Kernel- vs. User-Level Networking: A Ballad of Interrupts and How to Mitigate Them

by

Peter Cai

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, 2023

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

Networking performance has become especially important in the current age with growing demands on services over the Internet. Recent advances in network controllers has exposed bottlenecks in various parts of network processing. User-level networking, which bypasses the operating system's network stack and replaces it with one re-implemented in the userspace, is often framed as a silver bullet to mitigate any performance issues arising in the kernel network stack. However, there is often no comprehensive study on where this performance increase ultimately comes from.

This work aims to explore potential areas from which improvements in overall performance can arise. Most importantly, it is identified that asynchronous interrupts and their handling is a major source of overhead associated with the kernel network stack. Several proposals are presented with the goal of reducing the need for interrupts in the kernel network stack, simulating the execution model of user-level networking. It is shown that a small kernel modification with around 30 lines of code change results in a substantial performance increase without the need to replace the kernel network stack in its entirety.

Acknowledgements

I would like to thank Professor Martin Karsten, my supervisor, without whose acceptance and support this thesis would not have been possible, and Professor Peter Buhr, along with the entire C∀ team for their help.

I would also like to thank everyone with whom I have worked during the program in the lab – Gan Wang, Bryant Curto, Abdul Monum, and Mohammadamin Shafiei. Thank you for your kindness and all the memories we had together.

Dedication

This thesis is dedicated to my parents, without whose near unconditional support while I was young, I would not have chosen the career path into Computer Science, and without whose financial and emotional support, it would have been much more challenging to complete the Master's program.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ABI Application Binary Interface 8

API Application Programming Interface 2, 8, 16

CPT Cycles-per-(unit)-Time 17, 20

DNS Domain Name System 14

DPDK Data Plane Development Kit 6, 8–14, 21

EAL Environment Abstraction Layer 8

eBPF extended Berkeley Packet Filter 10, 52

FD File Descriptor 4

HTTP Hypertext Transfer Protocol 14, 22

IOMMU Input-Output Memory Management Unit 5, 6, 8, 10

IP Internet Protocol 4, 14, 24

IPC Instructions-per-Cycle 17–19, 21–25, 41, 43, 44, 47, 48, 51

IPQ Instructions-per-Query 17–25, 41, 44, 47, 48

IRQ Interrupt Request 4, 26–35, 37–39, 41–46, 48, 51, 52

LKL Linux Kernel Library 11–13

LLC Last-Level Cache 25, 49, 50

NAPI New API 30, 32, 35, 36, 38, 51

NFV Network Function Virtualization 14

NIC Network Interface Controller 2, 4–6, 8, 9, 11, 16, 23, 25, 28–30, 32–36, 38, 44, 45, 51, 52

NUMA Non-Uniform Memory Access 5, 16, 24, 29, 31, 37, 38, 40, 45–49, 51, 52

PMD Poll Mode Driver 6, 8, 10

QPS Queries-per-Second 19, 20, 49

QPT Queries-per-(unit)-Time 17, 19–24, 41, 47

RFS Receive Flow Steering 29

TCP Transmission Control Protocol 4, 6, 18, 24, 49

TLB Translation Look-aside Buffer 25

TLS Thread-Local Storage 12

UDP User Datagram Protocol 4, 6, 52

UML User-Mode Linux 13

VF Virtual Function 5, 8

VFIO Virtual Function I/O 5, 8, 10, 13

VFS Virtual Filesystem 16

XDP eXpress Data Path 10, 52

Chapter 1

Introduction

With growing demand on services and content served through the Internet, networking performance has become an especially popular topic of research. Many modern server applications are inherently I/O-bound, and specifically network-bound, with a majority of their execution time spent in the network stack (see Chapter 3). Therefore, the performance of the network stack often plays a decisive role in the performance of the application. It is thus crucial to understand the performance characteristics and bottlenecks of network stacks in order to improve their processing capability.

Recent literature (see Chapter 2) as well as reports from practitioners [28, 36, 42, 48] often attest to significant performance gains arising from user-level networking in comparison to using the kernel network stack. However, abandoning the kernel network stack and instead re-implementing it at the user level often implies severe limitations on how the application is designed and deployed. For example, some user-level network stacks restrict the use of multi-threading, and most of them require dedicated network and CPU resources to function, at least to some extent. A study on exactly which elements of user-level networking bring about the claimed performance increase would be beneficial in balancing the advantages against limitations of user-level networking, and in improving the performance of the existing kernel network stacks by applying these observations.

While no such comprehensive study exists today, by inspecting popular user-level networking implementations (see Chapter 3), two main aspects of user-level networking can be identified that improve performance, in contrast to kernel network stacks in their default configuration:

1. Customization: The reduced functionality of some user-level stacks directly leads to a corresponding reduction in memory footprint and processing overhead. The removal

of security-related system overheads (kernel-to-user memory copies, security checks, etc.), as an indirect result of customization, also contributes to a performance gain.

2. Alignment: Current user-level network stacks cannot directly receive hardware interrupts and thus use polling to interact with the Network Interface Controller (NIC). This leads to both spatial (core locality) and temporal (application-managed synchronous polling) alignment of network- and application-level processing, both of which are known to improve performance.

A customized network stack presents opportunities for optimization, but comes with significant caveats. As network protocols and their internals continue to evolve, a network stack often must be updated for the best functionality and performance with the rest of the Internet. High-efficiency and low-latency implementations are complex and error-prone, and very extensive testing is often needed to ensure correctness. Even well-known and mature network stacks, such as those found in Linux or *BSD, occasionally suffer from incorrectness. It is thus foreseeable that a custom and less mature implementation increases possibilities for errors. In addition, the Application Programming Interface (API) of a custom stack might need to differ from established APIs to fully realize the customization benefits. This presents challenges in integrating existing applications, especially considering their diversity. Similarly, user-level network stacks are not integrated with the operating system's scheduler and interrupt delivery mechanisms, requiring dedicated resource allocation and bypassing the kernel's resource management system, which potentially reduces overall system efficiency and utilization.

The main conjecture of this work is that better alignment of network- and application-level processing is possible without requiring massive changes or additions to a vanilla Linux system using the regular kernel network stack. It is shown that several simple configuration changes can bring the performance of the Linux kernel network stack much closer to user-level counterparts[1]. However, these configurations on an unchanged kernel often involve significant caveats similar to user-level networking. The core result of this work is a small kernel modification with around 30 lines of code change that replaces these restrictive configuration schemes, enabling the kernel network stack to simulate the execution model of many user-level network stacks. This code change is demonstrated to achieve a significantly higher throughput without compromising tail latency, and without many of the restrictions associated with user-level networking and/or a manually optimized kernel network stack through configurations. An up to 45% performance increase can be observed with the aforementioned modification compared to a vanilla Linux kernel.

[1] When their set of functionalities implemented is similar.

The rest of this work is organized as follows. Chapter 2 presents background and related work. In Chapter 3, the overhead of network processing is analyzed through a series of preliminary experiments on kernel- and user-level network stacks. Chapter 4 introduces various schemes to improve the alignment of the application and the kernel network stack are proposed. Chapter 5 comprehensively evaluates the effects of these schemes. Taken together, these chapters provide strong evidence substantiating the conjecture above. The thesis is concluded in Chapter 6 with an overview of future work with regard to both kernel- and user-level network stacks.

Chapter 2

Background and Related Work

2.1 Kernel- vs. User-Level Networking

Traditionally, most of the network protocol stack resides inside the operating system kernel as part of the basic infrastructure it provides to applications. The kernel typically ingests memory-mapped buffers of network data from the NIC after receiving a notification via an Interrupt Request (IRQ). Then, it passes these buffers through the link (Ethernet), network (IP), and transport (TCP, UDP, etc.) protocol layers. The buffers ultimately become part of a queue called *socket buffer* [7] (one per transport instance), from which the data is made available to applications through standard system calls (`read`, `write`, etc.).

Commensurate with the increase in link transmission speeds and NIC capabilities (now reaching 100s of Gbps), kernels, such as Linux, have been continually improving the performance of their network stacks. These improvements include efforts to streamline the network stack, such as reducing data copying and avoiding interrupts when possible, scaling multiple transmit (TX) and receive (RX) queues to multiple processor cores, and efforts to optimize communication and cooperation between kernel(s) and application(s). For example, early applications rely solely on blocking-based I/O system calls to synchronize with the kernel network stack under a thread-per-connection model, placing enormous pressure on the kernel's scheduler when the number of connections is large. Later, operating systems started to introduce I/O multiplexing based on `select`, `poll` [5] and their successors `epoll` [27] (Linux) / `kqueue` [43] (BSD), enabling one application thread to operate on a much larger set of File Descriptors (FDs), removing this bottleneck. The more recent `io_uring` [6] interface attempts to further reduce overhead by allowing the kernel and the

application to communicate I/O events and requests using a ring buffer, shared directly between the address spaces of the kernel and the application. While its potential for high performance is sometimes eclipsed by its lack of optimization and backwards compatibility, its maturity, if achieved, could mean a significant performance boost for kernel networking.

Nevertheless, the network processing path inside the Linux kernel involves a large amount of asynchronous notification delivery, even after the aforementioned attempts at streamlining the network stack. This complexity is in part due to the need for the kernel to remain generic and agnostic of application behaviour. Even though the kernel attempts to moderate interrupts when possible, they are still needed to drive network processing independent of applications. The receive path of the kernel network stack is executed largely in the interrupt-serving `softirq` context. Packet events are delivered across kernel threads, and sometimes across CPU cores or even Non-Uniform Memory Access (NUMA) domains because the `ksoftirqd` threads are decoupled from application threads.

Resulting from this perceived datedness and inefficiency of the kernel network stack, a recent line of work seeks to abandon the kernel network stack in its entirety. Such work includes library-based network protocol stack implementations that are executed in the context of user-level application processes. Unlike the kernel, however, a userspace application is typically not granted direct access to hardware queues provided by the network controller for security reasons. As a result, they usually leverage specialized provisions made in kernel-mode drivers of NIC allowing ring buffers to be mapped directly into the address space of an application. On the other hand, in the context of cloud computing with virtual machines, there is a need for secure direct access to hardware with minimal host overhead, resulting in hardware-based device virtualisation solutions such as the Input-Output Memory Management Unit (IOMMU) [2] along with the Virtual Function I/O (VFIO) kernel module, which can expose a Virtual Function (VF) of the NIC to a virtual machine. This approach can also be adapted for user-level networking, where an application, instead of a virtual machine, receives a memory mapping controlled by the IOMMU that grants direct access in both directions (application to hardware and vice-versa).

Levels of kernel involvement in network processing can vary even among stacks labelled as user-level. The term *kernel-bypass* is used to denote minimal involvement of the kernel in the data path, where the application takes complete control of at least a subset of hardware RX/TX queues. This design is used for the majority of high-performance user-level network stacks today. As a result, in most of this work, except when clearly indicated otherwise to highlight the differences, the two terms *user-level networking* and *kernel-bypass networking* are used interchangeably.

A side effect of processing network protocol stacks in userspace is that the user-level pro-

cess must often run in a constant polling loop on dedicated CPU cores. This requirement results from the fact that major operating systems available today provide no mechanism for routing interrupts to specific user-level processes. Instead of hardware interrupts themselves, only their *effects*[1] can be made available to applications. As a result, to receive interrupts in a userspace process necessarily means kernel processing, which kernel-bypass networking by definition cannot rely on. Even with IOMMU, an application process, without at least access to a virtualized *ring0*[2], cannot receive interrupts from hardware. This contrast between kernel- and user-level networking is illustrated in Figure 2.1. However, depending on the exact execution model chosen, even with user-level networking, it is possible to designate only a subset of cores to run in polling mode, and to rely on user-level communication for the rest of the application to receive data. Such a model is nevertheless very different from the fundamentally interrupt-based handling in kernel space.

2.2 User-Level Networking Frameworks

Increased interest in user-level, and more specifically, kernel-bypass networking from both academia and industry has resulted in a number of frameworks that abstract away some of the complexity of performing network processing within applications themselves. This section explores Data Plane Development Kit (DPDK) [25], the most popular one based on which there is a plethora of high-performance network stacks, along with a number of its alternatives.

2.2.1 DPDK

DPDK is a prominent framework widely adopted as a basis of kernel-bypass network stacks and applications. As a library, DPDK provides network stacks and applications built on top of it with common infrastructure and abstractions needed by almost all types of user-level networking. For example, DPDK includes various Poll Mode Drivers (PMDs) that interface with different mechanisms for accessing the underlying hardware RX/TX queues. A series of drivers under the umbrella term UIO[3] interface with DPDK-specific provisions made in kernel-mode NIC drivers to allow full control from userspace. The kernel counterparts of these drivers are also developed as part of the DPDK project. Additionally, there are PMD

[1] For example, TCP streams and UDP datagrams exposed after network stack processing.
[2] Kernel privilege level on x86.
[3] This acronym does not have an official full expansion. It can be understood as Userspace I/O.

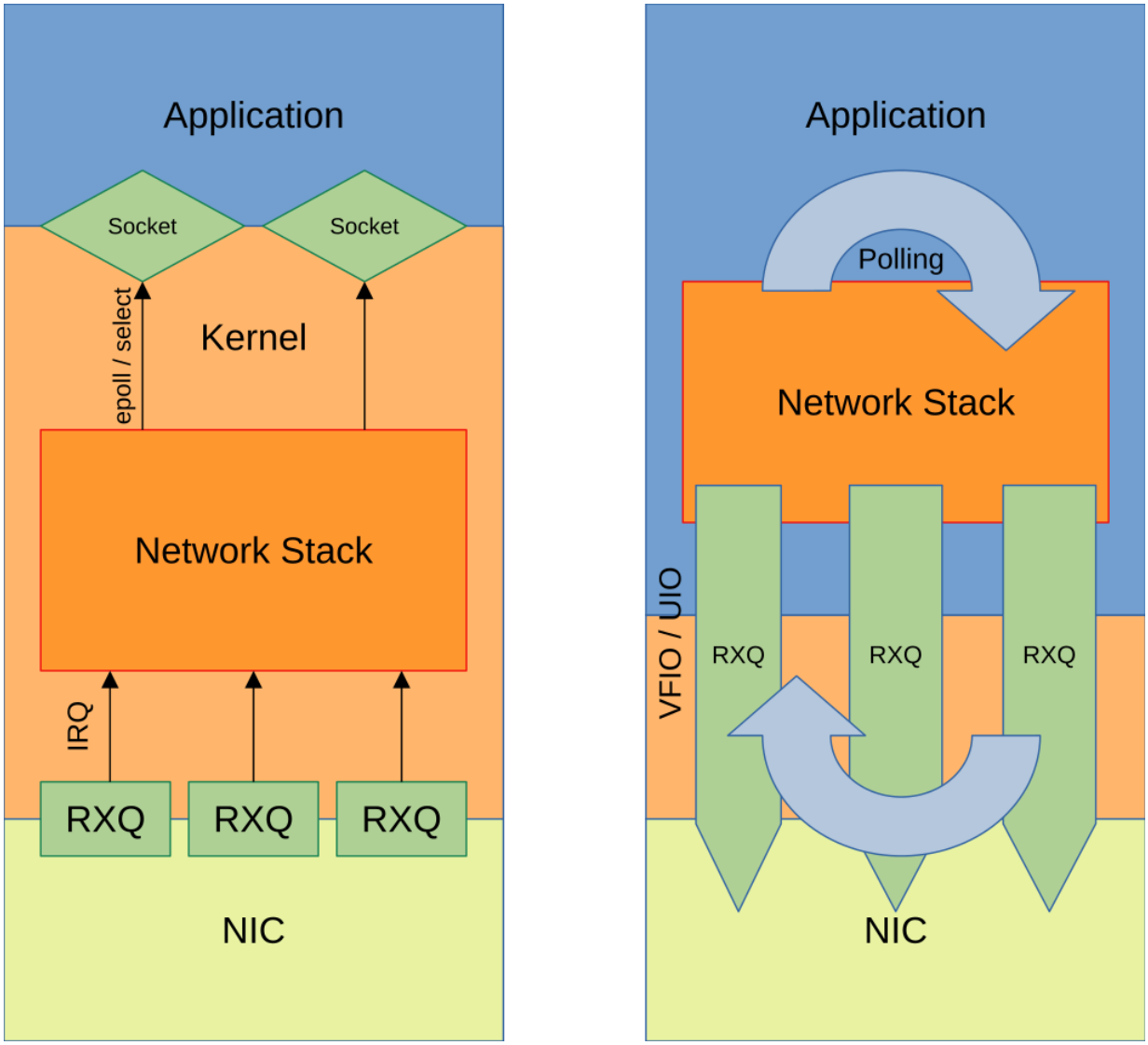

Figure 2.1: Kernel- vs. User-Level Networking

drivers for VFIO, which work with all NICs with virtual functions and IOMMU support without model- or vendor-specific adaptations, and newer partial (but more generic) kernel-bypass mechanisms such as `AF_XDP` (see Section 2.2.3). In some cases, kernel drivers of NICs, such as ones from Mellanox, support coexisting with DPDK and exposing part of the data path (e.g. a VF) to DPDK without IOMMU and VFIO (*bifurcated* drivers).

All PMDs expose a common interface to the rest of DPDK and applications built on top, enabling application code to be agnostic over the specific kernel-bypass mechanism in use. As the name may suggest, most if not all PMDs require polling-mode execution. A polling loop is managed by DPDK on each available core that drives execution of the application. The polling loop scans the RX rings for traffic, and if the result is non-empty, it then hands over control to the application for processing. Applications must, in response, return to the polling loop once any handling logic is complete. In contrast to the default kernel network stack, there is near-perfect coordination between the application and the network stack when running under this polling loop. This relationship comes at the cost of dedicated resource allocation, since typical DPDK applications must be assigned an exclusive set of cores in order to be constantly polling.

When generic kernel-bypass mechanisms cannot be used, DPDK becomes bound to specific ranges of Linux kernel versions. The nature of UIO necessitates its existence (partially) in kernel space, and since Linux provides no stable in-kernel Application Binary Interface (ABI) and API [22], there is no guarantee that a UIO driver developed for an older kernel even compiles on a newer kernel. In practice, even a minor version increment almost always introduces some ABI and API incompatibility. As a result, DPDK applications often require a more extensive update procedure than typical userspace applications, to which the in-kernel ABI and API is usually of minimal concern.

Beyond basic driver abstractions and management of the main polling loop, DPDK comes with various additional libraries to ease the development of kernel-bypass network stacks. The Environment Abstraction Layer (EAL) hides platform-specific details on memory mapping, memory allocation and thread creation. The *mbuf* and *mempool* libraries provide abstractions for memory allocation and deallocation from a pool shared by the application and the NIC to form RX/TX queues. The *Ring* library supplies the application with a lock-free and bounded implementation of queues to be used as the ring buffer. Over time, the collection of libraries included with DPDK has grown significantly, removing much of the development work from both the network stack on top and the application.

It is important to note that DPDK, as the name *Data Plane* implies, was originally developed (and still very often used) for implementing fast-path software-defined data planes on software routers and switches. However, due to its wide coverage of abstractions,

it has since grown far beyond simple raw packet processing and become the foundation of high-performance user-level network stacks.

2.2.2 PF_RING and PacketShader

While DPDK is the most mature and popular framework for both user-level packet processing and full network stack implementations, other frameworks and mechanisms do exist to enable userspace customization of network processing. This section documents two examples of DPDK alternatives, namely PF_RING and PacketShader. Note that even though all of these frameworks enable user-level packet processing or at least some level of customization, not all of them constitute as full kernel-bypass networking. While it would be intriguing to investigate full network stacks on top of these alternatives without full kernel-bypass to contrast with those based on DPDK, it is unfortunately the case that full network stacks realized on top of them with at least comparable performance claims are not common.

PF_RING [30] is a Linux kernel module for fast-path packet capturing, aiming to replace traditional solutions such as raw sockets and *libpcap* [46]. It extends the kernel with a new socket type, with which applications can receive raw Ethernet frames copied directly into userspace memory. Buffers used by applications are recycled similar to those utilized for NIC rings, improving efficiency. PF_RING can also perform preliminary parsing of some higher-level protocol headers, reducing the amount of code required for the application. In its *zero-copy* mode, PF_RING can achieve kernel-bypass networking, where rings used by NICs are directly mapped into userspace instead of being copied into a separate ring, thus kernel processing is almost entirely bypassed. Under this operating mode, PF_RING also requires driver- or hardware-specific provisions and polling-mode execution, similar to DPDK.

PacketShader [15] is a different approach to networking than both kernel- and other user-level networking solutions. While user-level networking solutions employ different strategies for data ingestion, they nevertheless execute on the CPU, in contrast with the parallelized nature of GPUs. Based on the observation that many routing and switching workloads are highly parallel in nature, the authors of PacketShader proposed that the GPU can be used instead to perform any transformation on the data, hence the name (Shader referring to a type of GPU program). To implement such a processing path, the authors produced their own engine for user-level packet processing. However, in newer versions of PacketShader, this I/O engine is abandoned in favor of DPDK.

2.2.3 eXpress Data Path (XDP)

As user-level (kernel-bypass) networking grows in popularity, developers of the Linux kernel are also exploring new approaches to potentially enable more customizability in the kernel network stack. One such approach is the eXpress Data Path (XDP) [16]. Instead of permitting userspace applications access to hardware queues, XDP employs the extended Berkeley Packet Filter (eBPF) [9] subsystem in the Linux kernel, providing userspace with injection points at the very start of the kernel network stack, immediately after network data is ingested into a kernel driver. Albeit incomplete, this approach bypasses most processing normally performed in the kernel stack, and instead delegates it to an eBPF program injected by an application. The possibility of inserting code into the kernel enables applications to customize the kernel network stack without excessive performance penalties associated with packet capturing. Before XDP, such performance was only possible using full kernel-bypass frameworks like DPDK. Note that XDP itself only addresses the first aspect of userspace networking mentioned in Chapter 1, customization.

Unfortunately, in order to allow programs submitted by userspace to be executed directly in the kernel, trade-offs have to be made in the eBPF virtual machine. The virtual machine in the Linux kernel carries out extensive security checks on all eBPF programs. Most importantly, such checks include a maximum cap on the time and space complexity of the code, which means that, at the very least, variable-dependent loops are forbidden. In other words, the eBPF virtual machine is not *Turing Complete*. This limits its computational capability. Consequently, XDP with eBPF is unsuitable for implementing a complete alternative network stack in the kernel.

Recent Linux versions have introduced a workaround for this limitation with a new socket type, `AF_XDP`, that redirects network traffic to applications, leveraging the same infrastructure of XDP. This mode is conceptually similar to the custom socket type implemented by PF_RING (see Section 2.2.2). `AF_XDP` is also supported by DPDK as a new PMD backend, making it possible for DPDK to run without full kernel bypass, and without VFIO/IOMMU or hardware support. However, the performance implication of this PMD backend on DPDK applications is unclear. To avoid unknown effects, this work uses native DPDK drivers instead of `AF_XDP` at this time.

2.3 User-Level Network Stacks

Since its release, DPDK has seen wide adoption in industry to realize user-level networking. In this section, two types of stacks based on DPDK – userspace ports of kernel stacks like

F-Stack / Linux Kernel Library (LKL), and complete re-implementations of network stacks such as Shenango / Caladan – are introduced as the focus of comparison for this work. Both provide relevant points of reference for different reasons.

Other user-level networking approaches exist, with Onload [50] being an example based on `AF_XDP` or `ef_vi`, a kernel-bypass mechanism specific to Xilinx NICs. Seastar [39], an asynchronous programming framework for C++, includes a network stack primarily targeting DPDK. mTCP [20] can switch among a number of user-level networking backends, including DPDK and PacketShader's I/O engine. Some library implementations of network protocol stacks are agnostic of any underlying packet ingestion framework, lwIP [41] being a prominent example. However, none listed above provide or claim levels of source compatibility, performance, or maturity that eclipse those described below.

2.3.1 F-Stack and Linux Kernel Library (LKL)

As alluded to in Chapter 1, maintaining a new network stack written from scratch is non-trivial and often bug-prone. However, to achieve the benefits of user-level networking, a network stack must be able to execute in user mode, which is not the case for most mature network stacks, as they are developed as part of operating system kernels. One workaround, however, is to provide shims for kernel services they depend on, such that these in-kernel network stacks can be executed in userspace, on top of user-level networking frameworks such as DPDK. F-Stack [47] and the LKL [26] are two prominent examples of this approach.

F-Stack is a port of the FreeBSD network stack to DPDK. The project includes the complete source code of the FreeBSD kernel, but replaces all functions pertaining to multi-threading, synchronization, and device drivers with empty stubs. Instead, the network stack, along with any application code, is executed within the main polling loop managed by DPDK. When network data is retrieved by the polling loop, it is emitted through the FreeBSD network stack, now with all system dependencies removed or replaced. This adaptation is facilitated through a virtual ethernet device (`veth`) registered with the FreeBSD network stack. After processing, an application-supplied callback will be invoked by F-Stack within the same synchronous execution path, from which application logic can be performed.

Although F-Stack advertises inclusion of the `kqueue` [43] system-call interface (but relegated to being a function instead of a system call) and a compatibility layer for `epoll`, they are ultimately only used for event retrieval in the application within the callback supplied to F-Stack. FreeBSD's original event notification mechanisms depend on the scheduler, which has been removed as part of the porting process to userspace. Neither

does F-Stack support multiple thread contexts for the event notification mechanism to function, as the porting effort has removed all multi-threading synchronization primitives. A program on top of F-Stack operates in the same synchronous execution model as any DPDK program, alternating between polling, network stack and application processing, limiting them to one single thread. To make use of multi-core systems, the application must be designed to spawn multiple processes without a shared address space as opposed to threads, such that a separate instance of F-Stack can be launched within each process.

It would be tempting to explore whether F-Stack can be extended to support true multi-threading. However, two major issues prevent this from being done as part of this work.

- Singletons. One simple idea of supporting multi-threading is to allow spawning multiple instances of F-Stack within one address space. However, as an operating system kernel, the source code of FreeBSD is littered with global variables that are initialized once for each address space. Large portions of its code depend on the existence of these variables. To remove them is to refactor all the functions to take a context parameter or use Thread-Local Storage (TLS) instead of assuming the existence of global singletons, requiring extensive code changes. In addition, such a "multi-threading" model does not eliminate most of the limitations as seen in the current multi-process operating mode, as each instance is bound to one system thread, and application logic is still limited to executing as a callback of the polling loop run by each instance.

- Performance. Another potential approach is to enable one instance of F-Stack to support multiple application threads. Or, even in the multi-instance multi-threading case laid out above, in order to decouple F-Stack instances from application threads, each instance also effectively needs to communicate with multiple application threads. This necessitates some type of userspace inter-thread/process communication, which can often lead to a heavy performance overhead. The inter-process signalling mechanism used by Shenango / Caladan (see Section 2.3.2) is tightly coupled to their user-level threading runtime, making it not applicable to F-Stack either.

To summarize, comparing against a customized F-Stack with multi-threading support cannot be considered fair in this study, both for other stacks and for F-Stack itself. In this work, any comparison against F-Stack is either performed in single-threaded mode, or with an application designed for operating as multiple processes.

On the other hand, LKL does not share this limitation of F-Stack. Unlike F-Stack, LKL is a *Library Operating System*, rather than merely a network stack. It is a complete port of

the Linux kernel to userspace, not unlike User-Mode Linux (UML) [8], though it does not rely on system call hijacking thanks to its integration with application code. LKL not only includes the network stack but also file systems, partial multi-threading, and even some device drivers through VFIO. A virtual Ethernet driver is provided as part of the network stack that employs DPDK for host access. Unfortunately, in order to satisfy in-kernel synchronization constraints, LKL implements a global lock that only allows "kernel" code to be executed on one single thread at any moment in time. This locking effectively limits network processing power to that of a single core. While the application can use multiple threads, this global lock presents a severe performance bottleneck.

2.3.2 Shenango and Caladan

In addition to industrial applications, DPDK and user-level networking enjoy increasing popularity in the research literature. Shenango [32], and its successor Caladan [12], are recent user-level network stacks built on DPDK. Both add significantly to the default behaviour of DPDK – instead of dedicating all cores to one application for polling, in both Shenango and Caladan, only one core is needed to run the DPDK polling loop as their `iokerneld` process. To allow application threads to sleep while idling, their work includes a fast-path inter-process signal delivery mechanism exposed through a custom kernel module, `ksched`. One polling `iokerneld` process can support multiple applications through this notification mechanism. A custom user-level threading runtime is provided as part of the application runtime, such that `iokerneld` can make scheduling decisions through `ksched` and through all application runtimes.

The majority of the network stack, other than the polling loop in `iokerneld`, runs in the application, similar to other DPDK-based network stacks. Each application receives data with its own polling loop called `softirq`, which effectively functions as a separate user-level thread in their custom threading runtime. Despite similar naming, unlike the `softirq` context in the Linux kernel, the `softirq` thread in the Shanango / Caladan runtime is unable to directly preempt a running application thread. Polling can only be performed when application code yields to the runtime, resulting in a similar execution model as other DPDK applications. The difference is that kernel threads in Shenango / Caladan can be put to sleep while idle-looping, relying on `iokerneld` to wake them up when events of interest arrive.

One important aspect to point out is that the Caladan / Shenango work mainly focuses on scheduling and its evaluation. Nevertheless, it is built upon the assumption that user-level network stacks have inherent performance advantages and presents extraordinary

performance gains over vanilla Linux. In this work, Caladan is used without interfering background applications such that the scheduler is minimally involved.

2.4 Applications of User-Level Networking

User-level network stacks are often targeted at high-performance server applications. F-Stack includes both *Nginx* [29], a well-known and popular HTTP server, and *Redis* [35], an in-memory database engine, in their repository as sample use-cases. The authors also claim to have deployed F-Stack as the network stack for their Domain Name System (DNS) service. Both Shenango and Caladan include a port of *Memcached* [10], which is a widely deployed in-memory key-value store often used as a caching daemon for web servers. Seastar's kernel-bypass networking optionally powers *ScyllaDB* [38], a real-time database engine developed by the same authors. In all of these cases, the application requires a complete Layer 4[4] protocol stack.

On the other hand, user-level processing of network packets is also a popular technique in software routing and switching. In this scenario, user-level processing frameworks such as DPDK or `AF_XDP` are used only as a method for efficient packet capturing, and network traffic does not terminate at the router or switch. The application includes minimal Layer 4 processing, if any, and most logic operates directly on raw Ethernet frames or IP packets. For example, NetVM [17] leverages the flexibility of DPDK's user-level processing to implement Network Function Virtualization (NFV). Rubik [24] seeks to simplify the programming of network middleboxes by designing a new special-purpose programming language that targets DPDK as its packet processing infrastructure. When hardware offloading is involved, with or without application-level networking, DPDK-based solutions are also often used as state-of-the-art "best-case scenario" for software processing [33, 18].

It is important to note that this work focuses on the server application aspect of user-level, specifically kernel-bypass networking. While findings presented may also apply to other scenarios, for example, for middleboxes that parse application-level traffic before forwarding, they are not evaluated as part of this work.

[4] Transport Layer

Chapter 3

Network Processing Overhead

This chapter attempts to assess the performance overhead associated with system services, and in case of user-level networking, the network stack and user-level runtimes. In selecting experiments and network stacks for the purpose of this chapter, one important concern is the need to distinguish between (potential) performance improvements arising from two sources:

- user-level, synchronous execution; and
- customization of the network protocol stack.

In practice, most user-level stacks, as discussed in Chapter 2, implement both improvements at the same time. To facilitate studying this distinction, F-Stack (see Section 2.3.1) and Caladan (see Section 2.3.2) are chosen as representative examples of user-level network stacks: F-Stack serves as an example of a production-proven full-featured kernel network stack repurposed for userspace, while Caladan is a fully customized user-level network stack written from scratch claiming outstanding performance. By contrasting these two stacks against each other and the vanilla kernel as a baseline, insight can be gained into which part of networking overhead is reduced or eliminated from the processing path through their respective user-level networking approaches.

Memcached and Nginx are used as application workloads on top of different network stacks. These two applications are commonly included as benchmark use-cases in user-level network stacks (see Section 2.4). They are widely deployed in production, yet simple enough such that performance characteristics of the underlying network stacks dominate

in benchmarks. Assessments in this chapter are performed using a server with 2 Xeon E5-2680 8-core processors in a NUMA configuration. Note that the NUMA boundary is never involved in benchmarks unless otherwise specified. Load generation is handled by a separate set of 7 servers. This experimental configuration is identical to that used in Chapter 5, where a more in-depth description can be found.

3.1 Methods

In order to study networking overheads, it can be tempting to attempt a detailed breakdown of the execution of each network stack. Unfortunately, such a breakdown can often be difficult and unreliable. As an example, the kernel network stack is tightly integrated with other subsystems. At the very least, it relies on the memory management subsystem for NIC and socket buffers, the Virtual Filesystem (VFS) subsystem for I/O system calls, and the scheduling subsystem for event notification. Which part of the overhead can be said to originate from the network stack is not well-defined, and modifications in the network stack may in reality affect execution of other subsystems in a non-negligible way. In addition, the kernel by definition also requires system calls to interact with userspace applications, the effects of which may not be local to the network stack either.

On the other hand, however, the execution path of any program consists of application code and system services (or runtime services, in case of user-level networking). The breakdown between application and system code is clear and should be less problematic to obtain. Changes in network stacks, while maintaining a similar API, is expected to mainly affect the system portion rather than the application itself, assuming minimal adaptation in application code. In a slight generalization of Amdahl's Law [3], one could ask how the affected portion (system) can lead to the overall performance improvements. In other words, assuming an unchanged application, what portion of its total execution overhead is attributable to system services and how much of this overhead can be eliminated?

3.1.1 Performance Model

To assess the performance of a network stack before and after a certain change, the following metrics are proposed to aid in this task. Any change in overall efficiency must result in changes in one or more of the following global metrics along with their respective application-system breakdown.

The application throughput performance, Queries-per-(unit)-Time (QPT), can be measured as

$$QPT = queries/time \tag{3.1}$$

The average CPU resource utilization, Cycles-per-(unit)-Time (CPT), is given by

$$CPT = cycles/time \tag{3.2}$$

On superscalar processors [40] with deep pipelines, Instructions-per-Cycle (IPC) is a well-established metric describing how efficient the pipeline can process a particular workload. Additionally, Instructions-per-Query (IPQ) is proposed here to capture how many instructions are executed for each application-level query. This metric is used as an estimate for functionality, i.e., the amount of processing that is done for each application query on average:

$$IPC = instructions/cycle \tag{3.3}$$

$$IPQ = instructions/query \tag{3.4}$$

It is not difficult to see that

$$QPT = \frac{CPT \times IPC}{IPQ} \Leftrightarrow \frac{QPT}{CPT} = \frac{IPC}{IPQ}. \tag{3.5}$$

Fundamentally, this model illustrates that an increase in performance or efficiency (QPT/CPT) must be accompanied by an increase in IPC or a decrease in IPQ or both, regardless of what types of overhead are involved. A decrease in IPQ could be caused by an algorithmic improvement or reduced functionality. In the case of user-level network stacks, assuming that there are no fundamental changes in the design of algorithms and data structures in the protocol stack, a dramatic decrease in IPQ most likely results from a reduction in functionality[1].

It is important to note that some overheads associated with operating systems, such as system calls, can also contribute to IPQ. For example, memory copies between userspace and the kernel require at least a number of instructions of the order $O(N)$, where N denotes the length of the buffers. However, such an overhead is often also experienced

[1]Referring to major protocol features that involve processing in the hot path, e.g. congestion control.

by user-level network stacks, especially in the TCP stack, where packets must be copied into a continuous buffer to satisfy TCP's stream semantics [11]. Other overheads, such as context switches, should show minimal direct impact on IPQ due to their comparatively low asymptotic complexity in terms of the number of instructions needed. Overall, IPQ should be a good approximation for functionality for user-level network stacks. While small changes in IPQ can occur due to other effects, any major difference found in IPQ would be a strong indication of significant changes in functionality. This expectation is further studied in the rest of this chapter by comparing against F-Stack, whose functionality should be similar to that of the Linux kernel; and Caladan, whose network stack is rewritten from scratch and may not share the same level of maturity and completeness.

On the other hand, an IPC increase typically means fewer stalled cycles due to improved efficiency of the processor pipeline, caused by increased cache hit rates (including for page translation and branch prediction) or similar effects. Any indirect overheads should manifest as a change in IPC, because by definition their "indirect" effects can only mean those on the processing pipeline and not on the code path itself.

3.1.2 Performance Data Collection

To collect performance data in order to compute metrics proposed in Section 3.1.1, the `perf` [44] tool, included as part of the Linux kernel, is used to monitor application behaviour during a number of representative experiments.

`perf` is composed of two main operating modes as two sub-commands: `perf stat`, and `perf record`. In `perf stat`, the tool monitors aggregate statistics exposed through hardware or software performance counters. These counters are usually managed by either the CPU or the kernel, adding very minimal performance overhead since the userspace tool does not need to perform much computation while the observed program is running. Two of the performance counters are relevant to this chapter:

- `instructions`: The number of instructions executed in the context of the monitored process(es) or CPU(s). This counter can be reported separately for kernel- and user-space (named `instructions:u` and `instructions:k` respectively).
- `cycles`: The number of cycles spent in the context of the monitored process(es) or CPU(s). Similar to `instructions`, this value can separately report kernel- and user-space execution (`cycles:u`, `cycles:k`).

Along with the Queries-per-Second (QPS) typically reported by benchmark load-generation tools, which equals QPT where $T = 1s$, these metrics are enough to separately compute all metrics mentioned in Section 3.1.1 for the kernel and the application. However, as user-level network stacks exist within the context of the application, the user-kernel separation provided by `perf stat` cannot distinguish between them and the application(s).

To further break down userspace execution, it is necessary to make use of `perf record`. This mode generates a record comprised of statistical sampling taken at a given period / frequency, with regard to a user-specified type of event. For example, when invoked with `perf record -c 1000 -e cycles`, the tool emits a sample in the record log every 1000 cycles, recording which function is executing at that time. `perf record` can also take samples at a frequency with regard to time instead for better consistency (with the `-F` parameter), in which case the kernel dynamically adjusts the period of sampling to approximate the specified frequency. With enough time and a sufficient number of samples, this tool provides an estimate of how many cycles, instructions, or any other type of events each function generates. Because user-level network stacks examined in this chapter are static libraries, and because they are open-source and compiled binaries can include debug symbols[2], execution in user-level network stacks can be approximately distinguished from the rest of the application by filtering on their known symbols. With reports emitted by `perf record`, the ratio of cycles or instructions can be calculated between the stack and application logic. This, combined with the output of `perf stat`, gives the instructions and cycles spent by the network stack versus the application.

There are two major caveats in this approach:

- `perf stat` and `perf record`, or multiple `perf record` invocations cannot be used at the same time. This restriction is due to the fact that `perf record` comes with a non-negligible overhead, and can interfere with itself when multiple copies of `perf` are executed simultaneously. Consequently, every IPC and IPQ calculation for network stacks essentially involves 3 distinct experiment runs with different `perf` commands invoked alongside the application:

  1. `perf stat` only, providing base statistics such as the total number of cycles and instructions executed
  2. `perf record` on cycles, providing the breakdown of cycles used by application, libraries and kernel

[2] A symbol table that maps the names and locations of most functions, including functions internal to the program and not exported for dynamic linkage.

3. `perf record` on instructions, providing the same breakdown of instructions.

It is assumed that application behaviour investigated in this chapter does not change fundamentally for the sole reason of a different `perf` command used.

- User-level network stacks have inlined code that does not exist as separate symbols inside static libraries. Without separate symbols, `perf record` cannot distinguish between them and application logic. This compile- and link-time optimization can lead to a slight over-estimation of applications' own overheads when user-level networking is in use.

Overall, however, this approach should provide a good estimation of each metric required for this chapter. Note that for consistency, `perf record` is used even for the kernel-user breakdown in cases where the kernel network stack is examined, regardless of the fact that `perf stat` already supports such a distinction. The intention of consistently using `perf record` is to cancel the effect of any potential bias inherent to `perf record`, whether beneficial or not to user-level networking.

3.2 Performance Assessment

The performance model is used for basic observations pertaining to the cost of I/O-heavy server applications, specifically Memcached and Nginx. Experiments reported below are performed with closed-loop clients that guarantee 100% CPU utilization on all server cores, eliminating the need to report CPT. For QPT, raw reports from the benchmark tools are adopted, which uses QPS, i.e. QPT where $T = 1s$.

In all reported experiments, IPQ and cycle (CPU cycle) measurements are divided into an *App* and *Sys* part denoting overhead in the application versus the rest of the system (libraries and kernel), according to techniques laid out in Section 3.1.2. In most cases, server software examined in this section is either unmodified upstream versions, or a port directly taken from the software repository of the respective network stacks, maintained by the same authors of the network stacks. The exception is Memcached on F-Stack[3], where a simple source-level transformation to use `epoll` is first performed, and then I/O functions along with `epoll` are replaced with compatible functions from F-Stack.

These experiments are shown as tables of performance metrics along with a brief report on the observations. A further discussion is presented in Section 3.3.

[3]Source code available at https://git.uwaterloo.ca/p5cai/memcached-fstack.

Table 3.1: Memcached: Vanilla vs. F-Stack (1 core)

	QPT	IPQ			Cycles		
	(T=1s)	App	Sys	Total	App	Sys	IPC
Vanilla	84124	1905	18512	20417	10.6%	89.4%	0.64
F-Stack (±%)	106468 (+27%)	1930	20579	22509 (+10%)	10.2%	89.8%	0.89 (+39%)

3.2.1 Memcached / F-Stack

This scenario compares Memcached in a default Linux setup with a Memcached version that uses F-Stack (on DPDK) for user-level networking. Note that this experiment is performed on a single core with single-threaded Memcached servers due to the limitations of F-Stack (see Section 2.3.1). The results are listed in Table 3.1, where 'Vanilla' denotes the default Linux setup.

Firstly, it is worth pointing out that only about 10% of Memcached's overhead is actually attributable to Memcached code itself. As confirmed by the rest of the experiments in this section, in I/O heavy server applications, system code constitutes the majority of program execution. While cycles consumed by libraries such as `libc` are not shown separately here, they are observed to incur very minimal (less than 1%) overhead.

In terms of throughput, it can be observed that replacing kernel networking with F-Stack results in an overall performance increase of 27%. The IPQ of F-Stack is slightly higher than the vanilla Linux kernel by around 10%, which is likely a difference between the specific versions of the Linux and FreeBSD network stacks adopted here under this specific load scenario, or (though unlikely) a result of porting Memcached to F-Stack. Regardless, the IPQ of the two cases shown here and their breakdown are still similar compared to some later experiments (specifically, Section 3.2.3, comparing Linux against Caladan). This implies that when functionality is similar (as is the case with Linux vs. FreeBSD), simply running a network stack in userspace may not drastically decrease IPQ enough to constitute multiple-fold performance gains. On the other hand, a significant IPC increase by 39% compensates for the added IPQ and leads to the substantial performance improvement associated with F-Stack.

Table 3.2: Nginx: Vanilla vs. F-Stack (8 cores)

	QPT	IPQ			Cycles		
	(T=1s)	App	Sys	Total	App	Sys	IPC
Vanilla	508828	5749	19245	24994	24.3%	75.7%	0.59
F-Stack (±%)	647441 (+27%)	6330	18037	24367 (-3%)	32.9%	67.1%	0.73 (+24%)

3.2.2 Nginx / F-Stack

To confirm the behavior of F-Stack on multiple cores, a similar experiment is repeated with Nginx. As Nginx is a web server, which does not require excessive shared states, it is designed to work as multiple *worker processes* by default. This enables Nginx to spawn multiple instances of F-Stack to make full use of multi-core systems.

Results for this scenario in Table 3.2 show a similar overall performance improvement of 27%. It is worth pointing out that for Nginx, cycles and IPQ for the application seem to be higher than those under the Memcached scenario. Nginx, in general, is a much larger code base than Memcached, which likely contributes to this difference. A production HTTP server must handle a plethora of different web browser and client OS implementations, along with various different protocols and mitigations for common abuses. These factors all increase the complexity of the application.

Comparing within Table 3.2, it can also be noticed that the port of Nginx to F-Stack has resulted in a slight increase in the application-level IPQ. While both use the same version of Nginx (see Chapter 5), the version provided by the authors of F-Stack makes use of dynamic redirection of I/O system calls as part of the integration effort with F-Stack. This, along with other rewriting and refactoring effort, likely results in such an increase. The inherent limitation of `perf record` (see Section 3.1.2) can also lead to a slight over-estimation of application IPQ and cycles when user-level network stacks are in use. Overall, however, the IPQ is still similar between the two cases, and the performance improvement is still driven by the IPC increase of 24%.

Table 3.3: Memcached: Vanilla vs. Caladan (6 cores)

	QPT	IPQ			Cycles		
	(T=1s)	App	Sys	Total	App	Sys	IPC
Vanilla	577653	1783	17549	19332	9.89%	90.11%	0.69
Caladan (±%)	2108154 (+265%)	2103	5282	7385 (-62%)	28.5%	71.5%	0.97 (+41%)

3.2.3 Memcached / Caladan

It is suggested in the Caladan research proposal [12] that an approximately 11-fold performance increase for Memcached results from Caladan's user-level network stack compared to vanilla Linux[4], even when the actual scheduling proposal central to Caladan is not in use. This scenario attempts to reproduce and break down such a phenomenal performance observation.

Results for running the same Memcached benchmark on the vanilla Linux kernel and Caladan are shown in Table 3.3. Note that these experiments are run on 6 cores, and for Caladan this means 1 scheduler (`iokerneld`, see Section 2.3.2) core and 5 application cores. The reason for this specific choice of cores is that Caladan experiences bottlenecks with its `iokerneld` process, which must handle all incoming packet notifications. Beyond 6 cores on this specific server, Caladan fails to deliver significantly higher throughput, meaning that around 6 cores represents the highest efficiency of Caladan for this specific hardware. In the original work, Caladan implements a feature named "directpath" by which the NIC is programmed to directly place frames in per-flow-specific buffers, which are consumed by worker threads, alleviating the `iokerneld` bottleneck. As this feature is not supported by the experiment hardware used for this work, these experiments have to be limited to a point before Caladan experiences such slowdowns.

In this scenario, Caladan achieves an impressive 3.65-fold performance improvement over the default Linux setup, which is a combination of an IPQ reduction by almost 2.6X, while the IPC is increased by 41%. The dramatic IPQ improvement is mostly contributed by the *Sys* portion (i.e., network stack), while the *App* portion sees a slight increase. Similar to the F-Stack case, this slight increase can be caused by the porting effort, where extensive refactoring is required, or static linkage and inlining that cause difficulties with `perf record`. Overall, the IPQ decrease in network stack processing far eclipses any such effect, resulting in its phenomenal performance.

[4]Figure 4 on Page 290: inflection points of solid green vs. solid blue line.

Table 3.4: Memcached/Vanilla: 8 cores vs. 4+4 cores/NUMA

Cores	QPT (T=1s)	IPQ			Cycles		
		App	Sys	Total	App	Sys	IPC
8	724077	1832	17570	19402	9.6%	90.4%	0.65
4 + 4 (±%)	601494 (-17%)	1851	17672	19522 (+1%)	8.6%	91.4%	0.55 (-15%)

To understand such a dramatic decrease in IPQ, especially when F-Stack, another established user-level network stack, does not show such a dramatic discrepancy with the Linux stack, a code inspection was performed on Caladan. This investigation revealed that the stack implements only the bare minimum functionality for TCP/IP processing that is needed to run these experiments. For example, the stack's TCP component does not implement round-trip time (RTT) estimation or maximum segment size (MSS) adjustments, but instead uses entirely constant values. Most importantly, it does not implement any congestion control. As outlined in Chapter 1, user-level networking brings about flexibility of customization, which can be seen as both a strength and/or a weakness. While it is true that a user-level network stack can omit advanced features not required by specific applications in order to reduce overhead, excluding basic features such as congestion control in TCP can be considered unfair in such comparisons.

In trying to compare the observations here with the 11-fold increase previously reported for Caladan, two further aspects need pointing out:

1. The exact configuration of the vanilla Linux system is not provided in the original paper.

2. The original experiments utilize 24 cores (or 48 hyperthreads) across two CPUs.

While the authors of Caladan state that NUMA is not considered, it is highly likely that NUMA effects have a detrimental impact on the interrupt-driven default setup, but do not affect a polling-based system like Caladan as much. Both these aspects are investigated further in the remainder of this work.

3.2.4 Memcached / NUMA

The final preliminary experiment investigates the effect of locality in general and NUMA in particular on network stack and application performance. It compares Memcached on 8

cores on a single socket with an equivalent dual-socket 4+4 setup. The results are shown in Table 3.4. While IPQ numbers are largely unchanged between both setups, it is clear that reduced locality comes with a performance penalty of about 17%, which is caused by a corresponding reduction in IPC.

3.3 Discussion

Observations reported in this chapter, in general, point to IPC, rather than overhead arising from actual processing associated with the network stack itself ("Sys" IPQ, cycles), as the driving factor in performance improvements, provided that the network stack is not simplified to fit only one or a few potential use case(s). As described in Section 3.1.1, IPC describes the efficiency of the processor pipeline, and specifically how often stalls occur in the pipeline.

Factors affecting pipeline efficiency include branch prediction misses, cache misses (L1, L2, L3, or Translation Look-aside Buffer (TLB)), or pipeline distortions forced by external sources such as interrupt handling. In addition to a similar IPQ, under the assumption that algorithmic factors remain similar between network stacks, it is safe to assume that no significant differences in branch prediction should occur. The memory footprint of all stacks far exceed typical L1 and L2 sizes, and as such their effect should be constant across all cases. While experiments studied later in this work show some variability in Last-Level Cache (LLC)[5] cache misses, they are not enough to explain the performance difference between network stacks. For data caching and locality in general, there also seems to be a limited impact. For example, the first experiment in Section 3.2.1 runs on a single core, so there are no locality issues by definition, yet the IPC increases by almost 40% with user-level networking. Both factors are further studied in the remainder of this work.

When functionality remains the same, if neither branch prediction and cache misses can explain the significant performance increase associated with user-level networking, the only possibility for such a difference is pipeline distortions. In this case, it means either hardware interrupts from the NIC, or software interrupts associated with system calls, which are not external but nevertheless distort the processing path. Note that as IPQ remains similar in the F-Stack cases, it is safe to assume that system calls present minimal direct overhead. Their effect, if significant at all, should be indirect ones manifesting in the processing pipeline, for example, from software interrupts and context switches. Even

[5]Level 3 cache on most modern x86_64 processors.

in this case, they are not expected to incur an excessive overhead thanks to optimizations in modern processors[6].

It is non-trivial on Linux to only remove system calls but not (hardware) interrupts. However, the opposite can be tested – it is theoretically easier to transform the kernel network stack into the same IRQ-less synchronous execution model as user-level network stacks. The rest of this work focuses on such a transformation for the kernel network stack. In particular, how much of the overhead associated with the kernel stack can be eliminated purely with better alignment rather than fundamental changes?

[6]Speculative execution bugs that may add more overhead on system calls also tend to be mitigated on latest hardware.

Chapter 4

Network Stack Alignment

Reports presented in Chapter 3 suggest that temporal and spatial alignment of the network stack may be at the core of a large portion of performance uplifts associated with user-level networking. Here, temporal alignment refers to a synchronous execution path, as opposed to asynchronous interrupt processing, while spatial alignment refers to core locality. As mentioned in Chapter 1 and Chapter 2, there is typically very strong alignment under user-level networking, both temporally and spatially.

To illustrate and corroborate the performance potential of alignment, a number of proposals with increasing practicality are presented in this chapter to reorganize IRQ handling for the Linux kernel network stack. The goal is to avoid asynchronous interrupt handling and simulate patterns of execution in a user-level network stack as much as possible. At the end of this chapter, a scheme is introduced with both high performance and practicality.

It is important to note that the presentation of these schemes is focused on RX interrupt handling, because it has a much larger effect than that of TX interrupts. The transmission (TX) path of the network stack does not rely on TX interrupts, but rather executes synchronously with the application's requests to transmit. TX interrupts are typically only used to recycle stale entries from TX queues, and sensible default configurations do not generate an excessive amount of them to constitute a significant effect.

4.1 IRQ Routing

Most modern hardware platforms provide programmable interrupt controllers, which can be configured through the operating system. In the Linux kernel, each IRQ number is

associated with a property called *affinity*, defining which CPU cores receive and potentially handle the corresponding interrupt. The respective CPU core also typically executes the deferred portion of interrupt handling (termed *softirq* in Linux). A set of in-kernel threads (one per core), `ksoftirqd`, are responsible for running callbacks defined by device drivers registering for interest on IRQs. Most processing in the RX path as required by the network stack is performed in the context of these kernel threads. Notifications signalling data availability are delivered asynchronously after network processing is completed by `ksoftirqd`.

Configurations on the specific way to service each IRQ have a tremendous effect on the network stack, as a majority of the RX path resides in the IRQ handler's context. Opinions differ among practitioners on the optimal strategy under various scenarios, with some advocating for *balancing* IRQs among CPU cores [45, 31, 21], while others believe that they should be *packed* onto a small number of dedicated cores [34], especially for latency-sensitive workloads.

4.1.1 IRQ Balancing

It is often recommended to balance total IRQ workload across CPU cores in order to achieve higher performance. Each IRQ number in a system is not guaranteed to always receive the same amount of workload. The *irqbalance* [49] daemon automatically observes traffic generated by each interrupt source and directs the highest volume interrupts to a single unique CPU core each, while lower-volume sources can share other CPU cores. This ensures that each CPU handles an approximately equal amount of IRQs and their associated workload. However, there are two caveats:

- The IRQ balancing process does not necessarily take into account the placement and scheduling of network-intensive applications, which results in less than optimal alignment.
- Consequently, the very nature of dynamic interrupt balancing can lead to performance variations that make reproducibility difficult.

As a result, disabling irqbalance and controlling interrupt routing statically is usually the preferred approach for reproducible high-performance networking experiments.

For an application deployed on N cores, a typical static approach is configuring N RX and N TX queues and assigning one RX and one TX interrupt per core. In networking-focused experiments, it is generally expected that the NIC attempts to balance each RX

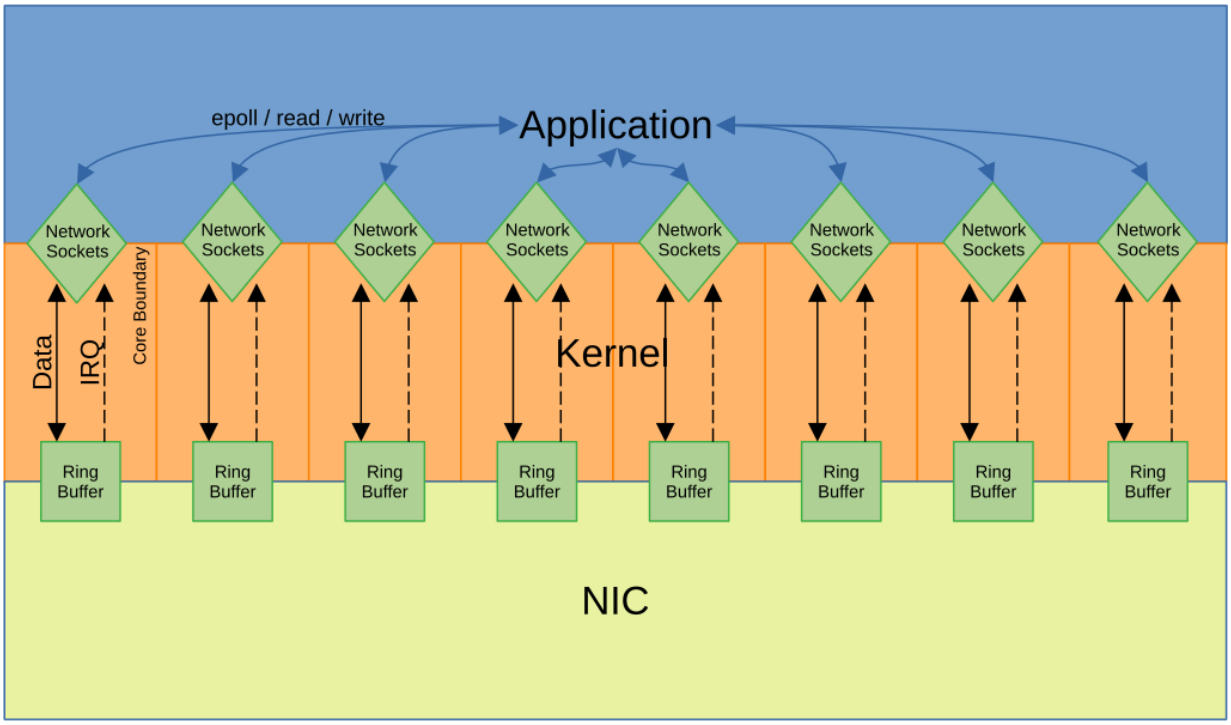

Figure 4.1: IRQ Balancing

and TX queue in the amount of traffic, resulting in an approximately equal amount of interrupts generated. Consequently, this simple static configuration should result in a near optimal balancing of IRQ workload, without needing dynamic schemes such as irqbalance.

The setup described above forms the baseline for the experiments reported in this paper and is conceptually illustrated in Figure 4.1. Note that in a realistic workload, a `ksoftirqd` thread, which processes and delivers notifications for network traffic, does not necessarily execute on the same core(s) as the corresponding application thread. Mechanisms exist in Linux that attempt to improve locality on the data path between NIC and application threads, such as Receive Flow Steering (RFS) as well as thread pinning in combination with the `SO_INCOMING_NAPI_ID` or `SO_INCOMING_CPU` socket options. However, none of these mechanisms significantly shift the baseline, at least not when all cores are in the same NUMA domain, so Figure 4.1 shows the most optimistic (but unrealistic) case of perfect spatial alignment.

4.1.2 IRQ Packing

In the Linux kernel, New API (NAPI) [37] serves as the framework of all modern NIC drivers. This framework abstracts away infrastructure for raw packet processing, but most importantly, it already includes an opportunistic mechanism to moderate interrupts, albeit incomplete compared to user-level networking.

In NAPI, after an interrupt is received, the respective interrupt number is temporarily masked. The kernel enters polling mode and retrieves available network packets from the RX ring where the interrupt is generated. This polling loop can run continuously for a period of time up to a certain *budget*, configurable through kernel parameters. When the budget is exhausted, the corresponding interrupt is re-enabled. If the RX ring is still not empty at this point, the kernel opportunistically restarts another polling episode if allowed by scheduling constraints. Therefore, the more cores the RX workload is distributed over (thus each core is less loaded by RX path processing), the less likely this new polling episode can happen. The presence of application workload on the same core also delays the onset of the new polling episode, resulting in interrupts generated before being suppressed again due to the next polling episode.

Based on this observation, an *IRQ packing* scheme is proposed. The objective is to force the kernel into an almost perpetual polling mode on a subset of cores by excluding the influence of application workload, and by minimizing the probability of fully exhausting the RX ring during any polling episode. This configuration scheme is expected to facilitate aggressive interrupt moderation in both the NAPI layer and the NIC driver. The resulting performance changes illustrate the cost of interrupts and their handling. Note that this objective is not necessarily the same as the recommendation of a similar assignment scheme in the context of real-time workloads [34].

Instead of distributing the interrupt load, under IRQ packing, interrupts are assigned to a minimal set of dedicated cores, while application threads are restricted to a set of different cores. The number of NIC queues is also set to the number of dedicated IRQ handling cores. This IRQ packing scheme is illustrated in Figure 4.2. In an ideal case, it uses just enough cores to handle all network traffic, causing all IRQ-handling cores to be constantly saturated. Although this does not spatially align the application with the network stack, IRQ packing makes interrupt mitigation extremely effective and suppresses most interrupts. The resulting performance improvements, shown in Chapter 5, indirectly confirm the conjecture that IRQ handling has a significant performance impact. However, IRQ packing is often not straightforward to configure, since it relies on configuration settings that are hard to adapt dynamically. Most importantly, CPU cores can only be allocated in whole integers and need to be fully saturated by network traffic to effectively

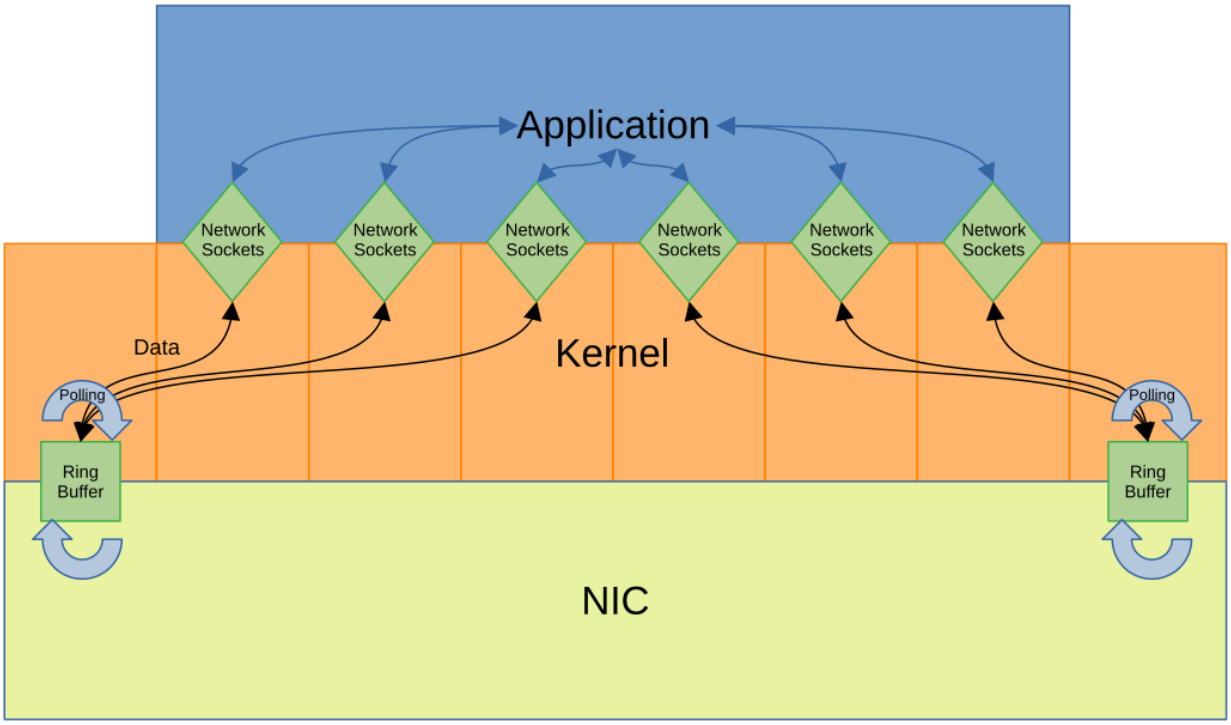

Figure 4.2: IRQ Packing

mitigate interrupts. These requirements result in a bottleneck – when a non-integer number of cores are required, assigning any more than the closest rounded-down number of cores would prevent effective polling. Some polling iterations end up empty, causing a fall back to interrupt-based handling.

Another potential limitation of this scheme is NUMA scalability. While spatial non-locality does not cause significant degradation when all cores are located on the same socket, the same cannot be said when a NUMA domain boundary is present. Under this scenario, it is not clear how IRQ packing should be configured. As no binding exists between cores serving IRQs and cores executing application code under this scheme, there can be significant NUMA penalties experienced by such a configuration. This issue with NUMA is examined further in Chapter 5.

4.2 Network Polling

While IRQ packing illustrates the overhead of IRQ handling, it cannot be regarded as a general-purpose scheme, as explained in the previous section. In the vanilla IRQ balancing scheme, on the other hand, IRQ arrivals compete with network processing and application processing for the same set of cores. The limitations of both of these schemes display a fundamental issue with IRQ-based network processing – minimal to no coordination exists between IRQ handling and the application.

In contrast, user-level (kernel-bypass) network stacks put the application in charge of the execution of the entire network stack (see Chapter 2). Interrupts are disabled globally, and the application coordinates execution by alternating between processing existing requests and polling the RX queues for new data. Modern programmable NICs address the exclusivity problem by allowing fine-grained control over which network traffic arrives in which RX queue.

This execution pattern can be emulated in the existing kernel network stack, albeit in an imperfect way. Mechanisms to promote application-initiated polling-based packet reception are already present in the Linux kernel. One such feature, though disabled by default, is `sysctl net.core.busy_poll`, sometimes used in combination with the `SO_BUSY_POLL` socket option. When this parameter is set, the kernel enters a short busy polling period, as defined by the value of the parameter, when an application uses any of the I/O multiplexing calls (`select()`, `poll()`, or `epoll_wait()`) and no events are immediately available. If network packets are received during polling, network protocol processing is performed in the same synchronous execution path, resulting in the desired cooperation between the application and the network stack similar to user-level networking. Note that this operating mode is fundamentally different and distinct from the opportunistic polling mode of NAPI mentioned in Section 4.1.2. While both are termed "polling" here, the short opportunistic polling episodes performed by NAPI are still tightly integrated in the interrupt-oriented processing path. They are not initiated by the application through any of the system calls, and have no direct knowledge of application workload. Here, with `sysctl net.core.busy_poll`, polling is only performed when the application explicitly informs the kernel of its lack of workload through one of the aforementioned system calls.

On the other hand, none of the mechanisms described above eliminates asynchronous interrupt handling sufficiently on their own. The kernel suppresses interrupts during a busy-polling episode, but afterwards interrupts are immediately re-enabled. Thus, while the application is processing data received previously, IRQs continue to arrive and distort the application's execution, incurring a large portion of the direct and indirect costs of IRQ

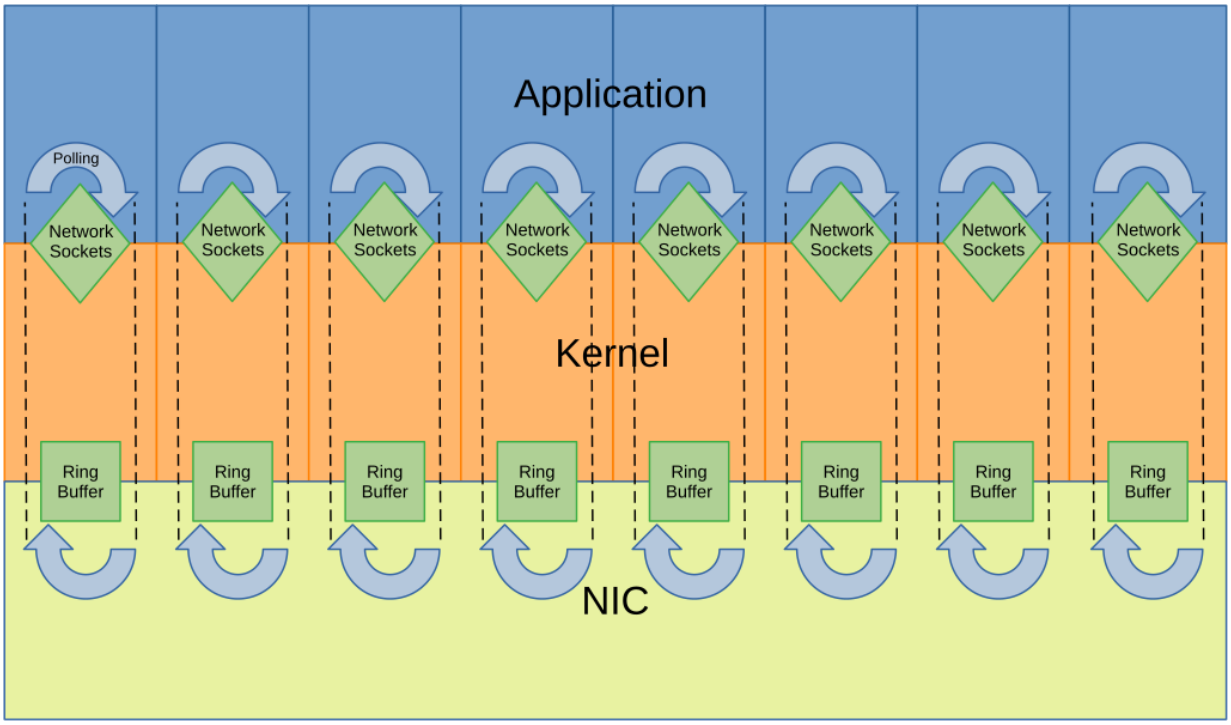

Figure 4.3: Network polling

handling. These IRQs also significantly limit the effectiveness of busy polling, because they allow some network packets to be available before the application calls I/O multiplexing again (e.g. `epoll_wait()`), preventing busy-polling episodes from being activated. Consequently, even with `sysctl net.core.busy_poll` enabled, applications still tend to fall into a "habit" of interrupt-based processing. To fully emulate the pattern of user-level network stacks, a separate mechanism is needed to mitigate these interrupts. The rest of this section explores such mechanisms, with the ideal case illustrated in Figure 4.3.

4.2.1 IRQ Suppression

The first potential approach to mitigating these (often unnecessary) interrupts and enforce network polling is to use the *interrupt coalescing* feature of modern NICs. This mechanism is a counter- and/or timer-based parameter to request the NIC to delay interrupt generation. These parameters are accessible through the `ethtool` program on Linux and can be set to high values to effectively suppress interrupt generation. In combination with

the existing kernel busy polling mechanism described above, this results in the majority of packets being received via polling, while interrupt generation is heavily limited.

As an example, assuming closed-loop clients, where network traffic is expected to be always available, one could use the following command[1] to request maximal coalescing.

```
ethtool -C <NIC> rx-usecs 65534 rx-frames 65534 adaptive-rx off
```

This command instructs the NIC to only generate one interrupt every 65534 ethernet frames, or every 65534 microseconds if not enough frames are received, eliminating the vast majority of interrupts. On the other hand, this tuning would affect performance negatively in all but closed-loop scenarios, because after an empty polling iteration, the application would block waiting for notification, which depends on IRQs being issued. The NIC cannot be notified of this fact, and IRQs can still only be fired after the number of frames or microseconds have passed as set in the `ethtool` command. Application utilization is thus reduced and time is wasted in the blocked state.

In general, such a tuning-based approach is not very robust and requires meticulous configuration for each application and in fact, each workload situation. Any tuning in the coalescing parameters involves a trade-off between latency and throughput, as the lack of interrupts during a low-rate (lower than that anticipated for a specific tuning) arrival phase causes delays before packets are retrieved. However, this scheme along with results shown in Chapter 5 does show that there is at least a possibility to realize a high level of coordination between application and network processing without abandoning or modifying the kernel network stack.

4.2.2 Kernel Polling

The missing piece for improved coordination between application and network stack is for the application to gain control over the masking of IRQs. As mentioned before, I/O multiplexing system calls in the Linux kernel (`epoll` etc.) already support busy polling before idle application threads are allowed to sleep. While interrupts are unmasked by default whenever a busy polling episode stops, a minor kernel modification can maintain masking while application logic is processing data retrieved from previous iterations.

The modification is implemented as part of `epoll_wait()`, the main component of the `epoll` series of system calls that enables applications to wait on events of interest. Instead of re-enabling the respective interrupt(s) as soon as `epoll_wait()` returns from its busy loop, IRQs stay masked until a subsequent `epoll_wait()` call finds no events of interest

[1] TX IRQs can also be tuned accordingly if required.

```
 1: procedure EP_POLL(ep)                          ▷ epoll_wait() implementation
 2:     avail ← are_events_available(ep)
 3:     loop
 4:         if avail then                                          ▷ Return path
 5:             events ← get_events(ep)
 6:             new_napi_id ← events[0].napi_id
 7:             if new_napi_id ≠ ep.last_napi_id then
 8:                 unmask_interrupts(ep.last_napi_id)
 9:             end if
10:             ep.last_napi_id ← new_napi_id
11:             mask_interrupts(ep.last_napi_id)
12:             return events                                      ▷ Copy to user
13:         end if
14:         avail ← do_busy_poll(ep)
15:         if avail then
16:             continue
17:         end if
18:         unmask_interrupts(ep.last_napi_id)
19:         sleep_until_notified(ep)
20:         avail ← are_events_available(ep)
21:     end loop
22: end procedure
```

Figure 4.4: Kernel Polling (Pseudo-code; Changes Highlighted)

and the application thread is about to be blocked. To facilitate this change, a new IRQ inhibition flag is added to NAPI instances, which correspond to RX queues. This kernel flag is set in the return path of `epoll_wait()` as long as the application has data to process. When the flag is set, NAPI advises the NIC driver against re-enabling hardware interrupts. With this change, no network interrupt is delivered while the application is busy receiving and processing data, and interrupts are only used as a fallback when the application is idle. The modification is illustrated with pseudo-code in Figure 4.4 and implemented by about 30 lines of kernel modifications at the boundary between generic event polling and NAPI code.

The resulting execution model mimics the execution model of typical user-level network stacks at full load and does not add any requirements compared to user-level networking. At lower loads, in fact, it is slightly better, because it can resort to blocking and inter-

rupt delivery, instead of having to continuously busy-loop during idle times. In order to maximize efficiency, there needs to be a 1:1 mapping between application threads and RX queues, as illustrated in Figure 4.3, so that no ambiguity exists on which queue should be polled by each application thread. A lack of such a mapping does not result in catastrophic failures as the polling loop can "float" to different RX queues, as shown in Figure 4.4 – only efficiency is affected due to the need to unmask interrupts while a thread switches between different queues to poll on. On the other hand, thread affinity to cores is not required for this scheme to function effectively.

The socket flag `SO_INCOMING_NAPI_ID` can be used to implement the aforementioned 1:1 mapping. When new connections are accepted, this flag indicates the ID of the in-kernel NAPI instance where the connection originates. A NAPI instance typically corresponds to a unique RX queue of the NIC. Under the condition that the number of threads and RX queues is set to equal, the application can simply distribute accepted connections according to the value of this flag. For example, it can be enforced that the $i \mod N$-th thread handles connections whose `SO_INCOMING_NAPI_ID` value is i and the total number of RX queues is N. This ensures a near-perfect polling pattern in each application thread.

While the currently proposed modifications are largely a proof-of-concept and not yet production-ready, there is a clear path towards a production-grade kernel polling scheme and possible adoption. As mentioned above, modern programmable NICs can alleviate cross-traffic concerns due to fine-grained flow classification and routing to specific RX queues. There is, however, still a security risk where (misbehaving) applications may stop calling `epoll_wait()` even with work remaining, resulting in interrupts being masked indefinitely. A fallback technical approach would use a kernel timeout set on the return path from `epoll_wait()`. If the application fails to re-enter the system call after the timeout, interrupts should be forcibly re-enabled. An administrative approach towards production-level robustness would encode the interrupt masking request in a privileged socket option or `epoll_wait()` flag, only available to threads with a suitable *capability* [14] flag.

Chapter 5

Evaluation

In this chapter, proposals laid out in Chapter 4 are evaluated in a controlled environment. Experiments are designed to confirm the impact of IRQs on network stack processing performance. Results from these experiments illustrate the potential of better aligned network stack processing, in particular, kernel polling.

5.1 Experimental Setup

5.1.1 Hardware

The evaluation is performed on a server with dual-socket octa-core Intel Xeon E5-2680 CPUs (NUMA setup, 16 cores in total). This server is equipped with 64 GiB of RAM (32 GiB per NUMA node), enough for all experiments reported in this chapter, and a Mellanox ConnectX-3 10 GbE network controller. In all experiments, Turbo Boost is manually disabled to rule out any unpredictable effects, and both CPUs run at their maximum non-Turbo Boost frequency of 2.7 GHz. Hyperthreading is avoided by only scheduling threads on the respective first hyperthread of each core. An additional 7 machines identical to the one described above are used as clients to generate load on the experiment server.

5.1.2 System Software

All servers in the experiments are set up with Ubuntu 20.04, with updates up to Q4 2022. User-level network stacks, such as Caladan, require an older version of the Linux kernel.

As a result, these experiments (primarily in Chapter 3) are performed on kernel version 5.4, provided by the official Ubuntu 20.04 repositories. Since these stacks bypass the kernel and require dedicated CPU cores and/or implement their own scheduling, the older kernel version is not expected to cause any distortion of performance observations - advantageous or disadvantageous. All other experiments are performed on kernel version 5.15, enhanced by the kernel polling patch described in Section 4.2.2 when appropriate.

It is also worth pointing out that even for experiments labelled 'Vanilla', a static IRQ assignment (see Section 4.1.1) is performed with each core mapped to one dedicated RX and TX queue on the NIC, in order to rule out inconsistencies in the default assignment or interference from dynamic IRQ assignment schemes such as *irqbalance*. Normally, this change should result in a slight performance increase for the vanilla kernel due to better locality compared to a true default setup. The kernel is booted with the boot setting `mitigations=off`, which disables various mitigations for older CPUs' security vulnerabilities. In addition, automatic NUMA balancing is turned off via the respective `sysctl` parameter to avoid interfering with the intended thread placement during experiments.

5.1.3 Benchmark Software

Memcached is an attractive target application for benchmarking network stacks and systems software. It is a widely deployed production-grade tool, while also being lightweight enough to expose the performance and efficiency of the underlying runtime system stack. Memcached 1.6.9 is used for all experiments using the Linux kernel's network stack. This is the earliest version with support for NAPI locality based on the `SO_INCOMING_NAPI_ID` socket option, which is required for kernel polling (Section 4.2.2). The reason for such a selection is that it is closest to Memcached 1.5.9, provided by the authors of Caladan as part of its software repository.

While porting Memcached to F-Stack, it was necessary to revert most of the changes in 1.6 to support user-level networking. For example, the `SO_INCOMING_NAPI_ID` socket option is irrelevant and must not be used when Memcached is running on a user-level network stack. To avoid mixing too many versions, Memcached 1.5.9 is also used for the F-Stack experiments. Any porting effort to Caladan or F-Stack consists of extensive code modification and refactoring that, at the very least, fully rewrites the main event loop of Memcached. This results in much greater differences than those between Memcached 1.5.9 and 1.6.9.

Memcached workload is generated with *Mutilate* [23], a well-established benchmark client for Memcached, using 8 threads (cores) on each of the 7 client machines, each client

thread creating 20 connections to the server with a pipeline depth of 1. The experiments use Mutilate's synthetic recreation of the Facebook "ETC" workload described in the literature [4] with 1,000,000 records.

Chapter 3 also presents experiments based on Nginx 1.16.1, since it is the latest version supported by F-Stack, and part of the F-Stack software repository. *Wrk* [13] is used for load generation, with 1000 concurrent connections repeatedly requesting the same small file[1].

5.1.4 Scripting

For consistency between experiment runs, a shell script[2] is developed to perform all the experiments in a predictable and reproducible manner, for example, by always re-applying the base IRQ assignment scheme mentioned above before the rest of the experiments. The script consists of different sub-scripts in the `experiments` directory for each type of experiment shown in both this chapter and Chapter 3. The `configs` directory contains parameters required for running the experiments, such as the paths to Memcached and Nginx executables, which must be adapted for the specific server where experiments are run. Finally, the `tunings` directory contains kernel and application-level configurations for each scenario (e.g. `ethtool` tuning for IRQ suppression).

The main entry point of the script is the file `run.sh`. Executing the shell file with different command-line parameters corresponds to different experiments shown in this work. Experimental output is always produced in the `data` directory, within subdirectories whose names reflect command-line parameters used. Typical command-line parameters to this file include:

- `-e <str>`, where `<str>` denotes the name of the experiment to run. This must correspond to a sub-script in the `experiments` directory.
- `-t <num>`, where `<num>` specifies the number of threads to run the experiments. The script also applies a limit on the CPU cores that can be used for the experiment using `taskset`. The limit is set as cores with IDs 0-`<num>`, except when `--first-cpu` is specified.
- `--first-cpu <num>`, which applies an offset of `<num>` to the IDs of CPU cores used for the experiment. On the server described earlier, cores 0-7 are from the first

[1] A plaintext file with content "Hello, world".
[2] Available at https://git.uwaterloo.ca/p5cai/netstack-exp/.

NUMA node, while cores 7-15 are from the second NUMA node. This configuration allows the use of the `--first-cpu` offset as a way to force the application to run across a NUMA boundary.

- `--extra-tunings <str>`, where `<str>` refers to a specific tuning configuration script to apply in the `tunings` directory.
- `--fstack` and `--caladan`, instructing the experiment to be performed on F-Stack or Caladan, respectively.

Some experiment sub-scripts also have their own command-line parameters. These can be specified after the aforementioned parameters to `run.sh` by separating them with `--`. Experiment-specific parameters are mainly intended to produce breakdowns shown in Chapter 3, where the parameter `--perf-record <type>` is supplied to produce `perf record` reports in the output directory. Parameters not mentioned in this section and their usage can be examined by inspecting the first few lines of each file in the repository.

Table 5.1: Memcached: Alignment Proposals, 8 cores

	QPT (T=1s)	IPQ			Cycles			pkts/ irq
		App	Sys	Total	App	Sys	IPC	
Vanilla	724077	1832	17570	19402	9.6%	90.4%	0.65	1.05
IRQ Packing (±%)	847669 (+17%)	1981	17549	19530 (+1%)	10.7%	89.3%	0.77 (+18%)	22.1
IRQ Suppression (±%)	967675 (+34%)	1842	17123	18965 (-2%)	11.7%	88.3%	0.85 (+31%)	262
Kernel Polling (±%)	947021 (+31%)	1853	16716	18569 (-5%)	11.9%	88.1%	0.82 (+26%)	15.6

5.2 Alignment

This section directly compares the performance characteristics of proposals in Chapter 4 against each other. Firstly, mirroring performance assessments presented in Chapter 3, Table 5.1 shows the sustained throughput performance and IPQ/IPC breakdown, as observed in a representative closed-loop experiment, for each of the proposals along with the vanilla configuration for reference. For the IRQ packing scheme, a configuration of 2 interrupt-processing cores and 6 application cores is set up for this experiment. This decision is based on the observation that 2 is the maximum number of cores that can be fully loaded by serving interrupts generated by this particular workload. IRQ suppression parameters are also manually tuned for maximum throughput in this particular experiment. In this case, it means maximum suppression on the RX side with `rx-usecs 65534` and `rx-frames 65534`. TX interrupts are also reduced using `tx-usecs 1024` and `tx-frames 256` while maintaining reasonable levels of ring buffer entry recycling in the kernel. Such a manual tuning configuration represents close to the best possible case in terms of throughput and the elimination of IRQs.

From Table 5.1, it can be noted that all alignment proposals result in a substantial performance increase over the vanilla configuration (Column 1, QPT). Moreover, it can be observed that, similar to the observations for F-Stack reported in Chapter 3, most of the performance improvement can be attributed to an increase in IPC, which closely mirrors the difference in throughput achieved by the respective proposed scheme. All schemes also significantly reduce the number of IRQs generated (or a higher number of packets processed per IRQ, as shown in the last column, pkts/irq), with IRQ suppression representing the best case scenario. There is a diminishing return in terms of the reduction of the number

of IRQs (or the increase in pkts/irq), because the first 15- to 20-fold reduction of IRQs, as shown by IRQ packing and kernel polling, already eliminates at least 90% of the IRQ overhead by definition. The remaining IRQs, however many there may be, cannot account for more than 10% of the original IRQ overhead. Therefore, while the IRQ suppression scenario sees a further 15x reduction in IRQs compared to kernel polling, the resulting throughput is similar.

However, maximum throughput is not sufficient to characterize the performance of I/O-heavy server applications. To fully assess the overall performance of each of the proposed schemes, a second experiment is used to assess the resulting tail latency behavior in relation to throughput. Clients generate a fixed rate of service requests in open-loop mode and the experiment measures the 99^{th} percentile latency achieved for the resulting throughput. Figure 5.1 shows this tail latency on the Y-axis (logarithmic scale) for varying throughputs for all alignment proposals. Each data point shows the average result of 20 independent repetitions of the same experiment. The resulting standard deviation is shown with error bars. It is again very apparent that all alignment proposals result in better performance compared to the vanilla kernel. In particular, they are able to maintain a lower tail latency up to higher rates of throughput. However, the curves differ significantly for the different schemes. This and other details are discussed next for each alignment scheme.

5.2.1 IRQ Packing

Figure 5.1 shows that IRQ packing maintains a very competitive tail latency, but Table 5.1 indicates its limited throughput capacity when compared to that of IRQ suppression or kernel polling. While IRQ packing promotes polling-based network processing, this processing is still performed in softirq kernel contexts, and is ultimately opportunistic in nature. At high load, IRQ packing increases the number of packets that are received per interrupt to about 22. Furthermore, IRQs do not distort application processing, but are delivered to dedicated cores that do not perform much other work. Overall, the performance of the IRQ packing proposal corroborates the conjecture that IRQ handling is a significant source of network processing overhead, as stated in Chapter 3, because IRQ packing results in minimal locality between the application and the network stack, yet the performance still increases significantly over vanilla.

IRQ packing shows good potential to improve network processing performance without any kernel modification. Unfortunately, its requirement of fully loading an integer number of cores severely limits the possibilities of adopting it as a general-purpose mechanism. This requirement also dictates that network processing is bottlenecked by the number of

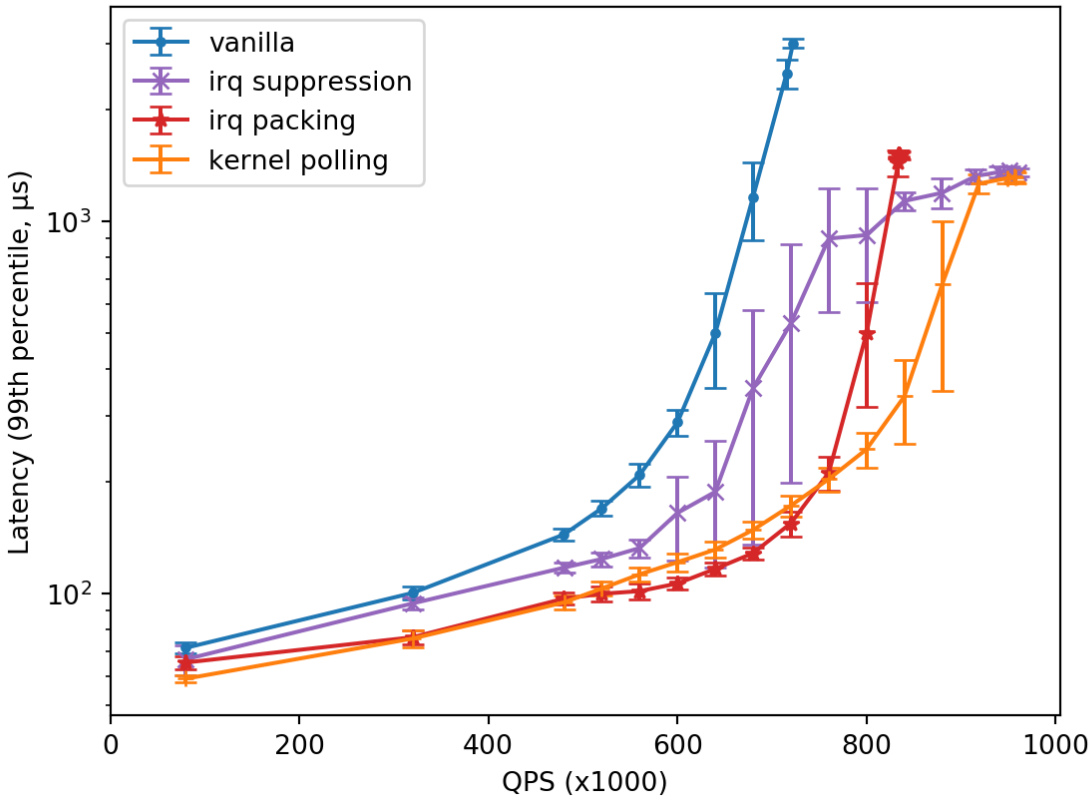

Figure 5.1: Memcached: Latency vs. Throughput, 8 cores (lower is better)

IRQ handling cores under all circumstances[3], which most likely also contributes to the limited throughput performance shown in Table 5.1 and visible in Figure 5.1.

5.2.2 IRQ Suppression

IRQ suppression results in an impressive 34% throughput increase over the vanilla version (Column 1 in Table 5.1), with a corresponding improvement in IPC. This increase further confirms the basic conjecture about IRQ handling, while also verifying that system calls

[3]Except in the extremely unlikely case where the network load exactly matches the processing power of an integer number of cores.

are not the main contributing factor, at least not in these experiments, because IRQ suppression is done purely through tuning of networking parameters without any modification in the kernel network stack. Note that compared to vanilla and IRQ packing, there is also a slight decrease in IPQ under IRQ suppression. This is due to the fact that in polling mode, asynchronous network processing logic is largely removed from the code path, resulting in the observed IPQ decrease.

However, as pointed out before, IRQ suppression requires fine-grained manual tuning for each target load scenario and even then, remains a fundamentally fragile mechanism. Its tail latency, while better than the vanilla case, is not competitive compared to the other two alignment schemes. In fact, the tail latency starts to grow much earlier than the saturation point where the system becomes overloaded.

Furthermore, Figure 5.1 shows very high variations in the measured 99^{th} percentile latency, even at relatively low load, which points to difficulties in coordinating between application and network stack. Given the current implementation of interrupt coalescing in NICs and kernel, parameters have to be chosen without taking into account application dynamics. Therefore, any given configuration implies an inherent trade-off between throughput capacity and tail latency. Specific choices of parameters can reduce tail latency for one target throughput, at the cost of overall throughput capacity and latency under even a slightly different target throughput. The converse is also true – the system can be tuned for maximum throughput capacity with maximum IRQ suppression, as is done for this experiment, at the cost of latency under non-full load. Based on these observations, it is questionable whether IRQ suppression could be deployed in dynamic workload scenarios, especially when a consistent tail latency is as important as throughput capacity.

5.2.3 Kernel Polling

The limitations suffered by IRQ packing and IRQ suppression, as evident in the last two sections, are not present for kernel polling, where a change in the kernel enables the decision whether to poll or use interrupts to be made automatically based on application workload. It shows strong performance in both maximum throughput and tail latency, as evident by Table 5.1 and Figure 5.1. There is a further decrease in IPQ even compared to IRQ suppression, because full elimination of asynchronous RX processing under high load can only be guaranteed with kernel polling.

With regard to tail latency, kernel polling is far superior to IRQ suppression. However, the IRQ suppression scheme achieves a slightly better IPC and maximum throughput

than kernel polling. The difference likely results from moderating both TX and RX interrupts in tuning for IRQ suppression, while kernel polling only disables RX interrupts whenever possible, as the interrupts reported for kernel polling in Table 5.1 are almost exclusively TX interrupts. Including TX interrupts in the scheme would require substantially more refactoring than the current RX-only approach, where code modifications are simple and non-intrusive, and can already facilitate the vast majority of the improvements. As mentioned earlier, IRQ suppression is intended to represent the best possible case for throughput in these experiments. This observation confirms that TX interrupts only have a limited impact on performance, at least in the Linux kernel.

5.3 Locality

Section 5.2 establishes the kernel polling scheme with a patched Linux kernel as the best-performing option for better network stack alignment among those described in Chapter 4. In addition, it is established that IRQs, instead of system calls or locality within a single NUMA node, are the driving factor in network processing performance, confirmed by experimental data with IRQ packing. However, locality might play a role when considering processor topologies with one or more NUMA boundaries, as shown in Table 3.4 in Chapter 3.

In this section, effects of NUMA boundaries are examined by comparing kernel polling, IRQ packing and the vanilla configuration. Specifically, IRQ packing should result in maximal spatial non-locality between the application and the network stack, since the separation of IRQ serving cores is central to this scheme. On the contrary, near-perfect spatial locality (at least at high load) automated by kernel polling is expected to require minimal communication across the NUMA boundary within the network stack. In other words, NUMA effects should manifest clearly in the comparison between these two proposals.

In producing Figure 5.2, the same latency-vs-throughput experiment from the previous section is repeated among the vanilla kernel, IRQ packing and kernel polling. For both vanilla and kernel polling, the application is spread across 4 cores in each of the two NUMA domains present on the system (4+4). For vanilla, as before, each core receives interrupts from one dedicated RX and TX queue on the NIC. As for IRQ packing, the 2 cores serving IRQs are configured to be on the second NUMA node, while the 6 application cores are kept on the first NUMA node. Application threads are not pinned to individual cores, and the scheduler is allowed to decide the placement of each thread within the specified groupings. These configurations should result in maximum communication across the NUMA boundary for all schemes examined while maintaining fairness of comparison. The

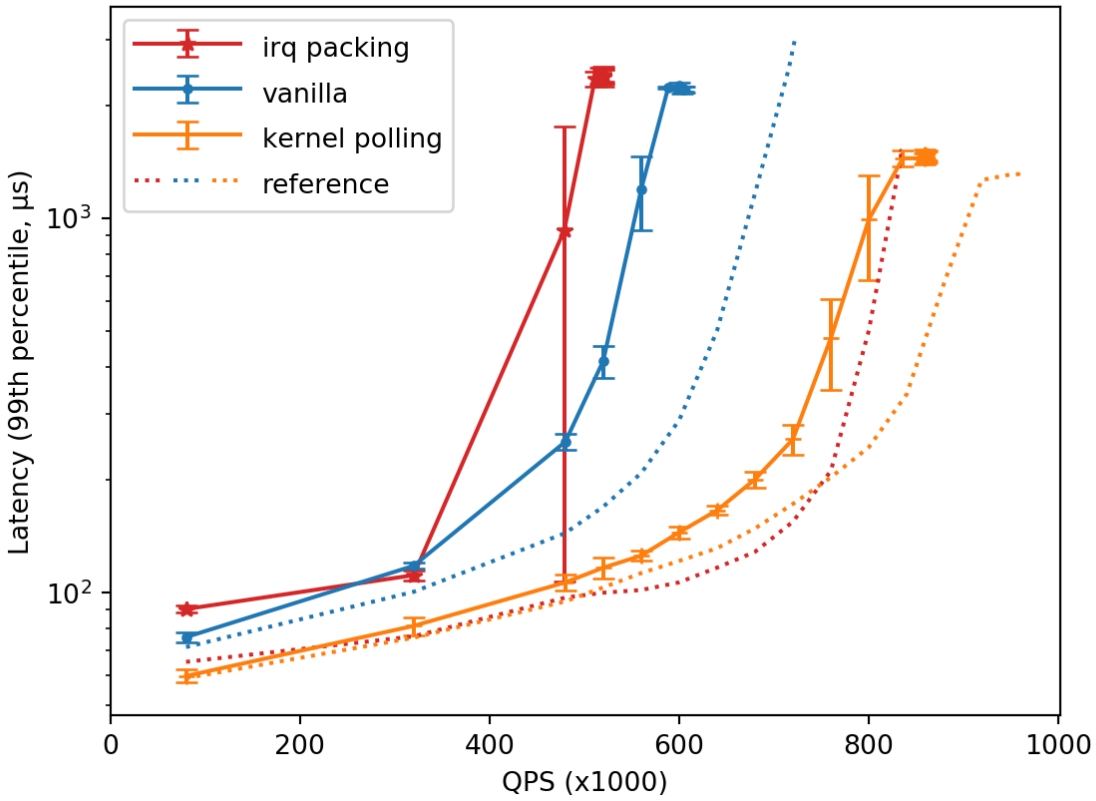

Figure 5.2: Memcached: Latency vs. Throughput, 4 + 4 cores

dotted lines in Figure 5.2 represent the same single-NUMA-domain cases from Figure 5.1 as reference.

From Figure 5.2, it can be seen that IRQ packing suffers the worst NUMA penalty, bringing down its performance to less than that of the NUMA configuration of vanilla kernel, along with high latency variation near its saturation point. On the other hand, kernel polling retains very good performance in both throughput capacity and tail latency thanks to its aforementioned automatic locality and appears to incur a relatively smaller NUMA penalty than the vanilla configuration. This fact confirms that in general, locality in between NUMA nodes does have a profound effect on performance, and that benchmarks and optimization proposals for network stacks need to take NUMA into account.

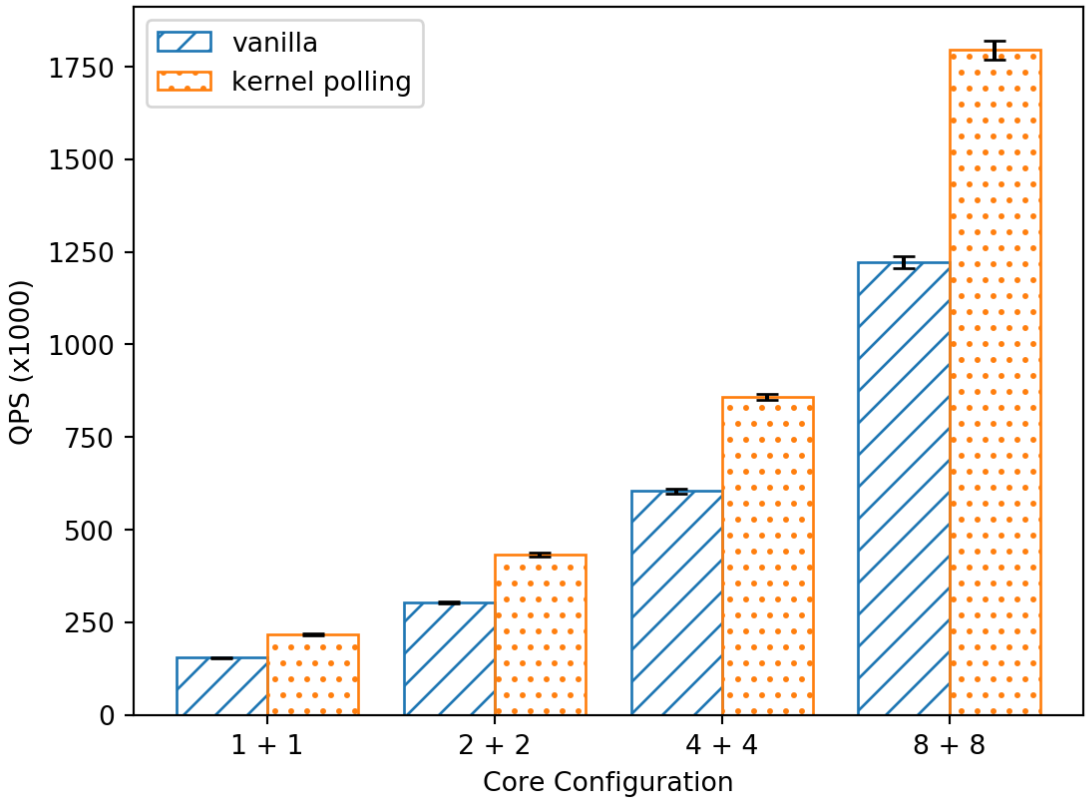

Figure 5.3: Memcached: Closed-loop Throughput (NUMA, higher is better)

To further evaluate the effect of NUMA, Figure 5.3 presents the closed-loop throughput of the vanilla kernel versus kernel polling for an increasing number of cores in each NUMA domain. Notice, both vanilla and kernel polling exhibit near-linear scaling with more cores even under NUMA, with very consistent throughput among experiment runs. The gap between vanilla and kernel polling is also largely constant, hovering at 43-46% for these experiments. Compared to 30-31% in the single-domain case (see Table 5.1 QPT column and Figure 5.1), this quantitatively verifies that for kernel polling, the network stack suffers less from NUMA overheads.

These results are consistent with Figure 5.4, where IPQ and IPC are shown for each of the NUMA configurations in Figure 5.3 for both vanilla and kernel polling. Both IPC

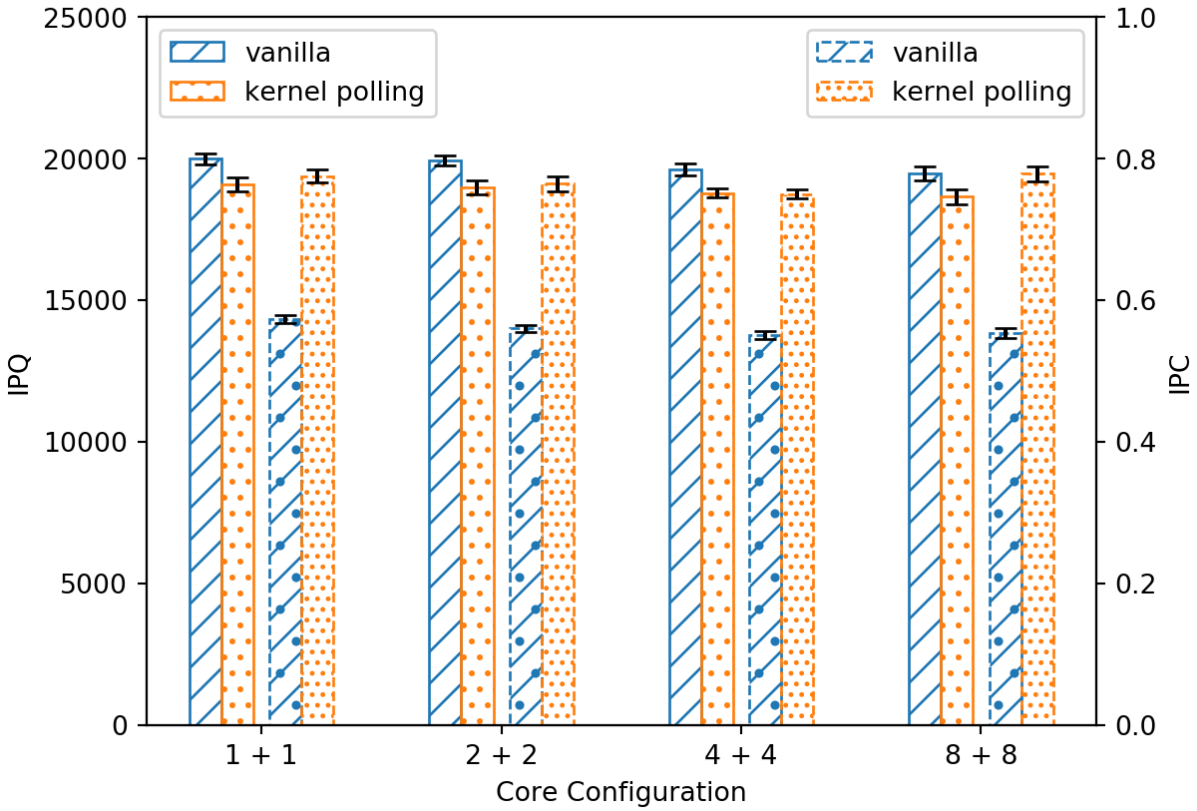

Figure 5.4: Memcached: IPQ / IPC (NUMA)

and IPQ remain largely constant as the number of cores increase in a NUMA configuration, reflecting the near-linear nature of the scalability of this workload. IPC still closely mirrors the performance difference, with a gap of 43% at 8 + 8 cores between vanilla and kernel polling, while IPQ remains similar with a decrease due to the elimination of the asynchronous processing path as described previously.

When taken together, the results reported here demonstrate that while locality does not play an important role within the same NUMA domain, NUMA overheads incurred by cross-domain communication can be substantial, as shown for IRQ packing. However, in all but the most extreme circumstances (IRQ packing), the vanilla Linux kernel with its internal packet routing and thread placement logic is largely successful in keeping NUMA

overheads limited. Kernel polling, as proposed in this paper, shows superior performance, as its automatic locality (see Section 4.2.2) further reduces NUMA overheads in the network stack.

5.4 Cache Capacity

While experimenting with Memcached, a phenomenon was observed where an increased number of connections, even at moderate values, results in a noticeable throughput decrease. This peculiarity of the kernel network stack appears to be closely connected to LLC misses. Figure 5.5 shows a set of closed-loop experiments with an increasing number of connections with kernel polling and the vanilla configuration, along with F-Stack for reference. Throughput for each examined number of connections is displayed as the curves, while LLC misses per query (i.e. normalized by QPS) are shown under the curves as bars. Results for the kernel network stack (kernel polling and vanilla) demonstrate an inverse correlation between throughput and the number of connections, while F-Stack appears to be unaffected. At 10 connections per client (560 total), kernel polling achieves a throughput about 8% higher than that of F-Stack, but this performance lead eventually dissolves as the number of connections increases. The kernel suffers from an increasing number of LLC misses per request, while for F-Stack, this number remains constant regardless of the number of connections. Note that the gap between the vanilla configuration and kernel polling remains an almost constant 30%.

The only meaningful explanation for this observation is that the effective cache footprint of the Linux network stack exceeds the LLC capacity of our particular server when handling a certain number of TCP connections. F-Stack appears to have a smaller memory footprint per connection on average, which removes LLC as a limiting factor on this specific hardware platform for these experiments. It might be a difference between Linux and FreeBSD networking, or can be considered a consequence of network stack customization.

Additionally, F-Stack seems to exhibit fairness issues at higher number of connections. This fact was observed using a customization[4] in Mutilate that reports the average and median data transmission rate over connections along with values at a few additional representative percentiles. At around 60 connections per client, the 10-th percentile transmission rate observed among the connections becomes 0. The same behaviour is not observed for other network stacks investigated in this work. This indicates an effectively smaller working set for F-Stack, which could also be part of the explanation for its lack of throughput

[4]By Thierry Delisle, unpublished.

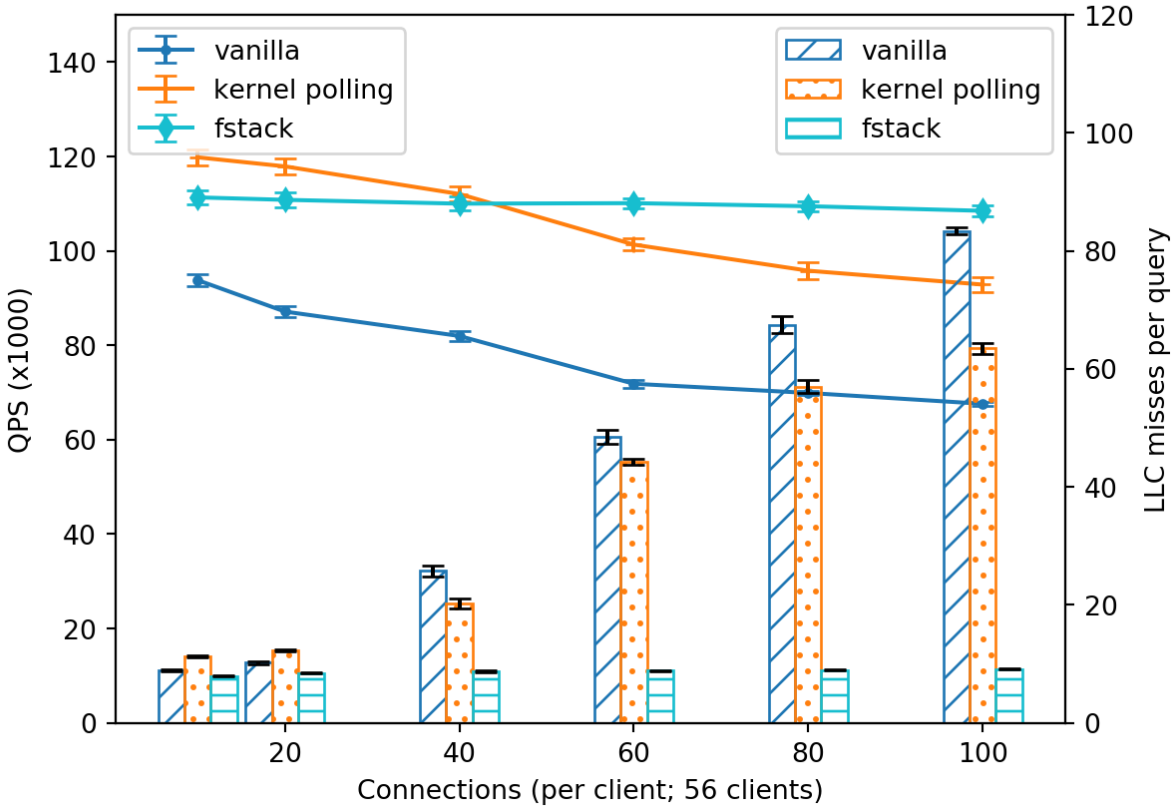

Figure 5.5: Memcached: Throughput & LLC misses, 1 core

degradation with an increased number of connections.

It is beyond the scope of the thesis to further research this specific peculiarity. All experiments reported in this work are run at a lower number of connections, before the effect described in this section becomes significant. A further comprehensive investigation would be necessary to fully understand the issue at hand and possible mitigations. Finally, as trends in hardware show an increasing amount of LLC, with some recent processors [1] approaching gigabyte-sized LLCs, the real-world relevance of this observation might be limited.

Chapter 6

Conclusion

This work presents an examination into the overheads of kernel- and user-level networking stacks. In doing so, IRQs and their handling are identified as a significant fraction of the overhead associated with the kernel network stack. As a secondary concern, core locality matters, though only eclipsing the overhead incurred by IRQs when operating across NUMA domains.

While user-level network stacks are often limited to interrupt-less processing, the kernel is not bound to relying on IRQs in principle. In practice, however, kernels prefer IRQ-based network processing due to generality and security concerns. This design results in an apparent disadvantage for the kernel, where serving IRQs results in less temporal (asynchronous / synchronous execution) and spatial (core locality) alignment compared to polling-mode processing employed by user-level network stacks.

It is important to realize that this discrepancy is of a practical nature rather than a fundamental one. In this thesis, a series of proposals were introduced to reduce this difference and improve the alignment between the kernel network stack and application. It is shown by evaluating these proposals that interrupt reduction is a key driving factor in the increase in IPC and overall performance. The best-performing scheme, kernel polling, can be implemented with a small (˜30 lines) and non-intrusive kernel change. Kernel polling increases throughput by up to 45% in a NUMA configuration without compromising tail latency. It also shows comparable performance to a comprehensive user-level stack, such as F-Stack.

Compared to user-level networking, kernel polling poses a minimal set of constraints on hardware and software. The only requirements on the NIC are that its driver must be modern enough to use NAPI, and must support receive side scaling and dynamic interrupt

moderation. Perfect alignment is guaranteed by executing the polling loop in the same application thread context with interrupts masked unless idling. Workload dynamics are handled automatically, with control handed over to the application itself, without the need for manually tuning configuration parameters. Compared to user-level networking, this method does not require reserving dedicated cores, nor pinning threads to physical cores. There is a clear path to adoption in production through capability-based permissions or a timer to guard against misbehaving applications.

Results presented in this work suggest the following. IRQs and their assignment are a major factor in deciding network processing performance. When presenting a comparison between several network processing schemes, it is necessary to carefully investigate similarities and differences in IRQ assignment and handling schemes to rule out their influence. Moreover, when proposing major restructuring of an existing software system, it is important to properly attribute relevant overheads and their reduction. In understanding the source of changes in performance characteristics, it may be possible to realize the same benefits without incurring the full cost of refactoring said system.

There are several avenues for future work arising from the findings reported here. While this work is focused on server-side network processing, it would be interesting to investigate whether kernel polling can lead to similar benefits in other application domains, such as software switches or middleboxes. Separately, a specific and detailed investigation of the Linux kernel stack might reveal opportunities to trim its per-connection memory footprint. Memory alignment, especially in NUMA scenarios, can possibly be improved by coordinating the scheduling of application threads in polling mode with ring buffer allocation in NIC drivers.

On the other hand, it can be envisioned that in the long run, high-performance networking efforts may converge in several ways. Kernel polling, as a patchset, can be adopted in the mainline kernel after adaptations for production-level security. The kernel network stack is offering increasingly many options for customization, for example, XDP and/or eBPF. New transport-level protocols, such as QUIC [19], tend to be implemented as user-level libraries with partial kernel involvement (e.g. on top of UDP). On the kernel-bypass networking side, it may become possible (with some restructuring of the kernel) to deliver interrupts directly to applications to avoid continuous polling. For either case, as shown in this thesis, there is often no silver bullet to improve performance by many folds, and it is crucial to thoroughly understand the overheads and trade-offs among the approaches.

References

[1] Advanced Micro Devices, Inc. 3rd Gen AMD EPYC Processors with AMD 3D V-Cache Technology Deliver Outstanding Leadership Performance in Technical Computing Workloads. https://www.amd.com/en/press-releases/2022-03-21-3rd-gen-amd-epyc-processors-amd-3d-v-cache-technology-deliver-outstanding, 2022. [Online; accessed 2023-04-08].

[2] Advanced Micro Devices, Inc. AMD I/O Virtualization Technology (IOMMU) Specification. https://www.amd.com/system/files/TechDocs/48882_3.07_PUB.pdf, 2022. [Online; accessed 2023-04-24].

[3] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. Association for Computing Machinery.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of SIGMETRICS*, 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] Jonathan Corbet. Ringing in a new asynchronous i/o api. https://lwn.net/Articles/776703/, 2019. [Online; accessed 2023-04-08].

[7] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux device drivers*, pages 528–531. " O'Reilly Media, Inc.", 2005.

[8] Jeff Dike. A user-mode port of the linux kernel. In *Annual Linux Showcase & Conference*, 2000.

[9] eBPF Foundation. eBPF. https://ebpf.io. [Online; accessed 2023-03-12].

[10] Brad Fitzpatrick. Memcached. https://memcached.org/. [Online; accessed 2023-04-08].

[11] Behrouz A Forouzan. *TCP/IP protocol suite*. McGraw-Hill Higher Education, 2002.

[12] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[13] Will Glozer. wrk - a HTTP benchmarking tool. https://github.com/wg/wrk. [Online; accessed 2023-04-04].

[14] Serge E Hallyn and Andrew G Morgan. Linux capabilities: making them work. 2008.

[15] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 40(4):195–206, 2010.

[16] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery.

[17] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 445–458, Seattle, WA, April 2014. USENIX Association.

[18] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanoPU: A Nanosecond Network Stack for Datacenters. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 239–256. USENIX Association, July 2021.

[19] Jana Iyengar and Martin Thomson. RFC 9000 - QUIC: A UDP-Based Multiplexed and Secure Transport. Internet RFC, Internet Engineering Task Force (IETF), May 2021.

[20] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association.

[21] Imran Khan. irqbalance: design and internals. https://blogs.oracle.com/linux/post/irqbalance-design-and-internals, 2023. [Online; accessed 2023-04-08].

[22] Greg Kroah-Hartman. The linux kernel driver interface. https://www.kernel.org/doc/Documentation/process/stable-api-nonsense.rst. [Online; accessed 2023-04-10].

[23] Jacob Leverich. Mutilate. https://github.com/leverich/mutilate. [Online; accessed 2023-04-04].

[24] Hao Li, Changhao Wu, Guangda Sun, Peng Zhang, Danfeng Shan, Tian Pan, and Chengchen Hu. Programming Network Stack for Middleboxes with Rubik. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 551–570. USENIX Association, April 2021.

[25] Linux Foundation. Data Plane Development Kit (DPDK). http://www.dpdk.org. [Online; accessed 2023-03-12].

[26] Linux Kernel Library Project. Linux Kernel Library. https://github.com/lkl/linux. [Online; accessed 2023-03-15].

[27] Robert Love. *Linux system programming: talking directly to the kernel and C library*, pages 97–98. " O'Reilly Media, Inc.", 2013.

[28] Marek Majkowski. Kernel bypass. https://blog.cloudflare.com/kernel-bypass/, 2015. [Online; accessed 2023-04-03].

[29] Nginx, Inc. Nginx. https://www.nginx.com/. [Online; accessed 2023-04-08].

[30] Ntop. PF_RING. https://github.com/ntop/PF_RING. [Online; accessed 2023-03-15].

[31] Oracle Corporation. Performance Tuning - Administering Oracle Coherence. https://docs.oracle.com/en/middleware/standalone/coherence/14.1.1.0/administer/performance-tuning.html. [Online; accessed 2023-04-08].

[32] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[33] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, Carlsbad, CA, October 2018. USENIX Association.

[34] Red Hat, Inc. Minimizing system latency by isolating interrupts and user processes. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/8/html/optimizing_rhel_8_for_real_time_for_low_latency_operation/assembly_binding-interrupts-and-processes_optimizing-rhel8-for-real-time-for-low-latency-operation. [Online; accessed 2023-04-08].

[35] Redis Ltd. Redis. https://redis.io/. [Online; accessed 2023-04-08].

[36] Marc Richards. Linux kernel vs DPDK: HTTP performance showdown. https://talawah.io/blog/linux-kernel-vs-dpdk-http-performance-showdown/, 2022. [Online; accessed 2023-04-03].

[37] Jamal Hadi Salim, Robert Olsson, and Alexey Kuznetsov. Beyond Softnet. In *Proceedings of the 5th Annual Linux Showcase & Conference - Volume 5*, ALS '01, page 18, USA, 2001. USENIX Association.

[38] ScyllaDB, Inc. Scylladb. https://www.scylladb.com/. [Online; accessed 2023-04-08].

[39] ScyllaDB, Inc. Seastar. https://github.com/scylladb/seastar. [Online; accessed 2023-03-15].

[40] John Paul Shen and Mikko H Lipasti. *Modern processor design: fundamentals of superscalar processors*. Waveland Press, 2013.

[41] Swedish Institute of Computer Science. lwIP. https://savannah.nongnu.org/projects/lwip/. [Online; accessed 2023-03-15].

[42] Maryam Tahhan and Donald Hunter. The hybrid networking stack. https://next.redhat.com/2022/12/07/the-hybrid-networking-stack/, 2022. [Online; accessed 2023-04-03].

[43] The FreeBSD Project. kqueue. https://man.freebsd.org/cgi/man.cgi?kqueue. [Online; accessed 2023-04-11].

[44] The Linux Foundation. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. [Online; accessed 2023-04-08].

[45] The Linux Foundation. Scaling in the Linux Networking Stack. https://www.kernel.org/doc/Documentation/networking/scaling.txt. [Online; accessed 2023-04-08].

[46] The Tcpdump Group. Tcpdump & libpcap. https://www.tcpdump.org/. [Online; accessed 2023-04-10].

[47] THL A29 Limited. F-Stack. https://github.com/F-Stack/f-stack. [Online; accessed 2023-03-15.

[48] Andree Toonk. Kernel bypass networking with FD.io and VPP. https://blog.apnic.net/2020/04/17/kernel-bypass-networking-with-fd-io-and-vpp/, 2020. [Online; accessed 2023-04-03].

[49] Arjen Van De Ven and Neil Horman. irqbalance. http://irqbalance.github.io/irqbalance. [Online; accessed 2023-03-17].

[50] Xilinx, Inc. OpenOnload. https://github.com/Xilinx-CNS/onload. [Online; accessed 2023-03-08].

APPENDICES

Appendix A

Kernel Polling Patch[1]

```
diff --git a/fs/eventpoll.c b/fs/eventpoll.c
index cf326c53d..71ec31a9b 100644
--- a/fs/eventpoll.c
+++ b/fs/eventpoll.c
@@ -395,7 +395,7 @@ static bool ep_busy_loop(struct eventpoll *ep, int nonblock)

  if ((napi_id >= MIN_NAPI_ID) && net_busy_loop_on()) {
    napi_busy_loop(napi_id, nonblock ? NULL : ep_busy_loop_end, ep, false,
-            BUSY_POLL_BUDGET);
+            BUSY_POLL_BUDGET, true);
    if (ep_events_available(ep))
      return true;
    /*
@@ -440,6 +440,10 @@ static inline void ep_set_busy_poll_napi_id(struct epitem *epi)
  if (napi_id < MIN_NAPI_ID || napi_id == ep->napi_id)
    return;

+ /* Ensure that IRQs are re-enabled for the NAPI instance being replaced */
+ if (ep->napi_id >= MIN_NAPI_ID)
+   napi_suppress_interrupts(ep->napi_id, false);
+
  /* record NAPI ID for use in next busy poll */
  ep->napi_id = napi_id;
 }
@@ -1779,6 +1783,7 @@ static int ep_poll(struct eventpoll *ep, struct epoll_event __user *
    events,
  u64 slack = 0;
  wait_queue_entry_t wait;
  ktime_t expires, *to = NULL;
+ unsigned int napi_id;

  lockdep_assert_irqs_enabled();

```

[1]Based on Linux 5.15.79

```
@@ -1806,6 +1811,16 @@ static int ep_poll(struct eventpoll *ep, struct epoll_event __user
    *events,

  while (1) {
    if (eavail) {
+     /* Disable IRQs as we already have enough work.
+      * Even if we fail in ep_send_events, as
+      * long as we have not timed out, we will end up
+      * polling again. */
+     if (!timed_out) {
+       napi_id = READ_ONCE(ep->napi_id);
+       if (napi_id >= MIN_NAPI_ID && net_busy_loop_on())
+         napi_suppress_interrupts(napi_id, true);
+     }
+
      /*
       * Try to transfer events to user space. In case we get
       * 0 events and there's still timeout left over, we go
@@ -1819,6 +1834,8 @@ static int ep_poll(struct eventpoll *ep, struct epoll_event __user *
    events,
    if (timed_out)
      return 0;

+   napi_id = READ_ONCE(ep->napi_id);
+
    eavail = ep_busy_loop(ep, timed_out);
    if (eavail)
      continue;
@@ -1826,6 +1843,11 @@ static int ep_poll(struct eventpoll *ep, struct epoll_event __user
    *events,
    if (signal_pending(current))
      return -EINTR;

+   /* Re-enable IRQs if we are about to enter a waiting state
+    * Otherwise we will wait forever if we have disabled interrupts before */
+   if (napi_id >= MIN_NAPI_ID && net_busy_loop_on())
+     napi_suppress_interrupts(napi_id, false);
+
    /*
     * Internally init_wait() uses autoremove_wake_function(),
     * thus wait entry is removed from the wait queue on each
diff --git a/include/linux/netdevice.h b/include/linux/netdevice.h
index 3b97438af..80f04f4eb 100644
--- a/include/linux/netdevice.h
+++ b/include/linux/netdevice.h
@@ -341,6 +341,7 @@ struct napi_struct {
  struct hlist_node napi_hash_node;
  unsigned int    napi_id;
  struct task_struct  *thread;
+ bool      suppress_interrupts;
 };

 enum {
@@ -478,6 +479,8 @@ static inline bool napi_reschedule(struct napi_struct *napi)
  return false;
 }
```

```
+void napi_suppress_interrupts(unsigned int napi_id, bool suppress);
+
 bool napi_complete_done(struct napi_struct *n, int work_done);
 /**
  * napi_complete - NAPI processing complete
diff --git a/include/net/busy_poll.h b/include/net/busy_poll.h
index 3459a04a3..13d7470ee 100644
--- a/include/net/busy_poll.h
+++ b/include/net/busy_poll.h
@@ -45,7 +45,8 @@ bool sk_busy_loop_end(void *p, unsigned long start_time);

 void napi_busy_loop(unsigned int napi_id,
        bool (*loop_end)(void *, unsigned long),
-       void *loop_end_arg, bool prefer_busy_poll, u16 budget);
+       void *loop_end_arg, bool prefer_busy_poll, u16 budget,
+       bool skip_schedule);

 #else /* CONFIG_NET_RX_BUSY_POLL */
 static inline unsigned long net_busy_loop_on(void)
@@ -109,7 +110,8 @@ static inline void sk_busy_loop(struct sock *sk, int nonblock)
  if (napi_id >= MIN_NAPI_ID)
    napi_busy_loop(napi_id, nonblock ? NULL : sk_busy_loop_end, sk,
             READ_ONCE(sk->sk_prefer_busy_poll),
-            READ_ONCE(sk->sk_busy_poll_budget) ?: BUSY_POLL_BUDGET);
+            READ_ONCE(sk->sk_busy_poll_budget) ?: BUSY_POLL_BUDGET,
+            false);
 #endif
 }

diff --git a/net/core/dev.c b/net/core/dev.c
index be51644e9..926f5a788 100644
--- a/net/core/dev.c
+++ b/net/core/dev.c
@@ -6545,6 +6545,21 @@ void __napi_schedule_irqoff(struct napi_struct *n)
 }
 EXPORT_SYMBOL(__napi_schedule_irqoff);

+void napi_suppress_interrupts(unsigned int napi_id, bool suppress)
+{
+ struct napi_struct *napi;
+
+ napi = napi_by_id(napi_id);
+ if (napi) {
+   napi->suppress_interrupts = suppress;
+   /* If we need to re-enable interrupts, we have to poll once to ensure IRQs get re-
    enabled */
+   if (!suppress) {
+     napi_schedule(napi);
+   }
+ }
+}
+EXPORT_SYMBOL(napi_suppress_interrupts);
+
 bool napi_complete_done(struct napi_struct *n, int work_done)
 {
  unsigned long flags, val, new, timeout = 0;
@@ -6613,7 +6628,7 @@ bool napi_complete_done(struct napi_struct *n, int work_done)
```

```
  if (timeout)
    hrtimer_start(&n->timer, ns_to_ktime(timeout),
            HRTIMER_MODE_REL_PINNED);
- return ret;
+ return !n->suppress_interrupts && ret;
 }
 EXPORT_SYMBOL(napi_complete_done);

@@ -6652,9 +6667,9 @@ static void __busy_poll_stop(struct napi_struct *napi, bool
    skip_schedule)
 }

 static void busy_poll_stop(struct napi_struct *napi, void *have_poll_lock, bool
    prefer_busy_poll,
-        u16 budget)
+        u16 budget, bool _skip_schedule)
 {
- bool skip_schedule = false;
+ bool skip_schedule = _skip_schedule;
  unsigned long timeout;
  int rc;

@@ -6698,7 +6713,8 @@ static void busy_poll_stop(struct napi_struct *napi, void *
    have_poll_lock, bool

 void napi_busy_loop(unsigned int napi_id,
        bool (*loop_end)(void *, unsigned long),
-       void *loop_end_arg, bool prefer_busy_poll, u16 budget)
+       void *loop_end_arg, bool prefer_busy_poll, u16 budget,
+       bool skip_schedule)
 {
  unsigned long start_time = loop_end ? busy_loop_current_time() : 0;
  int (*napi_poll)(struct napi_struct *napi, int budget);
@@ -6755,7 +6771,7 @@ void napi_busy_loop(unsigned int napi_id,

    if (unlikely(need_resched())) {
      if (napi_poll)
-       busy_poll_stop(napi, have_poll_lock, prefer_busy_poll, budget);
+       busy_poll_stop(napi, have_poll_lock, prefer_busy_poll, budget, skip_schedule);
      preempt_enable();
      rcu_read_unlock();
      cond_resched();
@@ -6766,7 +6782,7 @@ void napi_busy_loop(unsigned int napi_id,
    cpu_relax();
  }
  if (napi_poll)
-   busy_poll_stop(napi, have_poll_lock, prefer_busy_poll, budget);
+   busy_poll_stop(napi, have_poll_lock, prefer_busy_poll, budget, skip_schedule);
  preempt_enable();
 out:
  rcu_read_unlock();
```