking. Section 2.4.2 shows that several stages of virtual

networking are handled by the hypervisor, which can lead to resource contention, causing the VM's application cores to be starved. Therefore, allocating 4 cores is considered a balanced and reliable choice on the available hardware.

4.5.1 Closed-loop Maximum Throughput

Table 4.2: Throughput Metrics in VirtIO-net, Memcached 4 Cores Aligned

Testcase	QPS	IPC	CPQ	CPU Breakdown (%)		
				User	System	Softirq
No Coalescing	173,615	0.456	42,388	14	58	28
IRQ Suspend	227,455	0.565	34,016	18	80	2
Busy Wait	229,944	0.572	33,871	18	81	1

Table 4.3: Throughput Metrics in VirtIO-net, Memcached 4 + 2 Cores [2]

Testcase	QPS	IPC	CPQ	CPU Breakdown (%)		
				User	System	Softirq
No Coalescing	287,631	0.596	34,301	18	55	27
IRQ Suspend	323,113	0.659	31,662	20	78	2
Busy Wait	325,748	0.664	31,982	20	79	1

Section 4.4.1 explains that the observed throughput improvements are primarily driven by higher IPC and lower Cycles per Query (CPQ), which is reflected in Table 4.2 and 4.3. In a virtualized setup, the CPU must frequently handle `VM_EXIT` events (especially under high-traffic load), which involve transitioning between the guest and the hypervisor. These transitions introduce context switches that would lead to pipeline stalls and reduce instruction throughput, making improvements in IPC even more critical for performance in this environment. Therefore, reducing the number of `VM_EXIT` events improves overall performance. It is important to highlight that this thesis evaluates the effectiveness of `IRQ Suspend` only within the packed and aligned core configurations, without directly comparing the two. Furthermore, the performance results in Table 4.3 are higher than those in Table 4.2, primarily because the former allocates more virtual cores to application processing.

The `Busy Wait` approach eliminates interrupt overhead by continuously polling the NIC queues, which in turn reduces the frequency of `VM_EXIT` events. This leads to fewer context switches and, consequently, improved performance. Similarly, the `IRQ Suspend` mechanism achieves comparable results by remaining in polling mode during periods of high network

[2] The values reported in this table are collected exclusively from the four application cores, excluding any measurements from the IRQ handling cores.

traffic. This performance improvement is supported by the IPC increase. Table 4.2 shows that the IPC for `IRQ Suspend` and `Busy Wait` increases by 26% in comparison to the `No Coalescing` in the Aligned scenario, which shows a similar trend in Table 4.1. The packed scenario (see Table 4.3) shows an IPC improvement of approximately 10%, the smaller gain is attributed to the isolation of hardware interrupts from the application cores, which reduces interference during application execution.

Additionally, both the `IRQ Suspend` and `Busy Wait` configurations lower the proportion of time spent in the SoftIRQ context, both for the packed and aligned scenarios. This indicates a shift in the packet processing path (from the traditional SoftIRQ-based handling to a busy polling model) similar to the behavior described in Section 4.4.1, where packets are processed more directly by the application.

Table 4.3 indicates that the performance gains from `IRQ Suspend` and `Busy Wait` are smaller compared to the aligned configuration in Table 4.2. This is due to the complete separation of hardware interrupt handling from the application (VM) cores, which reduces the number of context switches between the HV and the VM. As a result, the `No Coalescing` configuration performs better in this scenario. Nevertheless, both `IRQ Suspend` and `Busy Wait` still demonstrate benefits, achieving approximately a 12% performance improvement. This improvement is primarily attributed to further minimizing context switches during the application execution.

4.5.2 Open-loop Latency

Building on the discussion in Section 4.4.2, the VM latency results in Figures 4.2b and 4.3b show that both the `IRQ Suspend` and `Busy Wait` reduce 99th percentile latency compared to the `No Coalescing` setup in different loads.

Even within a virtual machine, where interrupt handling and scheduling delays introduced by the hypervisor contribute additional overhead, pure busy polling remains effective in reducing tail latency by eliminating per-packet IRQ processing and SoftIRQ wake-ups. However, under low traffic conditions, the performance of `IRQ Suspend`, `No Coalescing`, and `Busy Wait` configurations is nearly identical. This result is due to the static coalescing software interrupts in the virtual NIC. Virtual NICs do not support adaptive coalescing; therefore, for a small number of packets (in this testbed, 1 packet), an interrupt is triggered. This results in better latency compared to an adaptive strategy. When an interrupt is triggered for one packet, the CPU immediately becomes active and begins polling for packets, resulting in low latency. However, as traffic increases, the frequency of interrupts also rises, disrupting the application's processing. This causes the CPU to switch more frequently between the application thread and the SoftIRQ context, which leads to increased latency.

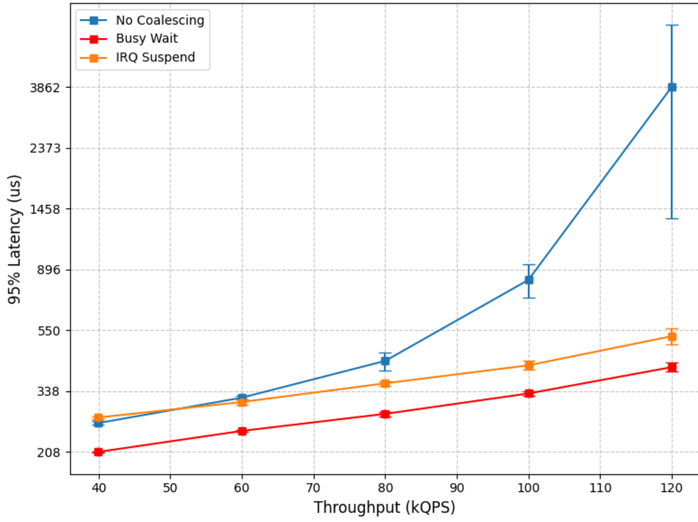
Figure 4.3b illustrates that the `No Coalescing` test case performs more competitively with `IRQ Suspend` and `Busy Wait` across a broader range of throughput levels. This improvement is attributed to the use of IRQ packing, where hardware interrupts are handled

on separate cores from those running the application. As a result, the number of context switches, which increases with rising throughput, is reduced compared to the aligned configuration. In the aligned setup, the virtual CPU must yield to allow the physical core to handle hardware interrupts and then resume execution, introducing additional overhead. By reducing this context switching, the `No Coalescing` configuration under IRQ packing achieves better performance. In addition, the `IRQ Suspend` and `Busy Wait` further reduce the number of context switches, leading to a better response time and latency.

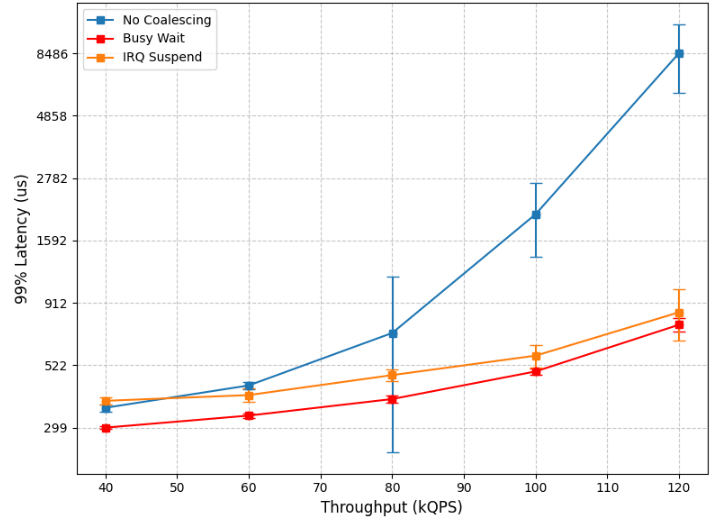
However, as shown in Figures 4.2b and 4.3b, the tail latency observed in the virtual environment is notably higher than that of the physical NIC. In addition, the standard deviation is notably higher for each of the curves. This increase stems from the additional virtualization layer that packets must traverse. Once a packet is routed by the HV, the corresponding virtual CPU needs to be awakened for processing. This extra processing stage, along with the required context switching between the HV and the guest system, adds to the overall latency. Moreover, the latency trend follows a similar pattern to that shown in Figures 4.1a and 4.1b, where latency remains stable under low throughput and rises as throughput grows. However, a key observation is that the magnitude of this increase is significantly greater in the virtualized environment. This can be attributed to the additional virtualization layer introduced by the bridge networking setup (i.e., the software switch). As traffic grows, more packets must be transferred between the hypervisor and the guest through the software switch, leading to increased packet accumulation at various stages of processing and, consequently, higher latency.

4.5.3 CPU Utilization

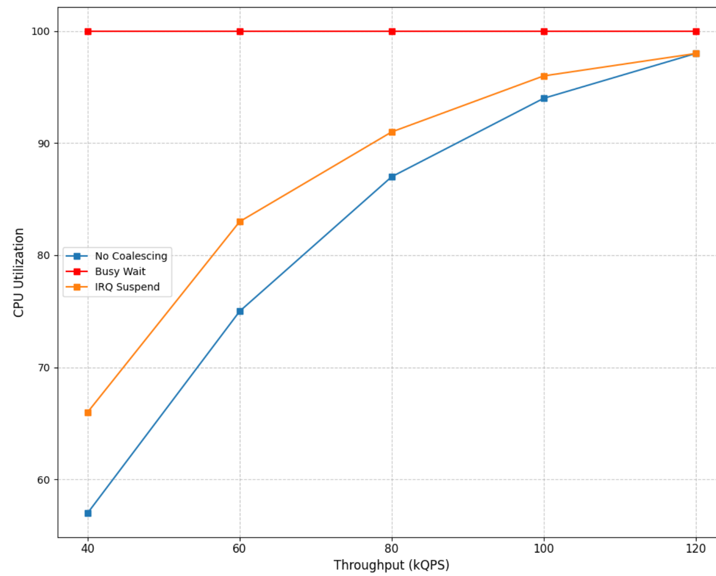
CPU utilization in Figure 4.2c and 4.3c follow the same trend (increase of the CPU utilization) as the physical machine tests (refer to Section 4.4.3) for all test cases. This is because the `No Coalescing` configuration sits on the interrupt to start the packet processing, while the `Busy Wait` does the constant polling loop on the `RX's` queues. In addition, `IRQ Suspend` enables the interrupts whenever necessary based on the application workloads, this leads to a better CPU utilization specially under low-load where the number of packets per interrupt could be low. This behavior shows that enabling interrupts at the right spot can potentially save energy and CPU cycles, even under the virtualized environment.



(a) 95th Percentile Latency



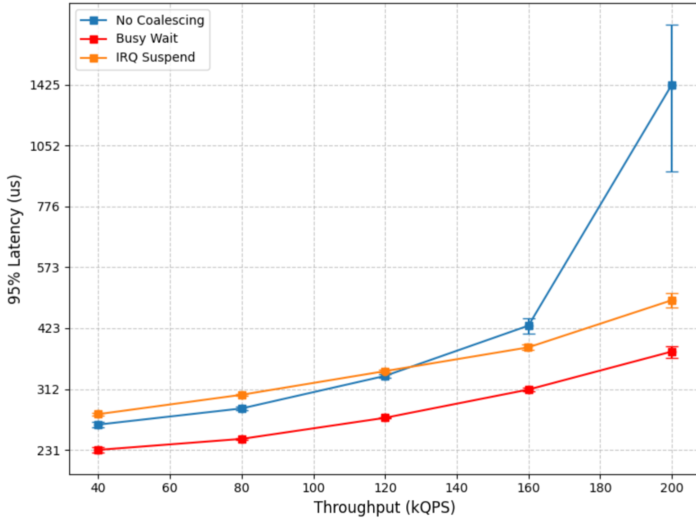
(b) 99th Percentile Latency



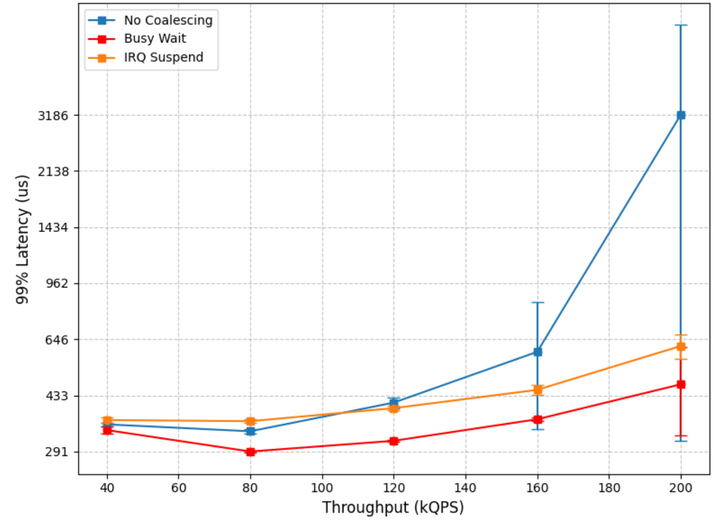
(c) CPU Utilization

Figure 4.2: Latency and CPU Utilization vs Throughput, Memcached 4 Cores Aligned in VirtIO-net Scenario (Lower is Better)[2]

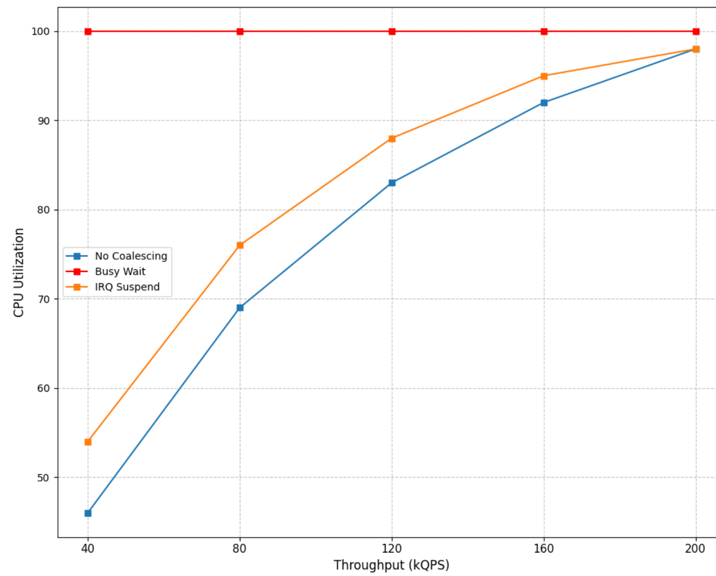
[2]The x-axis is limited to a range of 40K to 120K QPS because the `No Coalescing` configuration hits its peak throughput near 170K QPS, where latency becomes significantly high. Extending the range beyond this point would distort the scale of the plot.



(a) 95th Percentile Latency



(b) 99th Percentile Latency



(c) CPU Utilization

Figure 4.3: Latency and CPU Utilization vs Throughput, Memcached 4 + 2 Cores in VirtIO-net Scenario (Lower is Better)[3]

[3]The x-axis is limited to a range of 40K to 200K QPS because the `No Coalescing` configuration hits its peak throughput near 270K QPS, where latency becomes significantly high. Extending the range beyond this point would distort the scale of the plot. For seeing the trend of the latency, this range is enough and shows how performant the IRQ suspend is against other mechanisms.

4.6 Full Passthrough and Partial Passthrough Tests

Table 4.4: Throughput Metrics in Full Passthrough Scenario, Memcached 8 Cores

Testcase	**QPS**	**IPC**	**CPQ**	**CPU Breakdown (%)**		
				User	**System**	**SoftIRQ**
No Coalescing	425,874	0.524	49,111	18	41	41
Adaptive Coalescing	708,727	0.710	32,358	21	51	28
IRQ Suspend	794,629	0.786	28,957	23	77	0
Busy Wait	808,865	0.791	28,890	22	78	0

Table 4.5: Throughput Metrics in Partial Passthrough Scenario, Memcached 8 Cores

Testcase	**QPS**	**IPC**	**CPQ**	**CPU Breakdown (%)**		
				User	**System**	**SoftIRQ**
No Coalescing	382,133	0.498	49,223	17	40	43
Adaptive Coalescing	685,760	0.697	32,339	21	51	28
IRQ Suspend	789,291	0.788	29,071	23	77	0
Busy Wait	807,070	0.782	29,025	22	78	0

All tests in the previous section use a virtual NIC, where software interrupts are handled by the HV and the guest remains unaware of the hardware interrupts. However, as explained in Section 2.4.6, enabling PCI-passthrough (`Full Passthrough`) and SR-IOV (`Partial Passthrough`) allows for the guest to interact directly with the physical NIC, as if it were running on the HV. This is achieved through the VFIO-PCI driver, which redirects hardware interrupts to the virtual CPUs of the guest, removing the need for HV intervention. As a result, the guest's performance should relatively match that of the HV. A set of experiments, identical to those are described in Section 4.4, have been conducted using `Full Passthrough` and `Partial Passthrough` to validate that the guest achieves comparable performance to the HV and that both `IRQ Suspend` and `Busy Wait` maintain similar effectiveness.

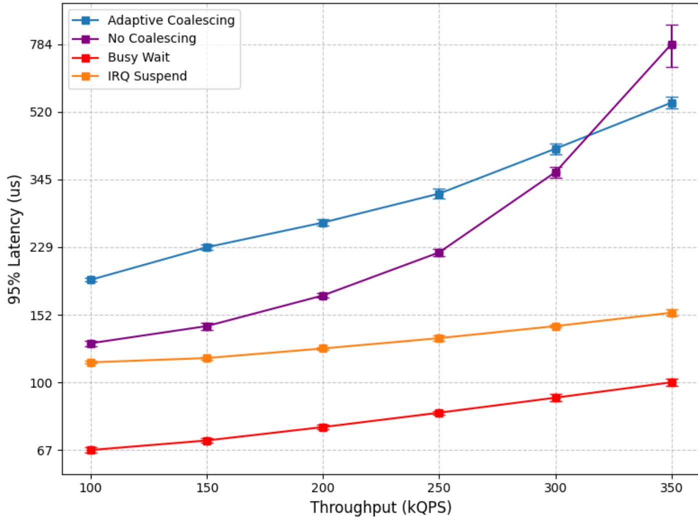
Table 4.4 and 4.5 present the throughput results across various scenarios utilizing `Full Passthrough` and `Partial Passthrough` for virtual networking. A key observation is that both `IRQ Suspend` and `Busy Wait` exhibit a more pronounced performance effect in virtualized environments compared to their impact on the physical machine. On the physical machine, `IRQ Suspend` and `Busy Wait` yield approximately a 6% performance improvement over the `Adaptive Coalescing` setup, and about 45% over the `No Coalescing` setup. In contrast, within the virtualized environment, these improvements increase to roughly 18% and 90-110%, respectively. This can be attributed to how virtual CPU scheduling affects performance, particularly maintaining virtual CPU activity without the HV interference during periods of high network traffic leads to improved throughput and overall efficiency. The VirtIO-net experiments conducted in the previous section attest to this finding as well.

Additionally, the throughput results for `IRQ Suspend` and `Busy Wait` are relatively close to those of the HV (both for `Full Passthrough` and `Partial Passthrough`), approximately 15–20% lower, as shown in Table 4.1 and 4.4. Part of the performance gap is due to the added overhead introduced by the IOMMU, which performs address translation. According to prior studies [45], the DMA addresses generated by the CPU require translation from I/O virtual addresses to the host's physical memory addresses, and this additional translation step introduces performance overhead.

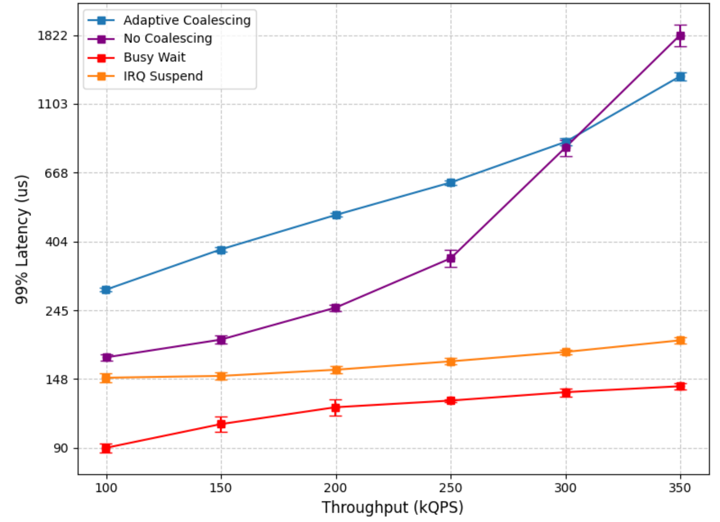
Another notable insight from the performance tables is that interrupt-driven configurations, specifically `No Coalescing` and `Adaptive Coalescing`, exhibit greater performance degradation, approximately around 25-61%, compared to physical machine results. This difference stems from how interrupts are processed, in addition to the other overheads. In a physical machine, the ISR is executed immediately upon NIC-generated interrupts. However, with `Full Passthrough` and `Partial Passthrough`, interrupts must first be rerouted via the VFIO-PCI layer before the virtual CPU can be notified. According to the literature [26], this extra layer of indirection introduces overhead and leads to performance degradation, especially in the `No Coalescing` setup.

Figures 4.4a and 4.4b exhibit the same latency trends observed in Figures 4.1a and 4.1b, with `IRQ Suspend` and `Busy Wait` outperforming the other configurations. At lower traffic loads, `IRQ Suspend` behaves similarly to `No Coalescing`, but as the traffic increases, its behavior aligns more closely with `Busy Wait`. The `Adaptive Coalescing` configuration mirrors the HV results, maintaining higher latency levels compared to the others. This is due to the adaptive coalescing policy of the NIC, which delays interrupt generation until a timer expires or a specified packet threshold is reached.

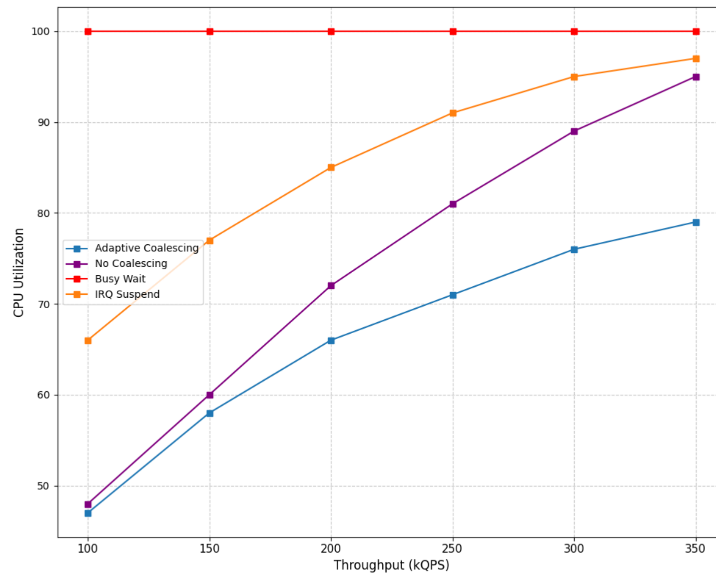
Figures 4.4c and 4.5c indicate that CPU utilization is higher in all test cases compared to the hypervisor-based experiments. One contributing factor to this increased CPU usage is the `KVM halt polling` mechanism [32]. In this mechanism, when a guest virtual CPU issues a halt instruction, the HV does not immediately trigger a `VM_EXIT` and reschedule another thread. Instead, KVM optimistically spins on the halted virtual CPU for a specified duration, polling for wake-up events such as interrupts. If no work is detected within that window, it then falls back to putting the virtual CPU to sleep. This increase of the CPU usage is evident in the VirtIO-net experiments in Figure 4.2c and 4.3c.



(a) 95th Percentile Latency



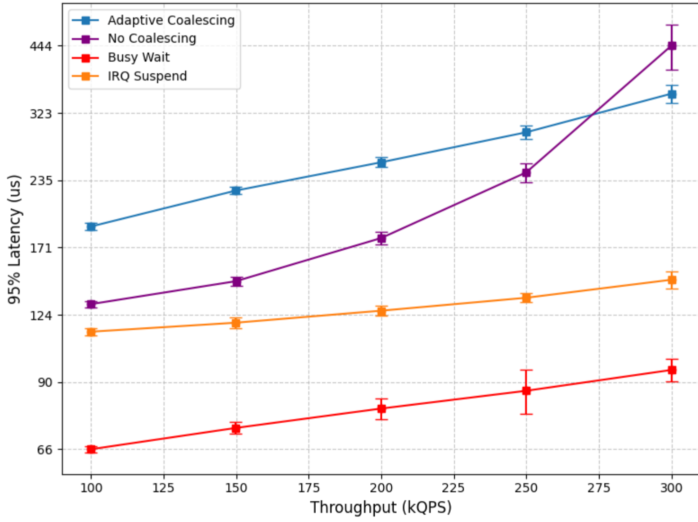
(b) 99th Percentile Latency



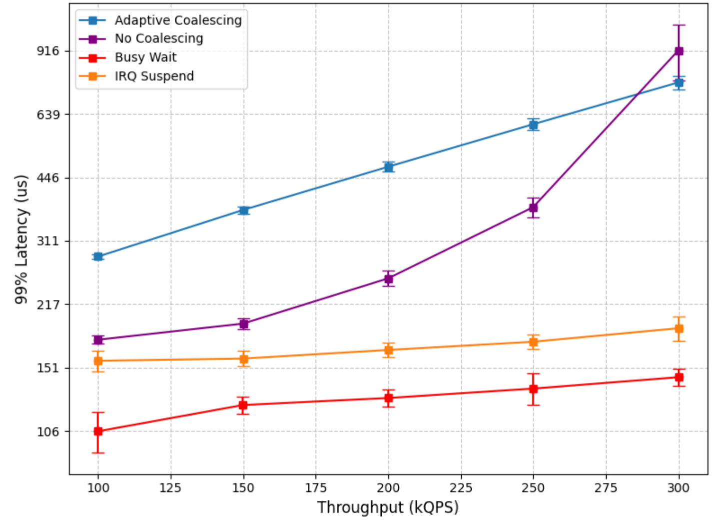
(c) CPU Utilization

Figure 4.4: Latency and CPU Utilization vs Throughput, Memcached 8 cores in Full Passthrough Scenario (Lower is Better)[4]

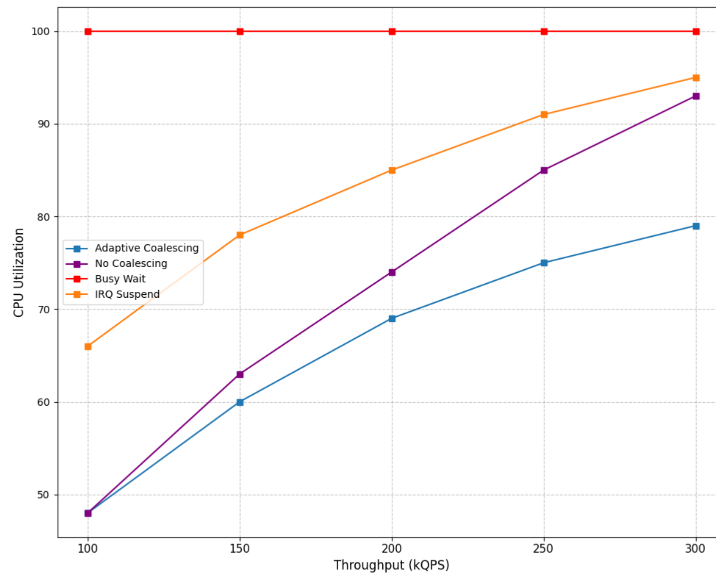
[4]The x-axis is limited to a range of 100K to 350K QPS because the `No Coalescing` configuration hits its peak throughput near 420K QPS, where latency becomes significantly high. Extending the range beyond this point would distort the scale of the plot.



(a) 95th Percentile Latency



(b) 99th Percentile Latency



(c) CPU Utilization

Figure 4.5: Latency and CPU Utilization vs Throughput, Memcached 8 Cores in Partial Passthrough Scenario (Lower is Better)[5]

[5]The x-axis is limited to a range of 100K to 300K QPS because the `No Coalescing` configuration hits its peak throughput near 380K QPS, where latency becomes significantly high. Extending the range beyond this point would distort the scale of the plot.

Chapter 5

Performance Breakdown

The performance results discussed in Chapter 4 indicate that the total number of instructions executed relative to the workload plays a major role in performance variation. The efficiency with which the CPU executes these instructions, reflected by the IPC, is a significant factor. Improvements in IPC generally stem from executing more instructions while reducing overall CPU cycles. The cache hierarchy helps in this regard by lowering the latency involved in fetching instructions from the main memory. This chapter analyzes how cache behavior in various test cases contributes to IPC improvements.

5.1 Cache Hierarchy

In modern processors, each CPU core typically contains a Level-1-cache (L1) cache, which is divided into separate instruction (L1i) and data (L1d) caches. These L1 caches are private to their respective cores, ensuring rapid access to frequently used instructions and data. Following this, each core usually has its own L2 cache, which is unified for both instructions and data, providing a larger storage area while still maintaining relatively low latency. Beyond the L2 cache, the Level-3-cache (L3), often referred to as LLC, is shared among all cores within the processor.

Inclusive and exclusive caching policies define how data is distributed among different levels of the cache hierarchy. In an inclusive design, any data present in a higher-level cache is guaranteed to also exist in the lower levels. In contrast, an exclusive cache ensures that each data block exists in only one cache level at a time; if it is moved to a higher level, it must be invalidated in the lower level. These policies influence the behavior of cache hits and misses, the flow of data between cache levels, and how coherence and overall cache capacity are managed. The hardware used for the experiments reported in this thesis employs an exclusive caching strategy across L1, L2, and an inclusive strategy at the LLC.

5.1.1 Journey of a Cache Line

The L1 cache is the first level accessed when a CPU core issues a memory request. On an L1 hit, the requested data or instruction is retrieved within a few cycles, eliminating the need to consult deeper cache levels. This low latency is due to the L1 cache's small size (typically 32–64 KiB), high associativity, and operating at the core's full clock speed. In the case of an L1 miss, the core allocates a miss-tracking entry and forwards the request to the L2 cache. If the evicted line in L1 is dirty, it must first be written back to L2 before the new cache line can be loaded.

If the L2 cache hits, it returns the full cache line to L1, where it is installed, possibly evicting an existing line if needed, and the requested data or instruction is delivered to the core. In inclusive cache hierarchies, a line present in L1 must also be present in L2, ensuring coherence. If the requested line is not in L2 either, the request proceeds to the LLC, with L2 assigning a miss-tracking entry to track the miss. Dirty evictions from L2 are written either to LLC (in inclusive systems) or to DRAM (in non-inclusive designs).

The LLC handles the final cache lookup. On an LLC hit, the line is returned to L2 and then forwarded to L1 and the core. In inclusive designs, the presence of a line in LLC guarantees that the line either exists, or previously existed, in L2 and L1. If the LLC also misses, the request is forwarded to main memory. Once retrieved, the data is allocated in the LLC, promoted to L2 (evicting lines if necessary), and finally delivered to L1 for use by the core.

5.2 Cache metrics

Section 5.3 explains that the `Perf` tool retrieves hardware performance counters by interfacing with the PMU. In this study, cache hit and miss statistics are gathered from these counters for each cache level, except the L2 cache. Specifically, the counters provide separate counts for data and instruction loads and misses at the L1 level, and aggregated counts for both at the last-level cache. However, accurately tracking data and instruction loads for the L2 cache is challenging. While the PMU offers a broad set of events to monitor L2 cache behavior, determining the exact number of data and instruction loads would require collecting a large number of counters. Since the number of available PMU counters is limited, monitoring too many of them simultaneously can impact application performance, leading to results that do not accurately reflect the application's true behavior. Additionally, this study begins by collecting instruction counts and CPU cycles, followed by separate experimental runs to capture cache-level events for the L1 and LLC. Introducing an extra experimental layer solely for measuring L2 hits and misses would likely add further inaccuracy to the data.

Therefore, L2 cache hit and miss statistics are derived indirectly using the data obtained from L1 and LLC counters, as detailed below:

$$\mathrm{L}_2 \text{ Loads} = L_1 \textit{ Instruction Cache Misses} + L_1 \textit{ Data Cache Misses} \tag{5.1}$$

$$\mathrm{L}_2 \text{ Misses} = L_3 \textit{ Loads} \tag{5.2}$$

There are a few caveats to consider when estimating L2 hits and misses using this method:

- The calculation does not include L2 write or store operations. However, since the benchmark primarily consists of read operations, this omission is acceptable in this specific context.
- Not every L1 miss results in an L2 access. In certain cases, such as data prefetching, data may bypass the L2 entirely.

Additionally, the hit counts for each cache level are computed by subtracting the number of misses from the total loads, since PMU counters do not report cache hits as a separate event. The following formula is used to derive these values at each cache level (L_x):

$$\mathrm{L}_x \text{ Hits} = L_x \textit{ Loads} - L_x \textit{ Misses} \tag{5.3}$$

5.3 Performance Assessment

All experiments in this section are conducted using a closed-loop configuration, where the server operates at full capacity with 100% CPU utilization, ensuring it is fully saturated and unable to process any additional load. Performance is measured using QPS, as described in Section 4.2, with higher QPS values indicating better throughput and improved efficiency. Alongside QPS, the performance table also includes IPC and CPQ. The CPS metric is excluded from analysis, as it remains unchanged across all test cases. This behavior is because the server maintains full CPU usage, meaning CPS equals the CPU frequency (2.6 GHz in this setup) multiplied by the duration of the experiment for each test case.

According to prior work [28], resolving a cache hit in the L1 cache typically consumes between 1 to 4 cycles, while L2 hits take approximately 8-12 cycles. For LLC hits, the latency is approximately 30-40 cycles. Cache miss penalties are approximated based on the latency of hits at the next cache level; for example, a L1 miss is treated as a L2 hit in terms of cycle cost. These numbers are approximated to capture a sense of the cache latencies. However, the actual number of effective stall cycles can vary significantly depending on microarchitectural factors, including pipeline depth, runtime data dependencies, and whether out-of-order execution is able to hide part of the latency.

Because the modern system uses a multilevel cache hierarchy, raw access counts for each cache level alone are insufficient to assess cache hierarchy performance. Instead, a

common metric called Average Memory Access Time (AMAT) [54] is used to evaluate cache efficiency by incorporating the hit and miss rates at each level. As the name implies, AMAT reflects the average time required to access data across the cache hierarchy. The equations below show how AMAT can be calculated in this setup, where 3 levels of cache exist:

$$\begin{aligned}
\mathrm{L_3\ AMAT} &= L_3\ \mathit{Hit\ Time} &+ L_3\ \mathit{Miss\ Rate} * \mathit{DRAM\ Hit\ Time} \\
\mathrm{L_2\ AMAT} &= L_2\ \mathit{Hit\ Time} &+ L_2\ \mathit{Miss\ Rate} * \mathrm{L_3\ AMAT} \\
\mathrm{L_1\ AMAT} &= L_1\ \mathit{Hit\ Time} &+ L_1\ \mathit{Miss\ Rate} * \mathrm{L_2\ AMAT}
\end{aligned} \tag{5.4}$$

From the above equations, the time that is required to answer each memory reference is:

$$\begin{aligned}
\mathrm{L_1\ AMAT} = L_1\ &\mathit{Hit\ Time} \\
&+ L_1\ \mathit{Miss\ Rate} \times \Big(L_2\ \mathit{Hit\ Time} + L_2\ \mathit{Miss\ Rate} \\
&\times \Big(L_3\ \mathit{Hit\ Time} + (L_3\ \mathit{Miss\ Rate} \times \mathit{DRAM\ Hit\ Time})\Big)\Big)
\end{aligned} \tag{5.5}$$

In the above equation, memory references refer specifically to L1 loads, as this study focuses solely on load operations. This simplification is justified by the nature of the workloads presented in Chapter 4, which are predominantly read-only.

Based on the Equation 5.5, to calculate the AMAT for the first-level cache, a hit time is needed. Therefore, this thesis assumes the following latencies for cache hits: L1 cache hits incur 1 cycle, L2 hits require 12 cycles, and LLC hits take 40 cycles. The miss latencies can be ignored since they appear as the next-level cache hits and are irrelevant for calculating AMAT.

5.3.1 Cache Behavior of Test Cases

Table 5.1 and 5.2 show the number of loads, hits, and misses that occurred at each cache level in addition to performance metrics such as IPC, QPS, and CPQ (refer to Section 4.3), which is taken from 4.1. The AMAT column shows the calculated AMAT for each of the cache levels. It is important to emphasize that the values used in the AMAT calculations do not represent the actual latencies for accessing each cache level. Hardware-level optimizations, such as pipelining, instruction-level parallelism, and prefetching, are not captured in these computations. As a result, the conclusions drawn from the cache hierarchy tables should be interpreted as conceptual and relative indicators.

Another important consideration is that the total number of executed instructions alone does not reflect how effectively the processor pipeline is utilized. The IPC metric is a relative metric that inherently includes the effects of the hardware-level optimizations. For example, Table 5.2 reports an IPC of approximately 0.829 for the `Adaptive Coalescing` scenario, implying that the processor executed around 0.829 instructions per cycle. If

Table 5.1: Cache and Performance Comparison: No Coalescing

Group	**Metric**	**Hit Cost**	**No Coalescing**		**Busy Wait**		**IRQ Suspend**	
			Measured	**AMAT**	**Measured**	**AMAT**	**Measured**	**AMAT**
L1 Data	Loads	–	6199	–	5836	–	5838	–
	Hits	1	5661	–	5487	–	5485	–
	Misses	–	538	–	349	–	353	–
L1 Instr.	Loads	–	23535	2.151	22263	1.821	22283	1.822
	Hits	1	21908	–	21238	–	21258	–
	Misses	–	1627	–	1025	–	1025	–
L2 Cache	Loads	–	2165	16.656	1374	17.852	1378	17.893
	Hits	12	1948	–	1208	–	1210	–
	Misses	–	217	–	166	–	168	–
LLC	Loads	–	217	46.452	166	48.434	168	48.333
	Hits	40	203	–	152	–	154	–
	Misses	–	14	–	14	–	14	–
Memory	Hit	100	14	–	14	–	14	–
AMAT × L1 Instr. loads	–	–	50623		40540	–	40599	–
Performance	CPQ	–	35841	–	24850	–	24975	–
	QPS	–	668071	–	963527	–	958737	–
	IPC	–	0.657	–	0.896	–	0.892	–

Table 5.2: Cache and Performance Comparison: Adaptive Coalescing

Group	**Metric**	**Hit Cost**	**Adaptive Coalescing**		**Busy Wait**		**IRQ Suspend**	
			Measured	**AMAT**	**Measured**	**AMAT**	**Measured**	**AMAT**
L1 Data	Loads	–	5764	–	5836	–	5838	–
	Hits	1	5398	–	5487	–	5485	–
	Misses	–	366	–	349	–	353	–
L1 Instr.	Loads	–	21933	1.866	22263	1.821	22283	1.822
	Hits	1	20851	–	21238	–	21258	–
	Misses	–	1082	–	1025	–	1025	–
L2 Cache	Loads	–	1448	17.554	1374	17.852	1378	17.896
	Hits	12	1277	–	1208	–	1210	–
	Misses	–	171	–	166	–	168	–
LLC	Loads	–	171	47.017	166	48.434	168	48.333
	Hits	40	159	–	152	–	154	–
	Misses	–	12	–	14	–	14	–
Memory	Hit	100	12	–	14	–	14	–
AMAT × L1 Instr loads	–	–	40926	–	40540	–	40599	–
Performance	CPQ	–	26451	–	24850	–	24975	–
	QPS	–	905219	–	963527	–	958737	–
	IPC	–	0.829	–	0.896	–	0.892	–

simply fetching an instruction from the L1 instruction cache costs about 1 cycle, this would suggest that decoding, execution, and other instruction path stages impose no cost, which is clearly unrealistic. Hence, absolute performance metrics do not capture the full picture of CPU efficiency. Only relative comparisons across scenarios can reveal the effects of

architectural enhancements and workload characteristics. Furthermore, this study excludes the L1d AMAT calculation, as its impact is relatively minor compared to that of the L1i. When a data cache miss occurs, the CPU can often continue executing other instructions, minimizing pipeline stalls, unlike an instruction cache miss, which would directly disrupt the instruction flow. However, to improve the accuracy of certain calculations, particularly those related to estimating L2 loads, L1d metrics are still included in the presented tables.

In Table 5.1, the L1i's AMAT for `IRQ Suspend` and `Busy Wait` improves by roughly 18% in comparison to `No Coalescing`, while the IPC sees an increase of about 36%. This indicates that enhancements in the cache hierarchy account for nearly half of the total IPC gain. A similar pattern is observed in Table 5.2 for the `Adaptive Coalescing`, where L1i's AMAT improves by 2.5% and IPC rises by approximately 7%. The remaining performance gains likely stem from improvements in the L1 data cache and TLB. The AMAT value suggest that the primary factor contributing to the overall efficiency of the cache hierarchy is the L1 instruction cache. This finding can be supported by examining the corresponding AMAT values at deeper cache levels. While deeper caches such as L2 and LLC show higher access rates (the value of AMAT is higher) under the `IRQ Suspend` and `Busy Wait` configurations compared to `No Coalescing` and `Adaptive Coalescing`, it is the L1 instruction cache that exhibits significantly better efficiency, showing that this improvement is the main source of the performance gain.

A key takeaway from the results is that batching interrupts reduces the number of instruction and memory references. This reduction is evident when comparing the cache behavior of `Adaptive Coalescing` to that of `No Coalescing`. As outlined in Section 4.4, the `No Coalescing` configuration disables adaptive coalescing and relies on static settings that trigger an interrupt after one packet, resulting in more frequent interrupts. In contrast, `Adaptive Coalescing` leverages adaptive settings, allowing the NIC to batch a larger number of packets before generating an interrupt. This batching reduces disruption to the instruction pipeline, as reflected by a higher IPC. Frequent interrupts can hinder instruction execution, causing more pipeline stalls and reducing efficiency. The decline in efficiency becomes evident when examining the number of CPU cycles (CPS row) needed to execute the instructions (L1 Instr. loads row). The CPS metric reflects the total number of cycles consumed, including both useful and stalled cycles. A higher CPS value indicates that more cycles are required to complete the workload. For example, in the `No Coalescing` configuration, the CPS value is approximately 35% higher than in the `Adaptive Coalescing` case, suggesting that each instruction takes significantly longer to execute. It is important to note that the `No Coalescing` configuration executes approximately 7% more instructions than `Adaptive Coalescing`; however, this increase is significantly smaller than the rise in CPS, indicating that the remaining CPS overhead primarily stems from stalled cycles rather than actual instruction execution.

Chapter 6

Conclusion

A hybrid interrupt-management strategy, called `IRQ Suspend`, is evaluated in this thesis, which dynamically alternates between busy polling and interrupt-driven modes for the reception and processing of network packets based on real-time application activity and traffic conditions. The evaluation spans physical hardware, VirtIO-net virtualization, `Full Passthrough`, and `Partial Passthrough` scenarios, and systematic comparisons are made against three approaches: pure interrupt-driven operation (`No Coalescing`), hardware adaptive coalescing (`Adaptive Coalescing`), and continuous polling (`Busy Wait`).

Empirical results indicate that both `IRQ Suspend` and `Busy Wait` outperform interrupt-driven baselines in throughput and tail latency, achieving higher IPC. This improvement is attributed to the spatial and temporal alignment of packet processing with application execution, resulting in better cache locality. Analysis of cache metrics reveals that part of IPC improvements are linked to more efficient cache usage. By keeping packet processing within the application context and minimizing disruptive interrupts, `IRQ Suspend` and `Busy Wait` enhance cache hit rates and reduce average memory access times. While `Busy Wait` maximizes performance, it does so at the cost of continuous CPU usage, even during idle periods. In contrast, `IRQ Suspend` adapts to workload intensity, conserving CPU resources under low load by reverting to interrupt-driven processing and ramping up utilization only when necessary. The results also demonstrate that relying exclusively on hardware-based adaptive coalescing does not yield optimal performance in terms of either latency or throughput. This limitation arises because hardware coalescing operates independently of the application's execution state, meaning that interrupts could potentially interfere with the application's processing.

In virtualized environments, the benefits of `IRQ Suspend` are even more pronounced than on a physical machine. The `IRQ Suspend` approach delivers about a 6% performance boost over adaptive coalescing on the hypervisor and achieves a 45% gain compared to the pure interrupt-driven approach. In virtualized environments, it improves performance by roughly 18% over adaptive coalescing and demonstrates a 90–110% increase in scenarios using `Full Passthrough` and `Partial Passthrough` for fully interrupt-driven configura-

tion. By reducing the frequency of VM_EXIT events and context switches between guest and hypervisor, IRQ Suspend mitigates the additional overhead imposed by software-emulated interrupts and virtual device layers. It is important to note that

This work opens several avenues for future exploration. While the IRQ Suspend mechanism is integrated into the epoll subsystem, the Linux kernel offers other I/O event handling frameworks, such as io_uring [14]. Extending this application-aware interrupt handling technique to alternative subsystems could broaden its applicability and enhance its utility across diverse I/O models. Furthermore, this study focuses on a virtualized environment configured with VirtIO-net and Vhost-net. A valuable extension would be to evaluate the proposed method under different hypervisors and network backends that utilize varying hardware interrupt polling strategies. Additionally, investigating the cache hierarchy in virtualized setups could offer deeper insights into performance behaviors and bottlenecks, helping to further clarify how interrupt handling influences system efficiency.

As shown in this thesis, there is no universally optimal solution for all workloads. A thorough understanding of the trade-offs introduced by different IRQ handling techniques would help guide server tuning and system design for workload-specific performance optimization.

References

[1] Advanced Micro Devices, Inc. AMD I/O Virtualization Technology (IOMMU) Specification. Technical Report 48882-PUB, Advanced Micro Devices, Inc., February 2025. Publication #48882-PUB, Revision 3.10.

[2] Thomas Nadeau Ariel Adam, Amnon Ilan. Introduction to virtio-networking and vhost-net. https://www.redhat.com/en/blog/introduction-virtio-networking-and-vhost-net, 2019. Accessed: April 3, 2025.

[3] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. Technical Report UCB/EECS-2009-28, UC Berkeley Electrical Engineering and Computer Sciences, 2009.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[5] Gaurav Banga, Jeffrey C Mogul, Peter Druschel, et al. A scalable and explicit event delivery mechanism for unix. In *USENIX Annual Technical Conference, General Track*, pages 253–265, 1999.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, New York, NY, USA, 2003. Association for Computing Machinery.

[7] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, pages 41–46, Anaheim, CA, April 2005.

[8] Hao Bi and Zhao-Hun Wang. Dpdk-based Improvement of Packet Forwarding. In *ITM web of Conferences*, volume 7, page 01009. EDP Sciences, 2016.

[9] Eric Blake. Understanding QEMU Devices. https://www.qemu.org/2018/02/09/understanding-qemu-devices/, 2018. Accessed: June 2, 2025.

[10] Peter Cai. Kernel-vs. user-level networking: A ballad of interrupts and how to mitigate them. Master's thesis, University of Waterloo, 2023.

[11] Peter Cai and Martin Karsten. Kernel vs. user-level networking: Don't throw out the stack with the interrupts. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 7(3):1–25, 2023.

[12] X. Chang, J. K. Muppala, Z. Han, and J. Liu. Analysis of interrupt coalescing schemes for receive-livelock problem in gigabit ethernet network hosts. In *2008 IEEE International Conference on Communications*, pages 1835–1839, 2008.

[13] Stack Overflow Contributor. Interrupt Handling for Assigned Device Through VFIO. Stack Overflow, 2015. Accessed: April 24, 2025.

[14] Jonathan Corbet. Ringing In a New Asynchronous I/O API. https://lwn.net/Articles/776703/, 2019. Accessed: June 27, 2025.

[15] Jonathan Corbet. Smarter IRQ Suspension in the Networking Stack. https://lwn.net/Articles/1008399/, 2025. Accessed: June 17, 2025.

[16] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly Media, 3rd edition, 2005. Chapter 7: Time, Delays, and Deferred Work.

[17] Joe Damato. [net-next,v9,0/6]: Suspend irqs during application busy periods. https://patchwork.kernel.org/project/linux-kselftest/cover/20241109050245.191288-1-jdamato@fastly.com, 2024. Patch series cover letter submitted to linux-kselftest mailing list.

[18] Joe Damato. net-next.git: Commit c6aa2a7778d8e3ba7c6f84c8095f0b89f0617830. https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=c6aa2a7778d8e3ba7c6f84c8095f0b89f0617830, 2025. Accessed: June 17, 2025.

[19] Eric Dumazet. [net-next] net: gro: add a per device gro flush timer. https://patchwork.ozlabs.org/project/netdev/patch/1415235320.13896.51.camel@edumazet-glaptop2.roam.corp.google.com/, November 2014. Patch submission to netdev@vger.kernel.org, Patchwork ID 407220, archived.

[20] Eric Dumazet. Busy polling: Past, present, future. In *Netdev Conference*, volume 2, 2017.

[21] Eric Dumazet and Luigi Rizzo. [net-next,1/3] net: napi: add hard irqs deferral feature. https://patches.linaro.org/project/netdev/patch/20200422161329.56026-2-edumazet@google.com/, April 2020. Patch submission to netdev@vger.kernel.org, April 22, 2020.

[22] Louay Gammo, Tim Brecht, Amol Shukla, and David Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Linux Symposium*, volume 1, 2004.

[23] Tal Gilboa. Net DIM - Generic Network Dynamic Interrupt Moderation. https://www.kernel.org/doc/html/v5.8/networking/net_dim.html, 2020. Accessed May 12, 2025.

[24] Sebastien Godard. Sar Linux User's Manual. https://linux.die.net/man/1/sar, February 2025. Accessed: May 14, 2025.

[25] Brendan Gregg. Linux Perf Examples. https://www.brendangregg.com/perf.html, 2024. Accessed: April 29, 2025.

[26] HaiBing Guan, YaoZu Dong, Kun Tian, and Jian Li. Sr-iov based network interrupt-free virtualization with event based polling. *IEEE Journal on Selected Areas in Communications*, 31(12):2596–2609, 2013.

[27] Tom Herbert and Willem de Bruijn. Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt, 2020. Accessed: May 8, 2025.

[28] Johannes Hofmann, Georg Hager, Gerhard Wellein, and Dietmar Fey. An analysis of core-and chip-level architectural features in four generations of intel server processors. In *International Conference on High Performance Computing*, pages 294–314. Springer, 2017.

[29] IBM Corporation. Interrupt coalescing. https://www.ibm.com/docs/en/aix/7.1.0?topic=options-interrupt-coalescing, 2025. Accessed: May 12, 2025.

[30] Intel Corporation. Intel Virtualization Technology for Directed I/O Architecture Specification. Document ID: 774206.

[31] Intel Corporation. Linux Base Driver for Intel(R) Ethernet Network Connection. https://www.kernel.org/doc/Documentation/networking/e1000.txt, 2013. Accessed: May 12, 2025.

[32] Intel Corporation. Kvm tuning guide on xeon-based systems. https://www.intel.com/content/www/us/en/developer/articles/guide/kvm-tuning-guide-on-xeon-based-systems.html, 2023. Accessed: June 8, 2025.

[33] Aamer Jaleel, Joseph Nuzman, Adrian Moga, Simon C Steely, and Joel Emer. High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 343–353. IEEE, 2015.

[34] Jonathan Corbet. Heuristics for software-interrupt processing. https://lwn.net/Articles/925540/, 2023. Accessed: May 15, 2025.

[35] Avi Kivity, Yoav Kamay, Dor Laor, Udi Lublin, and Anthony Liguori. Kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, Ottawa, Canada, July 2007.

[36] Hans J Koch and H Linutronix Gmb. Userspace i/o drivers in a realtime context. In *The 13th Realtime Linux Workshop*, 2011.

[37] Maxim Krasnyansky. Universal TUN/TAP Device Driver. https://docs.kernel.org/networking/tuntap.html. Originally by Maxim Krasnyansky, revised by Florian Thiel.

[38] Jonathan Lemon. Kqueue-a generic and scalable event notification facility. In *USENIX Annual Technical Conference, FREENIX Track*, pages 141–153, 2001.

[39] Jacob Leverich. Mutilate: High-performance Memcached Load Generator. GitHub repository. Accessed: April 25, 2025.

[40] Linux Foundation. Data Plane Development Kit (DPDK). https://github.com/DPDK/dpdk, 2015. Accessed: May 8, 2025.

[41] Linux Kernel Documentation. VFIO - Virtual Function I/O. https://www.kernel.org/doc/Documentation/vfio.txt. Accessed: April 24, 2025.

[42] Linux Kernel Maintainers. Socket(7) — Linux Manual Page. https://man7.org/linux/man-pages/man7/socket.7.html, 2024. Accessed: June 17, 2025.

[43] Linux Kernel Newbies. Kernelbuild - guide to building the linux kernel. https://kernelnewbies.org/KernelBuild, 2021. Accessed: May 5, 2025.

[44] Linux-tools-common Package Maintainers. perf-list(1) - List all symbolic event types. https://manpages.ubuntu.com/manpages/bionic/man1/perf-list.1.html, July 2018. Accessed: May 28, 2025.

[45] Jiuxing Liu. Evaluating standard-based self-virtualizing devices: A performance study on 10 gbe nics with sr-iov support. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–12. IEEE, 2010.

[46] Robert Love. *Linux System Programming: Talking Directly to The Kernel and C Library.* " O'Reilly Media, Inc.", 2013.

[47] Eugenio Pérez Martín. Virtqueues and virtio ring: How the data travels. https://www.redhat.com/en/blog/virtqueues-and-virtio-ring-how-data-travels, 2019. Accessed: June 24, 2025.

[48] Jeffrey C Mogul and Kadangode K Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, 1997.

[49] Gal Motika and Shlomo Weiss. Virtio network paravirtualization driver: Implementation and performance of a de-facto standard. *Computer Standards & Interfaces*, 34(1):36–47, 2012.

[50] Oracle Linux Engineering. irqbalance: Design and Internals. https://blogs.oracle.com/linux/post/irqbalance-design-and-internals, 2023. Accessed: May 8, 2025.

[51] Perf Wiki Contributors. Perf: Linux Profiling With Performance Counters. https://perfwiki.github.io/main/. Accessed: April 29, 2025.

[52] Ravi Prasad, Manish Jain, and Constantinos Dovrolis. Effects of interrupt coalescence on network measurements. In *International Workshop on Passive and Active Network Measurement*, pages 247–256. Springer, 2004.

[53] Proxmox Wiki Contributors. Pci passthrough. Proxmox VE Wiki. Accessed: April 24, 2025.

[54] Rajeev Balasubramonian and Norman Jouppi and Naveen Muralimanohar. *Multi-Core Cache Hierarchies*. Morgan & Claypool Publishers, 2011.

[55] Red Hat, Inc. Enabling Intel VT-x and AMD-V Virtualization Hardware Extensions in BIOS. Troubleshooting section.

[56] Red Hat Inc. How to set up a Network Bridge for Virtual Machines. https://www.redhat.com/en/blog/setup-network-bridge-VM, 2023. Accessed: June 3, 2025.

[57] Red Hat, Inc. irqbalance - Tool Reference. Online manual page, jan 2023. Red Hat Enterprise Linux 6 Performance Tuning Guide.

[58] Red Hat Inc. PCI Device Assignment with SR-IOV Devices. Red Hat Enterprise Linux 7 Virtualization Deployment and Administration Guide, January 2023. Accessed: April 24, 2025.

[59] Red Hat, Inc. Receive-Side Scaling (RSS). https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/6/html/performance_tuning_guide/network-rss, January 2023. Accessed: May 8, 2025.

[60] Red Hat, Inc. Configuring Virtual Machine Network Connections. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/8/html/configuring_and_managing_virtualization, 2024. Accessed: June 5, 2025.

[61] Red Hat, Inc. Deep Dive Into Virtio-networking and Vhost-net. https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net, 3 2024.

[62] Red Hat, Inc. Network Troubleshooting and Performance Tuning. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/10-beta/html/network_troubleshooting_and_performance_tuning/tuning-interrupt-coalescence-settings, 2025. Accessed: April 3, 2025.

[63] Red Hat, Inc. Virtualization Host Configuration and Guest Installation Guide, Red Hat Enterprise Linux 6: Chapter 13 – SR-IOV. https://docs.redhat.com/en/documentation/red_hat_enterprise_linux/6/html/virtualization_host_configuration_and_guest_installation_guide/chap-virtualization_host_configuration_and_guest_installation_guide-sr_iov, 2025. Accessed: June 29, 2025.

[64] Luigi Rizzo, Giuseppe Lettieri, and Vincenzo Maffione. Speeding up packet i/o in virtual machines. In *Architectures for Networking and Communications Systems*, pages 47–58. IEEE, 2013.

[65] Rusty Russell. Virtio: Towards a De-facto Standard for Virtual I/O Devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[66] Jamal Hadi Salim. When NAPI Comes to Town. In *Linux 2005 Conf.* Citeseer, 2005.

[67] Ortiz Sameo. VFIO: Virtual Function I/O Documentation. https://gist.github.com/sameo/1c6131b3fda3c9e5ca1f3229459a9773. Accessed: April 24, 2025.

[68] John Paul Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Science/Engineering/Math, 2004.

[69] Shixuan Zhao. Inject an Interrupt. https://nskernel.gitbook.io/kernel-play-guide/kvm/inject-an-interrupt, 2025. Accessed: June 24, 2025.

[70] James E. Smith and Ravi Nair. The Architecture of Virtual Machines. *Computer*, 38(5):32–38, 2005.

[71] W Richard Stevens and Thomas Narten. Unix network programming. *ACM SIGCOMM Computer Communication Review*, 20(2):8–9, 1990.

[72] The Libvirt Developers. libvirt: The virtualization api. https://libvirt.org/, 2025. Accessed: June 27, 2025.

[73] The Linux Kernel Developers. e1000: Intel® pro/1000 networking driver documentation. https://www.kernel.org/doc/html/latest/networking/e1000.html, 2025. Accessed: June 27, 2025.

[74] The Linux Kernel documentation team. NAPI: The Linux Kernel Networking API. https://docs.kernel.org/networking/napi.html, 2025. Version 6.16.0-rc3 (or see page for version), accessed 23 June 2025.

[75] Linus Torvalds and Linux kernel contributors. Linux kernel source tree: Commit 80b6f094756f. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=80b6f094756f, 2025. Accessed: June 17, 2025.

[76] VMware Inc. Security of the VMware vSphere Hypervisor. Technical white paper, VMware Inc., January 2014.

[77] David A Werden, Sameer Seth, and Ajaykumar Venkatesulu. TCP/IP Architecture, Design, and Implementation in Linux. *ACM SIGSOFT Software Engineering Notes*, 35(5):57–57, 2010.

[78] Matthew Wilcox. I'll do it later: Softirqs, tasklets, bottom halves, task queues, work queues and timers. In *Linux. conf. au*. The University of Western Australia, 2003.

[79] Jing Xie, Xuesong Li, Qingkai Meng, Xunli Fan, Niu Bo, and Fengyuan Ren. Comparing busy poll socket and napi. In *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 318–325. IEEE, 2019.

APPENDICES

Appendix A

Test Cases

A.1 No Coalescing

The configuration below outlines how to reproduce the `No Coalescing` test case. It includes settings for per-NAPI parameters, per-`epoll` instance busy polling, and NIC-level options.

```
# Per-NAPI configuration
napi_defer_hard_irq = 0
gro_flush_timeout = 0
irq_suspend_timeout = 0

# busy polling configuration per-epoll instance
prefer_busy_polling = false
busy_polling_budget = 0
busy_polling_period = 0

# NIC configuration
adaptive-rx = off
adaptive-tx = on
rx-usecs  = 1
rx-frames = 128
tx-usecs  = 8
tx-frames = 128
cqe-mode-rx = off
cqe-mode-tx = off
rx ring size = 1024
tx ring size = 1024
```

A.2 Adaptive Coalscing

The configuration below outlines how to reproduce the `Adaptive Coalescing` test case. It includes settings for per-NAPI parameters, per-`epoll` instance busy polling, and NIC-level options.

```
1  # Per-NAPI configuration
2  napi_defer_hard_irq = 0
3  gro_flush_timeout = 0
4  irq_suspend_timeout = 0
5
6  # busy polling configuration per-epoll instance
7  prefer_busy_polling = false
8  busy_polling_budget = 0
9  busy_polling_period = 0
10
11 # NIC configuration
12 adaptive-rx = on
13 adaptive-tx = on
14 rx-usecs  = 8
15 rx-frames = 128
16 tx-usecs  = 8
17 tx-frames = 128
18 cqe-mode-rx = on
19 cqe-mode-tx = off
20 rx ring size = 1024
21 tx ring size = 1024
```

A.3 Busy Wait

The configuration below outlines how to reproduce the `Busy Wait` test case. It includes settings for per-NAPI parameters, per-`epoll` instance busy polling, and NIC-level options.

```
# Per-NAPI configuration
napi_defer_hard_irq = 100
gro_flush_timeout = 5000000
irq_suspend_timeout = 0

# busy polling configuration per-epoll instance
prefer_busy_polling = true
busy_polling_budget = 64
busy_polling_period = 1000

# NIC configuration
adaptive-rx = on
adaptive-tx = on
rx-usecs  = 8
rx-frames = 128
tx-usecs  = 8
tx-frames = 128
cqe-mode-rx = on
cqe-mode-tx = off
rx ring size = 1024
tx ring size = 1024
```

A.4 IRQ Suspend

The configuration below outlines how to reproduce the `IRQ Suspend` test case. It includes settings for per-NAPI parameters, per-`epoll` instance busy polling, and NIC-level options.

```
# Per-NAPI configuration
napi_defer_hard_irq = 100
gro_flush_timeout = 20000
irq_suspend_timeout = 20000000

# busy polling configuration per-epoll instance
prefer_busy_polling = true
busy_polling_budget = 64
busy_polling_period = 0

# NIC configuration
adaptive-rx = on
adaptive-tx = on
rx-usecs  = 8
rx-frames = 128
tx-usecs  = 8
tx-frames = 128
cqe-mode-rx = on
cqe-mode-tx = off
rx ring size = 1024
tx ring size = 1024
```

Appendix B

Virtual Machine configuration

B.1 Aligned, CPU and NIC Configuration

The following outlines the VM configuration used in the virtualized experiments described in Section 4.5, where four cores were assigned to run the application, and the same set of cores also handled IRQ processing.

```
1  <domain type='kvm' id='3'>
2    <name>vm-ubuntu</name>
3    <uuid>021f52ee-61d9-4317-b19a-c93840571074</uuid>
4
5    <memory unit='KiB'>2097152</memory>
6    <currentMemory unit='KiB'>2097152</currentMemory>
7    <vcpu placement='static' cpuset='0-7'>8</vcpu>
8    <cputune>
9      <vcpupin vcpu='0' cpuset='0'/>
10     <vcpupin vcpu='1' cpuset='1'/>
11     <vcpupin vcpu='2' cpuset='2'/>
12     <vcpupin vcpu='3' cpuset='3'/>
13     <vcpupin vcpu='4' cpuset='4'/>
14     <vcpupin vcpu='5' cpuset='5'/>
15     <vcpupin vcpu='6' cpuset='6'/>
16     <vcpupin vcpu='7' cpuset='7'/>
17     <emulatorpin cpuset='5'/>
18   </cputune>
19   <resource>
20     <partition>/machine</partition>
21   </resource>
22   <features>
```

```
23      <acpi/>
24      <apic/>
25    </features>
26    <cpu mode='host-passthrough' check='none' migratable='on'/>
27    <clock offset='utc'>
28      <timer name='rtc' tickpolicy='catchup'/>
29      <timer name='pit' tickpolicy='delay'/>
30      <timer name='hpet' present='no'/>
31    </clock>
32    <devices>
33      <emulator>/usr/bin/qemu-system-x86_64</emulator>
34      <interface type='bridge'>
35        <mac address='52:54:00:8b:ef:f1'/>
36        <source bridge='br0'/>
37        <target dev='vnet5'/>
38        <model type='virtio'/>
39        <driver name='vhost' queues='8' rx_queue_size='256' tx_queue_size='256'>
40          <host mrg_rxbuf='on'/>
41        </driver>
42        <link state='up'/>
43        <coalesce>
44          <rx>
45            <frames max='7'/>
46          </rx>
47        </coalesce>
48        <alias name='net1'/>
49        <address type='pci' domain='0x0000' bus='0x07' slot='0x00' function='0x0'/>
50      </interface>
51    </devices>
52  </domain>
53
```

B.2 4 + 2 Cores, CPU and NIC Configuration

The configuration below illustrates the IRQ packing setup used in Section 4.5, in which two cores on the hypervisor are dedicated to network processing, while four cores are assigned for application-level tasks.

```
1  <domain type='kvm' id='3'>
2    <name>vm-ubuntu</name>
3    <uuid>021f52ee-61d9-4317-b19a-c93840571074</uuid>
```

```
4
5    <memory unit='KiB'>2097152</memory>
6    <currentMemory unit='KiB'>2097152</currentMemory>
7    <vcpu placement='static' cpuset='0-7'>8</vcpu>
8    <iothreads>2</iothreads>
9    <cputune>
10     <vcpupin vcpu='0' cpuset='0'/>
11     <vcpupin vcpu='1' cpuset='1'/>
12     <vcpupin vcpu='2' cpuset='2'/>
13     <vcpupin vcpu='3' cpuset='3'/>
14     <vcpupin vcpu='4' cpuset='4'/>
15     <vcpupin vcpu='5' cpuset='5'/>
16     <vcpupin vcpu='6' cpuset='6'/>
17     <vcpupin vcpu='7' cpuset='7'/>
18     <emulatorpin cpuset='6'/>
19     <iothreadpin iothread='1' cpuset='4'/>
20     <iothreadpin iothread='2' cpuset='5'/>
21   </cputune>
22   <resource>
23     <partition>/machine</partition>
24   </resource>
25   <features>
26     <acpi/>
27     <apic/>
28   </features>
29   <cpu mode='host-passthrough' check='none' migratable='on'/>
30   <clock offset='utc'>
31     <timer name='rtc' tickpolicy='catchup'/>
32     <timer name='pit' tickpolicy='delay'/>
33     <timer name='hpet' present='no'/>
34   </clock>
35   <devices>
36     <emulator>/usr/bin/qemu-system-x86_64</emulator>
37     <interface type='bridge'>
38       <mac address='52:54:00:8b:ef:f1'/>
39       <source bridge='br0'/>
40       <target dev='vnet5'/>
41       <model type='virtio'/>
42       <driver name='vhost' queues='8' rx_queue_size='256' tx_queue_size='256'>
43         <host mrg_rxbuf='on'/>
44       </driver>
45       <link state='up'/>
```

```
46        <coalesce>
47          <rx>
48            <frames max='7'/>
49          </rx>
50        </coalesce>
51        <alias name='net1'/>
52        <address type='pci' domain='0x0000' bus='0x07' slot='0x00' function='0x0'/>
53      </interface>
54    </devices>
55  </domain>
56
```