

A Thesis for the Degree of Ph.D. in Engineering

A fast and portable TCP/IP stack

March 2026

Graduate School of Science and Technology

Keio University

Kenichi Yasukata

Acknowledgement

I would like to express my profound appreciation to my advisor, Prof. Kenji Kono. This dissertation could not have been completed without his invaluable guidance and support.

I would like to extend my sincere thanks to the members of my thesis committee, Prof. Hiroaki Nishi, Prof. Hiroki Matsutani, and Prof. Michio Honda. Their insightful feedback and suggestions have greatly contributed to improving the quality of this dissertation.

Finally, I am sincerely grateful to IIJ Research Laboratory for supporting my doctoral studies.

Abstract

This dissertation presents three-layered studies that address a research question: how can we realize a fast and portable TCP/IP stack that portably achieves good performance across diverse environments and use cases?

The first study, focusing on the middle layer, explores the design of a TCP/IP stack and presents our prototype implementation called *iip*, an integratable TCP/IP stack. We overhaul the responsibility boundary between a TCP/IP stack implementation and the code provided by developers, and introduce an API that enables *iip* to allow for easy integration and good performance simultaneously.

The second study looks into the lower layer which is packet I/O frameworks. We discuss how to support multiple packet I/O frameworks to increase the likelihood of achieving reasonable performance by circumventing the issues imposed by specific packet I/O frameworks, and present the details of our prototype implementation for *iip* along with an abstraction model that hides the differences between the APIs of packet I/O frameworks.

The third study focuses on the upper layer that implements the logic for specific use cases, and specifically, we build a memory access restriction mechanism, based on a hardware feature called Memory Protection Keys (MPK), as an upper-layer subsystem to improve memory safety for use cases where a process runs *iip* in user space, while exploring techniques to mitigate its performance overheads.

Through the three studies, we introduce a design philosophy that partitions software involving TCP/IP communication into layers and limits the layers performing memory copies of packet data. This design philosophy enables us to implement a technique for reducing memory copies in the upper layer, on top of the middle and lower layers designed to allow for zero-copy I/O. This design philosophy is naturally aligned with a fast and portable TCP/IP stack, as layer partitioning is the key to portability and reduced memory copies contribute to performance.

Contents

1	Introduction	1
1.1	Problem Statement	1
1.2	Contributions	2
1.3	Organization	4
2	Design	5
2.1	Issues of Existing TCP/IP Stacks	5
2.1.1	Performance-optimized TCP/IP Stacks	5
2.1.2	Portability-aware TCP/IP Stacks	8
2.2	Overview	10
2.2.1	Dependencies	10
2.2.2	Functionality	11
2.2.3	CPU Core Assignment Models	11
2.2.4	Use of NIC Hardware Offloading Features	12
2.2.5	Zero-copy I/O Capability	12
2.2.6	Multi-core Scalability	13
2.3	API	13
2.3.1	Common Arguments	14
2.3.2	Callback Functions Implemented by Developers	15
2.3.3	Functions Provided by iip	16
2.4	Limitations	17
2.5	Evaluation	18
2.5.1	Small Message Exchange	18
2.5.2	CPU Core Assignment Models	20
2.5.3	Bulk Data Transfer	21

2.6	Summary	22
3	Supporting Multiple Packet I/O Frameworks	23
3.1	Motivation	23
3.2	Technical Background	24
3.2.1	Overview of Packet I/O Frameworks	24
3.2.2	NIC Device Drivers	24
3.2.3	Common Features of Packet I/O Frameworks	28
3.2.4	Common Usage of Packet I/O Frameworks	28
3.2.5	Running TCP/IP Stacks in User Space	29
3.3	Major Packet I/O Framework Implementations	29
3.3.1	Characteristics	29
3.3.2	APIs	30
3.4	Discussions on Beneficial Properties	35
3.4.1	Sidestepping Compatibility and Performance Issues	35
3.4.2	Compatibility with Existing Systems	36
3.5	Implementation for Supporting Multiple Packet I/O Frameworks	37
3.5.1	Overview	37
3.5.2	Coping with API Differences	38
3.5.3	The Additional Packet-oriented Abstraction Layer	40
3.5.4	Implementation of the Callback Functions of the iip API	42
3.5.5	Discussions on Generality	43
3.6	Experiment	44
3.6.1	Experiment Setup	44
3.6.2	Packet I/O Performance	45
3.6.3	Exchanging Small Messages over TCP Connections	48
3.6.4	Bulk Data Transfer over a TCP Connection	49
3.6.5	Configurations with Existing Systems	50
3.7	Summary	52
4	Restricting Memory Accesses	57
4.1	Motivation	57
4.1.1	Memory Accesses Unintended by Developers	57
4.1.2	Goal	60

4.1.3	Out of Scope	61
4.2	The Memory Access Restriction Mechanism	61
4.2.1	Overview	62
4.2.2	Memory Protection Keys (MPK)	66
4.2.3	pkey Settings	67
4.2.4	Copies between Memory Regions with Different pkeys . .	71
4.2.5	Switching the Memory Access Rights	72
4.2.6	Reducing the Memory Access Right Switching	73
4.2.7	Reducing Memory Copies for Packet Data	75
4.2.8	Limitations	77
4.3	Discussions on Other Mechanisms	77
4.3.1	Using the MPK Feature	77
4.3.2	Using the Common Page Table Feature	80
4.3.3	Using the CPU Features for Virtualization	80
4.3.4	Using the Features of Programming Languages	81
4.4	Experiment	82
4.4.1	Experiment Setup	82
4.4.2	Exchanging Small Messages over TCP Connections . . .	83
4.4.3	Bulk Data Transfer over a TCP Connection	86
4.5	Summary	86
5	Conclusion	88
5.1	Contribution Summary	88
5.2	Future Directions	90
	Bibliography	91

List of Figures

2.1	Performance for a TCP ping-pong workload which exchanges 1-byte payloads.	19
2.2	TCP bulk data transfer throughput (Chapter 2.5.3).	21
3.1	The structures of the major packet I/O frameworks (Chapter 3.2.1).	25
3.2	The structure of NIC hardware from the view of a NIC device driver (Chapter 3.2.2).	26
3.3	The overview of the implementation to support multiple packet I/O frameworks (Chapter 3.5.1).	38
3.4	Abstraction models of the packet I/O frameworks (Chapter 3.5.2).	39
3.5	Packet I/O performance (Chapter 3.6.2).	46
3.6	Performance for exchanging small messages over TCP (Chapter 3.6.3).	53
3.7	Bulk data transfer performance over TCP (Chapter 3.6.4).	54
3.8	Performance of inter-process communication over a VALE-based virtual switch (Chapter 3.6.5.1).	55
3.9	Using XDP to avoid NIC dedication (Chapter 3.6.5.2).	56
4.1	Part of the definition of netmap_ring (Chapter 4.1.1.1).	59
4.2	TCP/IP stack placement in kernel and user space (Chapter 4.1.1.3).	60
4.3	The overview of our memory access restriction mechanism (Chapter 4.2.1).	62
4.4	Memory regions having pkey2 and relevant regions (Chapter 4.2.3.1).	68
4.5	The technique to reduce memory copies for packet data (Chapter 4.2.7).	76

4.6	Performance for exchanging small messages (Chapter 4.4.2). . . .	84
4.7	Bulk data transfer performance (Chapter 4.4.3).	87

List of Tables

4.1	PKRU settings for the two memory access rights (Chapter 4.2.3). .	67
4.2	pkey assignment for each memory type (Chapter 4.2.3).	68

Chapter 1

Introduction

TCP/IP is a standardized protocol suite for computer networks. TCP/IP stacks implement procedures to comply with the TCP/IP standard, and they have been typically developed as part of Operating System (OS) kernels. On the other hand, the hardware innovation, enabling NICs to achieve tens of Gigabit per second throughput, made it challenging for the legacy TCP/IP stack implementations to sufficiently utilize the available network bandwidth. To address this challenge, research and industry communities have invented many performance-optimized TCP/IP stacks [2, 18, 32, 14, 7, 59, 52, 21, 37, 64, 65] and demonstrated their significant performance advantages over the legacy TCP/IP stack implementations in a variety of network-intensive workloads including web [23, 49], cache [56, 47], and proxy [50, 17] servers. However, the existing performance-optimized TCP/IP stack implementations often incur tremendous integration complexity (Chapter 2.1.1). While there have been various portability-aware TCP/IP stack implementations [9, 8, 6, 55] that are free from the issues of the existing performance-optimized TCP/IP stacks, their performance is substantially limited because they are not sufficiently aware of performance-critical factors (Chapter 2.1.2).

1.1 Problem Statement

The problem is that there has been no TCP/IP stack implementation that allows for easy integration and good performance simultaneously. Consequently,

developers, who wish to integrate a high-performance TCP/IP stack implementation to speed up their systems, have had limited and laborious options: for example, intensively modifying one of the existing TCP/IP stack implementations [33, 41, 4, 26, 52, 31, 42, 19, 25], building a new TCP/IP stack from scratch, accepting performance limitations of one of the existing portability-aware TCP/IP stack implementations, or giving up the integration.

Research gap. The causes of the problem are not confined to the TCP/IP processing component. In general, a TCP/IP stack depends on lower layers, including low-level I/O subsystems, while upper layers, such as application logic and feature extensions targeting specific use cases, in turn rely on the TCP/IP stack implementation. To achieve high performance portably across diverse environments and use cases, a TCP/IP stack must be carefully integrated with its lower and upper layers to avoid including significant performance bottlenecks. However, prior studies have not explored the integration schemes that enable developers to incorporate a TCP/IP stack and other layers effectively in terms of engineering and performance overheads. Note that the integration scheme for a lower layer differs from that for an upper layer because the directions of the dependencies are opposite from the view of a TCP/IP stack. For instance, on one hand, a desirable integration scheme for lower layers makes it straightforward to support varied implementations that provide equivalent features through different APIs. On the other hand, an effective integration scheme for upper layers provides developers with opportunities to implement extensions, such as memory safety enhancement, that can be deployed transparently to other components. Due to this research gap, the methodologies for effectively incorporating a TCP/IP stack and the important facilities implemented as other layers, including packet I/O frameworks (Chapter 3) and memory safety enhancement functionality (Chapter 4), have been shrouded in ambiguity.

1.2 Contributions

This dissertation presents three-layered studies that address a research question: how can we realize a fast and portable TCP/IP stack that portably achieves good performance across diverse environments and use cases?

- **Middle layer.** The first study, focusing on the middle layer, explores the de-

sign of a TCP/IP stack and presents our prototype implementation called iip, an integratable TCP/IP stack. To derive the design of iip, we review existing performance-optimized and portability-aware TCP/IP stacks and determine the factors that diminish their portability and performance, and then, we present the design of iip along with a series of solutions to address the issues. Besides these, we report the performance numbers of iip [58].

- **Lower layer.** The second study looks into underlying packet I/O frameworks. The motivation for this study is that even though the TCP/IP processing component is adequately optimized for performance, it cannot always achieve reasonable performance due to the issues imposed by packet I/O frameworks. We investigate an approach to increasing the likelihood of circumventing the issues of specific packet I/O frameworks by supporting the use of multiple packet I/O frameworks. We present the details of our prototype implementation for iip along with an abstraction model that hides the differences between the APIs of packet I/O frameworks. We conduct experiments to evaluate the performance of our prototype and demonstrate its beneficial properties [61].
- **Upper layer.** The third study focuses on an upper layer that implements the logic for specific use cases, and specifically, we build a memory access restriction mechanism as an upper-layer subsystem to improve memory safety for use cases where a process runs iip in user space. Our memory access restriction mechanism leverages a hardware feature called Memory Protection Keys (MPK) to cause a segmentation fault and enable actions such as stopping the program when iip attempts to access memory regions that it should not, or implementations other than iip try to access memory regions dedicated to iip. Similar to the lower layer, an upper subsystem can limit TCP/IP communication performance; therefore, we explore opportunities to mitigate the performance overheads of the upper layer and implement techniques that moderate the performance reductions caused by our memory access restriction mechanism. We conduct experiments to quantify its performance overheads and examine how our techniques mitigate the performance reductions [62].

Design philosophy. Through the three studies, we present a design philosophy that partitions software involving TCP/IP communication into layers and limits

the layers performing memory copies of packet data. In our prototype, the middle and lower layers are designed to allow for zero-copy I/O and provide the upper layer, which implements the logic specific to a use case, with the opportunity to reduce memory copies for sending and receiving data. Specifically, this design philosophy enables us to implement a technique for reducing memory copies in our memory access restriction mechanism, built as an upper-layer subsystem, and transparently apply it on top of the middle and lower layers, namely the TCP/IP processing component and packet I/O frameworks. This design philosophy is naturally aligned with the properties of a fast and portable TCP/IP stack, as layer partitioning establishes the foundation for portability and reduced memory copies enhance performance.

1.3 Organization

The rest of the dissertation is organized as follows. Chapter 2 describes the study that explores the design of a fast and portable TCP/IP stack. Chapter 3 presents the study on the support for multiple packet I/O frameworks. Chapter 4 presents the study on the MPK-based memory access restriction mechanism. Chapter 5 describes the conclusion of this dissertation.

Chapter 2

Design

This chapter describes the design of iip. First, we review the existing TCP/IP stack implementations and determine their issues (Chapter 2.1), and then, derive the design of iip to address the determined issues (Chapter 2.2 and Chapter 2.3), along with the discussions on the limitations of iip (Chapter 2.4). We conduct experiments to examine the performance of iip and report the observed performance numbers (Chapter 2.5).

2.1 Issues of Existing TCP/IP Stacks

To derive the design of iip, we review the issues of the existing performance-optimized and portability-aware TCP/IP stack implementations.

2.1.1 Performance-optimized TCP/IP Stacks

We consider that the existing performance-optimized TCP/IP stack implementations incur three issues: dependencies on other components (Chapter 2.1.1.1), functionality conflicts (Chapter 2.1.1.2), and limited choices for CPU core assignment models (Chapter 2.1.1.3).

2.1.1.1 Dependencies on Other Components

Existing performance-optimized TCP/IP stacks typically introduce many dependencies on CPU architectures, NICs, OSes, libraries, and compilers, and these de-

dependencies often substantially diminish the integratability of the TCP/IP stacks. For example, the paper about the Luna [65] TCP/IP stack says that the authors gave up the use of existing performance-optimized TCP/IP stacks and decided to build Luna from scratch because the existing implementations have compatibility issues: VPP [2] is not well-compatible with Mellanox NICs, and IX [4] depends on the Dune [3] kernel module which works only on specific versions of the Linux kernel and relies on outdated Intel NIC drivers. Another example, indicating the significance of this issue, is that many tailor-made performance-optimized networked systems [33, 41, 4, 26, 31, 42, 19, 25] do not employ existing performance-optimized TCP/IP stacks, instead, they take one of the existing portability-aware TCP/IP stacks, such as lwIP [8], as the basement of their TCP/IP processing components and intensively modify the code base of it to mitigate its performance issues (Chapter 2.1.2).

2.1.1.2 Functionality Conflicts

In many cases, existing performance-optimized TCP/IP stacks are proposed as part of an application development framework [7, 52, 37] or a new OS [41, 4, 64] having various functionalities such as a specific thread runtime. The issue is that their TCP/IP stack components often depend on such extra facilities, and they cause functionality conflicts that make it difficult for developers to integrate the TCP/IP stack components into other systems. For example, when developers wish to use a TCP/IP stack implementation of a specific application development framework and integrate it into a particular new OS, if the TCP/IP stack relies on a framework-specific thread runtime and the new OS also has its specific thread runtime, the developers face a functionality conflict caused by the two independent thread runtimes; as a result, the developers cannot use the TCP/IP stack of the application development framework on the new OS, as long as they do not modify either the TCP/IP stack or the new OS, because two thread runtimes normally cannot coexist on a single system.

2.1.1.3 Limited Choices for CPU Core Assignment Models

There are three CPU core assignment models for threads executing networking and application logic; we borrow the terms from a previous study [60] and call

them split, merge, and unified, respectively:

- **Split.** The split model runs the networking logic and the application logic on two different threads, and dedicates a CPU core to each of the threads. The setup of this model requires at least two CPU cores, and in multi-core environments, the numbers of CPU cores preserved for the networking and application logic are typically configured by users.
- **Merge.** The merge model runs the networking logic and the application logic on two different threads similarly to the split model, but executes the two threads on the same CPU core; this model can be applied for a single CPU core at minimum, and in multi-core environments, the single CPU core setup, having a pair of the threads executing networking and application logic on the same CPU core, is duplicated to available CPU cores.
- **Unified.** The unified model executes the networking and application logic on the same thread. Similarly to the merge model, one CPU core is the minimum number of CPU cores necessary for the setup of this model, and multi-core environments duplicate the single CPU core setup for available CPU cores.

These models have different properties in the following aspects:

- **CPU utilization.** The split model, where the threads executing networking and application logic run on two different CPU cores, leads to low CPU utilization because it introduces a CPU resource boundary that prohibits the two threads from yielding unused CPU cycles to each other; this means that even if one of the two threads is fully busy and the other thread is occasionally idle and has unused CPU cycles, the busy thread cannot take over the unused CPU cycles from the other thread, and on top of this, a prior study [1] has reported that it is not straightforward to determine how many CPU cores should be assigned to the application logic and TCP/IP processing to achieve the best performance. Contrarily, the merge and unified models, which execute the networking and application logic on the same CPU core, make no CPU resource boundary between the networking and application logic, and these two can yield unused CPU cycles to each other; consequently, these two models can achieve higher CPU utilization than the split model.

- **Inter-thread communication overhead.** The split and merge models use two threads for executing networking and application logic, therefore, they impose the inter-thread communication overhead, and that of the merge model tends to be higher compared to the split model because the merge model executes two threads on the same CPU core, and every transition between the execution of networking and application logic requires a context switch that incurs comparatively high CPU overhead, on the other hand, in the split model, the context switch is not necessary for the transition between the two threads because these two run on different CPU cores. However, the split model imposes a different type of overhead; the speed of data movement between the two threads is limited by the hardware-level synchronization necessary to ensure the cache coherency between the two CPU cores. Besides these, the unified model is free from these overheads because it does not split the execution of networking and application logic into different threads.

The issue is that many existing performance-optimized TCP/IP stacks do not allow developers to freely choose a desired CPU core assignment model; for example, mTCP [18] does not allow developers to choose the unified model, and TAS [21] only accepts the split model. Because of this issue, it is often difficult for developers to integrate existing performance-optimized TCP/IP stacks into other systems in a performance-optimal manner (Chapter 2.5.2).

2.1.2 Portability-aware TCP/IP Stacks

The issues of the existing portability-aware TCP/IP stack implementations are that they are unaware of NIC hardware offloading features (Chapter 2.1.2.1), lack zero-copy I/O capability (Chapter 2.1.2.2), and lack multi-core scalability (Chapter 2.1.2.3).

2.1.2.1 Unaware of NIC Hardware Offloading Features

In workloads involving bulk data transfer, we found that the commonly available NIC offloading features, particularly checksum offloading and TCP Segmentation Offload (TSO), are essential to effectively utilize high-speed links (Chapter 2.5.3). However, the existing portability-aware TCP/IP stacks are not aware of the NIC

offloading features and do not leverage them even if they are supported by the installed NICs, consequently, their bulk data transfer performance is substantially limited. Moreover, other commonly available offloading features, such as scatter-gather and Receive Side Scaling (RSS), are necessary for zero-copy transmission (Chapter 2.1.2.2) and multi-core scalability (Chapter 2.1.2.3), but, mainly due to the lack of awareness of NIC offloading features, the existing portability-aware TCP/IP stacks do not offer them. We note that modifying an existing portability-aware TCP/IP stack to employ the NIC offloading features is often complicated because, to effectively utilize them, the foundational components, such as the packet management mechanism and the API, should take the availability of NIC offloading features into account as of the design phase of the TCP/IP stack.

2.1.2.2 Lack of Zero-copy I/O Capability

In bulk data transfer workloads, zero-copy I/O is important for CPU cache efficiency (Chapter 2.5.3) and commonly supported by performance-optimized TCP/IP stacks. Nevertheless, many existing portability-aware TCP/IP stacks do not support zero-copy I/O. One of the reasons for this is that they often maintain packet data on internal buffers which are managed independently of a NIC I/O subsystem; consequently, they require memory copies between their internal buffers and the packet buffers associated with a NIC. Another reason is, as described in Chapter 2.1.2.1, the lack of awareness of NIC offloading features; particularly, the scatter-gather feature of NIC hardware, which enables TCP/IP stacks to instantiate a packet header and a payload on non-contiguous memory addresses, is necessary for an ideal zero-copy transmission mechanism, which can concurrently send the same payload to different hosts, to avoid the conflicts at the header space adjacent to the payload data on the memory.

2.1.2.3 Lack of Multi-core Scalability

For workloads that exchange small messages, we find that multiple CPU cores are necessary to sufficiently take the performance benefit of high-speed NICs (Chapter 2.5.1). However, the existing portability-aware TCP/IP stacks are not designed to scale their performance on multi-core machines; for example, lwIP [8]’s internal data manipulation does not pay attention to thread safety, and FNET [6]

and picoTCP [55] employ giant locks for concurrency coordination. Moreover, as mentioned in Chapter 2.1.2.1, they are not aware of the RSS feature of NIC hardware which allows for affinity-aware packet steering where packets for a certain TCP connection are always received at a specific NIC receive queue although the previous work [40, 11, 18, 7, 28] demonstrated that the NIC-based affinity-aware packet steering enables TCP/IP stacks to reduce data exchanges between different CPU cores and contributes to multi-core scalability.

2.2 Overview

We present the overview of iip and how it addresses the issues discussed in Chapter 2.1.

2.2.1 Dependencies

To address the dependency issue discussed in Chapter 2.1.1.1, iip pays attention to the following:

- **Programming language selection.** An inevitable dependency point is the programming language. We decided to implement iip in the C programming language because C is one of the most widely used programming languages and there are many full-blown C compilers for varied computation environments. The implementation of iip complies with the C89 standard so that old and future versions of C compilers can compile the source code of iip. We also pay attention to the compatibility with the C++98 standard so that iip can be integrated into C++-based systems.
- **Indirection for using external facilities.** To minimize dependencies on other components, iip does not directly employ features specific to CPU architectures, NICs, OSes, libraries, and compilers; consequently, the compilation of the iip code base does not require external libraries, including libc, and relevant header files. The approach, to enabling the TCP/IP processing code of iip to access platform-dependent features without introducing dependencies, is to define, as iip's API specification (Chapter 2.3), a set of functions that are

the wrappers to access platform-dependent functionalities and assume to be provided by developers.

We note that iip does not depend even on a standardized API such as POSIX, therefore, it can be integrated into systems that are governed by a specialized runtime or a new OS and do not comply with a standardized interface.

2.2.2 Functionality

To minimize the chance of causing the functionality conflict issue described in Chapter 2.1.1.2, iip only implements the functionality of TCP/IP processing, and as mentioned in Chapter 2.2.1, it does not depend on specific external components except the C programming language. However, developers, considering the use of iip, should be aware that iip can become a conflict point when the developers' code, which depends on iip, has to be integrated with a system that relies on another TCP/IP stack implementation. Therefore, our recommendation for developers is to implement an abstraction layer to access networking features by themselves and let their code access the features of iip through it so that they can easily replace iip with another TCP/IP stack implementation at the level of the abstraction layer.

2.2.3 CPU Core Assignment Models

As discussed in Chapter 2.1.1.3, many existing performance-optimized TCP/IP stacks do not allow developers to freely choose the CPU core assignment model although the model selection has a significant impact on application performance (Chapter 2.5.2). The root cause of this issue is that they implement the loops that execute their TCP/IP processing code, and developers cannot customize the behavior of the hard-coded loops although, to apply the unified model, developers need to colocate both the networking and application logic in the same loop (Chapter 2.1.1.3). To avoid this issue, iip does not implement a loop to execute its TCP/IP processing code, instead, iip only offers functions to be called in a thread so that developers can embed them into an arbitrary thread which is provided by the developers (Chapter 2.3).

2.2.4 Use of NIC Hardware Offloading Features

As described in Chapter 2.1.2.1, NIC hardware offloading features are essential for TCP/IP stacks to achieve high performance (Chapter 2.5). One of the main reasons why the existing portability-aware TCP/IP stacks do not leverage NIC hardware offloading features is that they are specific to NICs and less generic compared to the simple network I/O functionality. While *iip* also does not cover all offloading features of existing NICs, it leverages commonly available NIC offloading features, such as checksum, scatter-gather, TSO, and Large Receive Offload (LRO) if they are available; *iip* defines, as part of its API, callbacks to be implemented by developers (Chapter 2.3.2) to leverage the NIC offloading features without introducing dependencies on NICs (Chapter 2.2.1).

2.2.5 Zero-copy I/O Capability

As discussed in Chapter 2.1.2.2, zero-copy I/O capability is important for TCP/IP stacks to achieve high performance for bulk data transfer workloads (Chapter 2.5.3). To offer zero-copy I/O capability, *iip* pays attention to the following:

- **Packet buffer allocation.** In the API of *iip*, developers are responsible for implementing the allocator for packet buffers associated with a NIC (Chapter 2.3.2). This means that developers can instantiate packet buffers, associated with a NIC, in arbitrary memory address space, and can perform zero-copy I/O by putting packet buffers in the memory address space that their application logic can directly access.
- **Scatter-gather feature of NIC hardware.** As discussed in Chapter 2.1.2.2, a zero-copy transmission mechanism, which can concurrently send the same payload, has to avoid the contention for the header space adjacent to the payload data. To realize this, *iip* does not use the adjacent space of payload data for putting its header, and instantiates a header on an independent memory address and assembles these two at the packet transmission using the scatter-gather feature of NIC hardware (Chapter 2.3.3).

2.2.6 Multi-core Scalability

As described in Chapter 2.1.2.3, multi-core scalability is crucial for taking the performance benefit of high-speed NICs (Chapter 2.5.1). For multi-core scalability, similar to the previous work [40, 11, 18, 7, 28], iip minimizes the contention for the access to in-memory objects shared across multiple CPU cores by dedicating in-memory objects, such as the queue of accepted TCP connections, to each CPU core. To realize this without introducing dependencies (Chapter 2.2.1), particularly on a memory allocator and a thread runtime, iip assumes the assistance of developers through a data structure representing the context of TCP/IP processing which is defined as part of the API (Chapter 2.3.1). We implement the TCP/IP processing code of iip to be stateless and the states of protocol processing are maintained through the context object, and iip assumes that developers feed a dedicated context object to a thread executing the TCP/IP processing code of iip; since it only manipulates a fed context object, the threads do not experience contentions for the access to shared in-memory objects while executing iip's TCP/IP processing code. On the other hand, this approach requires packets for a certain TCP connection to be steered to the same thread because, on the entire system, there is only one thread that has the in-memory object representing the TCP connection and it is not shared with the other threads, therefore, iip assumes that developers steer received packets to a certain thread in an affinity-aware manner, for example, by using the RSS feature of a NIC (Chapter 2.1.2.3).

2.3 API

The API of iip consists of two types of functions: 1) callback functions implemented by developers (Chapter 2.3.2) and 2) functions provided by iip (Chapter 2.3.3). The entry point for the execution of the iip code is the functions provided by iip, and they internally call the callback functions implemented by developers.

2.3.1 Common Arguments

Pointers to the following objects are passed between developers' code and the API of iip:

- **Context object.** As described in Chapter 2.2.6, the TCP/IP processing code of iip itself is stateless, and the states of protocol processing, such as the states of TCP connections, are associated with a context object. iip assumes that developers allocate a dedicated context object and feed it to each thread executing the TCP/IP processing code of iip; we note that the developers' code just needs to pass the pointer to the context object to the API of iip and does not need to recognize the content of the context object. As a side note, while this context object is mainly designed for multi-core scalability (Chapter 2.2.6), iip's context-based TCP/IP processing is beneficial to the integration with simulators such as ns-3 [44] and specifically allows a simulator process to instantiate multiple nodes installing iip in the same memory address space without using extra tricks such as `dlopen` applied in the Direct Code Execution (DCE) [51] framework.
- **Packet object.** To avoid introducing dependencies (Chapter 2.2.1) on a packet representation data structure, the API of iip offers an abstraction to intermediate packet representation between developers' code and the TCP/IP processing code of iip; the developers' code, through the arguments of functions, passes the pointers to packet objects, whose representation data structure is defined by developers, to the TCP/IP processing code of iip, and iip wraps each of the passed pointers by an iip-specific packet representation data structure. We note that the developers' code does not need to know the content of this iip-specific packet representation data structure, and iip is also agnostic on the packet representation used in the code of the developers. Internally, iip processes packets based on the iip-specific representation. To manipulate packet representation data structures specific to developers' code, iip defines callback functions, to be implemented by developers, that allow iip's TCP/IP processing code to indirectly apply settings, such as the packet data length, to developer-specific packet representation data structures (Chapter 2.3.2).
- **Opaque object.** For the flexibility of callback function development (Chap-

ter 2.3.2), each of the functions defined by the API of iip takes a pointer to an opaque object which can be an arbitrary object instantiated by the developers' code, and when the code of iip invokes a callback function implemented by the developers, the pointer to the opaque object is passed to the callback; the developers can employ the pointer to an opaque object to implement their specific logic in a callback function.

2.3.2 Callback Functions Implemented by Developers

iip assumes that developers implement the following callback functions:

- **Generic memory allocator.** The generic memory allocator, whose functionalities are represented by `malloc` and `free` of the standard C library, depends on platforms because its low-level operations leverage OS-specific system calls, such as `mmap`, and its high-level free space management typically involves CPU-specific atomic operations for accepting concurrent memory allocation requests on multi-core machines; therefore, iip assumes the generic memory allocator is provided by developers. Internally, the TCP/IP processing code of iip employs the provided callback functions to, for example, allocate and release iip-specific packet representation data structures (Chapter 2.3.1).
- **NIC control.** The NIC control, including the capabilities to transmit a packet and access the NIC offloading features, is platform-dependent and, thus, is assumed to be provided by developers.
- **Packet management.** As discussed in Chapter 2.3.1, packet representation can be specific to developers' code; therefore, iip assumes that developers provide callbacks to allocate, release, and manipulate packet representation data structures specific to the code of the developers. Internally, iip's TCP/IP processing code uses these provided callback functions to allocate and manipulate the developer-specific packet representation data structures to be passed to other callback functions implemented by the developers, and release the developer-specific packet representation data structures which are pushed from the developers' code through the functions provided by iip (Chapter 2.3.3). Along with this, as described in Chapter 2.2.5, developers are expected to provide a packet buffer allocator and implement callbacks to access the features of it; the

TCP/IP processing code of iip uses them mainly to allocate packet buffers to put the data to be transmitted and to release consumed packet buffers. These packet management features, to be provided by developers, are assumed to maintain a reference count for each object that they manage so that, for example, multiple packet representation data structures can point to the same packet buffer and it will be released only when no pointer references it anymore; this behavior is particularly needed to allow iip to make shallow copies of a packet by duplicating packet representation data structures pointing to the same packet buffer.

- **Generic utilities.** iip assumes that generic platform-dependent utilities, such as a function to get the current time, are offered by developers through callback functions.
- **Input event handling.** The API of iip defines a series of callback functions that are invoked for different types of incoming events such as accepting a new TCP connection and receiving a TCP payload. Developers can implement application-specific logic, such as parsing a received TCP payload to figure out an embedded request and sending back a TCP payload as the response to the request, in these callback functions.

2.3.3 Functions Provided by iip

iip implements the following:

- **Periodic invocation of the code of iip.** As described in Chapter 2.2.3, iip does not implement a loop to execute its TCP/IP processing code, in other words, it does not maintain a thread to invoke timer-based events such as TCP retransmission timeout. To cope with this issue, iip implements a function assumed to be called by a thread provided by developers at certain intervals; this function requests the time for the next invocation through a variable whose pointer is passed as the argument for this function.
- **Push received packets to iip.** iip offers a function that allows developers' code to push received packets to the TCP/IP processing facility of iip; many of the callbacks for input event handling (Chapter 2.3.2) are called in this function.

- **Operations involving packet transmission.** `iip` offers functions to transmit packets over specific network protocols; for example, sending ARP, ICMP, TCP, and UDP data, trying to establish a TCP connection, and closing a TCP connection. Developers can use these functions to implement application-specific logic. The payload transmission will be conducted in a zero-copy fashion if `iip` finds, using the callback function described in Chapter 2.3.2, that the NIC supports the scatter-gather feature. For a zero-copy payload transmission, internally, `iip`'s TCP/IP processing code first calls the callback function (Chapter 2.3.2) to allocate a new packet buffer, and crafts a packet header on it, then, sends the crafted header with the payload, passed from the developers' code, over the NIC; in this procedure, the header and the payload are put on non-contiguous memory addresses and these two are concatenated by the scatter-gather feature of the NIC at the packet transmission. As seen here, `iip` does not use the adjacent space of the payload for putting its header, therefore, it can avoid the header space contention issue which is described in Chapter 2.1.2.2.

2.4 Limitations

We discuss the limitations of `iip`.

Integration overhead. The typical integration steps for `iip` are to implement the callback functions wrapping the access to platform-specific facilities (Chapter 2.3.2), develop a function to be executed by a thread that pulls packets from a NIC and pushes them to `iip` using the API while calling the function assuming to be invoked periodically to trigger timer-based events (Chapter 2.3.3), and implement application-specific logic in the callbacks for input events (Chapter 2.3.2). We think that engineering for these steps is not too costly because many callback functions just need to proxy the operation requests to underlying platform-specific functions.

Compatibility with existing programs. `iip` does not comply with the networking APIs of major OSes such as the POSIX socket, therefore, it is not compatible with existing programs made for the widely used OSes. A potential approach to obtain compatibility with existing programs is to apply `iip` through a software layer that emulates networking-relevant system calls using a system call hook mechanism such as the `LD_PRELOAD` trick, `zpoline` [63], and `svc-hook` [35].

Maturity of the implementation. iip is made from scratch, therefore, its implementation is less mature compared to the existing legacy TCP/IP stacks. However, we believe that the primary matter is the time spent on development and we can enhance the quality of the iip implementation by continuous improvement.

2.5 Evaluation

This section reports the performance numbers of iip.

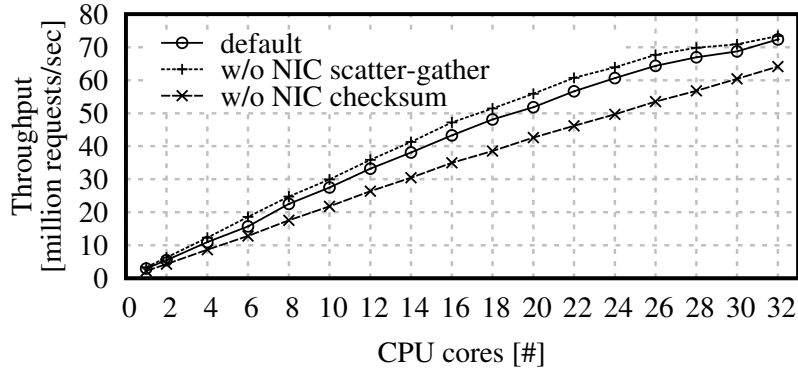
Experiment setup. For the experiments, we use two machines; each machine has two 16-core Intel Xeon Gold 6326 CPUs clocked at 2.90 GHz and 128 GB of DRAM. The two machines are directly connected via Mellanox ConnectX-5 100 Gbps NICs. Both machines install Linux 6.2. The benchmark programs use iip for TCP/IP processing and leverage Data Plane Development Kit (DPDK) [16] to perform packet I/O over the NICs. We refer to a setup, that adopts the unified model (Chapter 2.1.1.3) and activates all of the commonly available NIC offloading features along with zero-copy I/O, as the *default* setup, and we apply it on both machines unless otherwise stated. In all cases, the number of NIC queues is the same as the number of threads executing the networking logic, and we activate the RSS feature of NIC hardware to distribute incoming packets to the multiple NIC queues in an affinity-aware manner (Chapter 2.2.6).

2.5.1 Small Message Exchange

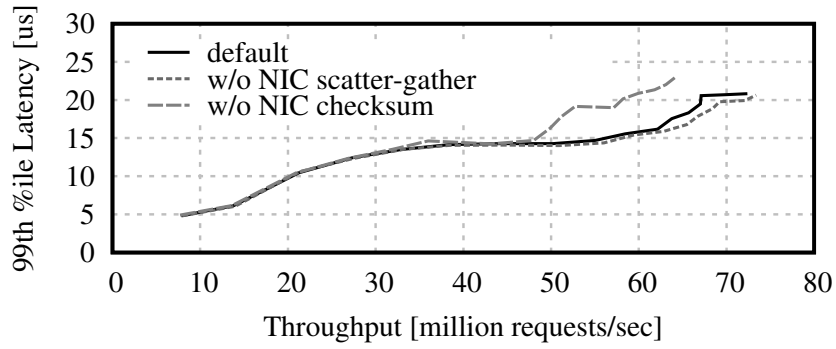
We evaluate the multi-core scalability of iip (Chapter 2.2.6) using a workload that exchanges small messages (Chapter 2.1.2.3).

Benchmark. We run a TCP ping-pong workload; we use one of the two machines to run a pinger process and the other machine executes a ponger process, and they exchange 1-byte TCP payloads as quickly as possible. In this experiment, the pinger process uses 32 CPU cores and adopts the unified model described in Chapter 2.1.1.3 for its CPU core assignment model, and the ponger process also adopts the unified model. We run the benchmark while changing the number of CPU cores assigned to the ponger process.

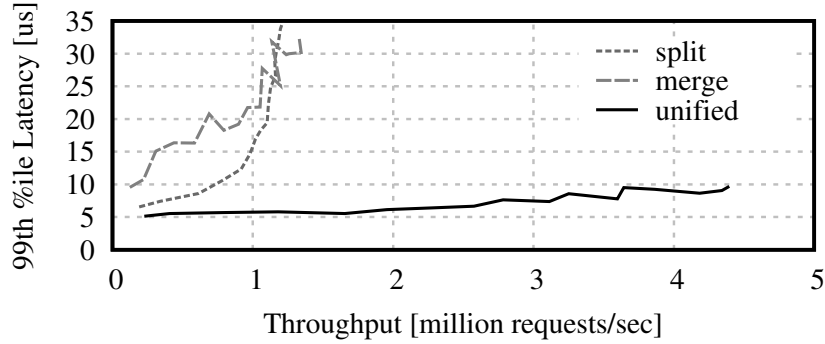
Throughput results. For the throughput measurement, we configure the number of TCP connections established between the pinger and ponger processes so



(a) Throughput according to the number of CPU cores (Chapter 2.5.1)



(b) 99th percentile latency and throughput of the 32 CPU core case (Chapter 2.5.1)



(c) 99th percentile latency and throughput of the three CPU core assignment models (Chapter 2.1.1.3) for the 2 CPU core case (Chapter 2.5.2)

Figure 2.1: Performance for a TCP ping-pong workload which exchanges 1-byte payloads.

that each thread of the ponger process will handle 32 concurrent TCP connections. Figure 2.1a reports the results. In the default setup, iip achieves 2.9 and 72.3 mil-

lion requests per second using 1 CPU core and 32 CPU cores, respectively. When the ponger deactivates zero-copy transmission by disabling the scatter-gather feature of the NIC and configures iip to perform memory copies between the memory regions maintained by the application logic and the packet buffers allocated in the TCP/IP processing code of iip, we observe slightly better throughput compared to the default setup; while this result is not intuitive, a previous study [43] reports the similar trend and explains that the software overhead necessary for the use of the scatter-gather feature of a NIC is more costly than the memory copy particularly when the size of the packet to be sent is small. When we disable the checksum offloading feature of the NIC on the ponger side, we observe 11~26% throughput degradation compared to the default setup case.

Latency results. We have measured the latency to exchange the 1-byte TCP payload for the case where the ponger process has 32 CPU cores while changing the number of TCP connections. Figure 2.1b reports the 99th percentile latency along with the observed throughput. Similarly to the throughput test above, we observe, compared to the default setup, slightly better performance when zero-copy transmission is disabled, and lower performance when the checksum offloading feature of the NIC is deactivated.

2.5.2 CPU Core Assignment Models

We quantify the effect of the CPU core assignment models (Chapter 2.2.3).

Benchmark. We use the same TCP ping-pong workload used in Chapter 2.5.1; we run it with different configurations where the ponger process uses 2 CPU cores and adopts the split and merge models, and compare the default setup adopting the unified model with them. In the cases of the split and merge models, the two threads communicate with each other over message queues. In the split model case, each thread continuously monitors the message queue in its primary loop to detect a new message pushed by the other thread. In the merge model case, the thread executing the application logic enters the sleep state to yield CPU cycles to the thread for networking when it has no data to be processed, and the thread executing the networking logic wakes it up using the event notification mechanism provided by the kernel after pushing a message to the message queue; to ensure that the thread for the application logic can continue to run while it has

the data to be processed, we configure the thread for the application logic to have a higher scheduling priority than the thread executing the networking logic.

Results. Figure 2.1c shows the 99th percentile latency versus throughput results. When the offered load is small, the split model exhibits lower latency compared to the merge model because the split model does not incur the context switch for the transition of the execution of networking and application logic. On the other hand, the merge model achieves higher throughput compared to the split model when the offered load is increased; this is because the merge model does not have the CPU resource boundary which prohibits, in the split model, the networking and application logic from yielding unused CPU cycles to each other. Besides these, the unified model is free from both of the issues, consequently, it achieves the best performance among these three cases.

2.5.3 Bulk Data Transfer

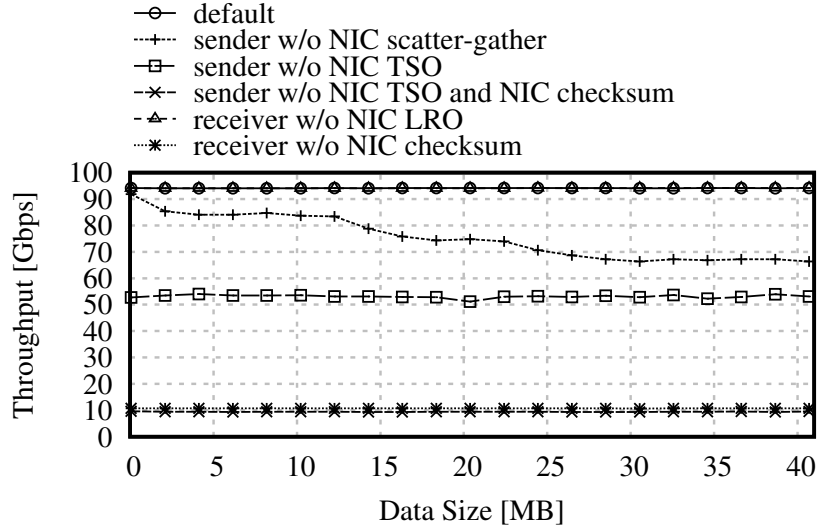


Figure 2.2: TCP bulk data transfer throughput (Chapter 2.5.3).

We examine the effect of each NIC offloading feature (Chapter 2.2.4) through a bulk data transfer workload.

Benchmark. We use one of the two machines as a receiver and the other as a sender, and we use a single CPU core on each machine. These two establish a

single TCP connection; for each case, the sender repeatedly sends a certain size of data, which is located in memory, to the receiver as fast as possible. Along with the default setup, we run this benchmark with five configurations: turning off 1) the scatter-gather feature of the NIC to disable zero-copy transmission on the sender side, 2) TSO on the sender, 3) both TSO and checksum offloading on the sender, 4) LRO on the receiver, and 5) checksum offloading on the receiver side.

Results. The throughput results are reported in Figure 2.2. The default setup almost reaches the NIC bandwidth limit. We found that the effect of zero-copy transmission, enabled by the scatter-gather feature of the NIC, gets bigger when the data size becomes larger; according to this result, we consider that zero-copy transmission mitigates the use of the CPU cache space, consequently, it leads to higher throughput particularly when the size of the data to be transferred is large. We observed that the throughput is almost halved once TSO is disabled, and when the sender additionally disables the checksum offloading feature of the NIC, the throughput goes down to just 10% of the default setup case. We did not see throughput degradation when LRO was turned off on the receiver side; note that this result does not mean that LRO does not contribute to the performance, and this only indicates that the receiver was fast enough without LRO, particularly for this workload. Similarly to the sender case, when the receiver turns off the checksum offloading feature of the NIC, we observe that the throughput goes down to approximately 10% of the default setup case.

2.6 Summary

In this chapter, we have presented the study on the design of a TCP/IP stack named *iip* that aims to allow for easy integration and good performance simultaneously. We have reviewed the issues of the existing performance-optimized and portability-aware TCP/IP stacks and derived the design of *iip* to address the issues. We also have reported the performance numbers of *iip* in workloads that exchange small messages and perform bulk data transfer, along with insights into how performance-critical factors affect communication performance.

Chapter 3

Supporting Multiple Packet I/O Frameworks

The lower layer, namely packet I/O frameworks, often incurs issues that diminish the performance advantages of the middle layer. In this chapter, we discuss how to support multiple packet I/O frameworks to increase the likelihood of achieving good performance by circumventing the issues specific to particular packet I/O frameworks, and present the details of our prototype implementation for `iip` to support three major packet I/O frameworks, DPDK [16], `netmap` [45], and `AF_XDP` [20] along with an abstraction model that hides the differences between their APIs. Besides these, we conduct experiments that investigate how `iip` performs on different packet I/O frameworks, and demonstrate the benefits of a TCP/IP stack implementation that supports multiple packet I/O frameworks.

3.1 Motivation

The motivation of this study is that most previously proposed performance-optimized TCP/IP stacks support only one packet I/O framework and the benefits of a TCP/IP stack implementation that supports multiple packet I/O frameworks have not been well discussed.

Prior implementations. There have been a few prior studies whose prototype implementations offer support for multiple packet I/O frameworks.

- **mTCP.** mTCP [18] is a performance-optimized TCP/IP stack and its publicly

available implementation supports DPDK, netmap, and the PacketShader [10] I/O engine (PSIO). However, the paper of mTCP is based on PSIO and does not include the discussion on the support for multiple packet I/O frameworks.

- **nethuns.** The nethuns [5] work aims to provide a common API that hides the differences of the APIs of different packet I/O frameworks. Nevertheless, the paper of nethuns focuses on the development of network functions and it does not discuss the use for a TCP/IP stack.

This study. As seen above, the benefits of a TCP/IP stack implementation that supports the use of multiple packet I/O frameworks have not been well investigated. This study aims to clarify how support for multiple packet I/O frameworks is useful to a TCP/IP stack implementation for achieving good performance across diverse environments and use cases.

3.2 Technical Background

This section provides the technical background of packet I/O frameworks and TCP/IP stacks running on them.

3.2.1 Overview of Packet I/O Frameworks

Figure 3.1 shows the structures of the major packet I/O frameworks, DPDK [16] (Figure 3.1b), netmap [45] (Figure 3.1c), and AF_XDP [20] (Figure 3.1d). The main difference between the common setup shown in Figure 3.1a and the setups adopting the packet I/O frameworks is that the setups with the packet I/O frameworks do not involve the kernel-space networking subsystem; by bypassing most of the kernel-space networking subsystem, the packet I/O frameworks enable applications in user space to perform high-speed packet I/O.

3.2.2 NIC Device Drivers

As shown in Figure 3.1, NIC device drivers are a foundational component of packet I/O frameworks. Here, we look into the interface that NIC hardware offers to software and how NIC device drivers use it to perform packet I/O over NICs.

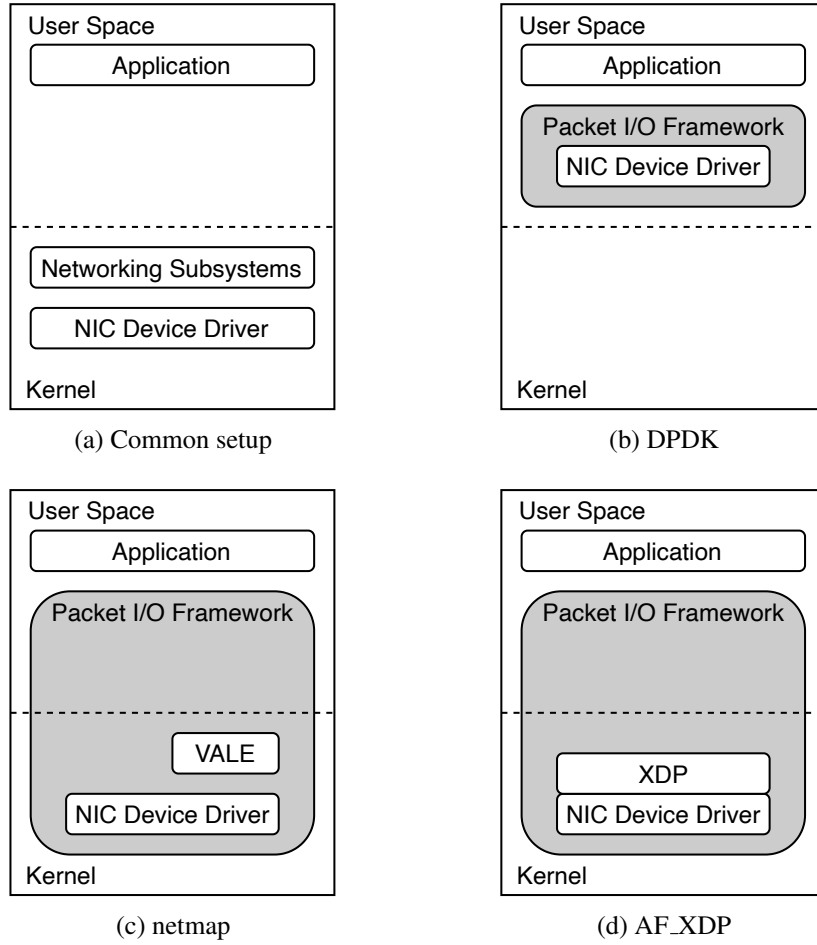


Figure 3.1: The structures of the major packet I/O frameworks (Chapter 3.2.1).

The view from a NIC device driver. Figure 3.2 shows the basic structure of NIC hardware from the view of a NIC device driver. While the implementation details of NIC hardware are vendor-specific, in general, the primary components are NIC registers, packet buffers, and descriptor rings.

- **NIC registers.** During the NIC setup, NIC registers are mapped to a memory region that can be accessed via the CPU instructions for reading and writing in-memory data. Each NIC register has a corresponding part in the memory region, and a NIC device driver reads and writes a particular NIC register by reading and writing the corresponding part of the memory region. Note that the

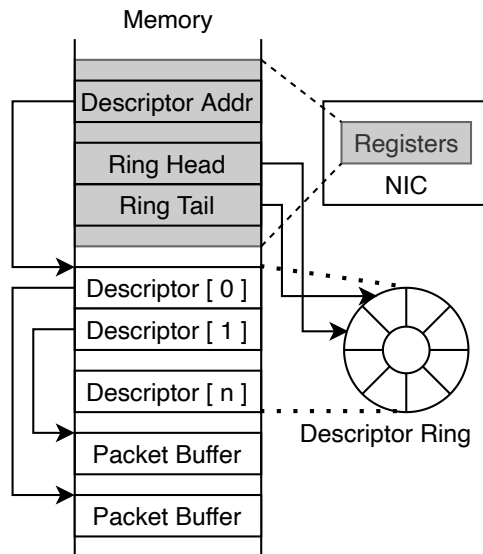


Figure 3.2: The structure of NIC hardware from the view of a NIC device driver (Chapter 3.2.2).

mapping of the NIC registers and their corresponding parts in a memory region is specific to each NIC hardware implementation.

- **Packet buffers.** A packet buffer is a memory region where NIC hardware places received data or software places the data to be transmitted over a NIC. Packet buffers are associated with receive and transmission queues of a NIC.
- **Descriptor rings.** A descriptor is a data structure, which is defined as part of the hardware specification of a NIC, to associate a packet buffer with a NIC queue. A descriptor works as a pointer to a packet and at least maintains two types of information: one is the position of a packet buffer, namely, its memory address, and the other is the length of the data placed on the packet buffer. Typically, the field for the position of a packet buffer is filled by software, namely, a NIC device driver to let the NIC hardware know the memory region that the NIC device driver wishes to use as a packet buffer. On the other hand, the data length field is filled by both software and hardware; the length of received data is set by the NIC hardware, and the length of the data to be transmitted is set by software. From the view of a NIC device driver, a descriptor ring is an ar-

ray of descriptors instantiated in a contiguous memory region. A NIC device driver tells the position of a descriptor ring on the memory to the NIC hardware through a specific NIC register designated for the descriptor ring setting.

Packet I/O. NICs maintain registers to represent the head and tail indexes of a descriptor ring used as a NIC queue. NIC device drivers perform packet I/O by reading and writing the values in them.

- **RX.** When a packet is received at a NIC, the NIC hardware puts it on the packet buffer pointed to by the position field of the descriptor located at the edge of an RX descriptor ring and fills the fields of the descriptor with the information including the length of the received data, and then, updates the index value in the NIC register to extend the length of the NIC receive queue. A NIC device driver reads the updated NIC register value then knows that a new packet has arrived. The NIC device driver traverses the information filled in the descriptor and finds the received data pointed to by the descriptor. After a NIC device driver has consumed the received packet, it updates the index value of the NIC register to shorten the length of the NIC receive queue. Through this NIC register update by the NIC device driver, the NIC hardware knows that the packet has been consumed by software. The descriptor that has pointed to the consumed packet will be reused by the NIC hardware for the reception of another packet when the index of the descriptor ring comes back in the next round.
- **TX.** When a NIC device driver wishes to transmit data on a packet buffer, it associates the packet buffer with a NIC send queue by setting the memory address of the packet buffer to the position field of a descriptor located at the edge of a TX descriptor ring. Afterward, the NIC device driver updates the index value on the NIC register to extend the length of the NIC send queue, and this update of the NIC register lets the NIC hardware know that the software wishes to send a packet; this NIC register update by the NIC device driver initiates the packet transmission. After the NIC hardware has transmitted the packet, it updates the index value of the NIC register to shorten the length of the NIC send queue. The NIC device driver knows that the packet has been transmitted by the NIC when it reads the NIC register value and finds that the length of the NIC send queue has shortened.

3.2.3 Common Features of Packet I/O Frameworks

Packet I/O frameworks commonly have two features.

- **Exposing packet buffers.** One of the two major features exposes packet buffers to the user-space memory address space of a process running an application program. Once this setup has been completed, the program running in user space can read and write data on the packet buffers without the interposition of the kernel. In other words, this setup exempts the program from the memory copies between user- and kernel-space memory regions, which are necessary in the common setups relying on the legacy networking subsystems of the kernel that isolates the packet buffers from user-space programs.
- **API for packet I/O.** The other major feature is to provide an API that allows a program running in user space to detect packets received at a NIC and trigger a packet transmission over a NIC. To provide the API, a packet I/O framework internally leverages its NIC device driver and manipulates the NIC registers representing the head and tail indexes of RX and TX descriptor rings.

3.2.4 Common Usage of Packet I/O Frameworks

The common usage of the features described in Chapter 3.2.3 is as follows.

- **Initialization.** First, a user-space program initializes a packet I/O framework. In this phase, the user-space program leverages the feature mentioned in Chapter 3.2.3 to map the packet buffers to its memory address space.
- **The main loop of a user-space program.** A user-space program adopting a packet I/O framework commonly proceeds with the following steps in its main loop. First, it detects incoming packets by using the API of the packet I/O framework mentioned in Chapter 3.2.3. When the user-space program finds new received packets, it performs its application-specific processing for each of them. If the application-specific processing needs to send a packet over a NIC, it uses the API of the packet I/O framework described above.

3.2.5 Running TCP/IP Stacks in User Space

Packet I/O frameworks enable user-space programs to bypass most of the kernel-space networking subsystems including the TCP/IP stack for packet I/O. As a result, when a user-space program employs a packet I/O framework for its packet I/O, it normally [59] cannot use the TCP/IP stack of the kernel and thus cannot communicate with external hosts over TCP/IP networks. A common approach to circumvent this issue is to run a TCP/IP stack implementation in user space.

Behavior. As mentioned in Chapter 3.2.4, a user-space program leveraging a packet I/O framework proceeds with three steps in its main loop: it first receives packets, executes the application logic, and then sends packets if necessary. In these steps, the TCP/IP stack in the user space is executed as part of the application logic. More precisely, the user-space program carries out the following procedures. The user-space program pushes received packets, which are pulled from the RX queue of the packet I/O framework, to the TCP/IP stack, and the TCP/IP stack performs TCP/IP processing for each of them. Afterward, the user-space program traverses the payloads of the received packets and conducts application-specific processing, such as generating a response to a request. If it requires sending data over TCP/IP, it pushes the data to the TCP/IP stack. The TCP/IP stack performs TCP/IP processing for it and crafts a packet, then transmits it by using the feature of the packet I/O framework.

3.3 Major Packet I/O Framework Implementations

We review the implementations of the three major packet I/O frameworks: DPDK, netmap, and AF_XDP.

3.3.1 Characteristics

We discuss the characteristics of each packet I/O framework.

NIC device drivers in user and kernel space. As shown in Figure 3.1b, DPDK maintains its NIC device drivers in user space. On the other hand, netmap and AF_XDP retain their NIC device drivers in kernel space.

- **User space.** The benefit of DPDK running NIC device drivers in user space

is that a user-space program does not need to interact with the kernel to use the features of a NIC device driver. Consequently, it does not require invoking a system call for triggering packet I/O, and this can contribute to packet I/O performance because an invocation of a system call generally incurs non-negligible performance overhead. A downside of this model is the engineering overhead for the development of NIC device drivers; NIC device drivers have been commonly implemented as part of OS kernels, and thus, many parts of the existing implementations for kernel space cannot be directly reused for the implementations for user space.

- **Kernel space.** The benefit of netmap and AF_XDP using the NIC device drivers in kernel space is that the engineering overhead for the NIC device driver development is usually smaller compared to DPDK because they can use the code base of full-fledged NIC device drivers implemented as part of OS kernels. On the other hand, a user-space program using them requires the communication with the kernel to trigger packet I/O and needs to invoke a system call that imposes performance overhead.

VALE and netmap. As illustrated in Figure 3.1c, netmap maintains a feature named VALE [46]. VALE is a virtual networking infrastructure that is implemented as part of the kernel module of netmap. VALE works as the virtual switch in the kernel, and performs packet forwarding between interfaces that comply with the netmap API.

XDP and AF_XDP. As shown in Figure 3.1d, AF_XDP works on top of a kernel-space functionality called eXpress Data Path (XDP) [13]. XDP implements a hook point in the kernel, and when it is activated, packets received at a NIC go through this hook point. Users can attach a user-provided Berkeley Packet Filter (BPF) [34] program to this hook point and the BPF program can implement custom logic choosing the destination of each packet: the standard kernel-space networking subsystem, an AF_XDP interface exposed to a user-space program, or discarded.

3.3.2 APIs

We review the APIs of DPDK, netmap, and AF_XDP.

3.3.2.1 The API of DPDK

We review the API of DPDK.

Packet representation. The packet representation in DPDK is centered around a data structure named `rte_mbuf`. It maintains a pointer to a packet buffer and the length of the data stored in the pointed packet buffer.

Memory management. DPDK implements the following functions to provide memory management functionality.

- **`rte_pktmbuf_pool_create`.** `rte_pktmbuf_pool_create` is a function that creates a pool of packet buffers that can be associated with a network interface. The packet buffers are instantiated in the memory address space of a user-space program that has invoked this function, meaning that the packet data received at the network interface immediately becomes visible to the user-space program, and the network interface can transmit the data that the user-space program has prepared on one of the packet buffers.
- **`rte_eth_rx_queue_setup`.** `rte_eth_rx_queue_setup` associates a packet buffer pool with an RX queue of a network interface. The underlying subsystem of DPDK uses the packet buffers in the associated pool for receiving packets at the associated network interface.
- **`rte_pktmbuf_alloc`.** A user-space program can allocate the memory for a packet to be sent by a function named `rte_pktmbuf_alloc`. It allocates `rte_mbuf` and a packet buffer, and associates these two together.
- **`rte_pktmbuf_clone`.** `rte_pktmbuf_clone` makes a shallow copy of a packet. It takes a pointer to `rte_mbuf` as its argument, and it newly allocates a memory region for `rte_mbuf` and copies the information from the original one, passed as one of its arguments, to the newly allocated one, and then, increments the reference count for the packet buffer pointed to by the original `rte_mbuf`.
- **`rte_pktmbuf_free`.** `rte_pktmbuf_free` releases the memory for a packet allocated by `rte_pktmbuf_alloc` and `rte_pktmbuf_clone`. It decrements the reference count of the packet buffer pointed to by `rte_mbuf`.

If the reference count of a packet buffer becomes zero, the packet buffer is released, namely, put back into a packet buffer pool.

Packet I/O. DPDK provides the following functions to perform packet I/O.

- **rte_eth_rx_burst.** A user-space program can use `rte_eth_rx_burst` to pull received packets from the underlying subsystem of DPDK. The main functionality of `rte_eth_rx_burst` is to tell the positions of the received packets to a user-space program; since the packet buffers are exposed to the user-space program, this function does not involve memory copies for moving the packet data. `rte_eth_rx_burst` takes an array of pointers to `rte_mbuf` as its argument, and it fills the array with the pointers to `rte_mbuf` objects pointing to received packets.
- **rte_eth_tx_burst.** `rte_eth_tx_burst` is a function to push packets to be sent to the underlying subsystem of DPDK. A user-space program first prepares the data to be sent on packet buffers, and then, makes an array of the pointers to `rte_mbuf` objects pointing to the packet buffers. Afterward, the user-space program calls `rte_eth_tx_burst` while specifying the array of `rte_mbuf` as its argument. Then, the underlying subsystem begins to send the data on the packet buffers specified through the array.

3.3.2.2 The API of netmap

We review the API of netmap.

Packet representation. `netmap_slot` is a data structure that points to a packet. It maintains the pointer to a packet buffer and the length of the data stored in the pointed packet buffer.

Ring buffer. netmap exposes ring buffers, that work as RX and TX queues, to a user-space program. A ring buffer is represented by a data structure named `netmap_ring`. It maintains an array of `netmap_slot` objects comprising a ring buffer along with the head and tail indexes of the ring buffer.

Initialization. During its initialization phase, a user-space program opens a character device implemented by the kernel module of netmap and obtains a file descriptor. This file descriptor is used for the interaction between the user-space program and the kernel-space subsystem of netmap. To request the kernel

to expose the RX and TX queues of a network interface, a user-space program executes the `mmap` system call while specifying the file descriptor obtained from the character device of netmap for one of the arguments of `mmap`. In this `mmap` system call, the kernel-space subsystem of netmap configures a shared memory region between the user-space program and the kernel. The shared memory region maintains `netmap_ring` and `netmap_slot` objects and packet buffers.

Packet I/O. A user-space program can request the kernel-space subsystem of netmap to perform packet I/O operations by passing the file descriptor to system calls like `select` and `poll`, which block the execution of the process and let it yield CPU cycles to other processes while waiting for a new I/O event, and `ioctl` for non-blocking packet I/O.

- **Receive.** A user-space program can request the kernel-space subsystem of netmap to synchronize a `netmap_ring` object exposed to the user-space program and the queue of a network interface by calling one of the system calls mentioned above. Specifically, if there is a packet received at the network interface, the kernel-space subsystem of netmap updates the index value of `netmap_ring`. After the system call returns, the user-space program detects the new packet by checking the updated index value in `netmap_ring`, and finds the received data in packet buffers pointed to by `netmap_slot` whose indexes are between the head and tail maintained by `netmap_ring`. After consuming the received packets, a user-space program updates the index maintained by `netmap_ring` to shorten the RX queue, and in the next invocation of one of the system calls, the kernel-space subsystem of netmap lets the network interface know that the number of the consumed received packets so that it can reuse the descriptors, which have pointed to the consumed packets, to receive new packets.
- **Send.** To send data, a user-space program first puts the data in packet buffers, and associates them with a network interface by configuring `netmap_slot` of `netmap_ring`, and then updates the index maintained by `netmap_ring` to extend the length of the TX queue. Afterward, it calls one of the system calls mentioned above, and the kernel-space subsystem of netmap tells the number of packets to be sent and their positions to the network interface, and triggers the data transmission over the network interface. After the kernel-space sub-

system confirms that the packets are transmitted from the network interface, it updates the index of `netmap_ring` to shorten the length of the TX queue so that the user-space program can reuse `netmap_slot`, which have pointed to the transmitted packets, to point to new packets to be sent.

3.3.2.3 The API of AF_XDP

We review the API of AF_XDP.

Packet representation. `xdp_desc` is the data structure that works as a pointer to a packet. Similar to `netmap_slot`, it maintains the memory address of a packet buffer and the length of the data on the packet buffer.

Ring buffer. AF_XDP defines two data structures to represent ring buffers. One of the two is `xsk_ring_prod` which is for a producer, and the other is `xsk_ring_cons` that is for a consumer. Similarly to `netmap_ring`, each of them maintains the indexes pointing to the head and tail of a ring buffer that consists of a set of `xdp_desc` objects.

Initialization. During its setup phase, a user-space program allocates an AF_XDP-specific socket by using one of the features of the kernel-space subsystem of AF_XDP. Along with this socket allocation, the user-space program tells the memory regions, which the user-space program wishes to use as packet buffers and `xsk_ring_prod` and `xsk_ring_cons` associated with them, to AF_XDP.

Packet I/O. A user-space program and the AF_XDP subsystem exchange the references to packets through the queues represented by the `xsk_ring_prod` and `xsk_ring_cons` data structures.

- **Receive.** The AF_XDP subsystem passes received packets to a user-space program through a ring buffer represented by the `xsk_ring_cons` data structure. The basic concept behind the mechanisms to transfer the information of packets received at a network interface and the number of packets consumed by the user-space program, built atop `xdp_desc` and `xsk_ring_cons`, is essentially the same as that of `netmap` based on `netmap_slot` and `netmap_ring`. The difference from `netmap` is that the AF_XDP subsystem assumes that a user-space program refills packet buffers which can be used for receiving new packets, and for this purpose, the user-space program passes unused packet buffers to AF_XDP through a ring buffer represented by `xsk_ring_prod`.

- **Send.** A user-space program passes packets to be transmitted over a network interface to the AF_XDP subsystem through `xsk_ring_prod`. Similarly to the receive mechanism described above, the basic mechanism for passing the references to packets and sending the requests to trigger packet transmission is essentially the same as that of netmap. The difference from the packet transmission interface of netmap is that the AF_XDP subsystem returns a packet buffer, whose data has been transmitted over a network interface, to the user-space program through `xsk_ring_cons`.

3.4 Discussions on Beneficial Properties

We discuss the beneficial properties of a TCP/IP stack supporting multiple packet I/O frameworks.

3.4.1 Sidestepping Compatibility and Performance Issues

A TCP/IP stack supporting the use of multiple packet I/O frameworks increases the chances for users to circumvent compatibility and performance issues imposed by a specific packet I/O framework.

3.4.1.1 Compatibility Issues

In the experiments for this study, as we describe in Chapter 3.6.1, we needed to use different versions of the Linux kernel for netmap and the other packet I/O frameworks. This is because the source code of netmap maintains patches for the NIC device drivers, which are part of the Linux kernel, and the patches, maintained by netmap, are sometimes not applicable particularly when a newer version of a NIC device driver includes changes that prevent the patch adaptation. As seen in this example, a packet I/O framework may incur compatibility issues and impose restrictions of applicable software and hardware on users. One of the benefits of a TCP/IP stack supporting multiple packet I/O frameworks is that users can have better chances to workaround the restrictions by using one of the packet I/O frameworks that is free from the compatibility issues.

3.4.1.2 Performance Issues

As reported in previous work [36], the performance of a packet I/O framework is significantly affected by the implementation of the adopted NIC device driver. For example, a packet I/O framework may not be able to achieve the ideal performance of a NIC if its NIC device driver does not support the use of NIC hardware offloading features. One of the benefits of a TCP/IP stack supporting the use of multiple packet I/O frameworks is that it increases the chances for users to employ a packet I/O framework having a NIC device driver that can utilize the potential of the NIC.

3.4.2 Compatibility with Existing Systems

Users can sometimes apply efficient setups by using existing systems compatible with only a specific packet I/O framework. One of the benefits of a TCP/IP stack supporting the use of multiple packet I/O frameworks is that it can increase the likelihood that users can apply efficient configurations with existing systems.

3.4.2.1 Using VALE with netmap

As described in Chapter 3.3.1, netmap has a virtual networking infrastructure called VALE. A benefit of VALE is that a VALE-based virtual switch can achieve higher efficiency than a DPDK-based virtual switch, especially for inter-process communication on the same physical machine.

No dedicated CPU cores for executing packet forwarding logic. A common setup to perform inter-process communication over a DPDK-based virtual switch instantiates a process that works as the virtual switch and dedicates at least a single CPU core to it. On the other hand, the packet forwarding logic of a VALE-based virtual switch is executed in the system call context of a sender process, therefore, unlike the DPDK case, the VALE-based inter-process communication setup does not need to instantiate a virtual switch process and thus does not require the CPU cores dedicated to the virtual switch process. In other words, the VALE case requires fewer CPU cores compared to the DPDK case.

3.4.2.2 Using XDP with AF_XDP

As mentioned in Chapter 3.3.1, the AF_XDP feature works with the kernel-space functionality called XDP that allows users to attach a user-made BPF program, which implements packet steering logic, to an in-kernel hook point that received packets go through.

Avoid NIC dedication to a user-space program. This XDP feature allows users to steer specific types of packets (e.g., TCP packets destined to particular TCP ports) to an AF_XDP interface exposed to a user-space program, and the other packets to the standard kernel-space networking subsystem. In other words, users do not need to dedicate a NIC to an AF_XDP-applied user-space program once they implement a BPF program that performs packet steering based on the packet types at the XDP hook point, while DPDK generally requires dedicating a NIC to a DPDK-applied user-space process.

3.5 Implementation for Supporting Multiple Packet I/O Frameworks

We present the implementation to support multiple packet I/O frameworks in iip.

3.5.1 Overview

The overview is illustrated by Figure 3.3.

Callback functions accessing packet I/O features. iip assumes that developers implement the callback functions, defined by the API of iip, to provide iip with access to platform-dependent features, including packet I/O. Specifically, the API of iip assumes two functionalities are provided as the callback functions for NIC-relevant operations.

- **NIC manipulation.** One of the two functionalities is to manipulate the NIC hardware for receiving and sending packets and activate available hardware offloading features of the NIC.
- **Packet management.** The other is the packet management functionality. It implements the memory allocator for packet buffers, which are associated with

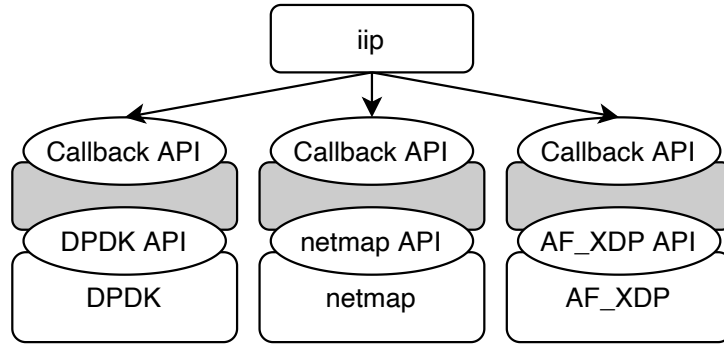


Figure 3.3: The overview of the implementation to support multiple packet I/O frameworks (Chapter 3.5.1).

a NIC and used for placing the received data or the data to be sent. It also defines a packet representation data structure that maintains a reference to a packet buffer and the length of the data located on the referred packet buffer, and provides a memory allocator for the packet representation data structure.

Using multiple packet I/O frameworks with iip. To use DPDK, netmap, and AF_XDP with iip, we implement the callback functions defined by the iip API particularly for the NIC manipulation and packet management functionalities for each packet I/O framework. In Figure 3.3, this implementation is shown by the shaded parts.

3.5.2 Coping with API Differences

To figure out how to cope with the differences of the packet I/O frameworks, we look into the differences in their APIs described in Chapter 3.3.2.

API differences. On one hand, the API of DPDK defines a packet representation data structure named `rte_mbuf`. On the other hand, the netmap and AF_XDP APIs do not define equivalent data structures. netmap and AF_XDP define `netmap_slot` and `xdp_desc`, respectively, to represent a pointer to a packet as a slot of a ring buffer comprising a queue of a NIC. This slot is conceptually the same as the descriptor of a NIC and its design assumes the packet information maintained by a slot, comprising a ring buffer, is overwritten by the information for a new packet in a short time. Therefore, this slot-based packet

representation cannot be used for TCP/IP stacks that, for example, cannot release a TCP packet until an ACK packet is received. In other words, the APIs of netmap and AF_XDP lack the packet management functionality necessary to implement the callback functions defined by the iip API.

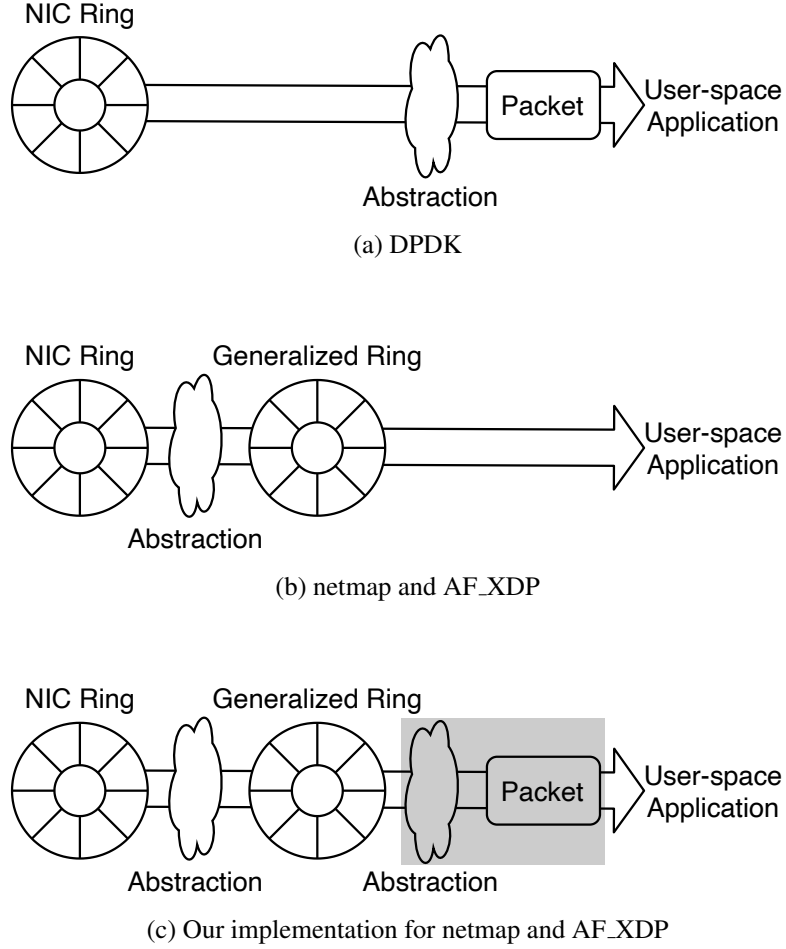


Figure 3.4: Abstraction models of the packet I/O frameworks (Chapter 3.5.2).

The abstraction model issue. We consider that an issue leading to the lack of the packet management feature in netmap and AF_XDP lies in their abstraction model. Figure 3.4 shows the differences in the abstraction models adopted by the DPDK, netmap, and AF_XDP APIs. The DPDK API provides the packet I/O functionality over the packet-oriented abstraction. On the other hand, the netmap

and AF_XDP APIs use the slot-based abstraction to expose a generalized ring buffer, which hides the hardware-specific ring format of the underlying NIC, to a user-space program. We find that the packet-based abstraction adopted in DPDK is more suitable for the development of the callback functions of the iip API, compared to the slot-based abstraction adopted in netmap and AF_XDP.

Introducing a new abstraction layer. To resolve this issue, we introduce a packet-oriented abstraction layer on top of the slot-based abstraction of the netmap and AF_XDP APIs. The additional abstraction layer that we have implemented is shown in the shaded part of Figure 3.4c.

3.5.3 The Additional Packet-oriented Abstraction Layer

The additional abstraction layer, that we have built on top of the netmap and AF_XDP APIs, consists of two functionalities. One is the definition of a packet representation data structure like `rte_mbuf`. The other is the packet management feature like `rte_pktmbuf_alloc`, `rte_pktmbuf_clone`, and `rte_pktmbuf_free` of the DPDK API; it allows users to allocate and release packet buffers which can be associated with a NIC, and packet representation data structure objects that work as the pointers to them.

3.5.3.1 A Packet Representation Data Structure

We define our own packet representation data structure that is conceptually equivalent to `rte_mbuf` of the DPDK API. It maintains a pointer to a packet buffer and the length of the data located in the pointed packet buffer. Along with this data structure definition, we have implemented a memory allocator for allocating and releasing the memory for our packet representation data structure objects. We pay attention to the performance overhead of the memory allocator because it works at least once for each packet.

The performance overhead of the standard memory allocators. The standard memory allocators such as `malloc` of the standard library are made for general purposes, and designed to be able to allocate different sizes of memory regions and accept concurrent requests by multiple threads, therefore, they usually incur the performance overhead deriving from the complexity of their memory allocation algorithms and concurrency control mechanisms.

A simplified memory allocation mechanism. To mitigate the performance overhead of memory allocation, our memory allocator implementation adopts a simple memory allocation mechanism. It instantiates an object pool of our packet representation data structure using a preallocated memory region, and maintains an array-based queue of pointers to the packet representation data structure objects. At the allocation of a packet representation data structure object, it picks up the one located at the top of the array-based queue. A released packet representation data structure object will be returned to the top of the array-based queue. We instantiate the pool and the array-based queue for each CPU core so that we can eliminate in-memory objects shared by threads running on different CPU cores and avoid the complexity of concurrency control; note that iip itself is designed to avoid access to shared in-memory objects.

3.5.3.2 Packet Buffers

DPDK maintains a packet buffer management feature. Once a packet buffer pool is made by `rte_pktmbuf_pool_create`, DPDK internally manages the usage of the packet buffers in the pool, and allocates and releases packet buffers when `rte_pktmbuf_alloc` and `rte_pktmbuf_free` are called. On the other hand, netmap and AF_XDP do not provide the packet buffer management feature whose functionality is equivalent to the one provided by DPDK.

Packet buffer allocator. We build a packet buffer allocator mechanism for netmap and AF_XDP. It creates a pool of packet buffers, which are allocated and exposed to the memory address space of the user-space program by the functionalities of netmap and AF_XDP mentioned in Chapter 3.3.2.2 and Chapter 3.3.2.3, in its setup phase. It also makes an array-based queue to maintain the references to the packet buffers in the pool. Similar to the memory allocator for the packet representation data structure described in Chapter 3.5.3.1, it picks up a packet buffer from the top of the array-based queue, and returns a released packet buffer to the top of the array-based queue. To provide the shallow-copy functionality equivalent to `rte_pktmbuf_clone`, it assumes the cases where multiple packet representation data structure objects refer to a single packet buffer. To determine the release timing of a packet buffer, it maintains a hash table to track the reference count of each packet buffer. The reference count for a packet buffer is

incremented when a packet representation data structure newly refers to it, and decremented when a reference to it is removed. A packet buffer is returned to the pool only when its reference count is decreased to zero.

3.5.4 Implementation of the Callback Functions of the iip API

We present how we have implemented the callback functions defined by the iip API to support DPDK, netmap, and AF_XDP.

Memory allocation. For the callback functions to allocate a packet representation data structure and a packet buffer, we employ `rte_pktmbuf_alloc` for the DPDK case. For the netmap and AF_XDP cases, we use the packet representation data structure and the memory allocation mechanisms presented in Chapter 3.5.3; it allocates a packet buffer and a packet representation data structure object, configures the packet representation data structure object to point to the allocated packet buffer, and then increments the reference count of the packet buffer.

Shallow copy. For the callback function for making a shallow copy of a packet, we use `rte_pktmbuf_clone` for the DPDK case. For the netmap and AF_XDP cases, we leverage the feature of the packet allocator presented in Chapter 3.5.3; to make a shallow copy, it allocates a new packet representation data structure and configures it to refer to a packet buffer pointed to by the copy target, and then, it increments the reference count of the packet buffer.

Memory release. For the callback functions to release a packet representation data structure and a packet buffer, we use `rte_pktmbuf_free` for the DPDK case. For the netmap and AF_XDP cases, we use the memory allocation mechanism presented in Chapter 3.5.3; it decrements the reference count of the packet buffer pointed to by a packet representation data structure object to be released, and if the reference count becomes zero, the packet buffer is released and returned to the packet buffer pool, and afterward, it releases the packet buffer representation data structure.

Packet RX. For the callback function to receive packets, we use `rte_eth_rx_burst` for the DPDK case. In the netmap and AF_XDP cases, this callback function checks the index of an RX ring buffer to detect incoming packets, and if it finds received packets, it allocates a packet representation data struc-

ture object using the feature presented in Chapter 3.5.3 for each of the received packets, and copies the information stored in `netmap_slot` and `xdp_desc` to the corresponding fields of the packet representation data structure; afterward, in the `netmap` case, it allocates a new packet buffer and replaces the pointer in `netmap_slot` with it to dissociate the packet buffer having the received data from the NIC queue, and in the `AF_XDP` case, while it is not necessary to manipulate the pointer field in `xdp_desc`, it allocates new packet buffers and passes them to `AF_XDP` through `xsk_ring_prod` to refill the packet buffers that can be used by `AF_XDP` for receiving new packets.

Packet TX. We use `rte_eth_tx_burst` for the callback function to send packets for the DPDK case. For the `netmap` and `AF_XDP` cases, we have implemented this callback function to first copy the pointer to a packet buffer and the length of the data placed on it, which are stored in a packet representation data structure object passed as its argument, to the corresponding fields of `netmap_slot` or `xdp_desc` of a TX ring buffer, and update the index field of `netmap_ring` or `xsk_ring_prod` to extend the send queue, and then, invoke a system call to tell the kernel-space subsystems to start the packet transmission.

3.5.5 Discussions on Generality

We discuss the generality of our abstraction model presented in Chapter 3.5.3.

3.5.5.1 Other TCP/IP Stacks

We discuss the applicability of our abstraction model to other TCP/IP stacks.

The characteristic of iip. The reason why we have implemented the packet management feature presented in Chapter 3.5.3 for `netmap` and `AF_XDP` is that `iip` performs TCP/IP processing for the data in packet buffers that are allocated by a packet I/O framework and can be associated with a NIC, rather than having memory regions allocated independently of a packet I/O framework, to reduce the memory copies between the packet buffers and other memory regions.

Applicability of our abstraction model. Therefore, our abstraction model will not be necessary for TCP/IP stacks that assume copying the packet data between packet buffers managed by a packet I/O framework and memory regions allocated by themselves. On the other hand, we consider that our abstraction model

is generally applicable to TCP/IP stacks that are designed to reduce memory copy by adopting the same approach as iip.

3.5.5.2 Other Packet I/O Frameworks

We discuss the applicability of our abstraction model to other packet I/O frameworks.

Applicability of our abstraction model. Our abstraction model, which introduces the packet-oriented abstraction layer, will not be necessary for packet I/O frameworks that already provide the packet-based abstraction like DPDK. On the other hand, we consider that our abstraction model can be generally applied to packet I/O frameworks adopting the slot-based abstraction like netmap and AF_XDP which expose generalized ring buffers to a user-space program.

3.6 Experiment

We conduct experiments to examine the efficiency of our implementation presented in Chapter 3.5.4 and investigate the beneficial properties described in Chapter 3.4 along with performance measurements.

3.6.1 Experiment Setup

We have used the following setup for the experiments.

Machine setup. We use a pair of machines. Each machine has two 16-core Intel Xeon Gold 6326 CPUs clocked at 2.90 GHz and 128 GB of DRAM. The two machines are directly connected via Intel X540 10 Gbps NICs; the main reason why we chose the 10 Gbps NICs rather than the 100 Gbps NICs used in Chapter 2.5 is that we found the three packet I/O frameworks have mature device driver implementations for the Intel X540 10 Gbps NIC and could achieve reasonably high performance. In the experiments with DPDK and AF_XDP, we use Linux 6.2, and for the experiments with netmap, we use Linux 5.15. As mentioned in Chapter 3.4.1.1, the reason why we use the different versions of Linux for netmap and the others is that we could not apply a patch set, provided as part of the netmap source code, to a kernel module of Linux 6.2, as of this writing.

Benchmark software. For the experiments, we use a benchmark program that we have made. This benchmark program maintains two features: one is exchanging messages between two hosts and the other is transferring data from one to another. For this benchmark program, we activate the non-blocking mode, which continually consumes CPU cycles to monitor the receive queue of a NIC to detect incoming packets, for DPDK, netmap, and AF_XDP cases.

NIC configuration. Unless otherwise stated, we disable the NIC checksum offloading feature of the NIC and iip's zero-copy transmission feature which relies on the NIC's scatter-gather feature. For the experiments using multiple CPU cores, we instantiate a pair of send and receive queues of the NIC for each CPU core and employ the RSS feature of the NIC to steer packets to the NIC receive queues. We have chosen the values for the interrupt moderation setting to enable netmap and AF_XDP to achieve reasonable performance in all workloads that we use for the experiments: for AF_XDP, we apply 0 us as the delay to invoke a hardware interrupt after the NIC receives a packet, and for netmap, we apply 500 us for this delay setting.

AF_XDP configurations. For the experiments with AF_XDP, we employ the XDP-aware NIC device driver implementation. We activate the feature that enables an AF_XDP-applied user-space program to avoid memory copies for packet data between user and kernel space. We activate the mode that triggers packet I/O of AF_XDP through an explicit notification sent by an AF_XDP-applied user-space program. We enable the busy-poll socket option for the socket of AF_XDP.

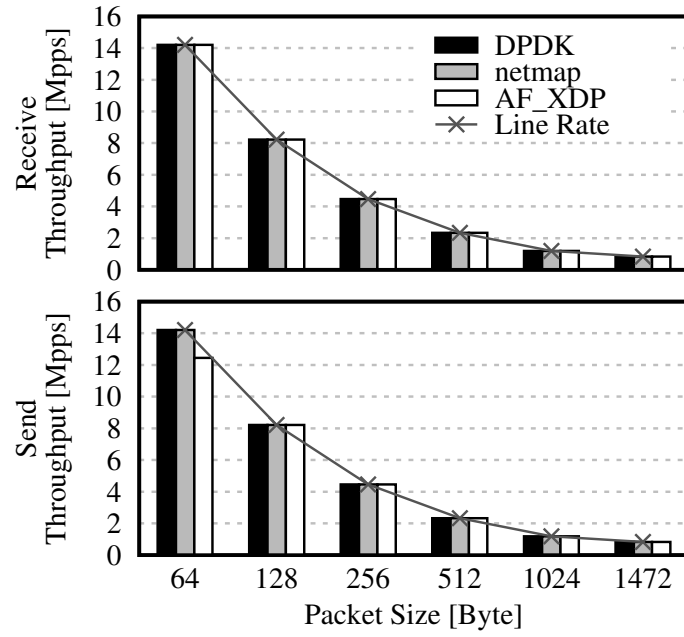
3.6.2 Packet I/O Performance

We run experiments to quantify the packet I/O performance of our implementation described in Chapter 3.5.4.

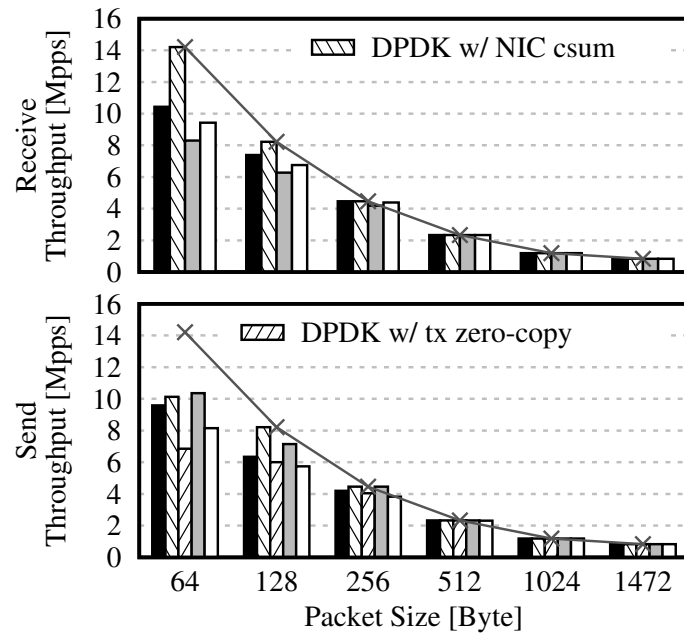
3.6.2.1 Packet I/O without iip

We report the packet I/O performance that does not include the processing of iip.

Benchmark. We have implemented a packet generator program that does not involve iip and directly leverages our implementation of the packet I/O infrastructure. Specifically, in the receive benchmark, the benchmark program uses one of the callback functions to allocate a packet representation data structure for each



(a) Packet I/O without iip



(b) UDP with iip

Figure 3.5: Packet I/O performance (Chapter 3.6.2).

packet buffer having received data, employs another callback function to associate them, and then calls a callback function to release the memory regions occupied for the received packet. In the send benchmark, the benchmark program calls a callback function to make a shallow copy of a preallocated UDP packet and then calls a callback function to transmit it over the NIC.

Results. Figure 3.5a shows the number of packets received on the receiver side in one second. In most cases, we have observed the NIC’s maximum throughput, indicated by the line labelled as Line Rate. Based on these results, we consider that the performance of our implementation is acceptably high.

3.6.2.2 UDP with iip

We measure the UDP send and receive performance using iip. The difference from the experiment in Chapter 3.6.2.1 is that the benchmark program executes iip to parse the UDP header of each received packet in the receive benchmark, and craft a UDP header for each packet to be transmitted in the send benchmark.

Benchmark. We use the same benchmark program on the two machines. We employ one of the two as the target machine and the other as the load generator machine. On the load generator machine, the benchmark program always leverages DPDK for its packet I/O; it uses two CPU cores to send UDP packets for the receive benchmark, and leverages one CPU core to receive UDP packets for the send benchmark. The target machine uses one CPU core for both receiving and sending UDP packets.

Results. Figure 3.5b shows the results. Compared to the results shown in Chapter 3.6.2.1, we observed lower throughput numbers particularly in small packet size cases because of the UDP packet processing overhead of iip. In both receive and send cases, we find that the checksum offloading feature of the NIC contributes to the performance. On the other hand, when the zero-copy transmission feature of iip is activated, we observe lower throughput numbers compared to the case where the zero-copy transmission feature is deactivated; the same trend has been reported in Chapter 2.5.1, and we consider this performance reduction derives from the scatter-gather feature of the NIC.

3.6.3 Exchanging Small Messages over TCP Connections

We measure the performance for a workload that exchanges small messages; this workload is represented by web and cache servers. For this measurement, we use one of the two machines as the target machine and the other as the load generator machine. The benchmark program on the load generator machine always uses DPDK for its packet I/O.

3.6.3.1 Using a Single CPU Core

We conduct experiments using a single CPU core on the target machine.

Benchmark. The process executing the benchmark program on the target machine uses a single CPU core. In the cases where the number of TCP connections established between the target and load generator machines is less than 16, the number of CPU cores used by the process running the benchmark program on the load generator machine is the same as the number of TCP connections, and otherwise, 16 CPU cores are used.

Results. Figure 3.6a shows the results; for throughput results, a higher value indicates better performance, and for 99th percentile latency results, a lower value indicates better performance. Similarly to the trend observed in Chapter 3.6.2.2, the checksum offloading feature of the NIC contributes to the communication performance, and we find that the zero-copy transmission feature built atop the scatter-gather feature of the NIC reduces the throughput. When both the checksum offloading feature and the zero-copy transmission feature are deactivated, the performance gap between the DPDK and netmap cases was minor. On the other hand, we have observed that the performance of AF_XDP is lower compared to the DPDK and netmap cases. We consider that the performance of the setup using AF_XDP can be improved by future updates of the Linux kernel since its architecture is close to that of netmap.

3.6.3.2 Using Multiple CPU Cores

We conduct experiments using multiple CPU cores.

Benchmark. We run the experiments while changing the number of CPU cores assigned to the process running the benchmark program on the target machine. The process executing the benchmark program on the load generator ma-

chine always uses 16 CPU cores. We configure the number of TCP connections established between the benchmark programs on the target and load generator so that each CPU core of the target machine will handle 64 TCP connections.

Results. Figure 3.6b shows the results. When the benchmark process on the target machine uses a few CPU cores, we observe a similar trend to the single CPU core case presented in Chapter 3.6.3.1. Additional CPU cores generally contribute to the performance. On the other hand, in the DPDK case, we find that the performance is limited by the zero-copy transmission feature which is based on the scatter-gather feature of the NIC.

3.6.4 Bulk Data Transfer over a TCP Connection

We measure the performance for a workload that transfers data from one to another over a TCP connection; this workload is typically seen in file servers. We investigate how the differences of the underlying packet I/O frameworks, the hardware offload features of a NIC, and the zero-copy I/O capability affect the performance in this workload.

Benchmark. For this experiment, we run the benchmark program on the two machines, and one of the two sends data to the other as fast as possible, through a single TCP connection. On one of the two machines, the benchmark program always uses DPDK for its packet I/O and activates the checksum offloading feature of the NIC. On the other machine, we run the benchmark program while changing the underlying packet I/O framework and its configurations. On both machines, a process running the benchmark program uses a single CPU core.

Results. Figure 3.7 shows the results.

- **Receive performance.** Regarding the receive performance, we have observed mostly the same throughput for all cases. We find that the checksum offloading feature of the NIC does not affect the performance in the environment that we have used for this experiment.
- **Send performance.** Regarding the send performance, we have observed that in all cases, the throughput numbers are higher than 8 Gbps. We find that the checksum offloading feature of the NIC improves the throughput. On the other hand, in the DPDK case, we find the improvement made by the zero-copy

transmission feature is minor; we consider this is due to using 10 Gbps NICs instead of 100 Gbps NICs as mentioned in Chapter 3.6.1, and based on the report for the experiment using 100 Gbps NICs in Chapter 2.5.3, the effect of the zero-copy transmission feature will be more significant if the throughput limit is higher than 10 Gbps.

3.6.5 Configurations with Existing Systems

We investigate the performance merits brought by existing systems compatible with a specific packet I/O framework, which are mentioned in Chapter 3.4.2.

3.6.5.1 Using netmap and VALE

We apply the configuration mentioned in Chapter 3.4.2.1 and measure the inter-process communication performance of two processes using a VALE-based virtual switch for their communication on the same machine.

Benchmark. To have the performance numbers of the ideal cases, we assign a CPU core to each of the two processes that perform communication over the VALE-based virtual switch. On the other hand, for the experiments using DPDK, we launch a process to execute a program called `l2fwd` which is distributed as part of the DPDK source code and works as a virtual switch, and we assign a single CPU core to the process running the `l2fwd` program. Therefore, the experiments for DPDK use three CPU cores, and the experiments for netmap use two CPU cores. For the experiments with DPDK, we put the `l2fwd` process in between the two processes performing the communication, and the `l2fwd` process and each of the two processes communicate over a `memif` interface that is a DPDK-specific virtual interface enabling one-to-one communication over shared memory.

Results. The results are reported in Figure 3.8.

- **Packet I/O performance.** Figure 3.8a shows the results of the same experiments as the ones shown in Figure 3.5a (Preallocated Packets) and Figure 3.5b (UDP with `iip`). In the Preallocated Packets case, the netmap setup exhibits higher performance compared to the DPDK setup. In the UDP with `iip` case, the DPDK setup slightly outperforms the netmap setup.

- **TCP communication performance.** Figure 3.8b shows the results of the same experiments as the one shown in Figure 3.6a. The reason why the performance of the DPDK setup is low in the 1 to 16 concurrent TCP connection cases is that the l2fwd program defers the packet forwarding until the length of its receive queue reaches 32 and waits for a preconfigured time; we note that this performance issue can be resolved by minor changes to the implementation of the l2fwd program, and thus, the results of the 1 to 16 concurrent TCP connection cases for l2fwd do not reflect the I/O performance of DPDK. On the other hand, in cases where 32 or more TCP connections are established and the DPDK setup is expected to achieve reasonable performance, we observed no significant performance difference between the DPDK and netmap setups.

Here, we see that a TCP/IP stack supporting the use of netmap offers users the option to adopt VALE for CPU-efficient inter-process communication.

3.6.5.2 Using AF_XDP and XDP

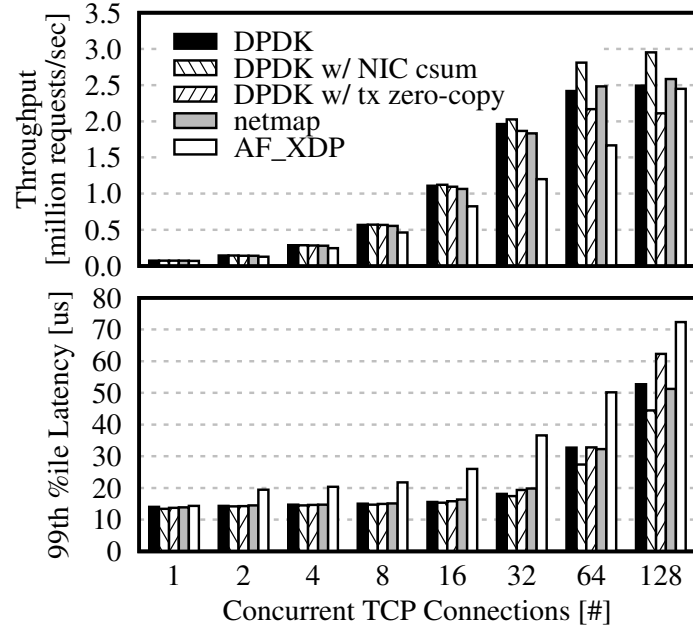
We apply the configuration explained in Chapter 3.4.2.2 and investigate how it affects the networking performance.

Benchmark. We have implemented a BPF program that pushes all received packets to the standard kernel-space network subsystem except the packets destined to TCP ports used by an AF_XDP-applied program, so that the AF_XDP-applied program and the standard kernel-space networking subsystem can share the same network interface. We run the same benchmark performed in Chapter 3.6.3 to examine the performance overhead of this configuration.

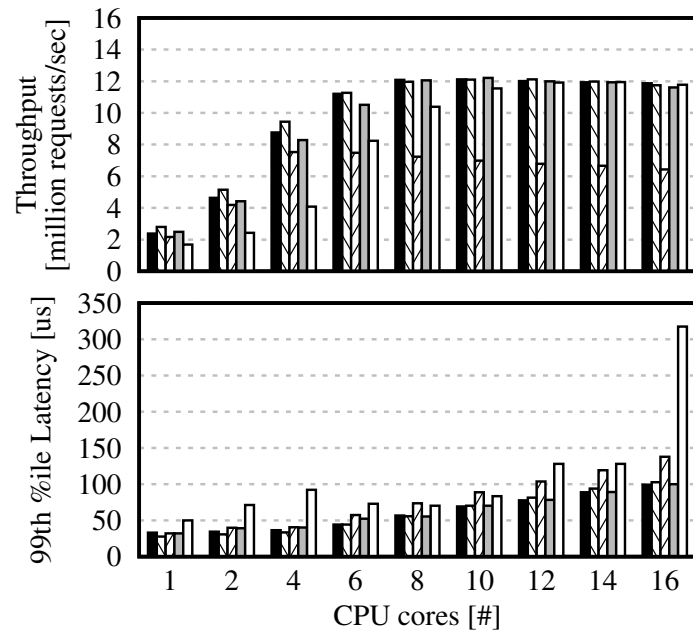
Results. Figure 3.9 shows the results. We note that the performance numbers for the cases where the network interface is dedicated to the AF_XDP-applied program are the same as those reported in Figure 3.6. We find that the performance overhead introduced by the BPF program is not significant. This means that a TCP/IP stack supporting the use of AF_XDP provides users with an efficient option to leverage the same network interface for an AF_XDP-applied program and other services backed by the standard kernel-space networking subsystem.

3.7 Summary

In the study presented in this chapter, we have discussed how supporting multiple packet I/O frameworks benefits a TCP/IP stack implementation in achieving good performance across diverse environments and use cases. We have presented the details of our prototype implementation for iip to support DPDK, netmap, and AF_XDP, and conducted experiments that evaluate our implementation and examine the beneficial properties.



(a) Using a single CPU core



(b) Using multiple CPU cores

Figure 3.6: Performance for exchanging small messages over TCP (Chapter 3.6.3).

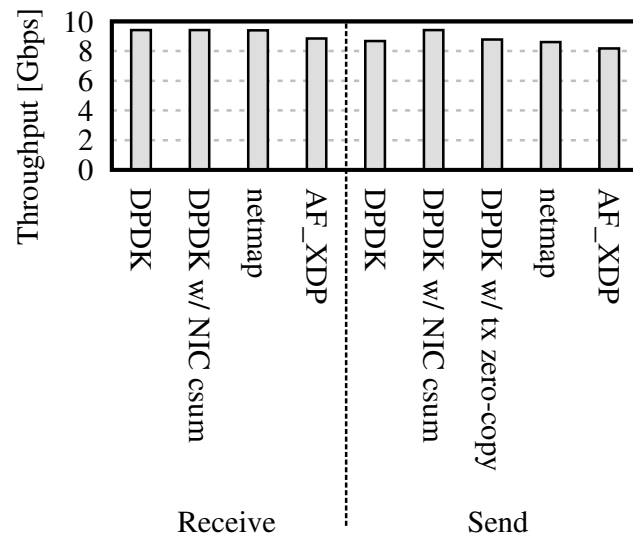
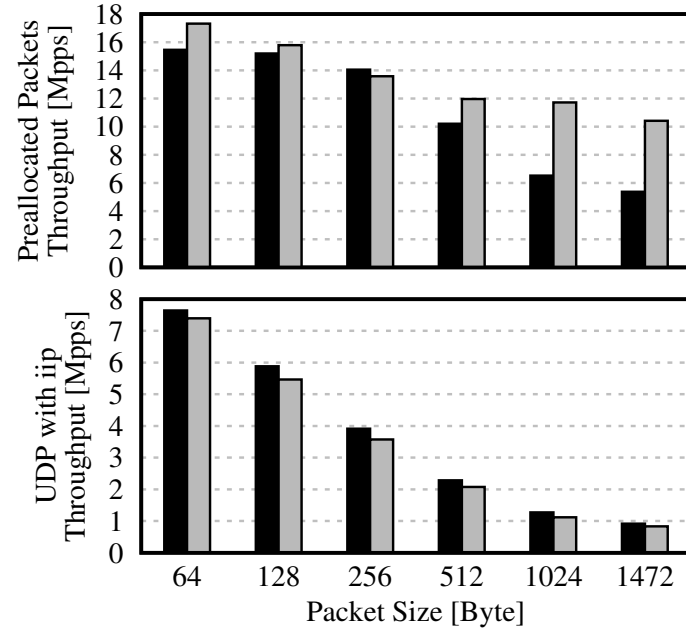
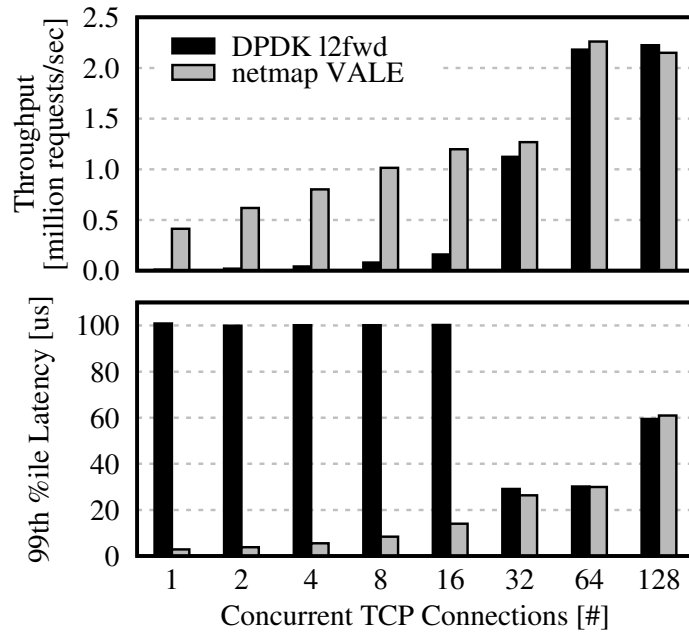


Figure 3.7: Bulk data transfer performance over TCP (Chapter 3.6.4).

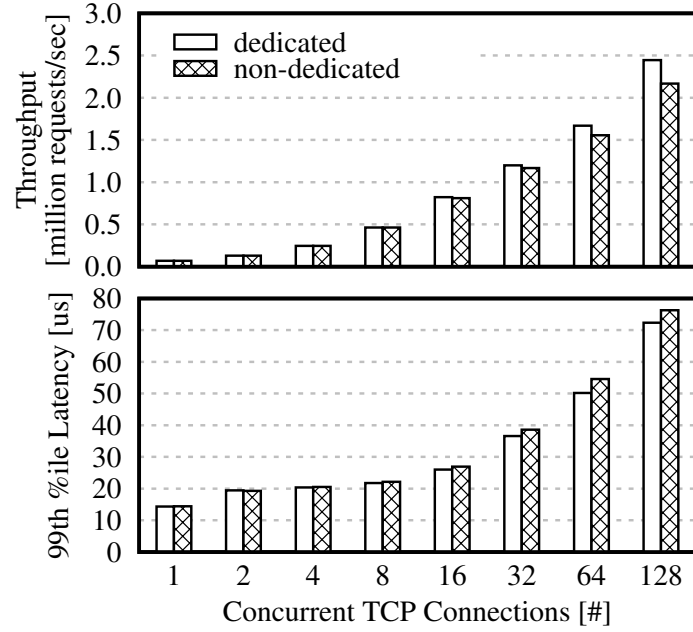


(a) Packet I/O

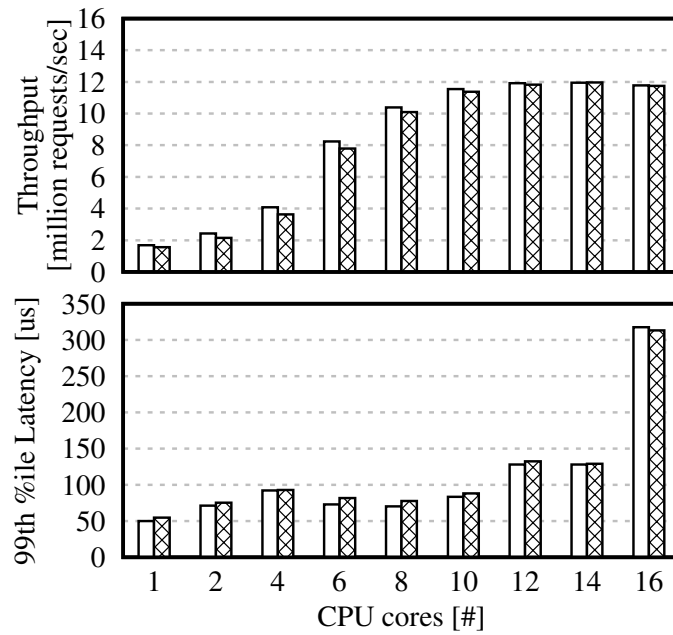


(b) TCP

Figure 3.8: Performance of inter-process communication over a VALE-based virtual switch (Chapter 3.6.5.1).



(a) Using a single CPU core



(b) Using multiple CPU cores

Figure 3.9: Using XDP to avoid NIC dedication (Chapter 3.6.5.2).

Chapter 4

Restricting Memory Accesses

This chapter focuses on an upper layer that implements the logic for specific use cases, and specifically, we build a memory access restriction mechanism as an upper-layer subsystem to improve memory safety. We explore opportunities to mitigate the performance overheads imposed by an upper-layer subsystem and implement techniques that moderate the performance reductions caused by our memory access restriction mechanism.

4.1 Motivation

The motivation of this study is to improve the memory safety for use cases where a process runs in user space. We implement the feature for enhancing memory safety as an upper-layer so that it can be transparently deployed without adding dependencies to the middle and lower layers.

4.1.1 Memory Accesses Unintended by Developers

In general, it is difficult to eliminate the possibility that a program performs memory accesses that are not intended by its developers.

Detecting unintended memory accesses by page table settings. Ideally, memory accesses, which are not intended by developers, should be detected so that appropriate actions, such as stopping the process, can be performed. The widely used general-purpose OSes configure the memory regions that a user-space

program can access through the page table setting, and a process that violates the memory access setting applied through the page table and caused a segmentation fault is generally stopped.

The limitation of the common page table feature. However, memory access restrictions applied through a page table do not always cause a segmentation fault for memory accesses that are not intended by developers. If a user-space program performs a memory access to a memory region that the page table setting allows the user-space program to access, a segmentation fault does not occur and the user-space program continues to run, even though it is not intended by the developers of the user-space program.

4.1.1.1 How Unintended Memory Accesses Occur

We present an example that leads to memory accesses unintended by developers.

Out of bounds. An out-of-bounds access to an array occurs when a program attempts to access an item in an array by an index that exceeds the length of the array. Particularly, if a user-space program performs an out-of-bounds access to an array and touches a memory region that the page table setting allows the user-space program to access, a segmentation fault does not occur.

Difficulty for memory debuggers. It is often not too difficult for memory debuggers to find an implementation performing an out-of-bounds access to an array when the length of the array is statically defined in the source code. On the other hand, it is complicated to detect an out-of-bounds access to an array whose length is not statically defined in the source code and the memory region for the array is dynamically allocated at runtime. We consider that the implementations, which decide the lengths of arrays at runtime, are often seen in programs that are aware of their memory usage and implemented to flexibly resize the sizes of the arrays depending on the underlying platforms.

An example in the netmap API. For instance, we can find an array, whose length is decided at runtime, in the netmap API. Figure 4.1 shows the definition of `netmap_ring` that maintains an array of `netmap_slot` objects whose length is not defined in the source code. The syntax above is known as a flexible array member that allows programmers to define an array without specifying its length. In the definition shown in Figure 4.1, the length of the array of `netmap_slot`

```
struct netmap_ring {  
    ...  
    const uint32_t    num_slots;  
    ...  
    struct netmap_slot slot[];  
};
```

Figure 4.1: Part of the definition of `netmap_ring` (Chapter 4.1.1.1).

objects is stored in the `num_slots` field, and the memory allocator allocates enough space subsequent to the memory region having the `netmap_ring` object to accommodate the array of `netmap_slot` objects whose length is set to the `num_slots` field. If a user-space program, using `netmap`, performs an out-of-bounds access, not intended by its developers, to the array of `netmap_slot` objects and the accessed memory region is configured to be accessible to the user-space program, a segmentation fault does not occur and the user-space program will continue to run. We note that arrays whose lengths are decided at runtime are commonly used and not specific to the `netmap` API.

4.1.1.2 Consequences of Unintended Memory Accesses

Memory accesses not intended by developers can lead to undesirable consequences if they are not appropriately detected and handled. For example, if an unintended memory access overwrites a memory region that has the data to be written to a file and the data is written to a persistent storage device, the content of the file will differ from what the developers expect. If a user-space program performs an unintended memory access and reads the data it should not and the data is sent over the network, an unintended information leak can occur.

4.1.1.3 The Code Size Increased by TCP/IP Stacks

Figure 4.2a illustrates the traditional model that places a TCP/IP stack implementation in kernel space. Figure 4.2b shows the model that puts a TCP/IP stack in user space and uses a packet I/O framework such as DPDK [16], `netmap` [45], and `AF_XDP` [20] for its low-level packet I/O; note that the functionalities imple-

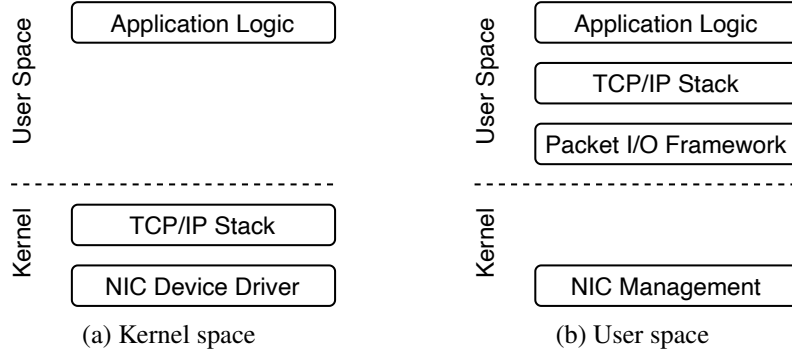


Figure 4.2: TCP/IP stack placement in kernel and user space (Chapter 4.1.1.3).

mented in the packet I/O framework component in user space and the NIC management component in kernel space depend on the implementation of a packet I/O framework, and for example, the kernel-space NIC management facility for DPDK just exposes the NIC device registers to a user-space program and the NIC device drivers, which manipulate them to perform packet I/O, are provided by DPDK and executed in user space, on the other hand, the kernel-space NIC management facility for netmap and AF_XDP includes NIC device drivers.

The code size issue. The model that places a TCP/IP stack in user space (Figure 4.2b) increases the size of the code executed in user space. The increased code expands the possibility that a process running a TCP/IP stack implementation in user space performs memory accesses that are not intended by developers.

4.1.2 Goal

The goal of this study is to provide a mechanism that applies intra-process memory access restrictions for reducing the possibility that memory accesses, which are performed by a process running a TCP/IP stack implementation in user space and not intended by developers, lead to undesirable consequences like the examples mentioned in Chapter 4.1.1.

Technical goal. The technical goal of this study is to build a mechanism that causes a segmentation fault and enables actions such as stopping the process that has caused the segmentation fault when iip, running in user space, attempts to access memory regions it should not or when implementations other than iip try

to access the memory regions dedicated to iip.

Using MPK. To this end, we employ a hardware feature called MPK. MPK provides a set of facilities enabling intra-process memory access restrictions that can cause segmentation faults and, as we discuss in Chapter 4.3, its performance overhead is lower than that of other hardware features that can be used for intra-process memory access restrictions. This study focuses on how to achieve the technical goal using the MPK feature.

4.1.3 Out of Scope

We note that security enhancement is not the goal of this study.

Security mechanisms built atop MPK. As we discuss in Chapter 4.3.1, there have been several prior proposals leveraging MPK to apply memory access restrictions to TCP/IP stack implementations. Their goal is to build robust security mechanisms on top of MPK. However, the widely used OS kernel does not assume that MPK is used as the foundation of a security mechanism. Consequently, there are various potential attacks that exploit the facilities of the kernel. To prevent these attacks, the implementations of the prior proposals require non-trivial changes to the kernel source code and thus diminish the handiness of their proposed security mechanisms.

The difference between prior security mechanisms and this study. Undesirable consequences caused by memory accesses are not only initiated by attacks but also by mistakes made by developers. We believe that a mechanism, which mitigates the possibility that memory accesses unintended by developers cause undesirable consequences, is generally useful particularly if it is easy for users to deploy. Therefore, this study explores a handy mechanism that does not require changes to the source code of the kernel while it does not aim to improve security.

4.2 The Memory Access Restriction Mechanism

This section presents our memory access restriction mechanism aiming to achieve the goal presented in Chapter 4.1.2.

4.2.1 Overview

Figure 4.3 shows the overview of our memory access restriction mechanism. It establishes an intra-process isolation boundary between a TCP/IP stack and the implementations other than the TCP/IP stack; the isolation boundary is shown by the shaded part in Figure 4.3. Once this mechanism is applied, a segmentation fault occurs when the TCP/IP stack attempts to access the memory regions it should not or when implementations other than the TCP/IP stack try to access the memory regions dedicated to the TCP/IP stack.

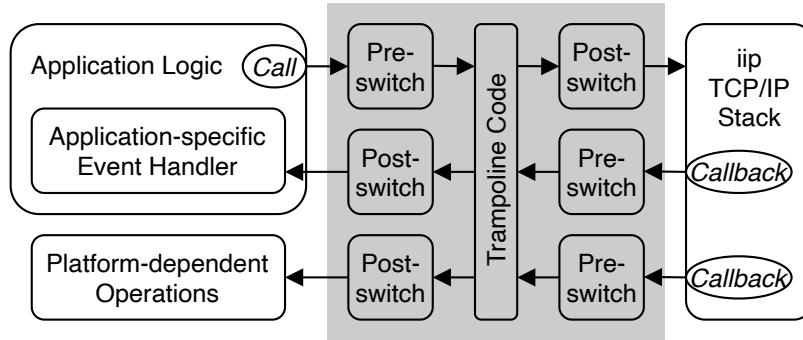


Figure 4.3: The overview of our memory access restriction mechanism (Chapter 4.2.1).

4.2.1.1 Causing Segmentation Faults by MPK

To realize memory access restrictions that can cause segmentation faults, we employ the MPK feature; the reason why we choose MPK is that, as we discuss in Chapter 4.3, its performance overhead is lower compared to other hardware features that can be used for building intra-process isolation boundaries.

MPK. MPK enables software to implement memory access rights that are applied to a CPU core, and, for each page, the software can apply settings that specify the memory access rights required for access to it. The MPK feature also allows software to switch the memory access right currently applied to the CPU core, at runtime. MPK causes a segmentation fault when software attempts a memory access to a page that is not allowed for the currently applied memory access right. We review the technical details of the MPK feature in Chapter 4.2.2.

4.2.1.2 Consequences of Segmentation Faults

In general, when a user-space program causes a segmentation fault, the kernel stops the process executing it.

A handler for a segmentation fault. On the other hand, if a user-space program implements a handler for a segmentation fault, the kernel does not stop the process running it, and the kernel delegates segmentation fault handling to the user-space program. OS kernels often provide user-space programs with APIs to register segmentation fault handlers.

Segmentation fault handling on Linux. For example, on Linux, user-space programs can use the `sigaction` system call to register segmentation fault handlers with the kernel. The kernel provides a function registered as the segmentation fault handler with the information about a segmentation fault including the memory address that has caused the segmentation fault and the values on the CPU registers that have been captured by the kernel at the segmentation fault, through the arguments of the segmentation fault handler function. The handler can potentially implement the procedures to figure out the issues that have caused the segmentation faults based on the information provided by the kernel and perform recovery so that the user-space programs can continue to run.

Our mechanism does not provide a segmentation fault handler. We note that our memory access restriction mechanism does not provide the implementation of a segmentation fault handler because each user-space program requires a specific implementation of the segmentation fault handler to appropriately recover from a segmentation fault.

4.2.1.3 The TCP/IP Stack Implementation for Our Prototype

In this study, we use `iip` for the TCP/IP stack of our prototype implementation.

The benefit achieved by portability. For its portability, `iip` accesses the features of external facilities including a memory allocator only through the callback functions specified as part of its API. This characteristic makes it straightforward to track the memory regions that can be accessed by `iip` and apply fine-grained memory access restrictions to a process running it.

Other TCP/IP stacks. The same characteristic is seen in other portable TCP/IP stacks such as `lwIP` [8], and we consider that the design of our memory

access restriction mechanism can be applied to TCP/IP stacks sharing the same portability property.

4.2.1.4 Adaptation

Our memory access restriction mechanism applies memory access settings using MPK so that different memory access rights will be required to access the memory regions dedicated to iip and the memory regions that iip should not access, and it switches the memory access right applied on the CPU core at the invocation of the iip API to realize the isolation boundary shown in the shaded part of Figure 4.3.

Existing implementations and our mechanism. Our memory access restriction mechanism is built as a pluggable extension to existing systems. In Figure 4.3, our memory access restriction mechanism is located only in the shaded part, and the other part is the standard setup of an iip-applied user-space program.

Three layers. Our memory access restriction mechanism consists of the following three layers.

- **Pre-switch functions.** The first layer is pre-switch functions which are the designated entry points for existing systems to go across the isolation boundary.
- **Trampoline code.** The second layer is the trampoline code that is called by a pre-switch function and performs the memory access right switching. The trampoline code preserves a set of CPU register values before switching the memory access right so that it can restore the CPU state, including the previously applied memory access right, when returning to its caller, namely, a pre-switch function.
- **Post-switch functions.** The third layer is post-switch functions that reshapes the arguments, which are passed to the pre-switch functions and modified because of the restrictions imposed by the memory access right transition mechanism described in Chapter 4.2.5, to comply with the argument specification format of a function that is called after the memory access right switching performed by the trampoline code.

Adaptation through function call replacement. Our memory access restriction mechanism is adapted to existing systems through function call replacement

so that it can be pluggable. Specifically, we employ the macro feature of the C programming language to replace the functions provided by iip and the callback functions implemented by developers, which are defined as the iip API, with the pre-switch functions of our memory access restriction mechanism. After this function call replacement has been completed, an invocation of the iip API is redirected to a pre-switch function of our memory access restriction mechanism; it calls the trampoline code that switches the memory access right applied to the CPU core, and afterward, the trampoline code calls a post-switch function of our memory access restriction mechanism, and it calls the original function that has been replaced with the pre-switch function while reshaping the arguments passed to the pre-switch function to comply with the argument specification format of the original function. We note that while we have employed the macro feature of the C programming language for our prototype, this adaptation model can be applied if we could use a mechanism enabling function call replacement, and in other words, it does not depend on the C programming language.

4.2.1.5 Behavior

We present how our memory access restriction mechanism changes the behavior of existing systems.

Receive. A user-space program pulls received packets from a packet I/O framework and calls a function provided by iip to pass the received packets to iip. When our memory access restriction mechanism is applied, this function is replaced with a pre-switch function provided by our memory access restriction mechanism; it calls the trampoline code to switch the memory access right and afterward the trampoline code calls the post-switch function that calls the function provided by iip to push the received packets to iip while reshaping the arguments passed to it.

Application-specific logic. Then, iip performs TCP/IP processing for each of the received packets and calls a developer-provided callback function which corresponds to the type of the received packet. This callback function is also replaced with a pre-switch function of our memory access restriction mechanism, and it calls the developer-provided callback function after going through the trampoline code and a post-switch function. Developer-provided callback functions

implement application-specific logic such as parsing the received TCP payload to extract an application-level request and generate and send back a response over the TCP connection.

Send. To send data using `iip`, the application-specific logic calls a function provided by `iip` to pass the data to be sent to `iip`. This function is replaced with a pre-switch function of our memory access restriction mechanism, and it calls the function of `iip` after going through the trampoline code and a post-switch function. Afterward, `iip` performs TCP/IP processing and crafts an appropriate header for the data to be sent, and then, it calls a developer-provided callback function that accesses the feature of an underlying packet I/O framework to trigger the packet transmission by going through the pre-switch function, the trampoline code, and the post-switch function of our memory access restriction mechanism.

4.2.2 Memory Protection Keys (MPK)

We review the functionalities of MPK. Note that we discuss particularly the MPK implementation of Intel CPUs.

Hardware facilities. A CPU core having the MPK support equips a 32-bit CPU register called PKRU, and implements CPU instructions called RDPKRU and WRPKRU to read and write the value on the PKRU register, respectively. Besides these, a particular 4-bit field in a page table entry is interpreted as the setting for the MPK feature.

Software-side implementation. Through the 4-bit field in a page table entry, software can assign one of 16 values, which are between 0 and 15, to each page. We call this value the pkey (protection key). Each of these 16 pkey values corresponds to a particular 2 bits of the 32-bit PKRU register, and the access right to the pages, a particular pkey is assigned to, is represented by the corresponding 2 bits of the PKRU register. Software can configure the current memory access right applied to a CPU core by writing an arbitrary value to the PKRU register using the WRPKRU instruction, and it can update the applied memory access right by overwriting the value of the PKRU register at runtime.

Kernel-provided API. In general, page tables are maintained in kernel space, and user-space programs cannot directly manipulate them. Therefore, kernels provide user-space programs with APIs to request the kernels to perform pkey

assignment settings on behalf of the user-space programs. For example, Linux offers a system call named `pkey_mprotect` for the pkey assignment.

Operations performed by user-space programs. On the other hand, user-space programs can execute RDPKRU and WRPKRU instructions without the interposition of a kernel. The usage of these CPU instructions is defined by hardware. Before a user-space program executes the RDPKRU instruction, it should set 0 to the ecx register. After the execution of RDPKRU, the value of the RDPKRU register is stored in the eax register. Before a user-space program executes the WRPKRU instruction, it should set 0 to the ecx and edx registers. Once the WRPKRU instruction is executed, the value stored in the eax register is written to the PKRU register.

4.2.3 pkey Settings

Our memory access restriction mechanism uses three pkeys to configure two memory access rights. In this study, we refer to the three pkeys as pkey0, pkey1, and pkey2. One of the two memory access rights is applied while the code of iip is executed, and the other is applied during the execution of the implementations other than iip; we call these two the memory access right for iip code and the memory access right for non-iip code, respectively. Our memory access restriction mechanism switches these two memory access rights at runtime by updating the value of the PKRU register.

Memory access right	pkey0	pkey1	pkey2
non-iip code	access granted	access granted	access denied
iip code	access denied	access granted	access granted

Table 4.1: PKRU settings for the two memory access rights (Chapter 4.2.3).

PKRU settings. Table 4.1 shows the PKRU settings for the two memory access rights. When the memory access right for non-iip code is applied, the user-space program can access the memory regions having either pkey0 or pkey1, but it is not allowed to access the memory regions having pkey2. When the memory access right for iip code is applied, the user-space program can access the memory regions having either pkey1 or pkey2, but it is not allowed to access the memory regions having pkey0.

Memory region type	pkey
Memory regions dedicated to iip	pkey2
Memory regions that can be accessed by both non-iip and iip code	pkey1
Other than the above	pkey0

Table 4.2: pkey assignment for each memory type (Chapter 4.2.3).

pkey assignment for each memory type. Table 4.2 shows the pairs of a memory type and an assigned pkey. Our memory access restriction mechanism assigns pkey2 to the memory regions dedicated to iip, pkey1 to the memory regions that both iip and non-iip code can access, and pkey0 to the other memory regions.

4.2.3.1 Memory Regions Having pkey2

As shown in Figure 4.4, memory regions, dedicated to iip and configured to have pkey2, are used for packet data, packet metadata, TCP metadata, and the stack.

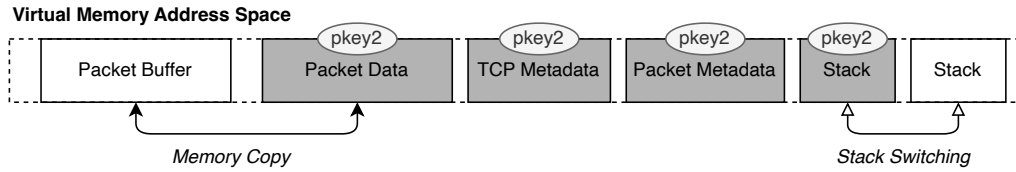


Figure 4.4: Memory regions having pkey2 and relevant regions (Chapter 4.2.3.1).

Reducing the size of the memory shared by iip and non-iip code. To increase the possibility that our memory access restriction mechanism successfully causes segmentation faults when memory accesses unintended by developers are attempted, we explore the pkey and memory configurations that can reduce the size of the memory shared between the iip and non-iip code. To this end, our memory access restriction mechanism allocates memory regions for the packet data and the stack; note that these memory regions are specific to the setup using our memory access restriction mechanism.

In-memory objects in the memory regions dedicated to iip. We provide the description of each in-memory object shown in Figure 4.4.

- **Packet data.** In this study, the packet data means the protocol headers and the payloads of packets. The content of packet data is the same as the data stored in packet buffers which are associated with a network interface by a packet I/O framework and store the data received or the data to be sent by the network interface. We use the two terms, packet buffers and the memory regions for packet data, to differentiate the memory regions; packet buffers are managed by a packet I/O framework, and the memory regions for packet data are used by iip when our memory access restriction mechanism is applied. As described above, we allocate the memory regions for packet data to reduce the size of the memory shared between iip and non-iip code; while iip itself can perform TCP/IP processing for the data in packet buffers, our memory access restriction mechanism conducts memory copies between the packet buffers and the memory regions for packet data so that iip can perform TCP/IP processing without accessing the packet buffers which belong to a packet I/O framework and have pkey0. The memory copies between packet buffers and the memory regions for packet data may negate the communication performance of iip, therefore, we introduce a technique to reduce memory copies between these two in Chapter 4.2.7.
- **Packet metadata.** The memory regions for packet metadata store iip's internal packet representation data structure objects; each of them maintains a pointer to packet data and its length. The memory regions for packet metadata are not specific to our memory access restriction mechanism, and these memory regions are allocated in a user-space program that leverages iip even if our memory access restriction mechanism is not applied.
- **TCP metadata.** The memory regions for TCP metadata, namely, TCP connection states store iip's internal data structure objects; each of them maintains the information of a TCP connection such as the sequence and ACK numbers. Similar to the memory regions for packet metadata, the memory regions for TCP metadata are not specific to our memory access restriction mechanism.
- **Stack.** The stack memory region is typically used as the temporary storage through a stack pointer implemented as a CPU register and the push and pop CPU instructions. As described above, our memory access restriction mecha-

nism allocates the stack memory region for iip to reduce the size of the memory shared between iip and non-iip code; non-iip code uses the default stack memory region having pkey0, and at the memory access right transition, our memory access restriction mechanism switches the stack pointer so that iip will use the stack memory region allocated by our memory access restriction mechanism rather than the default stack memory region used by non-iip code.

Memory allocation. The prototype implementation of our memory access restriction mechanism allocates the memory regions for the objects described above through a replacement of a function whose original implementation allows non-iip code to obtain the size of the memory required for iip, and non-iip code allocates the memory for iip based on the returned value. Our memory access restriction mechanism replaces this function with a function that returns a larger value so that the non-iip code will allocate memory regions large enough to accommodate the in-memory objects added for our memory access restriction mechanism.

pkey assignment. In our prototype, the pre-switch function checks whether the pkey2 assignment has been completed or not, and if not, it performs pkey settings for the memory regions that supposedly have pkey2.

4.2.3.2 Memory Regions Having pkey1

Our memory access restriction mechanism does not recommend the use of pkey1 because both iip and non-iip code can access the memory regions having pkey1.

Use cases of pkey1. However, underlying subsystems such as OS kernels and compilers which are powerful enough to change the control flow of a user-space program, and they occasionally insert procedures at points that are not obvious from the view of the user-space program, and the inserted procedures are not aware of the memory access right applied to the CPU core and thus sometimes attempt to access memory regions which cannot be accessed by the current memory access right and cause segmentation faults. In this case, it would be reasonable to assign pkey1 to memory regions accessed by procedures inserted by underlying subsystems.

Assumption. The memory regions, which should have pkey1, depend on the environments. Therefore, we do not explicitly define the memory regions that should have pkey1, and assume that detailed configurations are applied by users

of our memory access restriction mechanism.

The use of pkey1 in our prototype. In the prototype of our memory access restriction mechanism, pkey1 is assigned to the memory region where the code binary of iip is loaded, because this memory region is accessed not only by the iip code but also by some procedures of the dynamic linker/loader which is considered the non-iip code. We apply this setting during the initialization phase of a user-space program in a function to which we assign the constructor attribute which is an extension of GCC to specify a function called before the main function of a user-space program.

4.2.4 Copies between Memory Regions with Different pkeys

The standard memory copy mechanism, represented by `memcpy`, cannot move the data between memory regions having pkey0 and pkey2 because non-iip code cannot access memory regions having pkey2 and iip code cannot access memory regions having pkey0. Therefore, we implement a memory copy mechanism that enables memory copies between memory regions having different pkeys.

Two directions of data movement. Our memory copy mechanism supports two directions of memory copying: copy-in and copy-out.

- **Copy-in.** In the case of copy-in, our memory copy mechanism copies data from a memory region, that the current memory access right cannot access, to a memory region that can be accessed by the current memory access right, by proceeding with the following steps. First, it switches the memory access right to the one that can access the copy source by updating the PKRU register. Then, it stores the data of the copy source onto a set of CPU registers. Afterward, it restores the previously applied memory access right by updating the PKRU register. At last, it copies the values stored in the CPU registers, copied from the copy source, to the copy destination.
- **Copy-out.** In the case of copy-out, our memory copy mechanism copies data from a memory region, that the current memory access right can access, to a memory region that cannot be accessed by the current memory access right, by the following steps. First, it stores the data at the copy source onto a set of CPU registers. Then, it switches the memory access right to the one that can access

the copy destination by updating the PKRU register. Afterward, it copies the values in the CPU registers, which are copied from the copy source, to the copy destination. At last, it restores the previously applied memory access right by updating the PKRU register.

Use of the AVX-512 extension. Our prototype implementation leverages the zmm CPU registers, which are implemented for the AVX-512 extension, to temporarily maintain the data to be copied. A zmm CPU register is a 512-bit register and thus each can maintain a 64-byte value, and each CPU core maintains 32 of zmm CPU registers; 32 of 64-byte zmm CPU registers allow our memory copy mechanism to move up to 2048 bytes at once.

4.2.5 Switching the Memory Access Rights

As mentioned in Chapter 4.2.1.4, the memory access right switching is performed through the three layers: pre-switch functions, the trampoline code, and post-switch functions.

Manipulating the arguments. The pre-switch and post-switch functions are responsible for appropriately transferring the arguments between callers and callees separated by the isolation boundary of our memory access restriction mechanism. Particularly, the manipulation of the arguments is necessary when the number of arguments exceeds the number of CPU registers. In this case, the pre-switch function copies the arguments, which cannot be stored in the CPU registers, to a memory region that can be accessed after the memory access right is switched by using the memory copy mechanism presented in Chapter 4.2.4, and the post-switch function picks up the copied arguments and reshapes them to match the argument format defined by the callee.

The implementation of the trampoline code. The function, which implements the trampoline code, is normally called by a pre-switch function. In our prototype, the trampoline code function takes three arguments: a value to be set to the PKRU register, a value to be set to the stack pointer of a CPU core, and a pointer to a function to be called after the trampoline code finishes the memory access right switching. The trampoline code sets the supplied values to the PKRU register and the stack pointer of a CPU core, and calls the specified function which is normally a post-switch function. When the post-switch function returns, the ex-

ecution gets back to the trampoline code, and the trampoline code restores the values of the PKRU register and the stack pointer which were applied before the pre-switch function called the trampoline code function.

Nested calls. The memory access right switching can be performed in a nested manner, meaning that a pre-switch function can be called by a function called by a post-switch function; for example, a function, provided by iip for processing received packets and called by a post-switch function, may invoke a pre-switch function and trigger memory access right switching to call a developer-provided callback function which is a handler for the payload of a specific protocol. To support the nested invocation of memory access right switching, in our prototype, the trampoline code passes the PKRU and stack pointer values, which are applied when a pre-switch function has called the trampoline code, to the post-switch function, and we set these values to the arguments of the trampoline code function when calling the trampoline code function in the nested manner; the consequence is that the trampoline code implementation explained above sets the previously applied PKRU register and stack pointer values, and every return of a nested call properly restores the PKRU register and stack pointer values.

Security note. We note that the function of the trampoline code described above does not sufficiently take security into account, and it is vulnerable to, for example, an attack that passes a maliciously crafted PKRU value for its argument to apply an arbitrary memory access right.

4.2.6 Reducing the Memory Access Right Switching

A user-space program handing packets at a high rate requires switching the memory access rights frequently, and it experiences a performance reduction caused by the frequent memory access right switching. To mitigate the performance reduction, our memory access restriction mechanism implements a technique to reduce the frequency of memory access right switching.

Batched processing. The mitigation technique processes multiple packets within a single memory access right transition. The mitigation technique lets the iip and non-iip sides exchange packets by copying arrays of the packet information, which includes the pointer to packet data and the data length, to the other side by using the memory copy mechanism presented in Chapter 4.2.4.

Adaptation. We apply this mitigation technique to the following functions defined by the iip API.

- **The function to push received packets to iip.** The APIs of packet I/O frameworks usually allow user-space programs to pull received packets by a single API call. We pass the pulled packets to the function to push received packets to iip using this mitigation technique. Note that the number of the batched packets depends on the underlying packet I/O framework; if the intervals of packet arrivals are long enough and the received packets are pulled from the packet I/O framework one by one, the mitigation technique applied to this function does not contribute to the performance.

The callback functions for application processing. iip provides the code of developers with opportunities to execute application-specific logic by calling a developer-provided callback function for the payload of each received packet. To apply the mitigation technique, we make a queue of the packets, for which iip has completed TCP/IP processing, and let the pre-switch functions, which are called for the developer-provided callback functions to process the received payloads, push the packets, for which iip has finished TCP/IP processing, to the queue rather than immediately triggering the memory access right switching by invoking the trampoline code. The packets pushed to the queue are passed to the developer-provided callback functions within a single memory access right switching after iip finishes TCP/IP processing for all received packets that have been pushed at once by the function described above.

The function to push data to be sent to iip. We apply the mitigation technique to the function provided by iip to push the data to be sent to iip. We make a queue of the data to be sent and let the pre-switch function, which is called for the function of iip to pass the data to be sent to iip, push the data to the queue rather than triggering the memory access right switching. We pass all data in the queue to iip at once in a function designated for time keeping that is periodically called by a user-space program using iip.

4.2.7 Reducing Memory Copies for Packet Data

Memory copies for the data of packets can degrade the performance of bulk data transfer workloads. `iip` itself is designed to support zero-copy I/O and can perform TCP/IP processing without copying data between packet buffers, associated with a network interface, and other memory regions. On the other hand, as we mentioned in Chapter 4.2.3.1, to reduce the memory regions that can be accessed by both `iip` and non-`iip` code, our memory access restriction mechanism copies data between packet buffers and the memory regions for packet data and lets `iip` conduct TCP/IP processing on the memory regions for packet data. While the copies between packet buffers and the memory regions for packet data are inevitable for our memory access restriction mechanism, we implement a mitigation technique to reduce the data size of the memory copies.

Skipping the memory copies for payloads. The mitigation technique copies only the headers of the packets between packet buffers and the memory regions for packet data, and skips the copies for their payloads. The insight behind this technique is that a TCP/IP stack does not need to access the payload of a packet as long as it does not perform checksum calculation which can be done by the checksum offloading feature of NIC hardware.

Send. The behavior of the mitigation technique for the send path is shown in Figure 4.5a. For this case, we use the checksum offloading feature and the scatter-gather feature of a NIC. When a user-space program tries to pass the data to be sent to `iip` using the `iip` API, this mitigation technique passes only the pointer to the data and its length to the `iip` side. When the checksum offloading feature and the scatter-gather feature of NIC hardware are available, `iip` performs TCP/IP processing and generates a header for the data based on the information of its length, without accessing the content of the data. Afterward, the `iip` side passes the content of the generated header, using the memory copy mechanism presented in Chapter 4.2.4, to a packet I/O framework along with the pointer to the data. The packet I/O framework associates the header and the data referenced by the passed pointer together by using the scatter-gather feature of the NIC, and the NIC concatenates them at transmission.

Receive. The behavior of the mitigation technique for the receive path is shown in Figure 4.5b. For this case, we use the checksum offloading feature

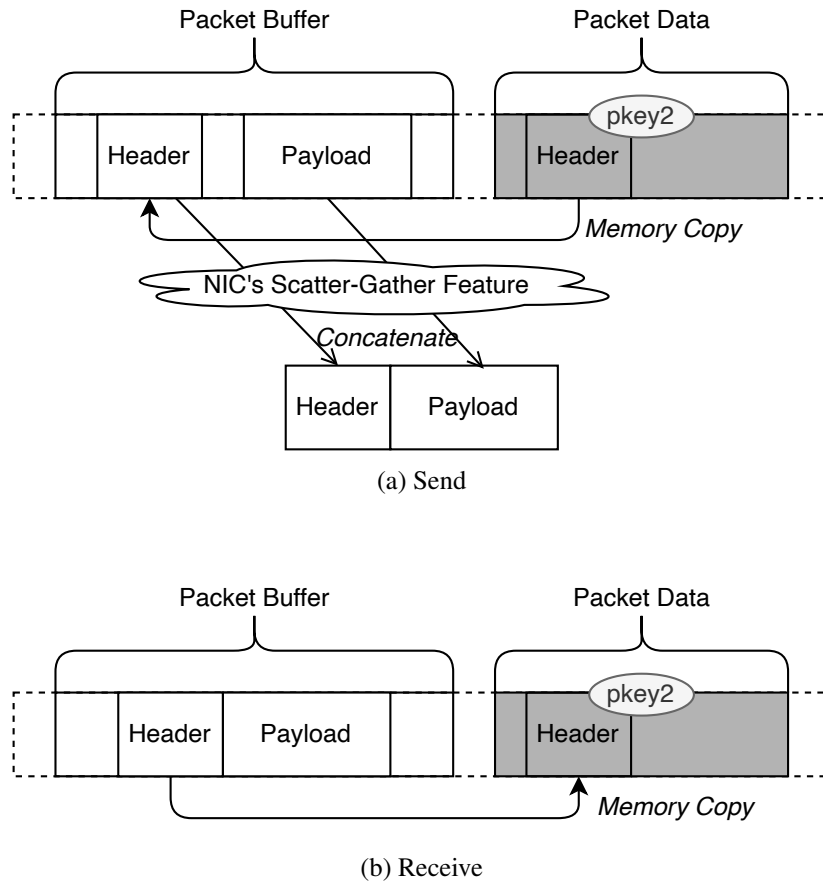


Figure 4.5: The technique to reduce memory copies for packet data (Chapter 4.2.7).

of a NIC. When a user-space program tries to pass a received packet to iip using the iip API, the mitigation technique passes only the header of the packet and the information of the correctness of its checksum value obtained from the checksum offloading feature to the iip side by using the memory copy mechanism presented in Chapter 4.2.4. Since the checksum information is provided by the NIC, iip does not need to calculate the checksum and thus it can perform TCP/IP processing although it has no access to the payload of the received packet.

4.2.8 Limitations

We discuss the limitations of our memory access restriction mechanism.

Combination use with other MPK-based systems. It is difficult to colocate our memory access restriction mechanism and other MPK-based systems that apply memory access restriction settings to the same memory regions because only a single pkey can be assigned to a single page. On the other hand, we consider that it is possible to colocate our memory access restriction mechanism and other MPK-based systems if they do not apply memory access restriction settings to the same memory regions.

The case where the designated API is bypassed. Some user-space programs may have implementations that attempt to access in-memory objects of iip without going through the iip API; these accesses will cause segmentation faults because our memory access restriction mechanism performs the memory access right switching at the entry and exit points of the functions defined by the iip API. To avoid these segmentation faults, the developers of the user-space programs have to manually insert the procedure to perform memory access right switching by modifying the source code of their programs. However, we consider that we do not face this issue often because the in-memory objects of iip are not intended to be exposed to outside of iip, and the accesses to them without going through the designated API are out of the common usage anticipated by iip.

4.3 Discussions on Other Mechanisms

We discuss other mechanisms that can be potentially applied to achieve the goal presented in Chapter 4.1.2.

4.3.1 Using the MPK Feature

There have been prior studies exploring the use of MPK to apply memory access restrictions [12, 27, 38, 48, 54, 57, 39]. However, as mentioned in Chapter 4.1.3, their focus is security enhancement and is different from the scope of this study. There are several prior proposals applying MPK-based memory access restrictions to TCP/IP stack implementations, and here, we review the differences between

them and this study.

Hodor. Hodor [12] is a mechanism that aims to improve security by applying intra-process memory access restrictions. Hodor supports the use of several hardware features, including MPK, the standard page table, and VMFUNC, to apply memory access restrictions. The paper of Hodor presents an example that restricts memory accesses of a process running a TCP/IP stack implementation called F-Stack [52]. While Hodor’s MPK support requires changes to the kernel and negates the handiness of the memory access restriction mechanism, its foundational components, which manipulate the MPK facility, can be used for realizing a system equivalent to our memory access restriction mechanism. On the other hand, the MPK feature can be used in a number of ways, and an appropriate usage of MPK depends on the goals that the developers wish to achieve; the design goal of the mechanism we have presented differs from that of the example presented in the paper of Hodor, therefore, the way MPK used in this study differs from its usage in the Hodor paper. Specifically, the major difference is that while the example shown in the Hodor paper uses the same memory access right for the TCP/IP stack and the packet I/O framework, we use different memory access rights for the TCP/IP stack and the packet I/O framework to reduce the size of the memory shared by the TCP/IP stack and the other components. This difference affects the design choices: our memory access restriction mechanism allocates different memory regions for a TCP/IP stack and a packet I/O framework and performs memory copies using the mechanism presented in Chapter 4.2.4, and Chapter 4.2.7 presents a technique to reduce the memory copies for mitigating the performance overhead. We consider the insights presented in this study to be useful for developers and researchers who wish to apply fine-grained memory access restrictions to their systems.

CubicleOS. The paper of CubicleOS [48] presents an example that applies MPK-based memory access restrictions to lwIP [8]. CubicleOS is built atop Unikraft [25] which is an OS designed for use cases running only a single application on a Virtual Machine (VM) and maintains a build system that allows users to build a kernel that retains a minimal set of functionalities required for executing a particular application so that they can reduce the resources occupied by the kernel. CubicleOS relies on the build system of Unikraft to apply its memory access restrictions and thus its applicability is limited to Unikraft environments

while we have presented, in Chapter 4.2.1.4, the adaptation technique based on function call replacement which works on general-purpose OSes.

FlexOS. The paper of FlexOS [27] also presents an example applying MPK-based memory access restrictions to lwIP. FlexOS is also implemented as the extension of Unikraft, but its adaptation scheme differs from that of CubicleOS and it modifies the source code of the adaptation target. According to the FlexOS paper, to use lwIP on FlexOS, 542 lines of addition and 275 lines of deletion have been required for the source code of lwIP. These changes are needed mainly because FlexOS inserts the procedure of memory access right switching to an existing implementation at compile time and this insertion is performed based on the annotations that are added by users to the source code of the existing implementations. While FlexOS maintains a mechanism that performs static analysis to automate a large part of the procedure for adding the annotations, manual changes are still necessary for cases where a function, that requires the memory access right switching before its invocation, is called through a function pointer. On the other hand, we have presented, in Chapter 4.2.1.4, the adaptation technique that uses the function call replacement and does not require changes to the existing implementations except for the cases described in Chapter 4.2.8.

SURE. SURE [39] is a serverless computing framework that leverages Unikraft in conjunction with MPK for running untrusted user code. Its trusted runtime, including the network I/O facility, and the untrusted user code are colocated in a Unikraft VM, and MPK is used for isolating the trusted and untrusted components. SURE provides untrusted user code with a SURE-specific API for access to the functionalities of the trusted runtime; this API internally implements a call gate mechanism that performs the MPK-relevant operations to change the privileges. The major difference between this study and the SURE work is that on one hand, in SURE, its MPK-based protection mechanism is tightly coupled with its foundational facilities including the communication channels used for the data movement between Unikraft VMs and NICs, on the other hand, this study presents an adaptation technique that enables us to implement our memory access restriction mechanism as a pluggable extension of iip and allows users to easily activate and deactivate it depending on their use cases.

4.3.2 Using the Common Page Table Feature

Memory access restriction mechanisms can be built atop the common page table feature without MPK. For example, general-purpose OSes use the common page table feature to provide isolation between processes.

Difference from MPK. MPK allows a user-space program to switch the memory access right without communicating with the kernel and the performance overhead of the WRPKRU CPU instruction used for memory access right switching is relatively low. On the other hand, on general-purpose OSes, a user-space program needs to interact with the kernel to update the settings of the page table, and moreover, the access to the CR3 CPU register, which is necessary for the page table setting, is a costly operation for CPUs.

Prior work. Our memory access restriction mechanism uses MPK to enable a process to employ multiple memory access rights in user space, and there have been prior proposals that implement the same functionality using the common page table feature [15, 29], while several past studies [54, 12] have shown that the performance overheads of MPK-based mechanisms tend to be lower compared to the mechanisms relying on the common page table feature.

4.3.3 Using the CPU Features for Virtualization

There are prior studies that leverage CPU features, such as Intel VT-x [53], commonly used as the basis of CPU virtualization in hypervisor software, to isolate a TCP/IP stack implementation from the code of application programs.

4.3.3.1 VMX Non-root Ring 0 and Ring 3

IX [4] is a performance-optimized OS that is designed to achieve high throughput and low latency while providing isolation between its networking subsystems and application programs by using the Intel VT-x feature through Dune [3]; the network subsystems, including the TCP/IP stack, are run in VMX non-root ring 0, and application programs are executed in VMX non-root ring 3. Note that the Dune-based protection can be removed, as demonstrated by the publicly available implementation of ReFlex [22], which is an extended version of IX. The difference between this study and the IX work is that, similar to the Hodor [12] work

discussed in Chapter 4.3.1, IX runs the TCP/IP stack and the NIC device driver with the same privilege, on the other hand, our memory access restriction mechanism presented in this study uses different memory access rights for a TCP/IP stack and a NIC device driver to achieve fine-grained memory access restrictions, and in Chapter 4.2.7, we have presented how to mitigate the performance overheads deriving from the memory access rights separation.

4.3.3.2 VMFUNC CPU Instruction

VMFUNC is a CPU instruction that can be executed by a program running on a VM and has been used for the foundational component of memory access restriction mechanisms. For a virtual CPU (vCPU) of a VM, the hypervisor configures the host physical memory that the vCPU can access through a special page table called the Extended Page Table (EPT). A CPU supporting VMFUNC allows software, typically a hypervisor, to associate multiple EPTs with a single vCPU, and the VMFUNC CPU instruction allows a program running on the vCPU to use a feature called EPT Pointer (EPTP) switching that switches the current active EPT to one of the EPTs associated with the vCPU. The fascinating part of the VMFUNC CPU instruction is that it does not cause an exit from the VM context, and therefore, the performance overhead of the EPTP switching feature is comparatively low.

Prior work. There have been previous proposals that employ the EPTP switching feature to provide intra-process isolation [30, 24, 12]. On the other hand, the mechanisms relying on VMFUNC require specific implementations in hypervisor software, and therefore, they often incur high deployment complexity. On top of that, previous studies [54, 12] have reported that the performance overhead of MPK is lower than that of the EPTP switching feature of VMFUNC.

4.3.4 Using the Features of Programming Languages

There have been programming languages that maintain the features that prevent memory accesses unintended by developers. For example, Java detects invalid memory access through its specific runtime, the Java Virtual Machine (JVM), and Rust prevents invalid memory access by compile-time checks.

Limitations of the features of programming languages. However, these features are not always applied to the runtimes and foundational libraries underpinning the programming languages. On top of this, developers cannot always use programming languages that have the features preventing memory accesses not intended by them because of, for example, their performance requirements and the language-level compatibility with the APIs of existing libraries. Therefore, we believe that it is valuable to explore mechanisms, like our memory access restriction mechanism, that prevent memory accesses unintended by developers without relying on the facilities of programming languages.

4.4 Experiment

We examine how our memory access restriction mechanism affects the communication performance of iip.

4.4.1 Experiment Setup

We use the following setup.

Machine setup. We use a pair of machines. Each machine has two 16-core Intel Xeon Gold 6326 CPUs clocked at 2.90 GHz and 128 GB of DRAM. The two machines are directly connected via Mellanox ConnectX-5 100 Gbps NICs. We activate the scatter-gather, checksum offloading, TSO, and LRO features of the NIC hardware. For the experiments using multiple CPU cores, we instantiate a pair of RX and TX queues of the NIC for each CPU core and we use the RSS feature of the NIC for spreading received packets to the NIC queues. Both machines install Linux 6.2.

Configurations. We conduct the experiments with five configurations.

- **baseline.** The baseline configuration does not apply our memory access restriction mechanism.
- **proposal.** The proposal configuration applies our memory access restriction mechanism.
- **proposal w/o rdpkru/wrpkru.** The proposal w/o rdpkru/wrpkru configuration applies our memory access restriction mechanism, but it omits the RDPKRU

and WRPKRU CPU instructions from the source code of our memory access restriction mechanism. We run experiments with this configuration to investigate the performance overhead deriving from the CPU instructions for the MPK support. Note that this configuration is just for the investigation, and our memory access restriction mechanism does not work properly when this configuration is applied.

- **proposal w/o batching.** The proposal w/o batching configuration applies our memory access restriction mechanism, but it deactivates the technique, presented in Chapter 4.2.6, for reducing the memory access right switching.
- **proposal w/o memcpy skip.** The proposal w/o memcpy skip configuration applies our memory access restriction mechanism while deactivating the technique, presented in Chapter 4.2.7, for reducing the memory copies for the data of packets.

Benchmark software. For the experiments, we use a benchmark program that we have made. The benchmark program works in user space and leverages DPDK and iip to provide two features: one is to exchange messages between two hosts and the other is to transfer data from one to another. We run this benchmark program on the two machines.

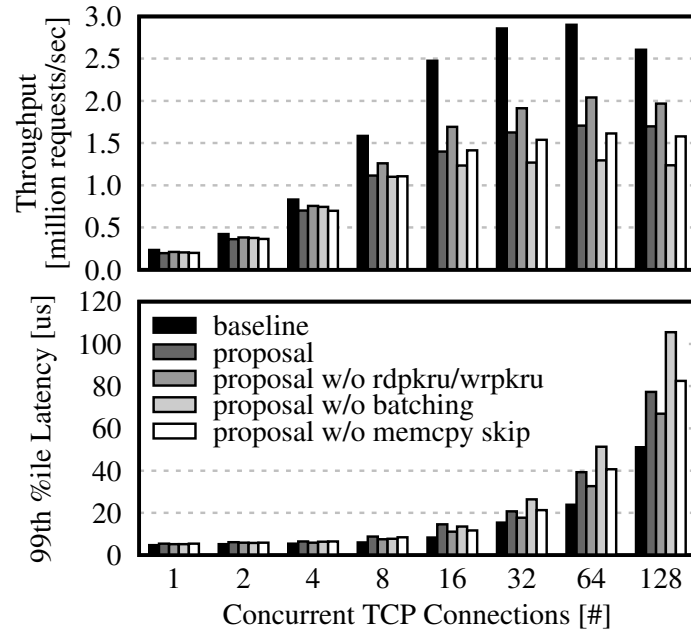
Target and load generator machines. We use one of the two machines as the target machine and the other as the load generator machine. On one hand, we apply the five configurations described above on the target machine, and on the other hand, we do not apply our memory access restriction mechanism on the load generator machine.

4.4.2 Exchanging Small Messages over TCP Connections

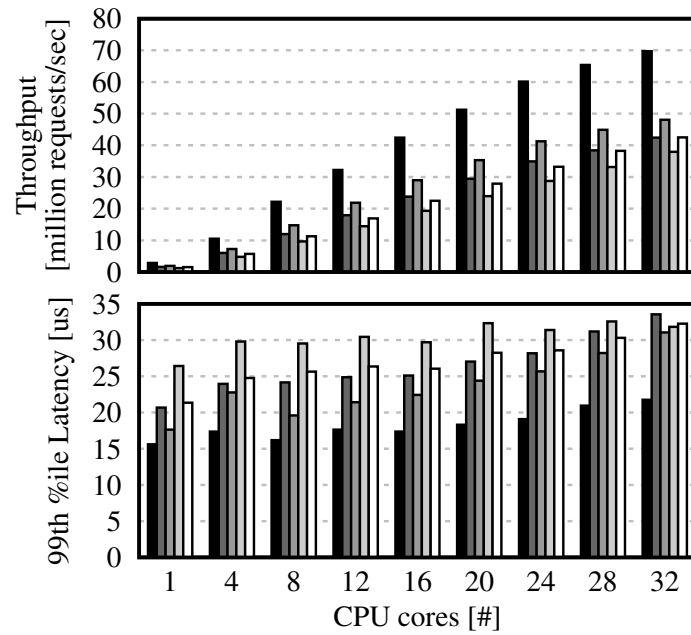
We run a workload that exchanges 1-byte messages between the two machines over TCP connections.

4.4.2.1 Using a Single CPU Core

We conduct the performance measurement using a single CPU core.



(a) Using a single CPU core



(b) Using multiple CPU cores

Figure 4.6: Performance for exchanging small messages (Chapter 4.4.2).

Benchmark. The benchmark program on the target machine uses a single CPU core. In the cases where the TCP connections are fewer than 32, the number of CPU cores used by the benchmark program on the load generator machine is the same as the number of TCP connections, and otherwise, the benchmark program on the load generator machine uses 32 CPU cores.

Results. Figure 4.6a shows the results, and our observations are as follows.

- **The overhead of our memory access restriction mechanism.** We have observed that the proposal configuration case, which applies our memory access restriction mechanism, exhibits lower performance compared to the baseline configuration case. The performance gap between these two configurations indicates the overhead introduced by our memory access restriction mechanism.
- **The overhead sources.** The overhead of the RDPKRU and WRPKRU CPU instructions is seen in the performance gap between the proposal and proposal w/o rdpkru/wrpkru configuration cases. Based on these results, we consider that the performance overhead of our memory access restriction mechanism stems not only from the MPK-related CPU instructions but also from operations performed before and after updating the PKRU register.
- **The effect of batched processing.** The gap between the results of the proposals and the proposal w/o batching configuration cases shows the performance gain brought by the technique, presented in Chapter 4.2.6, for reducing the memory access right switching.
- **The effect of reduced memory copies.** The gap between the results of the proposals and proposal w/o memcpy skip configuration cases shows the performance gain brought by the technique, presented in Chapter 4.2.7, for reducing the memory copies for the data of packets.

4.4.2.2 Using Multiple CPU Cores

We carry out the performance measurement using multiple CPU cores.

Benchmark. We run the benchmark program on the target machine while changing the number of CPU cores assigned to it. On the load generator machine, the benchmark program uses 32 CPU cores. We configure the number of TCP

connections so that each CPU core of the target machine will handle 32 concurrent TCP connections.

Results. Figure 4.6b shows the results. We have observed a similar trend to the one seen in the single CPU core experiment shown in Chapter 4.4.2.1. We find that our memory access restriction mechanism is not a significant bottleneck for multi-core scalability.

4.4.3 Bulk Data Transfer over a TCP Connection

We conduct performance measurement with a bulk data transfer workload that sends data from one to another as fast as possible.

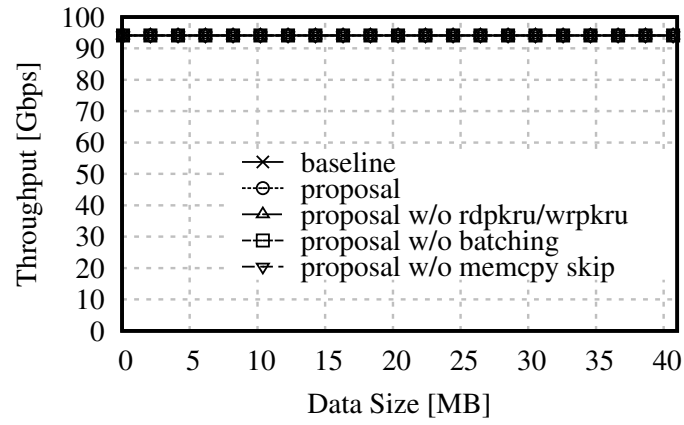
Benchmark. On both of the target and load generator machines, the benchmark program uses a single CPU core, and we use a single TCP connection for the bulk data transfer. The sender side repeatedly sends the same data prepared in a memory region, and we run the experiments while changing the size of the data.

Results. The results are reported in Figure 4.7.

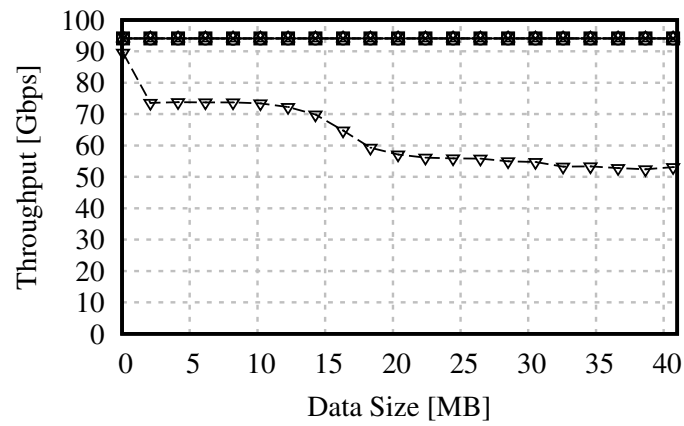
- **Receive.** The receive performance is shown in Figure 4.7a. We have observed the maximum throughput of the NIC in all cases.
- **Send.** The send performance is shown in Figure 4.7b. We have observed the maximum throughput of the NIC in all cases except the proposal w/o memcpy skip configuration case that disables the technique presented in Chapter 4.2.7. The gap between the proposal and proposal w/o memcpy skip configuration cases shows the effect of the reduced memory copies.

4.5 Summary

The study presented in this chapter has focused on the upper layer that extends iip for specific use cases, and specifically, we have built a memory access restriction mechanism as an upper-layer subsystem to improve memory safety for use cases where a process runs iip in user space. We have proposed techniques to mitigate the performance overheads of our memory access restriction mechanism, and conducted experiments to examine its performance overhead and quantify the performance gains brought by the proposed techniques.



(a) Receive



(b) Send

Figure 4.7: Bulk data transfer performance (Chapter 4.4.3).

Chapter 5

Conclusion

This chapter describes the conclusion of this dissertation.

5.1 Contribution Summary

This dissertation has addressed a research question: how can we realize a fast and portable TCP/IP stack that portably achieves good performance across diverse environments and use cases?

Three-layered studies. To answer the research question, we conducted three-layered studies. Each of them corresponds to the middle, lower, and upper layers, respectively.

- **Middle layer.** The first study, focusing on the middle layer, has presented the design of a fast and portable TCP/IP stack and our prototype implementation named *iip*. We have reviewed the existing performance-optimized and portability-aware TCP/IP stacks and determined their issues, and we have presented the API that enables *iip* to sidestep the issues and offer portability and good performance simultaneously. Specifically, the API of *iip* is designed to minimize dependencies on other components for portability and be sufficiently aware of performance-critical factors. We have conducted experiments to quantify the performance of *iip* and investigated how its awareness of the performance-critical factors contributes to its performance.
- **Lower layer.** The second study has investigated the lower layer which is packet

I/O frameworks. We have discussed the support for multiple packet I/O frameworks to increase the likelihood of achieving good performance in varied environments and use cases by circumventing the issues of specific packet I/O frameworks, and presented our prototype implementation for iip to support three major packet I/O frameworks, DPDK, netmap, and AF_XDP along with the packet-oriented abstraction model that abstracts away the differences between their APIs by hiding the underlying NIC queue representation based on a ring buffer data structure. We have conducted experiments to examine the performance of our prototype implementation and investigate the efficient setups realized by its beneficial properties.

- **Upper layer.** The third study has focused on the upper layer extending iip for specific use cases, and we have built a memory access restriction mechanism as an upper-layer subsystem to improve memory safety for use cases where a process runs iip in user space. Our memory access restriction mechanism is designed to cause a segmentation fault when iip accesses the memory regions it should not or implementations other than iip access the memory regions dedicated to iip. Besides this basic functionality, we have proposed techniques to mitigate the overheads associated with the MPK-based memory access restrictions. We have run experiments to evaluate the performance overheads of our memory access restriction mechanism and quantify how the proposed techniques contribute to the performance.

Design philosophy. Through the three studies, we have presented a design philosophy that partitions software involving TCP/IP communication into layers and limits the layers performing memory copies of packet data. In our prototype, the middle and lower layers are designed to allow for zero-copy I/O and only the upper layer performs memory copies for packet data; consequently, we could have the opportunity to implement the technique for reducing memory copies for packet data in our memory access restriction mechanism and it can be transparently applied on top of the underlying TCP/IP processing component and packet I/O frameworks. This design philosophy is naturally aligned with the properties of a fast and portable TCP/IP stack, as layer partitioning is the foundation of portability and reduced memory copies contribute to performance.

5.2 Future Directions

One of the future directions for this dissertation is to explore the methods to provide high-quality implementations of fast and portable TCP/IP stacks; particularly, we think of studies on testing and verification mechanisms that enable developers to have strong confidence in the correctness of the behavior of their TCP/IP stack implementations. Another future direction is to examine the portability aspect from a software engineering perspective; we expect that the property of *iip*, which minimizes dependencies on other components, would be beneficial for decentralized collaboration and accelerate development.

Bibliography

- [1] S. Awamoto and M. Honda. Opening up kernel-bypass tcp stacks. In *Proceedings of the 2025 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '25, USA, 2025. USENIX Association.
- [2] D. R. Barach and E. Dresselhaus. Vectorized software packet forwarding, jun 2011. Patent No. US 7961636 B1, Filed May 27th., 2004, Issued Jun 14th., 2011.
- [3] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, Oct. 2012. USENIX Association.
- [4] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, Oct. 2014. USENIX Association.
- [5] N. Bonelli, F. D. Vigna, A. Fais, G. Lettieri, and G. Procissi. Programming socket-independent network functions with nethuns. *SIGCOMM Comput. Commun. Rev.*, 52(2):35–48, June 2022.
- [6] A. Butok. FNET. 2005.
- [7] Cloudius Systems. Seastar. <http://www.seastar-project.org/>, 2014.

- [8] A. Dunkels. Full tcp/ip for 8-bit architectures. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services, MobiSys '03*, page 85–98, New York, NY, USA, 2003. Association for Computing Machinery.
- [9] egnite Software GmbH. Ethernut. 2001.
- [10] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. In *Proceedings of the ACM SIGCOMM 2010 Conference, SIGCOMM '10*, page 195–206, New York, NY, USA, 2010. Association for Computing Machinery.
- [11] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. MegaPipe: A new programming interface for scalable network I/O. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 135–148, Hollywood, CA, Oct. 2012. USENIX Association.
- [12] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. Hodor: intra-process isolation for high-throughput data plane libraries. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '19*, page 489–503, USA, 2019. USENIX Association.
- [13] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. 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.
- [14] M. Honda, F. Huici, C. Raiciu, J. Araujo, and L. Rizzo. Rekindling network protocol innovation with user-level stacks. *SIGCOMM Comput. Commun. Rev.*, 44(2):52–58, apr 2014.
- [15] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of the*

- 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS '16, page 393–405, New York, NY, USA, 2016. Association for Computing Machinery.
- [16] Intel. Data Plane Development Kit. <https://www.dpdk.org/>, 2010.
- [17] K. Jang, S. Han, S. Han, S. Moon, and K. Park. SSLShader: Cheap SSL acceleration with commodity processors. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, Mar. 2011. USENIX Association.
- [18] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. 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, Apr. 2014. USENIX Association.
- [19] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive scheduling for μ second-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, Feb. 2019. USENIX Association.
- [20] M. Karlsson and B. Töpel. The path to dpdk speeds for af_xdp. In *Linux Plumbers Conference*, volume 37, page 38, 2018.
- [21] A. Kaufmann, T. Stamler, S. Peter, N. K. Sharma, A. Krishnamurthy, and T. Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.
- [22] A. Klimovic, H. Litz, and C. Kozyrakis. Reflex: Remote flash $\approx$ local flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17*, page 345–359, New York, NY, USA, 2017. Association for Computing Machinery.
- [23] J. Kneschke. lighttpd. <https://www.lighttpd.net/>, 2003.

- [24] K. Koning, X. Chen, H. Bos, C. Giuffrida, and E. Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 437–452, New York, NY, USA, 2017. Association for Computing Machinery.
- [25] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, c. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici. Unikraft: fast, specialized uniker-nels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 376–394, New York, NY, USA, 2021. Association for Computing Machinery.
- [26] S. Kuenzer, A. Ivanov, F. Manco, J. Mendes, Y. Volchkov, F. Schmidt, K. Yasukata, M. Honda, and F. Huici. Unikernels everywhere: The case for elastic cdns. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, page 15–29, New York, NY, USA, 2017. Association for Computing Machinery.
- [27] H. Lefeuvre, V.-A. Bădoiu, A. Jung, S. L. Teodorescu, S. Rauch, F. Huici, C. Raiciu, and P. Olivier. Flexos: towards flexible os isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 467–482, New York, NY, USA, 2022. Association for Computing Machinery.
- [28] X. Lin, Y. Chen, X. Li, J. Mao, J. He, W. Xu, and Y. Shi. Scalable kernel tcp design and implementation for short-lived connections. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, page 339–352, New York, NY, USA, 2016. Association for Computing Machinery.
- [29] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel. Light-weight contexts: an os abstraction for safety and performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 49–64, USA, 2016. USENIX Association.

- [30] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15*, page 1607–1619, New York, NY, USA, 2015. Association for Computing Machinery.
- [31] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery.
- [32] I. Marinos, R. N. Watson, and M. Handley. Network stack specialization for performance. In *Proceedings of the 2014 ACM Conference on SIGCOMM, SIGCOMM '14*, page 175–186, New York, NY, USA, 2014. Association for Computing Machinery.
- [33] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the art of network function virtualization. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 459–473, Seattle, WA, Apr. 2014. USENIX Association.
- [34] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1993 Conference (USENIX Winter 1993 Conference)*, San Diego, CA, Jan. 1993. USENIX Association.
- [35] A. Moroo, H. Tazaki, and K. Yasukata. svc-hook: hooking system calls on arm64 by binary rewriting. In *Proceedings of the 26th International Middleware Conference, Middleware '25*, page 195–201, New York, NY, USA, 2025. Association for Computing Machinery.
- [36] J. Mostafa, S. Chilingaryan, and A. Kopmann. Are kernel drivers ready for accelerated packet processing using af_xdp? In *2023 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 117–122, 2023.

- [37] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. 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, Feb. 2019. USENIX Association.
- [38] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim. Libmpk: software abstraction for intel memory protection keys (intel mpk). In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference, USENIX ATC '19*, page 241–254, USA, 2019. USENIX Association.
- [39] F. Parola, S. Qi, A. B. Narappa, K. K. Ramakrishnan, and F. Risso. Sure: Secure unikernels make serverless computing rapid and efficient. In *Proceedings of the 2024 ACM Symposium on Cloud Computing, SoCC '24*, page 668–688, New York, NY, USA, 2024. Association for Computing Machinery.
- [40] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving network connection locality on multicore systems. In *Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12*, page 337–350, New York, NY, USA, 2012. Association for Computing Machinery.
- [41] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, Oct. 2014. USENIX Association.
- [42] G. Prekas, M. Kogias, and E. Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.
- [43] D. Raghavan, S. Ravi, G. Yuan, P. Thaker, S. Srivastava, M. Murray, P. H. Penna, A. Ousterhout, P. Levis, M. Zaharia, and I. Zhang. Cornflakes: Zero-copy serialization for microsecond-scale networking. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP '23*, page 200–215, New York, NY, USA, 2023. Association for Computing Machinery.

- [44] G. F. Riley and T. R. Henderson. *The ns-3 Network Simulator*, pages 15–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [45] L. Rizzo. netmap: A novel framework for fast packet I/O. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, June 2012. USENIX Association.
- [46] L. Rizzo and G. Lettieri. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, page 61–72, New York, NY, USA, 2012. Association for Computing Machinery.
- [47] S. Sanfilippo. Redis - Remote Dictionary Server. <https://redis.io/>, 2009.
- [48] V. A. Sartakov, L. Vilanova, and P. Pietzuch. Cubicleos: a library os with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 546–558, New York, NY, USA, 2021. Association for Computing Machinery.
- [49] I. Sysoev. nginx. <https://nginx.org/>, 2004.
- [50] W. Tarreau. Haproxy. <https://www.haproxy.org/>, 2001.
- [51] H. Tazaki, F. Uarbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabbous. Direct code execution: revisiting library os architecture for reproducible network experiments. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, page 217–228, New York, NY, USA, 2013. Association for Computing Machinery.
- [52] Tencent. F-Stack. <https://www.f-stack.org/>, 2017.
- [53] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005.

- [54] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. Erim: secure, efficient in-process isolation with protection keys (mpk). In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1221–1238, USA, 2019. USENIX Association.
- [55] M. Vincent. PicoTCP: the reference TCP/IP stack for IoT. In *FOSDEM 2014*, Feb. 2014.
- [56] A. Vorobey and B. Fitzpatrick. Memcached - a distributed memory object caching system. <https://memcached.org/>, 2003.
- [57] A. Voulimeneas, J. Vinck, R. Mechelinck, and S. Volckaert. You shall not (by)pass! practical, secure, and fast pku-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 266–282, New York, NY, USA, 2022. Association for Computing Machinery.
- [58] K. Yasukata. iip: An integratable tcp/ip stack. *SIGCOMM Comput. Commun. Rev.*, 54(1):21–28, Aug. 2024.
- [59] K. Yasukata, M. Honda, D. Santry, and L. Eggert. StackMap: Low-Latency networking with the OS stack and dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 43–56, Denver, CO, June 2016. USENIX Association.
- [60] K. Yasukata, F. Huici, V. Maffione, G. Lettieri, and M. Honda. Hypernf: building a high performance, high utilization and fair nfv platform. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 157–169, New York, NY, USA, 2017. Association for Computing Machinery.
- [61] K. Yasukata and K. Kono. Supporting the use of multiple packet i/o frameworks in a tcp/ip stack runnable in user space. *Computer Software*, 42(3):3_4–3_22, 2025.
- [62] K. Yasukata and K. Kono. Using memory protection keys to restrict memory access for a process running a tcp/ip stack in user space. *IPSJ Transactions on Advanced Computing Systems*, 19(1):1–17, 2026.

- [63] K. Yasukata, H. Tazaki, P.-L. Aublin, and K. Ishiguro. zpoline: a system call hook mechanism based on binary rewriting. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 293–300, Boston, MA, July 2023. USENIX Association.
- [64] I. Zhang, A. Raybuck, P. Patel, K. Olynyk, J. Nelson, O. S. N. Leija, A. Martinez, J. Liu, A. K. Simpson, S. Jayakar, P. H. Penna, M. Demoulin, P. Choudhury, and A. Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles, SOSP ’21*, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.
- [65] L. Zhu, Y. Shen, E. Xu, B. Shi, T. Fu, S. Ma, S. Chen, Z. Wang, H. Wu, X. Liao, Z. Yang, Z. Chen, W. Lin, Y. Hou, R. Liu, C. Shi, J. Zhu, and J. Wu. Deploying user-space TCP at cloud scale with LUNA. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 673–687, Boston, MA, July 2023. USENIX Association.