School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

High-performance Networking on seL4

by

Lucy Parker

Thesis submitted as a requirement for the degree of

Bachelor of Computer Science (Honours)

Submitted: November 2023

Supervisors: Prof. Gernot Heiser & Dr Peter Chubb

Student ID: z5059323

Abstract

Modern operating systems undergo rapid development on an extensive code base. This growth has steadily introduced bugs and security vulnerabilities into the trusted computing base, and the growing array of security patches and new features has led to a decline in system call performance. Unfortunately, monolithic kernel design means a single exploitable vulnerability can compromise the entire kernel. The alternative in microkernel-based design has historically been unpopular due to its impact on performance as it requires significantly more context switches than its monolithic counterpart. This thesis evaluates the seL4 Device Driver Framework which rectifies this concern, showcasing superior networking performance compared to a standard Linux system, while leveraging the security guarantees of the seL4 microkernel to provide a secure and performant networking subsystem. We extend the framework to flexibly and securely support multiple client applications and scale to multi-core systems, highlighting a number of ways in which system designers can enforce use-case specific policies.

Abbreviations

ARP Address Resolution Protocol

CC Communication Channel

CVE Critical Vulnerabilities and Exposures

DMA Direct Memory Access

FIFO First In, First Out

IDL Interface Definition Language

I/O Input/Output

IOMMU Input, Output Memory Management Unit

IoT Internet of Things

IP Internet Protocol

IPC Inter-process Communication

LOC Lines of Code

MAC Media Access Control

MPMC Multi Producer, Multi Consumer

MR Memory Region

MUX Multiplexer

NIC Network Interface Controller

OS Operating System

PD Protection Domain

PPC Protected Procedure Call

RPC Remote Procedure Call

Rx Receive

sDDF The seL4 Device Driver Framework

SPSC Single Producer, Single Consumer

TCB Trusted Computing Base

TCP Transmission Control Protocol

Tx Transmit

UDP User Datagram Protocol

Contents

List of Figures

Chapter 1

Introduction

Typical networking-focused systems face major problems with either security, performance or both. This thesis explores a novel I/O framework on seL4 for secure, high-performance networking-based systems and extends the framework to support multi-client systems.

1.1 The problem with I/O on monolithic kernels

Operating systems are designed to abstract over hardware while providing a consistent interface to applications. In a monolithic kernel such as Linux, the kernel houses device drivers, file systems and network stacks. This means that these services run in privileged mode, and user applications make system calls in order to utilise these services and perform I/O. However, there are two major problems with this design: it is inherently insecure and historically, performs poorly.

1.1.1 Monolithic kernels are fundamentally insecure

Modern operating systems see a rapid development rate on a huge code base. Figure 1.1 shows the exponential growth of the Linux kernel (extracted from [Linux Kernel Organization, Inc., 2023]) which sees over 70 thousand commits a year. With this growth, bugs and security vulnerabilities are steadily brought into the trusted computing base.

While much of this growth can be attributed to kernel extensions, such as drivers for new devices, these extensions should not be overlooked. A modern Linux kernel runs around 80 - 130 device drivers on a given configuration [LKDDb, 2023], and drivers account for a significant portion of critical vulnerabilities and exposures (CVE). Figure 1.2 shows the proportion of CVEs published in the last 5 years that can be attributed directly to drivers (extracted from [MITRE Corporation, 2023]), with an average of 38%.

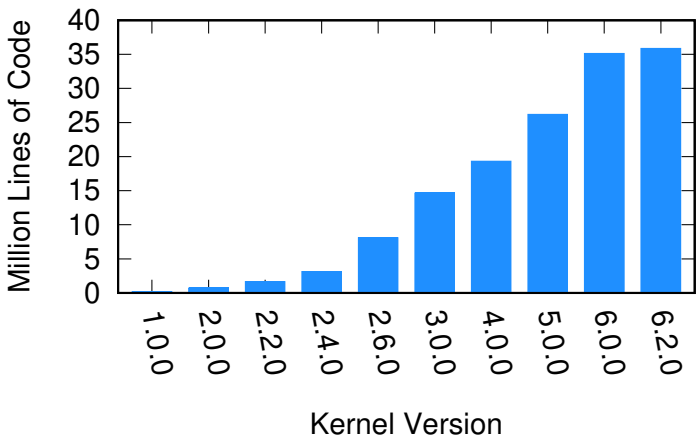


Figure 1.1: Growth of the Linux Kernel.

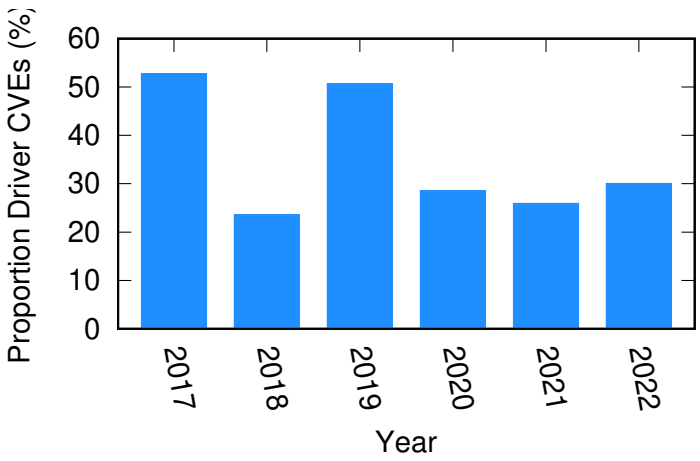


Figure 1.2: Proportion of Linux CVEs due to device drivers.

The reason behind such a high rate of Linux CVEs is due largely to the fact that drivers are typically written by hardware engineers as opposed to OS experts, and the growing complexity of the kernel, has made kernel programming extremely difficult [Swift et al., 2002].

While modern kernels deploy a number of security mechanisms to protect their execution, a large portion of vulnerabilities remains exploitable [Narayanan et al., 2020]. For example, even advanced defence mechanisms, such as code pointer integrity (CPI) and safe stacks, that protect the control flow of an application [Kuznetsov et al., 2014], remain vulnerable to data-only attacks. These attacks, which focus on altering or forging the critical data of an application can be combined with automated attack generation tools to load disabled modules or manipulate attributes of pages in memory [Ispoglou et al., 2018]. Unfortunately, the monolithic design means a single exploitable vulnerability potentially provides an attacker with access to the entire kernel.

1.1.2 Performance limitations of monolithic kernels

The monolithic design minimises the frequency of switching between kernel and user mode, enabling applications to trigger I/O with a single system call. However, the increasing number of security enhancements and new features introduced to Linux has steadily added overhead to core kernel operations and severely impacted system call performance. Ren et al. (2019) found that the *send()* and *recv()* system calls, used to send and receive a packet over the network, have slowed down by 100% between Linux versions 4.0 and 4.2. Furthermore, the monolithic design requires copying data between user and kernel space for I/O system calls in order to sanitise the request and protect the kernel from a clumsy/malicious application. The cost of this data copying adds consistent overhead to such system calls and can quickly become a bottleneck for high performance I/O systems.

1.2 How do we rectify this?

It's clear that we need to remove device drivers from the trusted computing base (TCB). However, simply running them as user level programs on a monolithic kernel requires careful design to prevent introducing more costly system calls and further degrading performance. Such designs are already mainstream but unfortunately are not without their own limitations. We explore these in Chapter 3. We examine how asynchronous APIs can improve performance in networking-based systems by reducing the system call overhead without addressing the security implications of in-kernel drivers in Section 3.1. In Section 3.2, we investigate I/O frameworks that relegate all I/O processing, driver included, to user space. Such designs provide strong fault isolation to drivers while improving performance through asynchronous data management. We also examine how isolation techniques can be used inside the OS to attempt to remove drivers from the TCB in Section 3.3. However, all of these designs are largely limited by the monolithic kernel design itself.

1.3 The Solution

Rather than grappling with the limitations inherent in current monolithic kernels, this thesis proposes developing a high performance networking system using a novel I/O framework on a microkernel instead. The microkernel design is based on minimality, providing only the functionality to securely abstract the hardware, and as such, already relegates device drivers and other OS services such as file systems and network stacks to run as user level programs. This provides strong fault containment for historically bug prone code and significantly reduces the TCB. In Chapter 2, we introduce the seL4 microkernel, underpinned by a comprehensive formal verification that guarantees correct isolation of user-level components [Klein et al., 2014]. However, the microkernel design poses potential performance degradation for I/O systems due to the

inherent number of extra context switches required. The seL4 Device Driver Framework (sDDF), introduced in Chapter 4, is a simple I/O framework that minimises this overhead and provides interfaces and protocols for developing performant networking systems [Parker, 2022]. The sDDF is characterised by a strong separation of concerns, where each task in the pipeline is handled by a simple, single-threaded component. Components communicate asynchronously via shared memory and seL4 notifications. As it stands, the sDDF already supports a statically defined, minimal networking system. However, prior to this work, the sDDF did not support more than a simple echo server application, and was confined to a single core platforms. In Chapter 5, we outline our design to expand the sDDF, enabling flexible and secure support for multiple client applications using plugin-compatible multiplexers each implementing a single, simple policy. We also make minimal adaptions to the lwIP [Dunkels, 2001] network stack library to better integrate with the event-driven programming model used by the sDDF and address diverse networking demands from potential client applications. Additionally, we present an alternative model for device drivers allowing some concurrent execution with the potential to scale efficiently to multi-core systems. We evaluate all these designs in Chapter 7, uncovering that despite the high number of context switches required, seL4-based networking systems significantly outperform Linux-based systems using the socket API. Our multi-client performance benchmarks demonstrate how system designers can implement different policies with our multiplexer designs, as well as configuring the overall system. We assess the scalability of the networking subsystem on multi-core configurations, unveiling significant overheads in the seL4 multi-core kernel. However, we demonstrate that the sDDF can be arbitrarily distributed across cores and effectively enables high-throughput networking for compute-heavy clients when a single CPU is insufficient. To conclude, we highlight the current limitations of the sDDF with our extensions, and directions for future work.

Chapter 2

Background

There have been a variety of attempts to address the performance limitations and security concerns of typical I/O systems. In order to best understand these solutions and their limitations, we must first understand how networking systems typically process packets in a monolithic kernel. We then introduce the seL4 microkernel, which comes with security guarantees and unparalleled performance that our solution will seek to take advantage of. Unfortunately its low level API can make development difficult. The seL4 Core Platform combats this by trading generality for simplicity and abstracting over seL4's objects.

2.1 Typical network packet processing in a monolithic kernel

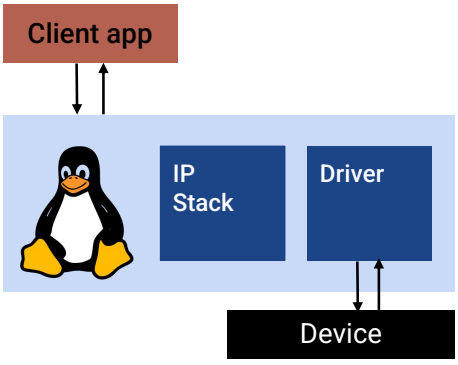


Figure 2.1: Network packet processing in a monolithic kernel.

A monolithic operating system houses both drivers and protocol stacks in kernel as shown in Figure 2.1. These components are typically written as libraries for the kernel to process a packet. In order for a user application to utilise the network, the application must open a socket through a system call. This system call creates an endpoint for

communication and returns a file descriptor that refers to that endpoint [Linux manual page, 2023]. The file descriptor can then be used to read (receive a packet) or write (transmit a packet) to the socket using the protocols specified when opening the socket.

To receive a packet on the network, the OS must first enqueue the addresses of buffers in memory to the device's control registers. The device can then write newly received data directly to these buffers. Once a packet has been written, the device will issue an interrupt to the OS to notify the OS. The kernel then processes this packet as defined by the IP protocol used in the packet. This typically includes reading the packet header, which contains critical information such as where the packet has come from, where it is headed, as well as the communication protocol used. This information advises the kernel about what to do with the packet. If the packet is addressed to a user-level application bound to a socket, eventually the header of the packet will be stripped before the data is copied to a user space buffer queue attached to the socket. This buffer can now be accessed using a read (*recv()*) system call. Note that much like a file read system call, this system call will block until there is data available to be read.

To transmit, a user application can write a packet using a write (*send()*) system call. The data will first be copied into an in kernel buffer. The kernel will process the packet and an appropriate header will be added to the head of the packet as per the protocol used with the socket. Eventually, this kernel buffer address will be enqueued to the device control registers which signals to the device that a packet is ready for transmit, after which, the system call returns. Once the device has transmitted the packet, it will issue an interrupt to notify the kernel that the buffer is no longer in use and can be re-used.

Operating systems commonly use interrupt hold off on the device as an attempt to batch hardware events, such as receiving or transmitting a packet, into a single interrupt. This means that the kernel will be able to more efficiently process multiple packets at a time. At high throughputs on the receive path, the kernel will even disable receive interrupts and switch to polling mode instead. This removes the interrupt handling overhead but requires the driver to continuously check for hardware events. Furthermore, in recent years, hardware has become more complex, and network devices are capable of commandeering some of the packet processing. This includes, for example, computing the packet checksum or packet coalescing. Many of such hardware extensions have been capitalised on by monolithic kernels to reduce the software overhead of packet processing.

However, the typical method of networking in a monolithic kernel is both insecure, due to the amount of untrusted code running at privilege level, and performs poorly, due to data copying operations and the increasing cost of system calls, particularly those used for sending and receiving a packet [Ren et al., 2019].

Figure 2.2: seL4 architecture

2.2 seL4

seL4 is a highly performant, highly secure microkernel. It has been formally verified to be functionally correct down to the binary, along with upholding the security properties confidentiality, integrity and availability [Klein et al., 2014]. However, as a microkernel, it provides only the minimum to securely abstract the hardware and nothing more [Heiser et al., 2022]. As such, it presents a very low level API, and it does not provide resource management, file systems, network stacks or device drivers and instead prescribes these services to run as user level programs. The kernel makes no distinction between such services and applications, which results in the horizontal architecture shown in Figure 2.2.

seL4 is also a capability-based system. A capability is a communicable, unforgeable token that references a kernel object with an associated set of access rights. This provides fine grained access control and means user-level components must have a capability in order to access an object and means that seL4 guarantees components only have access to objects, including data, that developers permit them to. Additionally, seL4 guarantees ensure full isolation of user-level components, meaning that failure of any component is completely contained and will not bring the rest of the system down. Finally, seL4 is the world's fastest microkernel and performs within 25% of the limits imposed by hardware [Mi et al., 2019].

However, the microkernel design can potentially degrade performance as it requires significantly more context switches than its monolithic counterpart. This is because OS services, such as device drivers, typically run as an isolated components, and client applications wishing to use such services must communicate using synchronous/asynchronous IPC and shared memory. Furthermore, interrupt processing is forwarded to the user level handler through asynchronous system calls and user level applications are also responsible for initiating any cache management operations through system calls when required for memory consistency. These system calls, which aren't typically required in a monolithic system, can quickly add up.

2.3 The seL4 Microkit

The seL4 microkit, formerly known as the seL4 Core Platform, is an operating system framework on seL4 with minimal abstractions over seL4 low-level objects [Heiser et al., 2022]. It aims to ease the development process of applications on seL4 by abstracting over architecture-specific details as well as seL4's low-level API. Microkit trades generality for simplicity and restricts systems to a predominantly static architecture. This means components are fixed at build time. The abstractions are:

1. **Protection Domain (PD)**: a process abstraction that enforces an event driven programming model.
2. **Communication Channel (CC)**: the ability for two PDs to communicate via notifications or PPCs.
3. **Memory Region (MR)**: a range of physical memory that may be mapped into 1 or more PDs address spaces.
4. **Notification**: a semaphore-like synchronisation primitive.
5. **Protected Procedure Call (PPC)**: synchronous RPC between PDs.

Microkit promotes the correct use of seL4 by employing an event driven driven programming model for PDs. Events are triggered by either notifications or PPCs sent along a CC. Microkit also restricts the use of seL4's IPC primitives to remote procedure calls, which is only possible if the callee has strictly higher priority than the caller. This prevents PDs from either directly or indirectly calling themselves. Finally, while microkit predominantly only supports static systems, there is some supported dynamism which enables the re-initialising of PDs and dynamically loading code into PDs [Heiser et al., 2022].

Chapter 3

Related Work

The wider community is well aware of the problem space created by untrusted/unreliable code running inside the kernel and there have been a variety of attempts to deprivilege components and/or improve the performance in networking-focused systems. In Section 3.1 we explore new asynchronous APIs to minimise the system call overhead. In Section 3.2 we examine user level I/O frameworks that bypass the OS altogether to deprivilege device drivers and remove the system call overhead. Finally, in Section 3.3 we consider isolation techniques that take advantage of hardware virtualisation support to remove drivers from the TCB.

3.1 Asynchronous I/O Frameworks

In order to combat the performance overheads of typical network packet processing, a number of new APIs have been developed with an asynchronous programming model.

3.1.1 Netmap

Netmap aims to decrease packet processing costs on monolithic kernels through an asynchronous, zero copy API [Rizzo, 2012]. Specifically, netmap removes the cost of dynamically allocating buffers per packet by preallocating fixed size packet buffers when the network device is first opened. These buffers are shared between kernel and user space to remove the cost of data copying operations. Finally, netmap removes the cost of blocking system call operations by providing an asynchronous API to user level applications. Applications open a special device file /dev/netmap using *ioctl()* and a flag to indicate registration. This function will return the size of the memory region containing shared data structures for buffer management. On the receive path, another *ioctl()* system call with the receive flag will return the number of packets available for reading. These packets can then be found in buffers ready to be dequeued from a shared

receive queue. On the transmit path, applications write packets intended for transmit into shared buffers and insert into the shared transmit queue. Another *ioctl()* system call is used with the transmit flag to notify the OS about the new packets to send. Importantly, the *ioctl()* call with either the receive or transmit flag is non blocking. This asynchronous interface is able to significantly reduce the system call overhead by providing a simple mechanism to batch packet processing. With this, the netmap API achieves significantly higher transmit throughput than the socket API.

3.1.2 io_uring

io_uring is another asynchronous API for I/O frameworks on Linux [Axboe, 2019]. As opposed to netmap, io_uring provides a more generalised interface for different classes of I/O processing and not just network-based applications. Figure 3.1 shows the 2 shared ring buffers between kernel and user space: one for submitting a request, another for results of completed requests. These ring buffers are set up using *io_uring_setup()* and then mapped into user space with *mmap()*. To issue a read or write request, applications must provide a user level buffer, set appropriate flags in the submission queue entry and add this entry to the tail of the requests ring buffer. User applications inform the kernel of updates to the requests ring buffer using a new system call *io_uring_enter()*. This system call is blocking, and returns once the kernel has processed the new requests and the results are available in the completed queue. To provide an asynchronous interface, there is an optional polling mode, whereby the kernel will continue to poll for updates in the requests queue. Polling mode removes the need for system calls, which can significantly improve throughput.

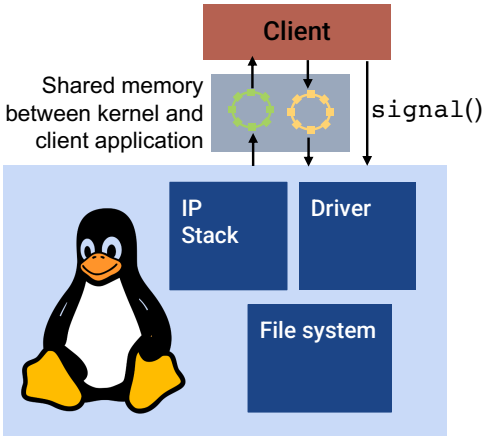


Figure 3.1: System design with io_uring.

3.1.3 Takeaways

Both netmap [Rizzo, 2012] and io_uring [Axboe, 2019] reduce performance overheads of I/O on monolithic kernels by providing an asynchronous interface to enable batching,

or the complete removal of system calls. However, both APIs rely on in-kernel device drivers and do not explore the security vulnerabilities of such a design. Furthermore, netmap requires exclusive device access and thus the usual in-kernel device sharing mechanisms do not work. Instead, if the NIC is to be shared, the pool of data buffers will be mapped into every user process that shares the NIC. The result is that a process has access to data belonging to other processes. Also, the polling mode of io_uring would monopolise the CPU even when there is no work to be done, consequently driving up power usage of the system and it is unclear what policy is applied to this mode when there are multiple client applications using it, or if there is other work to be done by the kernel.

3.2 User-space device drivers

Running device drivers as user level programs is the simplest way to deprivilege the software and provide strong fault isolation. However, this can lead to a significant increase in the number of system calls required and this cost can degrade performance significantly. In order to overcome this, kernel bypass frameworks have been developed to support user level IO asynchronously.

3.2.1 DPDK

DPDK (Data Plane Development Kit) provides data management libraries along with a set of network drivers to offload network packet processing to user space and bypass the OS in a Linux environment [Linux Foundation, 2020]. Figure 3.2 shows the system design differences between using an in kernel network driver vs a DPDK library.

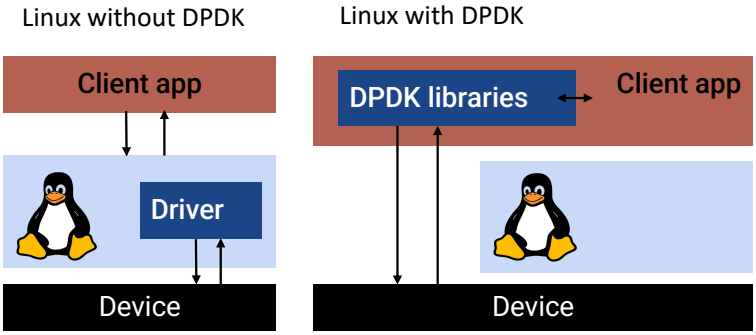


Figure 3.2: System design without and with DPDK.

Specifically, it provides [Linux Foundation, 2020]:

- Lock free, bounded, multi producer, multi consumer (MPMC) ring buffer queues for efficient data management.

- Buffer management to preallocate fixed sized buffers.
- A library of polling mode drivers.

DPDK enables faster packet processing than a typical in-kernel driver as it removes the cost of context switching between kernel and user space. Instead of receiving an interrupt, drivers must poll for hardware events. This means the driver needs to continuously read the device's event register. This has the consequence of driving the CPU utilisation of the system up as the system cannot be idle while waiting for an event. In addition, the framework itself is zero copy, so DPDK can further reduce packet processing costs as it no longer needs to copy packet data into and out of user space. Finally, the framework uses huge pages for large memory pools, which decreases the amount of look-ups and other page management costs while ensuring relevant pages stay pinned in memory and aren't migrated in the background due to Linux's page migration policy.

3.2.1.1 BBQ

BBQ is a block-based bounded queue design that aims to increase performance of ring buffer queues in frameworks such as DPDK [Wang et al., 2022]. DPDK uses multi producer, multi consumer queues, which are designed with the intention to split the workload across several processors. However, these queues have cache interference when performing consecutive operations and this degrades performance. This stems from the MPMC queue design as it needs to keep track of two heads and two tails to keep track of where a producer can produce to without interference from another producer (and likewise for consumers). Pseudo code for interacting with MPMC queues is shown Figure 3.3a and Figure 3.3b [Wang et al., 2022].

```
enqueue(data) {
again:
  ph = load(prod.head)
  pn = ph + 1
  if (pn > load(cons.tail)+ SIZE)
    return FULL
  if (!cas(prod.head, ph, pn))
    /* Another producer is
      using this slot */
    goto again
  entry[pn] = data
  /* Wait until earlier slots
    have successfully
    committed */
  while(load(prod.tail) != ph)
  store(prod.tail, pn)

  return OK
}
```

(a) Producer pseudo code

```
dequeue() {
again:
  ch = load(cons.head)
  cn = ch + 1
  if (cn > load(prod.tail))
    return EMPTY
  if (!cas(cons.head, ch, cn))
    /* Another consumer is
      consuming this slot */
  goto again
  data = entry[cn]
  /* Wait until earlier slots
    have successfully
    committed */
  while(load(cons.tail) != ch)
  store(cons.tail, cn)

  return data
}
```

(b) Consumer pseudo code

Figure 3.3: Psuedo code for MPMC Queues

The performance degradation comes from two possible places:

1. Cache misses occur in a multiprocessing environment every time a producer reads the consumer's tail (on line 5 in Figure 3.3a), even though the tail might be far behind and not of concern. This is because the cache belonging to the core running the producer will not have the required data and it must be fetched from the owner (either from main memory or the local cache of a core running a consumer). Likewise, the consumer will also always miss on the producer's tail (on line 5 in Figure 3.3b) although it may be far ahead.
2. When multiple threads try to read the same entry, they must continuously try again as shown at lines 7 - 10 in Figure 3.3a and Figure 3.3b

BBQ proposes a block-based approach instead, where the queues are divided into blocks and the producers only start using a block once it has been fully consumed. This avoids the cache miss on reading the consumer's tail. Each block contains 4 variables to keep track of the actions: allocated, committed, reserved and consumed. Producers produce between allocated and committed, but these variables aren't updated until the block is fully consumed (this is to prevent cache misses by the consumer). Compared with DPDK, BBQ isn't as affected by CPU over-subscription. This is because in the MPMC DPDK queues, the producers can form a waiting chain, where the last thread to allocate an entry can only commit that entry once the previous thread has committed its entry. While this waiting chain would have minimal impact if the threads are scheduled in the optimal order to each make progress, this is often not the case. As producers in BBQ commit independently using an atomic Fetch-and-Add instruction, this scenario does not occur.

Overall, the BBQ approach attempts to solve performance issues of MPMC queues in a multiprocessing environment, which stem from the concurrency involved in such queues.

3.2.1.2 CleanQ

CleanQ takes the opposite approach of BBQ and instead outlines the need for a simple and *reliable* ring buffer design that does not impede performance in frameworks such as DPDK [Haecki et al., 2019]. CleanQ removes the inherent complexity of MPMC queues and instead returns to single producer, single consumer (SPSC) queues. These queues are significantly simpler, and as a result, Haecki et. al. are able to infer correct guarantees from the design. CleanQ uses the concept of 'ownership' to base their formal invariants. Consider 'entities' to be either a producer or a consumer acting on the queues and 'objects' to be items passed around by these queues, then the invariants guaranteed by the CleanQ design are:

1. An object has at most one owner,
2. If an entity owns an object, it has exclusive use of it,
3. An entity knows whether it owns an object or not,
4. Ownership can be transferred.

These guarantees eliminate many subtle bugs in lock-free queue design. Furthermore, the CleanQ design achieves higher throughput than DPDK MPMC queues in a networking context, which ultimately highlights that a simple design does not negatively impact performance.

3.2.1.3 Takeaways

DPDK completely bypasses the OS to offload packet processing to user space. The zero copy design in user space achieves higher throughput than typical in-kernel packet processing [Linux Foundation, 2020]. However, all drivers are polling mode only. This design decision stems from the high cost of interrupt handling on Linux, but it enables the application to monopolise the CPU as the driver must continuously poll for events. While the polling has little impact in high throughput networking scenarios where there is consistently work to be done, at lower throughput, the CPU utilisation will not decrease with the workload and the system will use significant power without making progress. Moreover, implementing a device driver as a library inside an application limits the device to a single client. Ultimately, the short-comings of monolithic kernels, namely, the high cost of interrupt processing and context switching, have influenced the design and reduced the flexibility. DPDK also relies on cache-coherent architectures to achieve performance, as the minimal support available for cache management operations

from userspace introduces costly system calls. Finally, MPMC queues used in the DPDK data plane are inherently complex due to the concurrency involved and have the potential to degrade performance [Wang et al., 2022]. The simpler and cleaner design of ring buffer queues as shown in CleanQ, can not only achieve higher throughput, but are also verifiable [Haecki et al., 2019].

3.2.2 Ixy

Ixy is a user-space packet framework on Linux that aims to educate developers on driver development [Emmerich et al., 2018]. It utilises the *uio* and *vfio* interfaces to access the device from user space. Using these APIs, Ixy develops a user space polling mode device driver for the ixgbe 10Gbps NIC with low packet forwarding costs. Significantly, the ixy device driver reduces the *ixgbe* in-kernel driver from over 10K LOC to just over 1K LOC.

3.2.2.1 Linux uio

uio (User space I/O) exposes interfaces for user level device drivers by memory mapping files in the *sysfs* pseudo file system on Linux [Koch, 2006]. Interrupts are handled by reading this file once mapped. For example, a *read()* system call will block until an interrupt is triggered. The integer value read from this file represents the total interrupt count. *uio* exposes the device registers via *mmap()* when using a particular offset calculated based on the number of devices using *uio*. However, *uio* hides the issue that DMA addresses must stay in resident memory and that Linux page migration does not guarantee that physical addresses won't change behind the scenes. To overcome this issue, *uio* users can use huge pages as these won't be migrated by the Kernel.

3.2.2.2 Linux vfio

vfio (Virtual Function I/O) extends *uio* and adds support for IOMMU use [The Kernel Development Community, 2006]. The IOMMU is a memory management unit mapping device-visible virtual addresses to physical addresses, and lies on the system bus between I/O devices and main memory. When used, the main advantage of the IOMMU is protecting memory from malicious/faulty devices attempting errant memory accesses. *vfio* introduces additional files in the *sysfs* file system to expose the IOMMU to user space and provides data structures and library functions that can be used to map/unmap memory in the IOMMU for safe DMA through *ioctl()* system calls on Linux.

3.2.2.3 Takeaways

Although the ixy user level device driver is not as performant as its counterpart on the DPDK framework, it reduces a large and complex device driver to a much smaller driver without significant performance degradation. The lower performance of ixy when compared to DPDK can in part be attributed to DPDK's use of SIMD instructions to handle batches of packets at a time [Emmerich et al., 2018]. SIMD units are hardware components that perform the same operation on multiple data operands concurrently. In packet processing, this parallelism can boost performance on checksum calculations. However, much like DPDK, ixy is reliant on cache-coherent architectures and also does not support device sharing.

3.2.3 Linux User-level Device Drivers

Leslie et al. (2005) developed an asynchronous framework for user-level device drivers on Linux [Leslie et al., 2005]. Device registers are mapped into user-level drivers by calling *mmap()* with the approparite section of *\dev\mem*. When processing an interrupt, the kernel increments a semaphore, also mapped into a special file. Much akin to *uio* (though published prior to the release of *uio*), the user process can read this special file to obtain the value of the semaphore. The driver and kernel communicate asynchronously via lock-free, ring buffer queues in shared memory, and share DMA I/O buffers as already provided by Linux. For example, when an ethernet device receives a packet and triggers an IRQ, the kernel will increment the relevant semaphore to deliver this interrupt to the user-level driver. The driver will process this and enqueue the packet to the kernels network stack via these shared queues. The kernel will check for such a packet every time the driver returns from the interrupt handler. This way, a client application can use the driver via the socket API. Leslie et al. (2005) implemented an IDE block device driver and an Ethernet device driver on Linux 2.6.6. The user-level disk driver achieved identical performance compared with a typical in-kernel driver, and the network driver achieved 93% of the throughput for only 7% increase in CPU utilisation.

3.2.3.1 Takeaways

The User-level Device Drivers demonstrate that deprivileging a driver does not significantly degrade performance, despite the extra number of context switches required for high-performance I/O. Unlike DPDK, this design does not impose any restrictions on device sharing, however, still uses the kernel IP stack and file systems. These subsystems comprise of large, bug-prone code bases running in the trusted computing base. Furthermore, the design relies on kernel provided I/O buffers on cache-coherent architectures, and introduces costly system calls without such architectures.

3.2.4 Snap

Snap proposes a modular architecture, comparable to a microkernel design, by utilising asynchronous communication mechanisms and minimal state sharing [Marty et al., 2019]. The design intentionally leans away from typical socket programming for networking systems and instead opts for an event driven model. They present a user space network driver that uses asynchronous communication and ring buffers in shared memory for data transfer. Interrupts are delivered to an event file, which can be polled from user level. Alongside this, Snap takes a unique approach to scheduling. Rather than designing a generic scheduling policy that aims to be applicable across a large set of scenarios, they instead propose 3 different designs that could be swapped as required:

1. Tasks are pinned to cores to take advantage of cache efficiency.
2. Tasks are spread dynamically. Threads are scheduled when interrupts are delivered to process the event on the next available core.
3. Tasks are spread dynamically as above but onto as few cores as possible. This aims to take advantage of cache efficiency when possible, but as work queues up, the allocation needs to be queried.

3.2.4.1 Takeaways

The Snap design deprivileges untrusted drivers by running them in user space and the design, although running on top of a monolithic kernel, uses event driven programming. This design is shown to scale well to multiple clients and the swappable scheduling policy removes complexity from the trusted computing base while demonstrating efficiency.

3.2.5 Microdrivers

Instead of running the entire driver at user level, Microdrivers split the device driver into two separate components; one of which runs at kernel level and the other at user level in order to achieve the performance of in kernel drivers combined with the fault isolation of user level drivers [Ganapathy et al., 2008]. Figure 3.4 shows the architecture of microdrivers. The kernel mode component contains performance critical and frequently used functionality. For a network driver, this includes interrupt handling and sending and receiving packets. The rest of the code, predominantly initialisation and error handling, goes into user level where the performance degradation of context switching between kernel and user level does not have noticeable impact. In order to speed up the user mode component, microdrivers duplicate functions in both components and use shared memory for frequently accessed data structures.

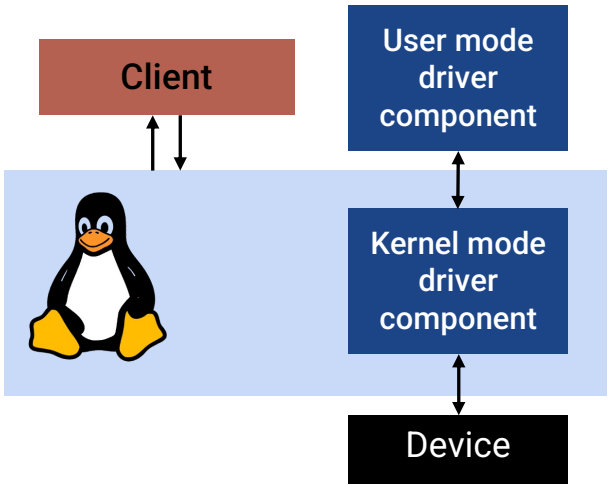


Figure 3.4: Microdrivers architecture

3.2.5.1 Takeaways

Although 70% of driver code consists of non-performance-critical functions and can be relegated to user space [Ganapathy et al., 2008], this does not provide good fault isolation. The remaining 30% of driver code is just as untrustworthy, and faults have higher impact as this code is on the critical path. Furthermore, by sharing data structures between kernel and user space, some of which could contain sensitive kernel data, higher trust needs to be placed on the user mode component, thus diminishing any safety guarantees obtained by running the code at user level. Finally, the research was not able to produce reliable performance results which undermines any performance gains of running part of the driver at kernel level. Thus, splitting a driver into a user level component and a kernel level component is not viable and could achieve the worst of both designs: poor performance and poor fault isolation.

3.3 Isolating Kernel Components inside the OS

Another method to deprivilege untrusted code inside the OS is to use isolation techniques. These approaches rely on hardware virtualisation extensions in order to re-use legacy code and avoid the redesign and redevelopment of OS components like drivers while also supporting device sharing.

3.3.1 Nooks

A Nook provides an isolated environment for driver execution inside a monolithic OS [Swift et al., 2002]. The driver still runs inside the kernel address space, but within a different protection domain. This can be achieved using virtual memory protection

and lowering the privilege level to remove access to privileged instructions. For shared resources, the OS can call into isolated device drivers with a wrapper function in order to track resource usage, verify data passed in/out, flush the TLB and perform a software trap. Similarly, a driver accessing a shared data structure is trapped so the OS can update the data on the drivers behalf.

3.3.1.1 Takeaways

Unfortunately there is a lack of detail into the Nooks architecture and the work doesn't account for potential performance degradation due to some of the design decisions. Firstly, in order for the OS to update data on the drivers behalf, it would need to first verify this data. How this data is verified and to what extent could cost cycles on a critical path. Furthermore, interrupts are forwarded to the isolated driver. Nooks measured interrupt handling on a isolated driver to cost double the cycles of an in kernel driver, yet there is no accounting for this in the performance. This cost is significant and would limit drivers to polling mode only.

3.3.2 LXDs

LXDs (Lightweight Execution Domains) enable isolation in a monolithic OS using hardware-assisted virtualisation (VT-x) [Narayanan et al., 2019]. The design utilises a small microkernel that runs inside the monolithic OS to provide both synchronous and asynchronous communication channels between kernel processes and isolated components. Figure 3.5 outlines the architecture of this solution. Instead of a typical function call to transmit data that would call into an in-kernel driver library, it instead sends a message via the LXDs microkernel to the isolated driver. The isolated driver has direct access to the NIC and can then perform the transmit.

In order to achieve this design, they prescribe methods to decompose kernel code to help analyse the interaction patterns between the kernel and its drivers. They propose a new IDL (Interface Definition Language) to auto generate glue code based on these patterns. This glue code creates copies of shared data structures to ensure such structures aren't shared between domains, and translates function calls into an isolated domain into remote procedure calls [Narayanan et al., 2019].
To better isolate device drivers, the design pins a driver to a particular core and enables communication via cross-core message passing. Synchronous message passing via the LXDs microkernel is used initially for establishing regions of shared memory that are then set up for asynchronous communication in the form of ring buffer queues. Cross-core communication then just relies on cache coherence, which has a lower cost (around 380 cycles) than using hardware mechanisms for address space isolation (around 850 cycles)
Instead of blocking waiting for another core running an isolated driver to complete its work, the design proposes to instead spawn a light weight thread when communicating asynchronously that can wait for the response. The original thread can then keep

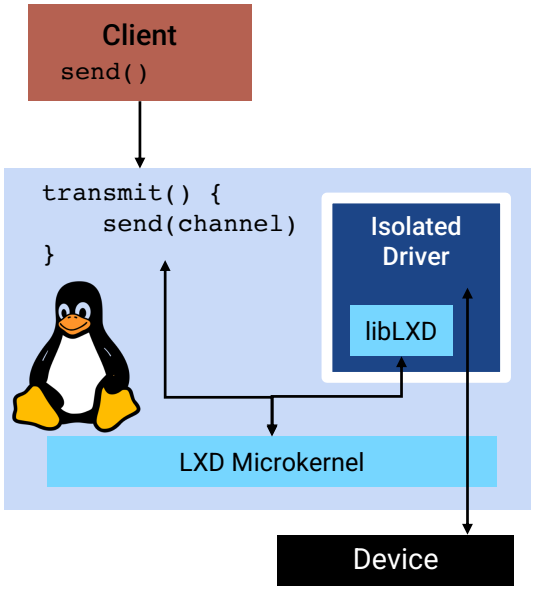


Figure 3.5: System design of LXDs with an isolated NIC driver

executing. When this is not possible, for example, when the next instruction relies on the result of the call, continuations can be used as well.

3.3.2.1 Takeaways

Although the design was able to achieve throughput within 80% of a non-isolated NIC driver, the design has several limitations. Firstly, the design relies on utilising multiple cores to achieve comparable performance. This is not always practical, and has the potential to waste resources. Although the paper does not display CPU utilisation figures for their benchmarks, by pinning a polling driver to a particular core, even at low loads this core will be executing at 100%. Additionally, the architecture involves adding a microkernel type component to the trusted computing base inside the OS. Although it may be small, there is no comment on the complexity of this new addition and this extra layer must be trusted in order to remove the driver from the TCB. This design also introduces more concurrency into the kernel by spawning threads to wait for cross-core communication. Finally, the research is limited to device drivers on multicore hardware and does not include other layers and components in the networking processing plane such as the IP stack. The Linux kernel IP stack is significantly more complex and difficult to isolate in such a manner.

3.3.3 Isolation using VMFUNC

Another method of isolating drivers and other kernel extensions is to use recent hardware mechanisms such as VMFUNC and extended page table switching. This design

involves executing the OS on top of a minimal hypervisor to make use of these hardware mechanisms [Narayanan et al., 2020]. VMFUNC is an Intel primitive that changes the extended page table underneath a virtual machine without exiting into the hypervisor. As the VMFUNC instruction is significantly faster than switching privilege levels, this research proposes using this instruction in a trampoline mechanism to quickly context switch into an isolated component. Figure 3.6 demonstrates the overall architecture. Similarly to the LXDs architecture, the in kernel driver is replaced by stub code that crosses the isolation boundary using vmfunc trampoline code before performing the transmit on the device.

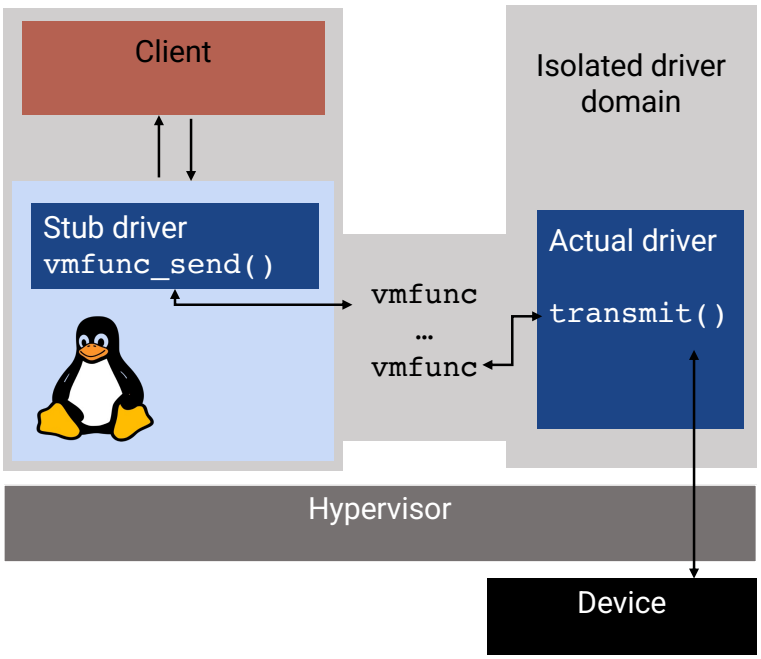


Figure 3.6: System design using vmfunc trampoline to isolate NIC driver

However, this architecture is not necessarily secure and extra measures must be taken. These are:

1. Ensure virtual address spaces of isolated domains, kernel and user processes do not overlap. This is to prevent a malicious component from executing VMFUNC on its own accord and having the ability to access to data it should not. The same applies to physical addresses.

2. ensure isolated domains have read-only access to their page table. This is to ensure an isolated component cannot change its address space layout and thus violate the above.

3. Sensitive data is protected by either mediating it through the hypervisor (such as control registers) or saved and restored across domain switches (general, segment and extended state registers). This happens in the trampoline code.

3.3.3.1 Takeaways

Isolating a network driver using the above design achieved throughput within 1% to 11% of the native polling driver on a 10Gbps NIC. Although the isolation technique used seems to add minimal overhead, some overhead is hidden in the interrupt delivery for non-polling drivers. The design proposes mapping the interrupt descriptor table is into both the kernel and the isolated domains to prevent the need to trap into the hypervisor on interrupt. The interrupt is then first handled by the kernel domain before interrupting the isolated driver where applicable. However, this allows an untrusted component access to the interrupt descriptor table and the ability to disable interrupts and never return control to the OS. To overcome this, they propose using a preemption timer in the hypervisor to ensure the kernel domain is making progress. Overall, the cost of delivering an interrupt to an isolated domain is over 1000 cycles, and the cost and trade-offs of using a preemption timer in the hypervisor is not measured. Furthermore, to ensure an untrusted isolated component did not alter the interrupt descriptor table without disabling all interrupts (eg. it could disable interrupts of some other device), the OS would need to check the interrupt descriptor table on every return. This cost is not measured. Overall, mapping the interrupt descriptor table into untrusted domains is inherently insecure and should be avoided.

3.4 Summary

Typical network packet processing is both insecure and non-performant. This is because complex, bug prone software systems run in the trusted computing base, and the complexity of such systems has led to reduced performance. The solutions outlined above aim to rectify this, but are largely limited by the monolithic kernel design itself. Asynchronous IO frameworks improve performance without addressing the security vulnerabilities. User level drivers achieve high throughput at the cost of high utilisation and are limited to single client applications (with the exception of Section 3.2.3). Although techniques to isolate drivers inside the OS promote device sharing, these techniques are not extensible to other kernel subsystems, they do not completely protect the OS and can also degrade performance. However, there are still some significant learnings we can take away from the above work. Firstly, Rizzo (2012), Axboe (2019), Linux Foundation (2020) and Marty et al. (2019) demonstrate significant performance gains through the use of an asynchronous API. Emmerich et al. (2019) showed that a complex device driver for a high throughput NIC could be simplified 10 fold and still achieve performance. Haecki et al. (2019) demonstrated that simple SPSC queues can outperform complex MPMC queues, and Marty et al. (2019) demonstrated that a modularised approach with flexible policy achieves competitive throughput in a real world application of data centres.

Chapter 4

The seL4 Device Driver Framework

Unfortunately, many approaches to solving the issues with I/O are limited by the monolithic kernel design itself. This presents a strong argument to redesign a performant yet secure I/O framework on a different architecture completely: the microkernel. However, the microkernel design has the potential to degrade performance as it requires significantly more context switches than its monolithic counterpart. In order to overcome potential performance degradation in I/O systems due to this, we require a simple framework that minimises system calls when possible for designing performant I/O systems on a microkernel architecture. This introduces the seL4 Device Driver Framework as an excellent starting point for this project.

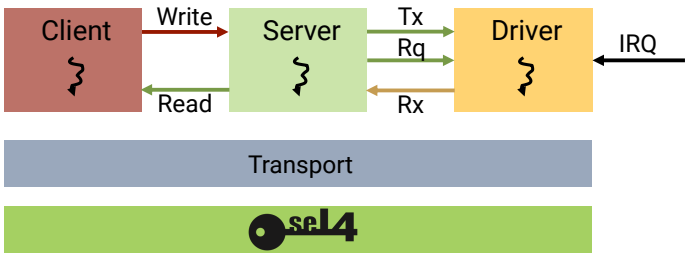


Figure 4.1: The seL4 Device Driver Framework (sDDF) [Parker, 2022]

The seL4 Device Driver Framework (sDDF) aims to rectify any performance degradation of a microkernel design by providing interfaces and protocols to write performant yet secure user level device drivers on seL4 [Parker, 2022]. It currently supports a statically defined, minimal networking-focused system, and is developed on top of the seL4 microkit. The sDDF prioritises a strong separation of concerns by componentising each task in an I/O framework such that each component has only a single job. This keeps each component inherently very simple, not only minimising any debugging effort required by developers but also makes these components accessible to verification. The

sDDF design is based on top of a simple transport layer that provides protocols for communication between drivers and other components in the system.

4.1 Driver Model

The device driver translates a hardware specific device protocol into a hardware independent (but OS specific) device class protocol. It is inherently event driven, as it only needs to react to either a client request or an interrupt generated by hardware, and in this way, it maps perfectly onto the seL4 microkit model. Clients make requests through shared memory and seL4 asynchronous notifications which simplifies the device driver code to an event loop reacting to either client requests or hardware events as shown in Figure 4.2.

```
main() {
    init()
    while(true)
        event = Wait()
        if (event & IRQ)
            handle_irq()
        if (event & CLIENT_REQ)
            handle_request()
}
```

Figure 4.2: Driver pseudo code

4.2 Transport Layer

The sDDF transport layer consists of 3 distinct shared memory regions, data structures for data management and access protocols. These memory regions are:

1. **Metadata region**: the control registers of a device shared between the device itself and its driver. This region is volatile as it is accessed directly by the device and is mapped uncached.
2. **Data region**: buffers containing data that's shared between the device and PDs that require access to the data. This region is also accessed directly by the device, but is mapped cached as we assume the device will only access it when instructed. Before and after such accesses, we need to invalidate or clean buffers to ensure cache coherency.
3. **Control region**: data structures to manage buffers in the data region. This is shared between two PDs in the system.

4.2.1 Control region

The control region consists of lockless ring buffer queues. These queues are single-producer, single-consumer (SPSC) which keeps the implementation simple. The queues keep track of addresses in the data region as shown in Figure 4.3. For simplicity, the figure only shows the control region on the transmit path between a server and driver. Each pair of communicating PDs in the sDDF have their own shared control region but share the data region with other PDs. The sDDF uses two queues per direction, per control region. For example, as per Figure 4.3, the Transmit Used (TxU) queue stores buffer addresses which contain data ready for transmit. The Transmit Free (TxF) queue stores buffer addresses available for reuse. This is duplicated for the receive path. The queues enable data to be passed between components in batches, thus minimising the number of system calls requried.

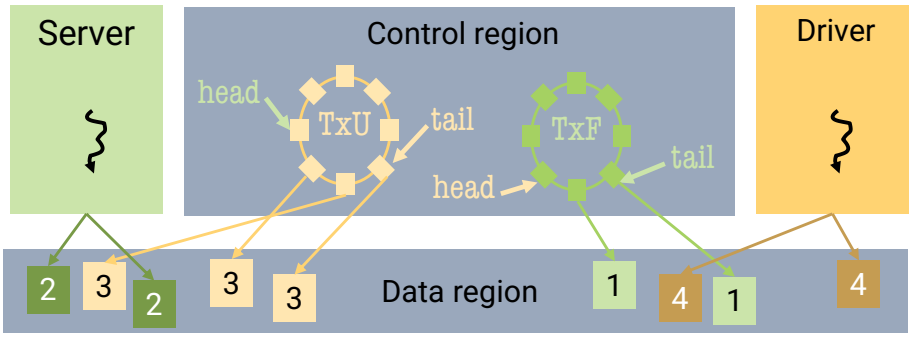


Figure 4.3: Control region between driver and server on transmit path [Parker, 2022]

The ring buffer queues themselves are very simple. Figure 4.4 shows the data structures. Each ring buffer has a separate head and tail pointer, and entries between the tail and head point to a valid buffer in the respective data region. Each ring buffer has exactly one producer (the server in Figure 4.3) and one consumer (the driver in Figure 4.3). The producer only ever updates the head and the consumer only ever updates the tail.

```
struct buffer_descr {
    void   *address;
    size_t length;
}
struct ring_buffer {
    uint32_t head;
    uint32_t tail;
    struct buffer_descr buffer[RING_SIZE];
}
```

Figure 4.4: Ring Buffer Queues

Lock free updates to these queues are possible by utilising the property that reads

and writes of small integers are atomic. We simply use a *write memory barrier* before updating the head or tail pointers as shown in Figure 4.5 to ensure that no reads or writes are re-ordered by the compiler or processor across this point. As the code ensures there is at least one unused buffer between the head and tail, the data race is benign and the memory barrier is sufficient to ensure consistency.

```
bool full(struct ring_buffer *ring)
{
    return (ring->head - ring->tail + 1) % RING_SIZE == 0;
}
bool empty(struct ring_buffer *ring)
{
    return (ring->head - ring->tail) % RING_SIZE == 0;
}
void enqueue(struct buffer_descr *buffer,
             struct ring_buffer *ring)
{
    assert ( !full(ring) );
    ring->buffer[ring->head % RING_SIZE] = *buffer;
    barrier();
    ring->head += 1;
}
struct buffer_descr *dequeue(struct ring_buffer *ring)
{
    struct buffer_descr *buffer;
    if (empty(ring)) {
        return NULL;
    } else {
        *buffer = ring->buffer[ring->tail % RING_SIZE];
        barrier();
        ring->tail += 1;
        return *buffer;
    }
}
```

Figure 4.5: Ring Buffer Queue Management

4.3 Device Sharing

In order to share a device with potentially multiple client applications, the sDDF proposes a simple PD whose sole concern is multiplexing the hardware. This multiplexer is implemented at layer one, meaning each client application has its own virtualised MAC address, and the multiplexer keeps a 1 to 1 mapping of MAC addresses and client

CC ids. Multiplexing at layer two or three would also be possible, but would mean confining applications to a particular protocol or set of ports. The multiplexer can be separated into two separate components. One component handles incoming traffic (Rx Mux), and the other, outgoing traffic (Tx Mux) as shown in Figure 4.6. Control regions are used by each of these components to communicate with the component on either side. In order to prevent clients from the ability to access each others' data, the shared data region must be split into separate pools, and a simple copy component per client is responsible for copying data from one memory region to another.

1. **Shared Rx data region**: This data region is accessed directly by the device for writing newly received data, as well as into the Rx Mux's address space and each of the copy components.
2. **Client Rx data regions**: This data region is mapped into a particular client's address space as well as this client's copy component. There is a separate client Rx data region per client. The copy component copies data from the shared RX data region to the client's own Rx data region.
3. **Client Tx data regions**: This data region is accessed directly by the device to transmit data, but is also mapped into the Tx Mux's address space and a particular client's address space. There is one client Tx data region per client. This enables a zero copy interface on the transmit path.

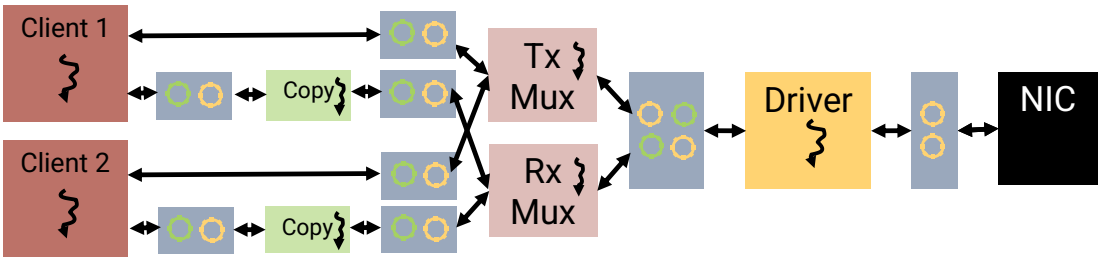


Figure 4.6: Multiple client applications on the sDDF

A multiplexer servicing multiple components inherently requires a policy to determine the order in which it will process work, however, the sDDF is currently limited to a signle client and as such, does not contain any policy at this stage. Furthermore, a layer one multiplexer with multiple client applications requires special handling of broadcast protocols.

4.4 Control Flow

Each component in the sDDF, aside from the client applications, is a single-threaded event-driven program. The components are simply reacting to an update in either the control or metadata regions, signalled by an seL4 notification. To minimise the number

of system calls, each component processes as many buffers as it can before signalling the next relevant component. The driver runs at the highest priority in the framework. This ensures timely handling of interrupts, as well as immediate reaction to requests. In order to prevent the driver from monopolising the processor in the case of high traffic on the receive path, we can limit the size of the receive queue in the metadata region and/or the size of the receive queues in the control regions. The remaining priority ordering is as follows:

$$\text{Driver} > \text{Tx Mux} > \text{Rx Mux} > \text{Clients and per client copiers.}$$

This priority ordering enables components on the receive path to process as many packets as possible and thus batch system calls. Due to the bounded size of the queues, only a limited number of packets can be processed in one invocation which provides flow control for lower priority components. Note that client applications may have different priorities. In the case that client A has higher priority than client B, then it may make sense that client B's copier has lower priority than that of client A. However, to enable batching, clients should run at lower priority than their copy component. On the transmit path, components are invoked as soon as packets are ready to be transmitted which keeps transmit latencies low.

4.4.1 Current Limitations

The prototype, as of the start of this thesis, did not support multiple client applications and was limited to single core systems. As such, the multiplexer components were very simple and did not contain any policies. Furthermore, the design was limited as it expected a client application with symmetric traffic on the receive and transmit paths. In a real networking system, this would not be the case. For example, a web server would have much higher throughput on the transmit path than the receive path. Finally, all previous evaluations were limited to running on a single core only which is not the typical set up for high throughput networking systems.

4.5 Thesis Problem Statement

This thesis addresses challenges in developing and evaluating a networking system based on the seL4 microkernel and the seL4 Device Driver Framework (sDDF). The overarching problem statement encompasses multiple key objectives. Firstly, the aim is to extend and evaluate the sDDF framework to provide flexible and secure support for multiple client applications, addressing diverse networking demands. Secondly, this thesis focuses on assessing the scalability of the framework to multi-core systems, investigating its efficiency across multiple cores to optimize performance in a multi-core environment. Additionally, this thesis aims to evaluate the framework's resilience against untrusted client applications, examining potential security vulnerabilities and proposing a simple solution to protect the subsystem in a given threat scenario from misbehaving clients.

These objectives collectively contribute to advancing microkernel-based networking systems, offering a foundation for flexible, scalable, and secure support for multiple client applications while addressing security concerns associated with untrusted clients.

Chapter 5

Design

To achieve high performance I/O on seL4, this thesis will extend the current seL4 Device Driver Framework prototype to securely support multiple client applications on a multi-core system. While doing so, we will evaluate the sDDF design while improving any bottlenecks in the current implementation and optimise the system for high performance networking systems.

When extending the sDDF, we wish to maintain the current prototype's design goals. Specifically, these are:

- **Radical Simplicity**: Each component in the framework should be kept as simple as possible. While verification is outside the scope of this thesis, keeping each component simple will aid any future verification effort while assisting developers to reason about it.
- **Strict separation of concerns**: Each component has one and only one job to do.
- **Swappable policy**: Any policy enforced by the framework should be swappable for another policy for different use cases. This enables the framework to be flexible and support different use cases while keeping the implementation simple.
- **Secure**: The framework should be secure by taking advantage of the security properties provided by seL4. seL4 guarantees complete isolation of user-level components, and its capability system provides fine grained access control. Untrusted clients must not be able to interfere with one another, nor any of the trusted shared components such as the multiplexers.
- **Performance**: The solution should be performance competitive with the current prototype, as well as with other existing I/O frameworks (see Chapter 3).

To accomplish our goal, we need to:

1. Implement policies for the multiplexers, Section 5.1
2. Support broadcast protocols, Section 5.2
3. Support different client applications that imitate different use cases, Section 5.3
4. Support execution across multiple CPUs, Section 5.4
5. Perform a threat analysis of the framework to outline, and where possible, improve any security vulnerabilities in our design, Section 5.5
6. Evaluate and optimise: evaluate the solution by benchmarking the system throughout development and improve any performance bottlenecks.

5.1 Multiplexer Policy

A multiplexer servicing multiple components inherently requires a policy to determine the order in which it will process work. Rather than designing a generic policy that aims to cater to all possible use-cases, we instead design several different multiplexers that implement a simple policy for a specific use-case.

Both the Rx and Tx multiplexers are trusted components, and thus are kept intentionally simple to leave them accessible to formal verification. The main tasks of the multiplexers, regardless of policy implemented, are as follows:

- Given a virtual address from the client/copier, translate it to a physical address before handing it over to the driver.
- Given a physical address from the driver, translate it to a virtual address before handing it over to the client/copier.
- Sanitise any buffer addresses from the client before forwarding them.
- Sanitise any addresses from the driver.
- Transmit buffers belong to the client, and therefore all free buffers on the transmit path must be returned to the appropriate client.
- Receive buffers belong to the driver, and therefore all free buffers on the receive path must be returned to the driver.

5.1.1 Receive Policy

All incoming packets are processed in FIFO order. The multiplexer contains a mapping of virtualised client MAC addresses and client queues. After dequeuing an incoming packet, the multiplexer must first invalidate the cache (to ensure subsequent reads are fetched again from memory) and read the packet header. The packet header contains the destination MAC address and if the packet is addressed to a client on the system, the buffer address is forwarded to the appropriate client. If a client's Receive Used (RxU) queue is full, the multiplexer drops the packet and the buffer address is returned to the driver. This can be prevented for higher priority clients by appropriate choice of client's RxU queue sizes. Appropriate sizes can be selected based on experiments discussed in Chapter 7.

The multiplexer must also return free buffers to the driver to be reused again. If a clients Receive Free (RxF) queue is full, the multiplexer could stall the client while it waits to enqueue free buffers. Should this be the case, the order in which the multiplexer processes the client RxF queue contains policy as it may unblock clients. We propose instead to ensure the clients RxF queue is the same size as the number of receive buffers circulating and thus prevent the client becoming blocked on waiting for this queue to be processed. This design removes the need for policy on processing free buffers.

5.1.2 Transmit Policy

All incoming requests from clients are processed subject to a particular policy. We implement 3 different policies in Chapter 6:

1. **Round-robin**: whereby the multiplexer processes an outgoing request one at a time, alternating between clients.
2. **Priority-based**: whereby the multiplexer prioritises client requests as per their given priority, processing as many requests as available for the highest priority client possible.
3. **Throughput-based**: whereby the multiplexer limits the available outgoing bandwidth per client at a time.

The transmit multiplexer also returns free buffers from the driver to the client in FIFO order, regardless of policy implemented on processing transmit requests. However, the multiplexer has no way of knowing which client a particular buffer belongs to before it has been dequeued. Should a particular client's Transmit Free (TxF) queue be full, the multiplexer could become blocked waiting to enqueue a free buffer. We propose client TxF queues are at least the size of the number of buffers belonging to that client in order to prevent this.

Similarly, the drivers Transmit Used (TxU) queue should be at least the size of the sum of all client transmit buffers. This ensures the multiplexer will not become blocked on this queue and thus disrupt the policy implemented to process client requests.

5.1.3 Enforcing Policy Through System Design

Depending on the greater system design, we expect some transmit policies not to work. For example, given a single-core configuration, the higher priority multiplexer will always be invoked as soon as a client has made a request to transmit. This means that we can assume any other clients do not yet wish to transmit, or have not yet had the opportunity to run to make a request. Therefore, the multiplexer does not need to consider all the client queues when invoked and can just service requests made by the client that signalled. To enforce policy on multiple clients in such a scenario, we rely on appropriate choices of client queue sizes and the scheduling parameters of each client. For example, to enforce a round-robin policy on two clients, we run both clients at equal priorities, and limit their TxU queues to one. This ensures that each client will be stalled until its single request has been processed, and another client will have the opportunity to run in the interim. The multiplexer will then only process a single request per client at a time, though at the detriment of performance.
Similarly, should we require a priority based policy, we can assign appropriate priorities to the clients scheduling parameters and limit lower priority client's queue sizes, thus relying on the scheduling of the system to enforce this policy.
Finally, we can limit the client's queue sizes on both the transmit and/or receive paths to limit the amount of either transmit and/or receive throughput available to each client. Appropriate sizes can be selected based on experiments discussed in Chapter 7.

5.1.4 Design Constraints

Our multiplexer design has introduced 2 small constraints that will impose on system design. To summarise, these are:

- All client's RxF queues must be the same size or greater than the total number of receive buffers circulating in the system. This is to prevent a client from becoming blocked, waiting for this queue to be processed and thus requiring further policy in the Rx multiplexer to handle free buffers. This constraint has no impact on the implementation, but must be considered when configuring queue sizes across the system.
- All client's TxF queues must be the same size or greater than the number of Tx buffers belonging to that client. If there is a Tx copy component in the system, this applies to the TxF queues between client copiers and the Tx multiplexer.

This is to prevent a client's TxF queue from becoming full, and thus blocking the multiplexer when processing free buffers.

5.2 Broadcast Protocols

Some protocols broadcast traffic to all systems by addressing the Ethernet packet with a specific MAC address of ff:ff:ff:ff. However, as the receive multiplexer is implemented at layer one and thus based on MAC addresses, it will not know how to handle such packets. In particular, we need to support the Address Resolution Protocol (ARP), as it maps an IPv4 address to a MAC address and thus is required for any network communication via Ethernet.
One possible solution is to copy these packets to all client applications. This would require additional policy in the receive multiplexer that ensures all client side RxF queues have enough free buffers available for the multiplexer to dequeue and then copy broadcast packets into. However, there are many different broadcast protocols, for example the Dynamic Host Configuration Protocol (DHCP), and the arrival of such packets can be nondeterministic. This makes it difficult to determine how many might arrive simultaneously and thus how many buffers should be available in all client side RxF queues at all times. If there are no any free buffers available for ARP, then a client will not be able to receive any IPv4 traffic.
Instead, we implement a separate component to handle broadcast traffic. It interfaces with the multiplexers in the same way as any other client, as shown in Figure 5.1.

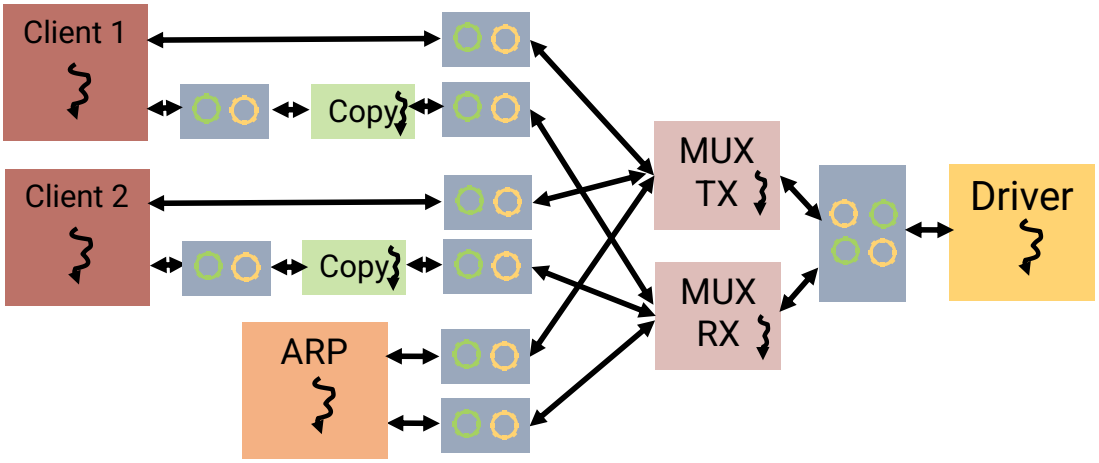


Figure 5.1: Multiple client applications with an ARP component on the sDDF

5.2.1 Separate ARP Component

A separate ARP component only requires the minimal functionality to respond to the Address Resolution Protocol on behalf of any client applications running on the system. In keeping with the design goals, this component will be kept very simple with the aim of

enabling formal verification. Consequently, such a goal also makes this component easy for developers to reason about and debug. Therefore we consider this component to be trusted to maintain the integrity of its shared queues with the multiplexer components as well as interfacing with clients and responding correctly to ARP on behalf of the clients.
While supporting other broadcast protocols, such as DHCP, is out of scope of this thesis, the simplicity of the ARP component design enables its functionality to be extended easily to support other broadcast protocols in the future. Such an extension would entail adding additional packet processing logic to our ARP component.

5.3 Client Applications

The client application consists of a simple echo server and uses lwIP [Dunkels, 2001] as an IP stack library to process the network packet headers. The lwIP API stores packet data, including pointers to the payload, in a `pbuf` struct. The `pbufs` are allocated from static-sized memory pools and can be chained together to produce a scatter-gather list. On the receive path, `pbufs` aren't chained together as the incoming payload already contains all the required packet headers. However, when transmitting a packet via the UDP or TCP API, the `pbuf` will only wrap around the actual payload, and in order to add the required headers, lwIP allocates additional `pbufs` containing these headers that are added to the head of a `pbuf` chain as shown in Figure 5.2.

Figure 5.2: Example lwIP `pbuf` list

Once a packet is ready to be transmitted, a chain of `pbufs` must then all be copied into a single sDDF buffer before being enqueued to the multiplexer. The result of this design means that if there isn't a transmit buffer available for this chain of `pbufs`, the packet is dropped and the `pbufs` are freed. While this is an acceptable outcome, it unfortunately means that a significant amount of packet processing is then wasted. In order to combat this on the simple echo server and reduce wasted cycles, packets are only processed on the receive path if transmit buffers are available. This stalls the client application until it receives a notification that more transmit buffers are available.

However, this solution is very limiting as most networking applications do not have symmetric traffic. For example, a client application could receive a higher number of packets than it transmits, and thus the design stalls the receive processing without reason. On the other hand, a client application with higher transmit demands would bypass the check for transmit buffers and result in packets being dropped after packet processing.

To remove the premature check for transmit buffers on the receive path, we need to temporarily store the transmit context by storing the `pbuf` chains when there are not enough transmit buffers available to the client. Once more transmit free buffers are available (by which the client will be notified), the client can resume the transmit context.

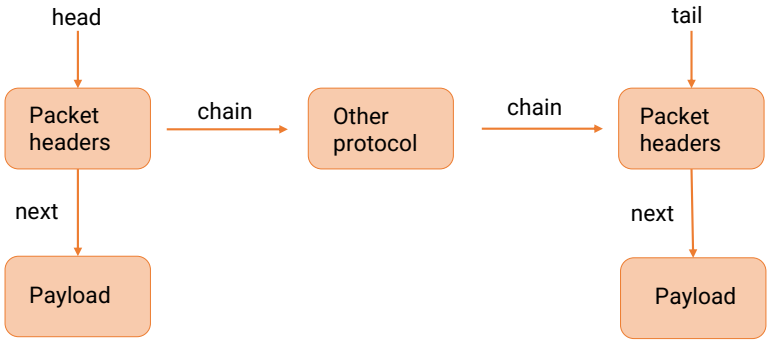


Figure 5.3: Example lwIP `pbuf` chain

We store the transmit context by simply linking `pbuf` chains together and storing the head and tail of the list. Figure 5.3 shows how multiple chains of `pbufs` can be linked together. When a chain of pbufs is ready to be transmitted, but there are not any free transmit buffers, we append the chain to the tail of the list and increment the reference count of the chain. This ensures the `pbufs` won't be freed until they have actually been sent. When a client is notified of the availability of more free transmit buffers, we dequeue `pbuf` chains from the head of this list. This only requires adding a single extra field to lwIP `pbuf` structs. This simple change removes any restriction on the receive path, sets up the client application to no longer expect symmetric traffic. We implement simple, example applications with asymmetric traffic to evaluate how our framework copes with such loads in Chapter 7.

5.4 Multi-core Systems

The sDDF is implemented on top of the seL4 microkit, which already supports multicore systems and so there are minimal changes required to achieve this. As the transport layer is lock-free and system calls are kept to a minimum, the framework itself *should* be capable of exploiting the parallelism of multicore hardware. We explore possible system designs in Chapter 7. Furthermore, it is possible to split the driver into two separate components. One component would be responsible for handling client initiated events (predominantly transmit requests), while the other component would react to hardware events. This separation has the potential to achieve more performance on a multicore system as each of these components could run concurrently on separate cores.

5.4.1 Two-threaded driver on multi-core systems

A device driver has multiple tasks: it must react to hardware events as signalled by an interrupt and it must react to software events, such as a client request. Namely, the driver has 4 tasks:

1. Dequeue incoming packets from the hardware receive ring and enqueue them to the RxU queue.
2. Dequeue free buffers from the RxF queue and enqueue them to the hardware receive ring.
3. Dequeue used transmit buffers from the TxU queue and enqueue them to the hardware transmit ring.
4. Dequeue transmitted buffers from the hardware transmit ring and enqueue them to the TxF queue.

While these tasks involve interfacing with both hardware and software queues, each task involves producing/consuming to different queues and reacting to different events. Task 3 occurs after a client transmit signal, tasks 1 and 4 occur after an interrupt, and task 2 occurs either preemptively after task 1 to ensure the hardware receive ring is ready to be received, or after a client signal that more free buffers are available. We can split the driver up into two separate components and simplify the workload as shown in Figure 5.4.

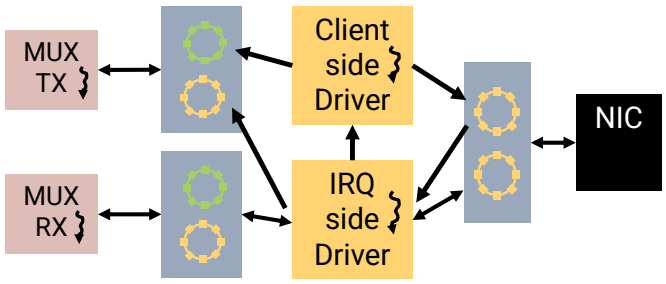


Figure 5.4: Driver architecture with two components

The client side driver is responsible for only task 3. It receives a signal from the transmit multiplexer when packets are ready to be transmitted, and enqueues them to the tail of the hardware transmit ring. If the hardware transmit ring is full, it waits until it receives a notification from the IRQ side driver when space has freed up.
However, introducing an additional component to the framework will incur additional performance overheads due to extra context switches required. We first analyse these overheads in Chapter 7 before concluding whether this design is appropriate on multi-core systems.

5.5 Security

Component communication in the sDDF relies on shared memory and asynchronous notifications. While asynchronous communication has considerable performance benefits (see Chapter 3), it means we trust each component to abide by the protocol. Future work, left out of scope of this thesis, will explore verifying all trusted components. However, the system cannot trust application code. Our design should ensure a misbehaving client application does not have the ability to interfere with other clients or the rest of the system.

Clients receive data by dequeuing metadata, including buffer addresses, from a shared ring buffer, and transmit data by enqueuing such metadata. However, as these shared ring buffers are used by the components on either side of the client (e.g. a multiplexer), we currently trust the client to maintain the integrity of the queues, and not write to buffers during DMA (i.e. once a packet has been enqueued for transmit and before the buffer returns in the free queue). Specifically, we trust the client to:

1. Update the tail pointer when dequeuing incoming packets from its RxU queue.
   While the rest of this queue can be mapped in as read-only to the client, if the client does not correctly increment this pointer, the copy component will incorrectly see this queue as full/empty and potentially could either write over client's data or stop enqueuing new packets. As each client application has its own copy component, this potential vulnerability is isolated only to the misbehaving client.

2. Write the correct metadata to the RxF queue once buffers can be recycled.
   Should a client not do this appropriately, for example, it could give a faulty buffer address/length, the copy component will not use this buffer. This vulnerability is isolated to just the client and its copy component.

3. Update the head pointer when enqueuing free buffers to its RxF queue.
   Like 2., this will only cause the copy component to not reuse newly enqueued buffers.

4. Signal the copy component after enqueuing free buffers to its RxF queue.
   If the client does not inform the copy component of this update and the copy component is blocked on enqueuing incoming packets as there aren't enough free buffers to copy into, the copy component may not wake up. This could block any receive traffic to the misbehaving client but will not affect other components.
   Alternatively, if a client signals its copy component unnecessarily, this has the potential to increase networking latency. As the copy component runs at higher priority than its client, a signal from the client to its copy component will cause an immediate context switch. This has the potential to also affect other applications that run at equal or lower priority than the copy component on the same CPU, as they may delayed to be scheduled as a result. seL4 already provides mechanisms

to protect against such a scenario in scheduling context objects. We can limit copy components' scheduling parameters such that they will not monopolise the CPU. Then, should a client excessively signal its copier, the copy component will use up its scheduling budget and become blocked, thus reducing any Rx bandwidth to that client. These limits will depend on greater system design, in particular, the scheduling parameters of all other applications running on the same CPU.

5. No longer write data to buffers once enqueued to the clients RxF queue.
   Should the client write data to buffers already enqueued in the RxF queue, it will potentially write over incoming data. This will only affect the misbehaving client application, but can also be protected against by mapping Rx buffers to clients as read-only.

6. Enqueue a buffer to the RxF queue only once per use.
   If a client enqueues the same buffer multiple times to the RxF queue, it will potentially cause the copy component to write new incoming data over other used packets. This may mean the client loses Rx packets. Similarly, this will not affect any other applications.

7. Write the correct metadata to the TxU queue.
   Should the client enqueue faulty metadata, such as a faulty address, the transmit multiplexer can detect this by ensuring the address lies inside that clients Tx DMA region, and will refuse this request without impacting any other components.

8. Update the head pointer of the TxU queue.
   If the client does not correctly update this pointer, the multiplexer will not see newly enqueued packets and they may never be sent.

9. No longer write to the packet after its enqueued to the TxU queue.
   Should the multiplexer be responsible for ensuring the packet is well formed with appropriate Ethernet headers, the client could potentially alter this data after sanitation and thus potentially send out corrupted packets. While modern NICs are typically equipped to prevent such errors, we don't want to always assume the device itself is trustworthy and corrupted packets have the potential to compromise other devices on the network.

10. Signal the transmit multiplexer after enqueuing used transmit buffers.
    If the client does not signal the multiplexer, enqueued transmit packets may never be sent. This only affects the client itself. Alternatively, if the client signals the Tx multiplexer unnecessarily, it will increase latencies. Similarly to the receive path, a signal from client to Tx multiplexer causes an immediate context switch. Unnecessarily invoking the multiplexer will delay other applications running on the same core at lower priority than the multiplexer.

11. Update the tail pointer of the TxF queue after dequeuing free transmit buffers.
    If this is not done and the multiplexer incorrectly sees the queue as non empty, it may write over client data and thus potentially overwrite and therefore lose the clients transmit requests. If the multiplexer incorrectly sees the queue as full, it will panic. This is because the multiplexer assumes the client's TxF queues

are never full (as when it dequeues a new free buffer from the driver, it cannot know in advance where to return the buffer). If one client misbehaves, then it may prevent the multiplexer from dequeuing more free buffers from the driver side and returning them to clients. This has the potential to corrupt the entire transmit path for all client applications.

12. Enqueue a buffer to the TxU queue only once per use.
    If a client enqueues the same buffer multiple times to the TxU queue, it will potentially cause the device to send the packet multiple times. This may affect lower priority clients by monopolising the device. An appropriate choice of Tx multiplexer policy can protect against this consequence, and thus only affecting the misbehaving client.

13. Keep track of all buffers.
    Rx and Tx buffers belong to clients and it is up to the client to not lose buffer pointers. However, should a client lose buffers, it could potentially limit that clients available throughput. Clients do not have access to shared buffers so this will not affect any other clients.

From the above analysis, and assuming all components other than the client applications are trustworthy, we can conclude the receive path is secure. A trusted component, the copier, sits between the untrusted client application and the shared multiplexer, and in each potential vulnerability whereby the client does not abide by the protocol, the copy component will not enqueue/dequeue more incoming packets and thus starve only the misbehaving client. This assumes the RxU queue between the receive multiplexer and copy component is appropriately sized. Should this queue size exceed the number of shared buffers available for all clients, and a client misbehaves, it could potentially starve other clients. This is because the copier of a misbehaving client will not enqueue more incoming packets to the client (either temporarily or permanently), and the RxU queue to the copier could then fill up. If there are limited shared buffers available and a high proportion are enqueued to the copier of a faulty client, there will be less buffers available for other clients. To prevent such a scenario, we ensure there are enough shared buffers on the receive path for all RxU queues between the multiplexer and each copy component to be full at the same time.

However, the transmit path is not protected from misbehaving client applications. While scenarios 7, 8 and 13 only impact the client itself, scenario 9 has the potential to compromise other devices on the network without a trusted device, and scenario 11 can block all transmit traffic, including that of other clients.

To remove these vulnerabilities, we can mimic the receive path by injecting a small, trusted component in between each untrusted client application and the shared multiplexer as shown in Figure 5.5. The component can be configured as per the system design, or even removed all together, as required by the threat scenario.

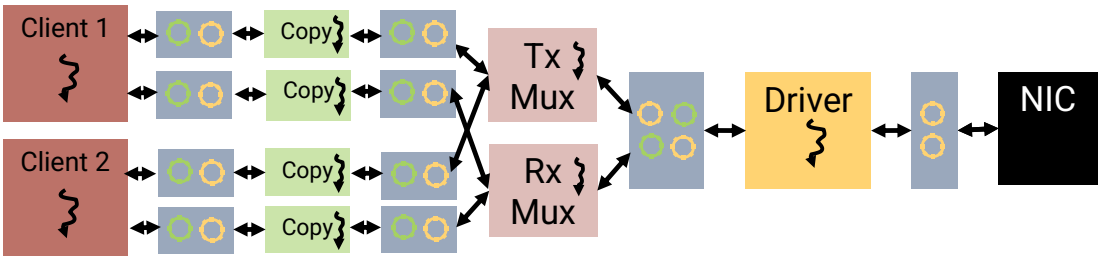


Figure 5.5: Architecture with an additional transmit copy component

To interface with a completely untrusted client application, this new component needs to:

- Ensure metadata enqueued to the multiplexer is sane.
- If the device is untrusted, check outgoing packets are well-formed and copy them into a separate region not mapped into the clients address space.
- Interface with the multiplexer correctly by incrementing the head/tail pointers as required.
- Ignore any transmit packets that do not abide by the protocols outlined above.

Like the receive path copy components, should a client misbehave when interacting with the shared queues, or fail to signal as per the protocol, the new Tx Copy component will ignore any requests or no longer enqueue free buffers to the client, thus only disturbing that client and not affecting any other components in the system.
Contrary to the Rx Copy component, this does not introduce a shared Tx data region. This additional component would introduce some additional performance overheads. These include the cost of reading packet headers, an extra system call on the transmit path, and potentially the cost of copying the entire packet. We measure these overheads in Chapter 7.

Chapter 6

Implementation

Prior to this thesis, the seL4 Device Driver Framework already supported a minimal, single client networking system. This included an Ethernet driver, layer-one Rx multiplexer (limited to a single client), copy component, a Tx multiplexer (also limited to a single client and containing no policy) and a simple echo server client application. To properly support multiple client applications, we extend the Rx multiplexer to multiplex to multiple different client applications, and implement 3 new transmit multiplexers. We also implement a simple timer driver to interface with our transmit multiplexer when required, and finally, a new ARP component to handle broadcast traffic. We configure all queues between these components to further implement different policies in Chapter 7.

6.1 Transmit Multiplexer Policies

We implement 3 different transmit multiplexers, each implementing a different policy. We present our pseudo code designs for how each multiplexer process transmit packets from the client. All multiplexers return free buffers from the driver to their respective clients in the same FIFO order and hence this function has been omitted for brevity. Notably, the implementations are very simple, aiding any debugging effort required, but also easing future formal verification endeavours.

6.1.1 Round-robin Policy

We implement a round-robin policy by processing a single client's transmit request at a time. This is done in a loop to ensure we process any batched requests as outlined in Figure 6.1.

```
void process_transmit_ready() {
  enqueued = 0;
  old_enqueued = 0;

  while(!ring_full(driver->used_ring)) {
    old_enqueued = enqueued;
    for each client {
      /* Process a single used buffer at a time */
      if (!ring_empty(client->used_ring) &&
          !ring_full(driver->used_ring)) {

        buf = dequeue(client->used_ring);
        phys = get_phys_addr(buf);
        enqueue(driver->used_ring, phys);
        enqueued += 1;
      }
    }
      /* we haven't processed any packets since
        last loop, exit */
    if (old_enqueued == enqueued) break;
  }
}
```

Figure 6.1: Transmit round-robin Policy

6.1.2 Priority-based Policy

We implement a priority-based policy in Figure 6.2. We assume the list of clients is ordered from highest priority to lowest. Thus it is possible to change a clients priority at run time by manipulating this list.

```
void process_transmit_ready() {
  enqueued = 0;
  old_enqueued = 0;
  while(!ring_full(driver->used_ring)) {
    old_enqueued = enqueued;
    for each client {
      while(!ring_empty(client->used_ring) &&
            !ring_full(driver->used_ring)) {

        buf = dequeue(client->used_ring);
        phys = get_phys_addr(buf);
        enqueue(driver->used_ring, phys);

        enqueued += 1;
        /* if a higher priority client has since
            made a request, go again */
        if (!any_ring_empty(clients[:client])) break;
      }
    }

    if (old_enqueued == new_enqueued) break;
  }
}
```

Figure 6.2: Transmit Priority-based Policy

6.1.3 Bandwidth-limited Policy

We implement a bandwidth-limited policy in Figure 6.3. The multiplexer will process client transmit requests, calculating how much bandwidth the client has used based on each request size in a given time window. Should a client exceed its limit within the time window, the multiplexer will request a time out from the timer driver for the remaining time. As the multiplexer must regularly communicate with the timer driver to get the time and set timeouts, we design a simple timer driver API as shown in Figure 6.4.

```
void process_transmit_ready() {
  curr_time = get_time();
  for each client {
    if (curr_time - client->last >= TIME_WNDW) {
      client->last = curr_time;
      client->bandwidth = 0;
    }

    while (!ring_empty(client->used_ring) &&
           !ring_full(driver->used_ring) &&
           client->bandwidth < client->max) {

      buf = dequeue(client->used_ring);
      phys = get_phys_addr(buf);
      enqueue(driver->used_ring, phys);
      /* recalculate the clients bandwidth used in
        the current time period */
      client->bandwidth = calculate_bandwidth(buf_size);
    }

    if (!ring_empty(client->used_ring) && !client->timeout) {
      set_timeout(TIME_WNDW - curr_time - client->last);
      client->timeout = true;
    }
  }
}
```

Figure 6.3: Transmit Bandwidth-limited Policy

```
uint64_t get_time() {
  microkit_ppcall(TIMER, microkit_msginfo_new(GET_TIME, 0));
  return microkit_mr_get(0);
}

void set_timeout(uint64_t micros) {
  microkit_mr_set(0, micros);
  microkit_ppcall(TIMER, microkit_msginfo_new(SET_TIMEOUT, 1));
}
```

Figure 6.4: Timer Driver API

The timer driver is a high priority, passive server. It receives interrupts from the device, and forwards these by notifying the appropriate client or multiplexer. The bandwidth-limited multiplexer reacts to such notifications by clearing the appropriate client's timeout and recalling process_transmit_ready. Note that this implementation could be made more efficient by mapping the timer registers read-only into the multiplexer's address space. This would remove the need for a frequent system call to get

the time. However, these implementations are meant only as example prototype rather than a prescribed implementation and further optimisations have been left for use-case specific implementations.

This bandwidth-limited policy can also be combined with the round-robin or priority based policy to order each clients requests. Such a case would be useful in reducing the transmit latencies of latency sensitive applications.

6.2 ARP Component

We implement a separate ARP component that is responsible for handling all broadcast traffic. As client MAC addresses are assigned at system design time, our ARP component has a static mapping of client IDs to virtualised MAC addresses. For client IDs, we use the clients communication channel ID as assigned by microkit.

6.2.1 Client Interface

As ARP is based on both MAC addresses and IPv4 addresses, the new component requires an interface with client applications. This interface will allow clients to register a new IPv4 address, change an IPv4 address, or remove one. As adding, changing and removing an IPv4 address is typically very infrequent for networking systems and requires minimal data exchange, we define this interface using a protected procedure call (PPC) from the client to the ARP component. The ARP component can use both the client ID and the MAC address provided in the arguments to store the clients IPv4 address.

```
void register_ip4(new_ip4_addr, mac_addr[6])
{
  /* split the MAC address across two registers so it fits */
  microkit_mr_set(0, mac_addr[0:4]);
  microkit_mr_set(1, mac_addr[4:6]);
  microkit_mr_set(2, new_ip4_addr);
  microkit_ppcall(ARP, microkit_msginfo_new(REGISTER_IP, 3));
}

void change_ip4(old_ip4_addr, new_ip4_addr, mac_addr[6])
{
  microkit_mr_set(0, mac_addr[0:4]);
  microkit_mr_set(1, mac_addr[4:6]);
  microkit_mr_set(2, old_ip4_addr);
  microkit_mr_set(3, new_ip4_addr);
  microkit_ppcall(ARP, microkit_msginfo_new(CHANGE_IP, 4));
}

void remove_ip4(old_ip4_addr, mac_addr[6])
{
  microkit_mr_set(0, mac_addr[0:4]);
  microkit_mr_set(1, mac_addr[4:6]);
  microkit_mr_set(2, old_ip4_addr);
  microkit_ppcall(ARP, microkit_msginfo_new(REMOVE_IP, 3));
}
```

Figure 6.5: Client Interface to ARP Component

Currently, the ARP component can only store a single IPv4 address per client, which is enough for our testing purposes, however this limit can be extended easily in the code base should the need arise.

Chapter 7

Evaluation

We now evaluate and analyse our designs using a series of different benchmarks and running the system under different configurations. We run all benchmarks on the Freescale i.MX 8M Mini quad applications processor with 2GB RAM, running at 1.2 GHz with a 1Gbps NIC. The configurations for each system tested are specified by each benchmark, and the system comprises of the following PDs:

1. Ethernet Driver
2. (only as specified) Client Side Ethernet Driver
3. Rx Mux
4. Tx Mux: implementation specified in each benchmark
5. ARP Component
6. Timer Driver
7. 1 Rx Copy Component per Client application
8. 1 or more UDP Client applications as specified in each benchmark. The clients use lwIP [Dunkels, 2001] as an IP stack library for network packet processing.
9. (only as specified) Tx Copy Component

Figure 7.1 shows our evaluation set up for a single client application. For all echo applications, we use ipbench as a distributed load generator to send 1472 byte-sized UDP packets to test the seL4-based system. ipbench runs on a 4-node x86 cluster, connected to the same switch as our test system, and counts the number of successful replies from the target system to measure the achieved throughput and latencies. All benchmarks use a sample size of 200,000 packets unless otherwise specified.

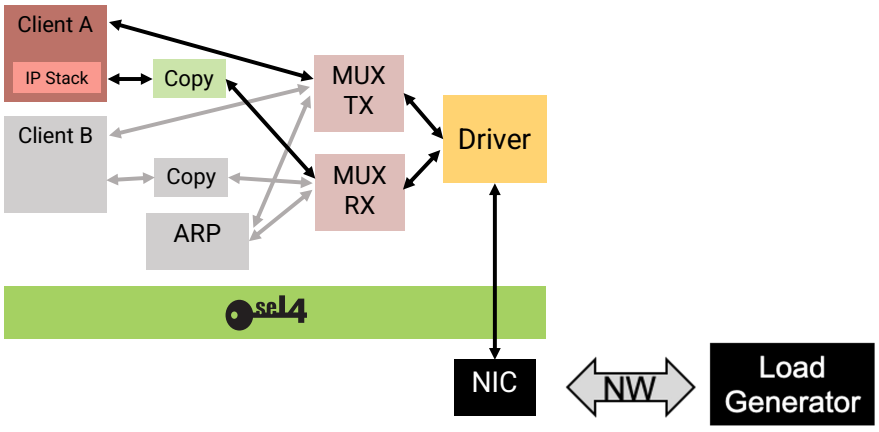


Figure 7.1: seL4 evaluation setup

For some benchmarks, we purposely overload our echo applications. A typical networking system would perform some computation on incoming data, such as a HTTP web server, which would need to fetch web pages from memory to process incoming HTTP requests. Although developing such a test scenario is out of scope for this thesis, we still want to test our framework under high loads where the CPU cannot keep up with the traffic to ascertain the absence of a performance collapse, as well as ensuring that the system abides by any policy implemented in a multi-client set up. We alter our echo application to perform 10 additional data copies and checksum calculations where specified.

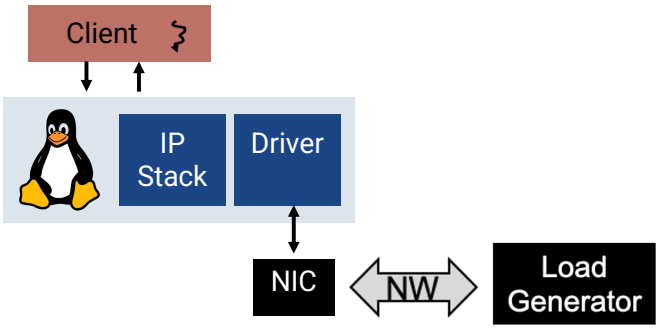


Figure 7.2: Linux evaluation setup

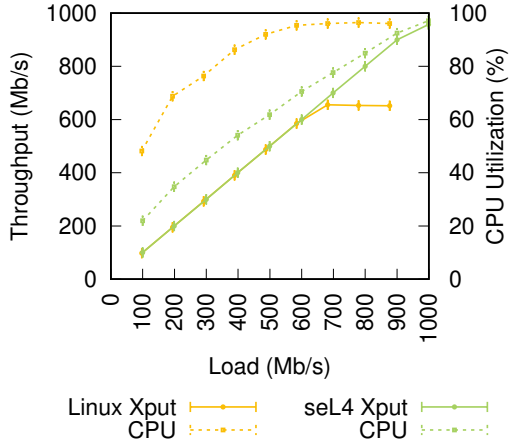
We also compare some echo server configurations against a Linux system (version 6.1.1) on the same hardware. Figure 7.2 shows our evaluation setup. We use a simple user-level application which reads and then writes all packets immediately back using the socket API. In order to determine the CPU utilisation of both systems, we run an idle thread on each core at low-priority to count the number of idle cycles of that core.

For all asymmetric client applications, we implement our own custom benchmarking applications to run on an Intel Xeon W-1250 running at 3.3 Ghz, with 64GB RAM and equipped with a 10Gbps NIC. For our transmit-dominant client, our benchmarking

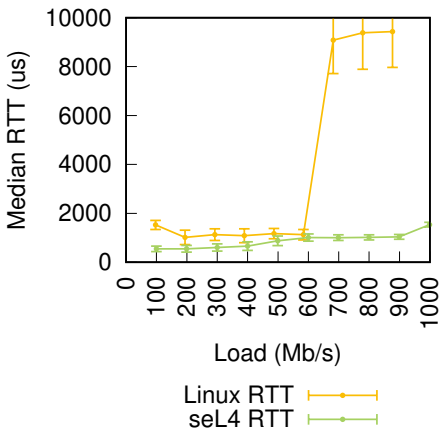
application receives packets using *recv()* and records start and stop time to determine the total transmitted throughput. For our receive-dominant client, our benchmarking application transmits packets using *send()* and we count the number of successfully received packets by the client itself to determine the total received throughput.

7.1 Single Client Performance

We first establish a baseline for our system by measuring the networking performance of a single echo server application. We also compare against a Linux system and can already establish that our design outperforms a typical monolithic setup for simple echo servers.



(a) Linux vs seL4 Networking Performance



(b) Linux vs seL4 Networking Latencies

Figure 7.3: Networking performance

Figure 7.3a compares the performance (achieved throughput and CPU utilisation over applied load) of both systems. We can see that the seL4-based system's throughput scales with applied load and manages to saturate the wire, while Linux scales linearly to about 650 Mb/s and then plateaus. When comparing the CPU utilisation of both systems, we can see that the reason behind this plateau in throughput is because Linux uses significantly more CPU cycles to handle a given load than the seL4-based system. Furthermore, the CPU utilisation of our system also scales sub-linearly with increasing load which indicates efficient batching. The batching is a result of the sDDF's asynchronous transport layer, which enables data to be passed around in batches and thus minimises the number of system calls requried per packet.

Figure 7.3b compares the round trip time (RTT) latency with standard deviations over applied load of both systems. The latency measured is the median result taken from a

sample size of 200,000 packets. We can see that the sDDF demonstrates a much faster round trip time over Linux.

Despite the high number of system calls required in our design, this evaluation demonstrates the minimal impact this count has on overall performance. The seL4-based system requires roughly 15K cycles per packet, whereas the Linux based system requires 43K cycles per packet. An asynchronous signal on seL4 is less than 1000 cycles [seL4 Foundation, 2023], thus an additional system call per packet would only add an overhead of roughly 6%. Compared with the cost of packet processing, this is insignificant, whereas a system call in a monolithic kernel can be of the order of 10s of thousands of cycles.

7.2 Asymmetric Client Applications

Now that we can support asymmetric traffic in the client application, we implement simple example applications that emulate different applications that could be deployed on a real system. We wish to test applications with different requirements: an application with higher transmit demands and minimal incoming traffic, an application with higher receive demands and minimal outgoing traffic and finally, an application that initiates communication before transmitting large amounts of data.

7.2.1 Transmit-dominant application

We implement a simple client application that transmits 10,000 UDP packets, of 1472 bytes each, for every UDP packet received. Ethernet maximum transmission unit is 1518 bytes, which leaves a maximum of 1472 bytes in payload once you account for 14 bytes in Ethernet headers and 32 bytes for UDP/IP headers. In order to interface with lwIP to initiate additional transmits, we first allocate a `pbuf`, and write 1472 bytes of data to the `pbuf`'s payload. As `pbufs` are allocated from a memory pool, they can also run out. Should this occur, we temporarily stop creating transmit requests and store the number of packets we have already sent so we can transmit the rest once more memory has freed up. The bottleneck on memory availability for lwIP `pbufs` occurs because `pbufs` are queued while the application waits for more transmit buffers as outlined above. Thus, when transmit buffers become available, so to does more memory for `pbufs`. We can thus resume transmitting once the transmit multiplexer has notified the application that more transmit buffers are available. The benefit of temporarily pausing creating transmit requests is that it also ensures the application is available to process any incoming traffic in the interim.

7.2.2 Receive-dominant application

We also implement a simple receive dominant client application that for every 10,000 UDP packets received, it transmits a single one. This application is much simpler to implement as all memory management is already dealt with so we just adapt the echo server to only echo every 10,000th packet.

7.2.3 Client Initiated Transmit

Not all networking devices act as servers that only react to incoming traffic. A common use case of a non-reactive application is a web client which initiates network communication with another networking device. To test such a scenario, we implement another client application which first initiates a handshake with another device, then transmits 1,000,000 packets. The initial handshake is there to ensure the external benchmarking application is ready to receive all incoming packets. We interface with lwIP as described in Section 7.2.1.

7.2.4 Results

We now measure the total received throughput by a receive-dominant application (Rx Mostly), the transmitted throughput by a transmit-dominant application (Tx Mostly) and the transmitted throughput of a client-initiated transmit application (Tx Initiated) and compare these tests against the achieved throughput of the echo server.

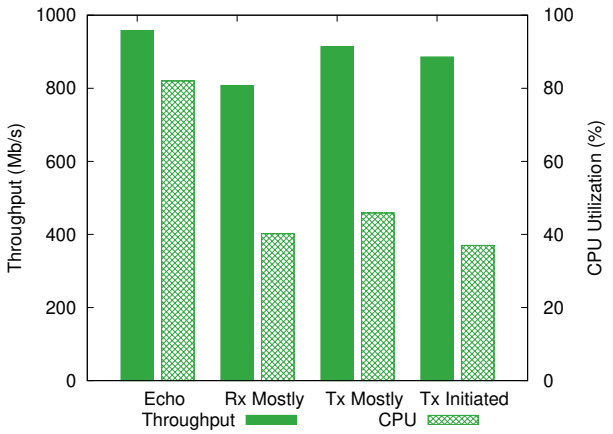


Figure 7.4: Networking performance of different client applications.

Figure 7.4 shows the measured throughput as well as the CPU utilisation of each test application. Compared with our echo benchmarks, our asymmetric applications both utilise 50% less CPU for 80+% of the throughput. While we would be able to achieve higher throughput in these examples by receiving/sending more than 10,000 packets for every packet in the other direction, thus skewing the ratio of send/receive in favour

of throughput, this would defeat the purpose of these examples as we are only wanting to test asymmetric traffic as opposed to uni-directional. From both of these results, we can conclude our system is able to cope well with asymmetric UDP traffic. In our client-initiated benchmark, we are able to achieve 885Mbps with less than 40% CPU. When considering the cycle count cost per packet, this is inline with our expectations. To achieve wire speed, the echo benchmarks require roughly 8.7K cycles per packet, or can be considered roughly 4.3K cycles per packet per direction. Our client-initiated transmit benchmarks require roughly 3.9K cycles per packet, or, 90% of the cycle count to achieve 92% of the throughput. Although these tests are fairly simple, they provide a good sanity check to confirm our system can keep up with different workloads and present a good starting point for developing more complex workloads in the future.

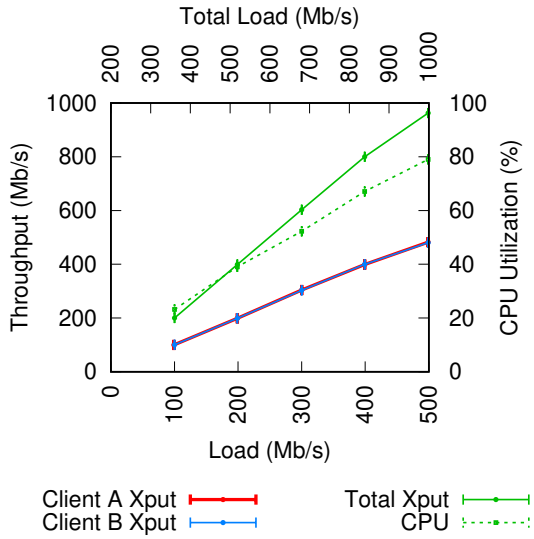
7.3 Single-core, Multi-client systems

Our design has a large number of different parameters to experiment with, including multiplexerf policies, queue sizes and the scheduling parameters of client applications running on the system. We select a small variety of example systems to evaluate our different policy designs. We evaluate these systems with two separate ipbench instances running on different clusters sending UDP packets at the same time to their respective clients. The order in which they arrive at our system entirely depends on the order in which packets arrive at the network switch connecting our system to the clusters running ipbench.

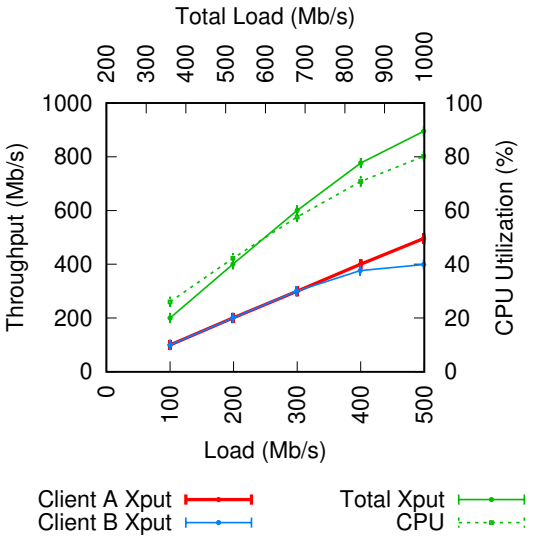
We start first with evaluating single-core configurations running two echo server applications. Each system comprises of an Ethernet driver, Rx Mux, Tx Mux, 2 copy components (1 for client A, 1 for client B) and two simple echo servers; client A and client B. As these systems run on single-core, a round-robin or priority-based policy in the Tx Mux would not make sense as the multiplexer runs at higher priority than both the clients and thus will be invoked as soon as it is signalled by either client. We instead first evaluate a simple Tx Mux that only responds to the client that signalled it, and enforce policy through our system design. We also evaluate our bandwidth-limited multiplexer on its ability to efficiently enforce different bandwidth limits for different clients.

7.3.1 Enforcing Policy through System Design

We first evaluate a system where both clients run at equal priority, with their respective copy components at equal, higher priority than their client, and both clients have a queue size of 512 for all interfacing queues. We rely on the kernel scheduler to ensure fairness and that clients are scheduled in round-robin order.



(a) Networking performance of two echo servers at equal priorities.



(b) Networking performance of two echo servers at equal priorities, with limited RxU queue of client B.

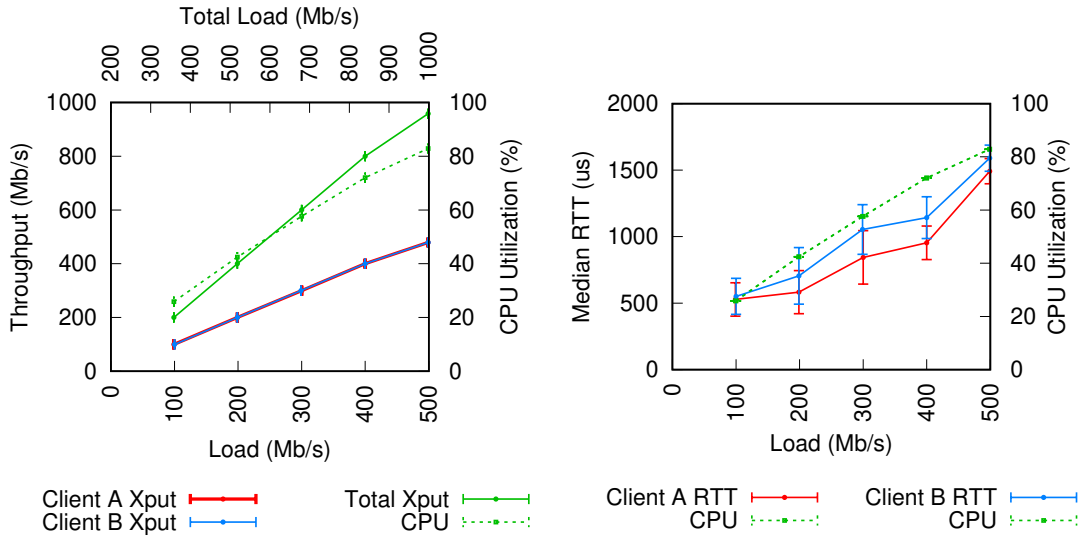
Figure 7.5a shows the achieved throughput for client A and B and the CPU utilisation of the entire system. Both clients achieve the same throughput, and their combined total reaches wire speed.

We now limit the RxU queue between the Rx Mux and the copy component of client B to just 16 to measure the impact this will have on client B's throughput. Figure 7.5b shows the resulting network performance. Client A still achieves the requested throughput but client B is limited to 400Mbps. This is because the RxU queue to client B will become full much sooner, causing the Rx Mux to discard subsequent packets for client B and thus limiting the total throughput available to client B.

We now incorporate different priorities to our system and measure the impact this has on each clients achieved performance. We reduce both client B and its copy component to lower priority than client A. Thus the priorities are ordered as follows:

7. Driver
6. Tx Mux
5. Rx Mux
4. ARP
4. Copier of client A
3. Client A
2. Copier of client B
1. Client B

We set all queue sizes to be equal.



(a) Throughput achieved by each client.

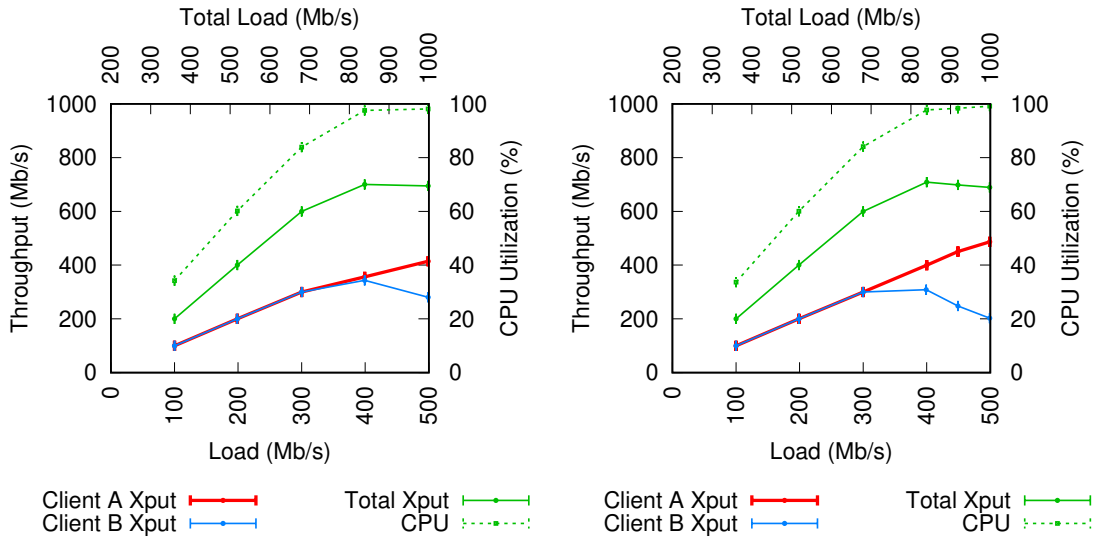
(b) Median latencies for each client.

Figure 7.6: Networking performance of two echo servers at different priorities.

We see in Figure 7.6a the throughput of the lower priority client remains unaffected as there isn't enough contention on the CPU to limit this client's CPU bandwidth and both clients achieve their requested load. However, the latencies are affected by this. Figure 7.6b shows the median round trip time (RTT) for each client. Client A achieves lower latencies than client B as it is running at higher priority and thus is scheduled to run first.

If there was more contention for the CPU, then we would expect the priority assignment

of each client to have an effect on their achieved throughput. We now add extra overhead to each packet's round trip to simulate an environment where this is the case. We have each client copy each packet and calculate a checksum 10 times.



(a) Equal queue sizes.

(b) Limit RxU queue of client B to 32.

Figure 7.7: Networking performance of two overloaded echo servers at different priorities.

Figure 7.7a shows the achieved throughput for both clients. Neither client achieves its requested load, with client A, the higher priority client, maxing out at 400Mbps and client B seeing a performance collapse to 300Mbps. By the total achieved throughput and CPU load, it is clear the system is overloaded at this point. However, it is clear that client B as the low-priority client is starving client A by monopolising shared Rx buffers, as client A is not able to achieve its requested load. We now limit client B's RxU queue to 32 to prevent this and ensure there are adequate Rx buffers available to higher priority clients as per Section 5.1.1. Figure 7.7b shows the resulting performance. The total throughput achieved is not affected by this change, and the system still becomes overloaded at 400Mbps. However, client A is now able to achieve its requested load as client B's available Rx throughput is limited.

The results all indicate that limiting a client's queues will limit its available throughput. We can use this property to implement a bandwidth-limited policy in the system. We now limit the RxU queue between the Mux and Copier of client B to just 16, and also limit the TxU queue between the Mux and client B to 16. Client B and its Copier still run at lower priority than client A, and both clients are just simple echo servers (with no additional overload).

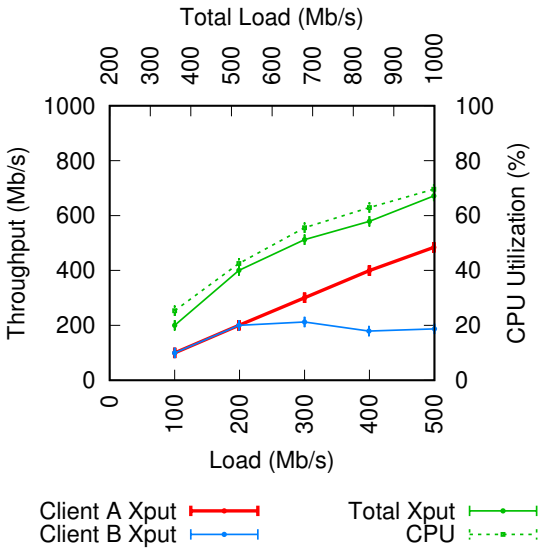


Figure 7.8: Networking performance of two echo servers with limited queues to client B.

Figure 7.8 shows the resulting performance. Client A still achieves its requested load, but client B is limited to just 200Mbps. The total CPU load is also reduced compared to the system without any queue limits, indicating less load on the limited client.

We now take a closer look at how limiting a clients Tx queues impacts its available throughput. We benchmark a single echo client for maximum throughput, and vary the size of both the clients TxU and TxF queues.

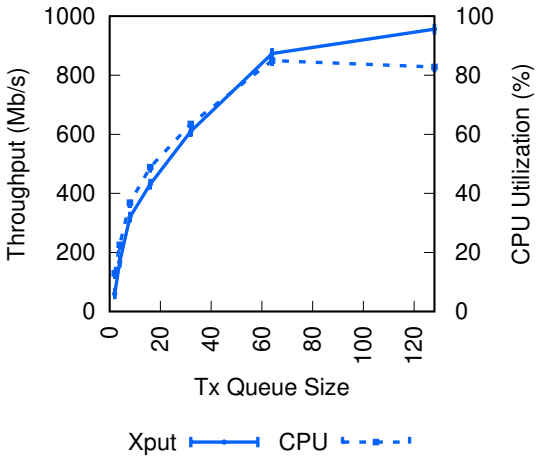


Figure 7.9: Impact of queue sizes on networking performance.

Figure 7.9 shows the throughput achieved and CPU utilisation against the increasing queue sizes. Both measurements increase dramatically before plateauing. Notably, there is an efficiency improvement shown in the CPU utilisation as throughput increases

and the queue size doubles from 64 to 128. This is because a limited transmit queue for an echo application causes incoming packets to be potentially processed by the driver and multiplexer before being discarded as the client's RxU queue is full. This occurs in echo applications due to a shortage of transmit buffers stalling a client's ability to process packets. The result is that some excess work is done on packets that get discarded, leading to a higher average cycle cost per packet. Ultimately, despite this overhead, we see that limiting a client's transmit queues can effectively limit the available transmit throughput to that client, and with a queue size of 128, a client is able to achieve wire speed.

7.3.2 Bandwidth-limited Tx Multiplexer

We can also limit client's available bandwidth using policy in the Tx Mux. We evaluate Section 6.1.3 with two echo servers, running at equal priority and with equal queue sizes, and limit client B to just 100Mbps in the Tx Mux. Client A is not limited.

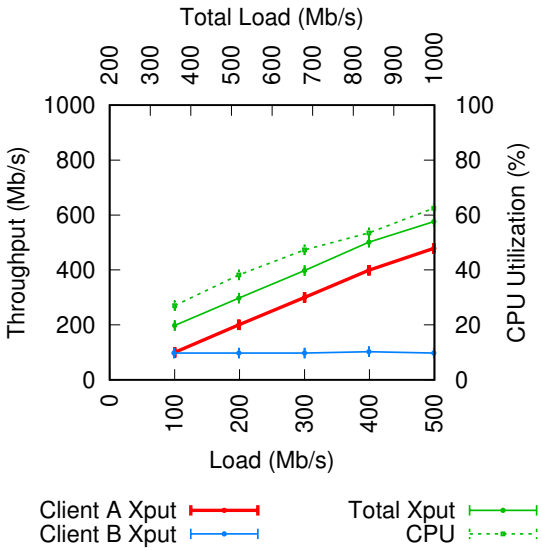


Figure 7.10: Networking performance of two echo servers with bandwidth-limited multiplexer.

Figure 7.10 shows the achieved throughput for both clients and the total CPU load of the system. As expected, client A which does not have a bandwidth limit, achieves its requested load and client B is limited to just 100Mbps. Furthermore, the total CPU load for the total achieved throughput (600Mbps) is only 62%. Compared to the CPU load of the same achieved throughput without any bandwidth limiting (see Figure 7.5a) of 52%, this is less than a 20% increase. While we expect there to be some overhead as the multiplexer must communicate regularly with a timer driver, thus incurring additional context switches, some of this overhead can be reduced with an optimised implementation as outlined in Section 6.1.3. This is left for future work.

In order to properly evaluate our round-robin and priority-based multiplexer designs, we must first evaluate how our design scales to multi-core. From there, we can design systems where the clients run on separate cores and will be scheduled irrespective of one another, thus requiring further policy in the multiplexer.

7.4 Multi-core

Scaling to multi-core will incur overhead from two separate causes. Firstly, seL4 configured with symmetric multi-processing (SMP) increases the cost of every kernel invocation. This is not only due to the cost of acquiring the kernel lock, but also because the SMP kernel is less optimised and some fast paths do not apply. Secondly, inter-core communication costs significantly more than intra-core due to the addition of inter-processor interrupts (IPI) for inter-core signalling and cache-line bouncing from any shared data.

While detailed analysis of these overheads is left for future work, in order to properly analyse how our system performs on multi-core configurations, we first establish a baseline of these costs. We first evaluate two separate configurations using the SMP kernel: pinning every component to a single-core and splitting the load across two cores (out of an available 4).

We compare these to our baseline without the SMP kernel. Each system is comprised of 5 PDs: an Ethernet driver, RX Mux, TX Mux, Copy Component, Client running a simple echo server. Our two core system pins the driver and Tx Mux to one core, and the Rx Mux, Copier and Client pinned to the other. We also compare these systems against a typical Linux set up, whereby the user-space echo server application is pinned to one core, and we allow the Linux scheduler to schedule the remaining 3 cores.

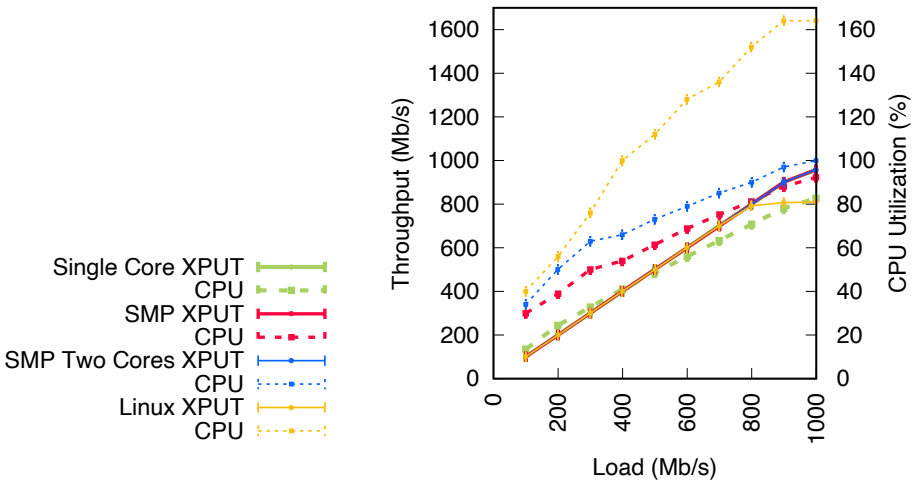


Figure 7.11: Networking performance comparison of multi-core systems.

Figure 7.11 shows the achieved throughput against the CPU utilisation for our 3 different seL4-based systems and Linux. As expected, each of our 3 seL4-based systems achieve wire speed (the green and red lines are directly behind the blue line) whereas the Linux system maxes out at just 800Mbps. All seL4 systems also outperform the Linux system, which utilises more than 1.6 cores for 800Mbps. Further investigation into the Linux system reveals the core running the echo server is maxed out at this point, and hence why the system can't achieve higher throughput.

However, the total CPU utilisation of each of our seL4 systems differs substantially. The difference between the single-core CPU load (dotted green line) and the single-core SMP CPU load (dotted red line) outlines the overhead of using the SMP kernel. To achieve 100Mbps, the normal kernel configuration utilises 13.5% of the CPU, whereas the SMP kernel configuration utilises 29.9% of the CPU; about 121% overhead. While this is substantially higher than expected given there is no inter-core communication for either of these configurations, detailed exploration of these overheads is left for future work. Comparing the single-core SMP CPU load against the two core SMP CPU load (dotted blue line) outlines the overheads introduced by inter-core communication between our 5 PDs; just over 20% at 100Mbps. Note that not all system calls will introduce IPIs in this configuration, as, for example, the Copy component only communicates intra-core with the Rx Mux and Client.

Ultimately, distributing PDs across different cores will incur different overheads. We explore these overheads with another benchmark comparing 4 separate configurations. Each different configuration isolates one PD on one core, and runs all other PDs on another core. We test these configurations with the same echo server benchmark with the same 5 PDs as our previous multi-core benchmark.

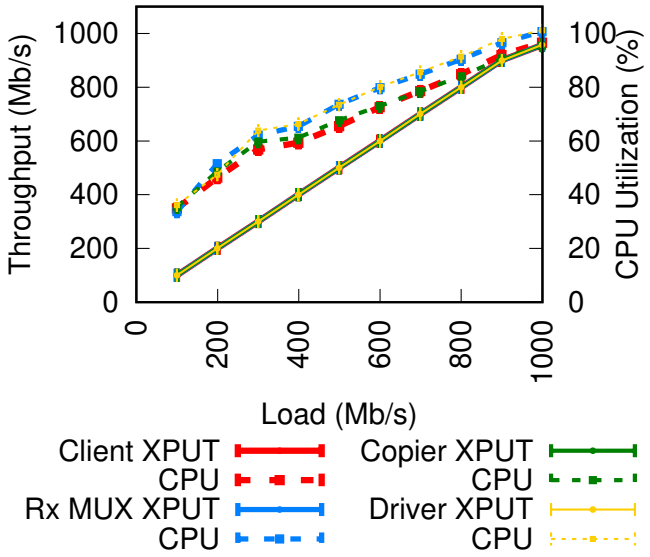


Figure 7.12: CPU utilisation with select components isolated on a separate core.

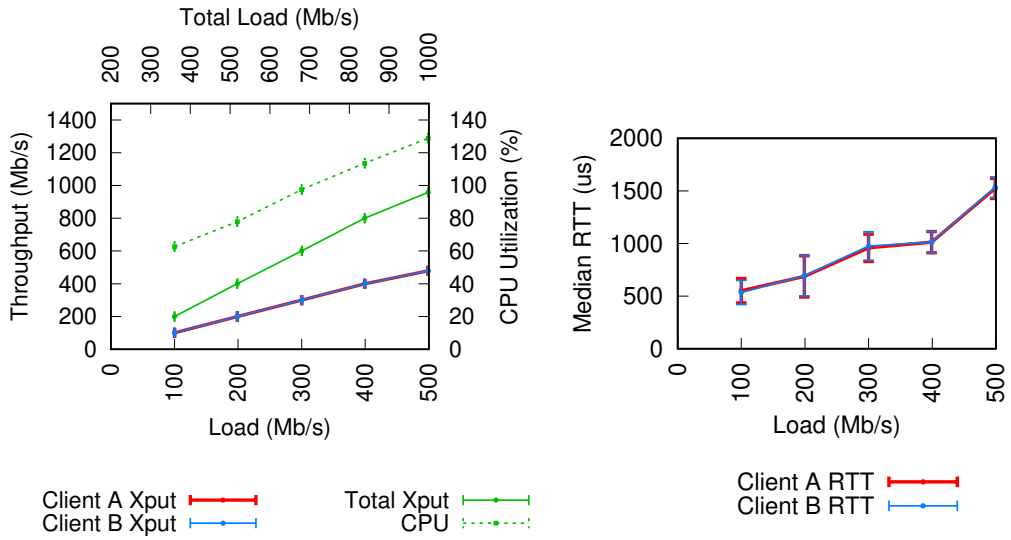
Figure 7.12 shows the results when isolating the driver, Rx Mux, Copier and Client. The yellow lines show the throughput and CPU utilisation when the driver is pinned to a separate core. Likewise, the red for the Client, the blue for the Rx Mux and green for the Copier. As expected, each different configuration achieves wire speed but the CPU utilisation of each system differs. Notably, the configuration where the driver is isolated incurs the most overhead, followed closely by the configuration isolating the Rx Mux. This is expected due to the higher number of system calls per packet each of these PDs makes and thus these systems will incur more overheads from intra-core signalling and kernel lock contention. Overall, the difference in CPU load of each of these systems is only 4%, indicating that components can be arbitrarily distributed across cores. Ultimately, the optimal allocation for a system is use case dependent.

Now we have a baseline evaluation for multi-core systems, we can design simple systems with which to evaluate our other transmit multiplexer designs.

7.4.1 Round-robin Tx Multiplexer

A round-robin policy in our Tx Multiplexer only makes sense when clients are running on separate cores. This is because the Tx Mux runs at higher priority than the clients, and will be invoked as soon as a client has signalled it. If both clients are on the same core, then the Tx Mux will be invoked after one client has finished enqueuing packets for transmit, but before the other client has been scheduled. We evaluate our round-robin Tx Mux on a 2 client multi-core configuration, where each client is pinned to a separate core shared with its respective copy component, and another core hosts the

Driver, Rx Mux, ARP Component and Tx Mux.



(a) Throughput achieved and CPU load of two echo servers.

(b) Median latencies of two echo servers.

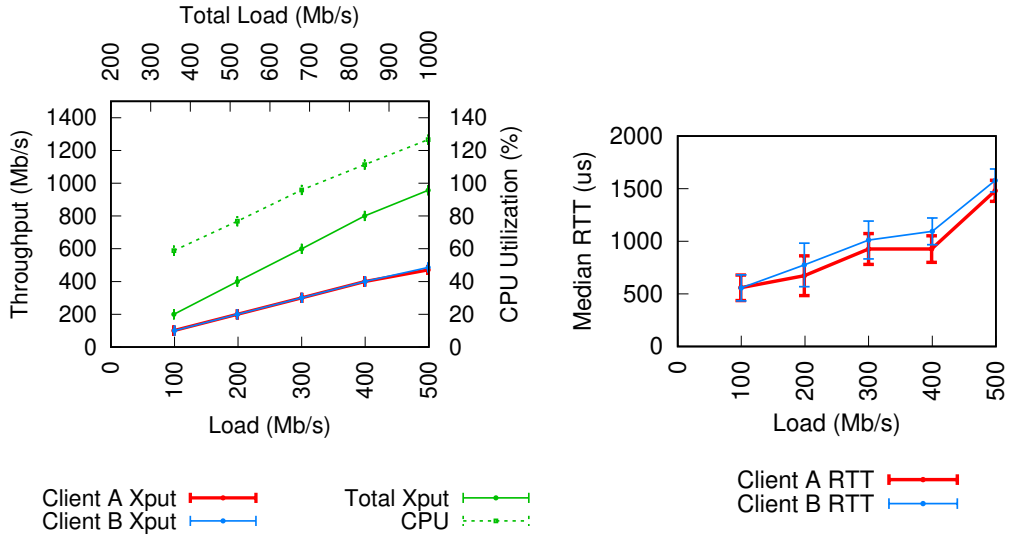
Figure 7.13: Networking performance of two echo servers with a round-robin multiplexer

The total achieved throughput as well as that of each client is shown in Figure 7.13a. As expected, both clients achieve their requested throughput, with their sum equalling wire speed. Importantly, both clients experience the same networking latencies as shown in Figure 7.13b by their respective median round trip times. This demonstrates our round-robin Tx Mux enforces fairness between two clients effectively.

The system incurs significant overhead when compared to our equal priority, single-core example in Figure 7.5a. To achieve a total of 200Mbps (100Mbps per client), our multi-core system utilises 63% of the CPU (split across 3 cores) compared with only 23.5% CPU utilisation of our single-core example. To better understand these overheads, we run our equal priority, single-core example again (see Figure 7.5a) but with the SMP kernel instead. We find that to achieve a total throughput of 200Mbps, the system utilises 51% of the CPU, and the system maxes out at just over 700Mbps. From this we can conclude that most of the overheads in these systems stem from just using the SMP kernel, and an additional 22% accounts for the costly intra-core communication across 3 cores as well as any overheads of the multiplexer itself.

7.4.2 Priority-based Tx Multiplexer

We also evaluate our priority-based Tx Mux using the same system design, where we will instead expect a higher priority client to achieve lower latencies. We set up our priority-based Tx Mux implementation to prioritise client A, followed by client B and finally ARP at lowest priority. As both clients run on separate cores, their scheduling parameters with respect to one another has no impact and so we allocate them the same scheduling priority as one another. Likewise with their copy components, which are higher priority than their client and running on the same core.



(a) Throughput achieved and CPU load of two echo servers.

(b) Median latencies of two echo servers.

Figure 7.14: Networking performance of two echo servers with a priority-based multiplexer.

Both clients achieve their requested throughput as shown in Figure 7.14a and as expected, Figure 7.14b shows that client A has lower latencies over client B. Notably, client A's latencies are also lower on this system, with a median RTT of 1480us at wire speed, when compared to our system with a round-robin policy, where client A has a median RTT of 1524us at the same throughput. This demonstrates our policy implemented in the Tx Mux works effectively to prioritise client A over client B.
Furthermore, both our round-robin multiplexer and our priority-based multiplexer introduce similar overheads as indicated by their respective CPU utilisation measurements. We conclude from these experiments that our round-robin and priority-based multiplexers work effectively, with each experiment displaying expected consequences of the policy enforced by the multiplexer, but more work is needed in investigating the

SMP kernel to develop more efficient multi-core systems.

7.4.3 Two-threaded Driver

For some multi-core systems, the driver could be a performance bottleneck if the core it is running on is saturated. We can offload some of the driver's work to another core with our two-threaded driver design. In doing so, we will introduce additional overheads stemming from an additional context switch per packet, as well as communication between the two driver components required at high loads. We first measure these overheads by comparing our single-core system with a simple echo server client against the same system with our two threaded driver design. We compare the overall throughput and CPU utilisation of both systems. As our two-threaded driver design only makes sense in multi-core systems, we configure both systems with SMP enabled but initially confine them to a single-core.

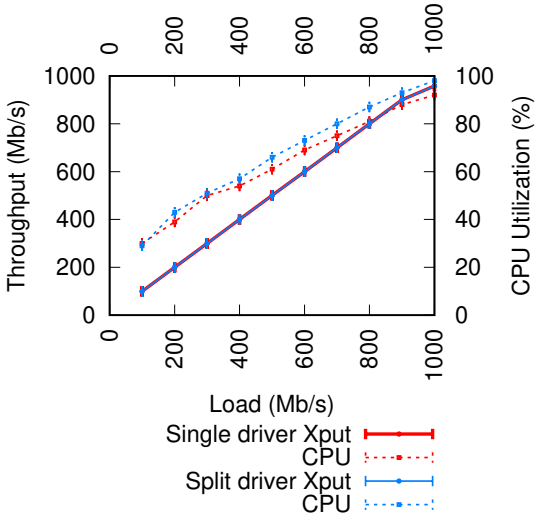


Figure 7.15: Performance comparison of two-threaded driver on multi-core.

Figure 7.15 shows the results. Both systems achieve the total requested throughput. The CPU utilisation of our two-threaded driver (pink dotted line) shows that at low throughputs, the overhead is minimal. However, at high loads, we see a 6% increase in CPU load. We hypothesise this overhead is due to the extra notification required between the two driver components when the hardware transmit queue becomes full (an unlikely occurrence at low loads).

We now design a system where the two-threaded driver design may be applicable. The driver will only be a bottleneck in a multi-core system when the core it is running on is saturated, and there are no other available cores with sufficient bandwidth to run

the driver on. To replicate such a scenario, we configure our system with 4 compute-heavy client applications. We set up 4 echo servers to copy each packet 100 times and calculate a checksum 100 times. We configure each client to run on its own core, along with its copy component, and use our round-robin transmit multiplexer to ensure fairness across the clients. As our hardware platform only has 4 cores, the driver must share a core with at least one client and its copy component. The core assignment is as follows:

Core 0: Driver, Copier of client A, client A

Core 1: Rx Mux, Copier of client B, client B

Core 2: Copier of client C, client C

Core 3: Tx Mux, Copier of client D, client D

We test this system with 4 separate ipbench instances, each sending UDP packets to a client and counting received packets to determine throughput achieved by this client. We request up to 250Mbps for each client for a combined total of 1Gbps on the system.

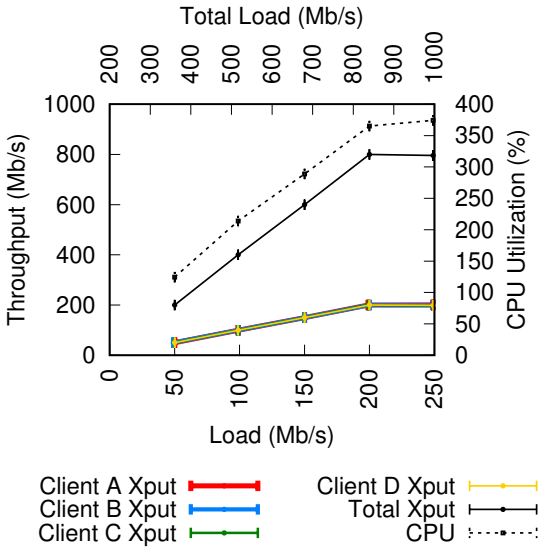


Figure 7.16: Networking performance with 4 clients on multi-core.

Figure 7.16 shows the achieved throughput for each client, as well as the total throughput achieved by the system and the total CPU load. Each client is able to achieve 200Mbps but no more. We now configure this system with our two-threaded driver, and offload some of the driver's work to core 2 to test whether clients are able to achieve higher load. The core assignment is as follows:

Core 0: IRQ-side Driver, Copier of client A, client A

Core 1: Rx Mux, Copier of client B, client B

Core 2: Client-side Driver, Copier of client C, client C

Core 3: Tx Mux, Copier of client D, client D

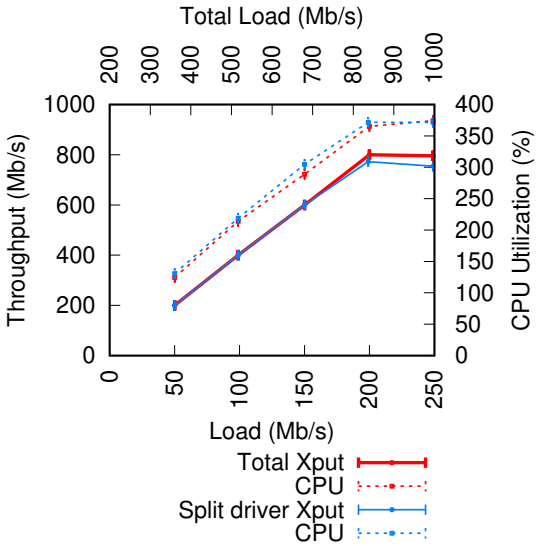
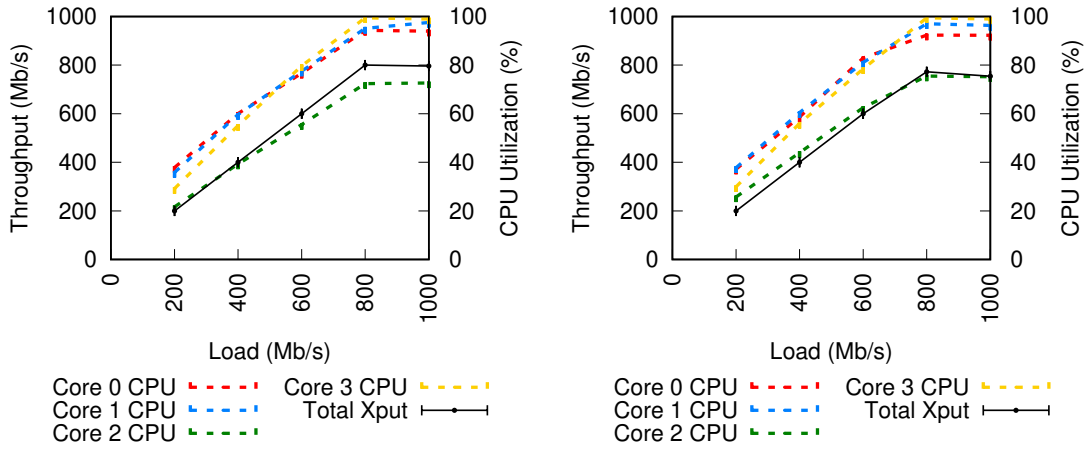


Figure 7.17: Performance comparison of two-threaded driver with 4 clients on multi-core.

Figure 7.17 shows the performance comparison of both systems total throughput achieved and CPU load. The system with the two-threaded driver actually achieves slightly lower throughput than the system with the single driver at the same CPU load. We now take a closer look at the CPU utilisation of each core.



(a) Single driver system.

(b) Two-threaded driver system.

Figure 7.18: Core utilisation of an overloaded 4 client system.

Figure 7.18 shows each core's utilisation against the requested load for both systems. In both systems cores 0, 1 and 3 are all saturated. This tells us that although the driver is a potential bottleneck, both the Rx and Tx multiplexers are too. In Figure 7.18b we see core 0 has slightly lower utilisation at wire speed compared with Figure 7.18b, showing that our two-threaded driver design has successfully offloaded some of the driver load to another core. However, as our multiplexers are also both bottlenecks in this example workload, the two-threaded driver design is not enough to improve performance. We hypothesise that splitting the multiplexers into two separate components, one component responsible for processing used packets and the other for processing free packets, may help reduce load on those cores, but such a design is only applicable on systems with more available CPUs. We leave this exploration for future work.

7.5 Transmit Copy Component

We now measure the overheads introduced by our additional transmit copy component. We implement a new component between the Tx Mux and our client application as per Figure 5.5. While the implementation of this component depends on system design and the threat scenario, we measure the total possible overhead added if the component is responsible for all possible tasks listed. These are:

- Ensuring metadata enqueued to the multiplexer is sane.
- Checking outgoing packets are well formed
- Copying outgoing packets into a separate region not mapped into the client's address space.

- Interfacing with the multiplexer correctly by incrementing the head/tail pointers as required.
- Ignoring any transmit packets from the client that do not abide by the protocols.

We measure these overheads on a simple echo server system. We run this new component at higher priority than the client and the Rx Copy component to ensure invocation as soon as packets are ready to be transmitted and thus keep latencies low. We use a simple Tx Mux configured for a single client and the priority assignment of the system is as follows:

Driver > Tx Mux > Tx Copy > Rx Mux > Rx Copy > Client

We compare the achieved throughput and CPU utilisation of this system to the same system without this additional component.

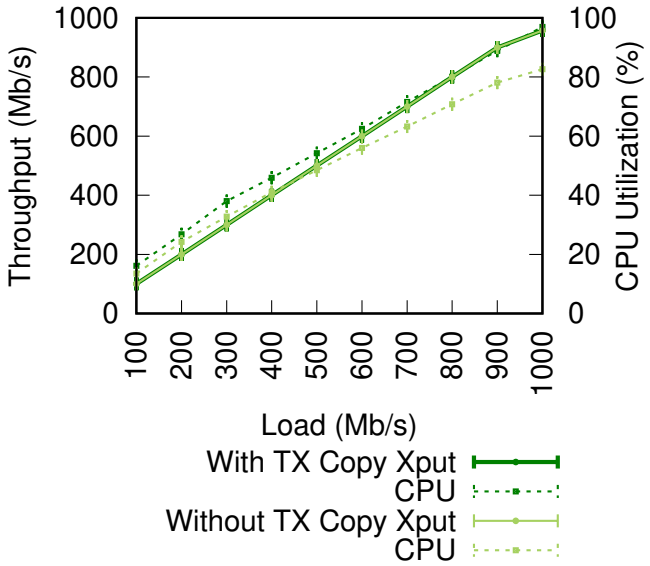


Figure 7.19: Performance overhead with Tx Copy component.

Figure 7.19 shows the resulting performance with and without the Tx Copy component. As expected, both systems achieve wire speed. At 100Mbps, our system utilises 16% CPU with our Tx Copy component, compared to 13.5% CPU without the additional component. This is about an 18% overhead. To achieve 958Mbps, our system utilises 97% CPU with our Tx Copy component, compared with 82.5% CPU utilisation without it, thus showing a 17% increase. This overhead is inline with our expectations as we are comparing systems with 6 and 5 components respectively, thus a 20% increase in component count alone. Of course, this overhead can be reduced depending on the threat model, as some systems might rely on the NIC to ensure packets are well formed, or use the IOMMU to sanitise all buffer addresses.

7.6 Summary

Overall, our single client echo benchmarks outperform Linux significantly, both in single and multi-core set ups. This demonstrates a highly componentised, microkernel-based architecture is a feasible design despite the extra number of context switches required. Our exploration into asymmetric traffic demonstrates our framework can handle such loads efficiently.

Our multi-client results indicate the framework design, along with our new multiplexer designs can flexibly implement a variety of different policies that may be required depending on system use-case. However, not all combinations make sense and system design heavily influences inner policies. For example, Figure 7.7a demonstrated that a higher priority client could be starved by a lower priority client without appropriate choice of queue size, and simply limiting a clients queue sizes can reduce a clients available throughput. System designers can use these properties to implement a use-case specific policy

Although our design scales well to multiple clients, our investigation into multi-core systems has unveiled significant overheads from just enabling the SMP configuration in seL4. Thorough investigation of these overheads is left for future work. In spite of such overheads, our design can be arbitrarily distributed across cores and effectively enables high-throughput networking for compute-heavy clients when a single CPU is insufficient. From our analysis, we conclude our two-threaded driver design is not beneficial due to the overheads outweighing any possible benefit in systems where the driver is a bottleneck in throughput.

Finally, although a security analysis of our framework reveals a misbehaving client does have the ability to compromise the entire transmit path for all clients, these vulnerabilities can be easily eliminated with an additional component between the client and the shared multiplexer. This component introduces acceptable overheads for the stronger security guarantees provided, but can also be easily configured for the systems threat scenario and performance requirements.

Chapter 8

Conclusions

Traditional networking systems often grapple with significant challenges related to security, performance, or a combination of both. This thesis conducted an evaluation of the seL4 Device Driver Framework for networking systems, expanding its functionality to flexibly support multiple client applications on multi-core systems. The highly modularized design of the sDDF not only surpasses the performance of traditional monolithic setups in Linux, despite the number of context switches required, but also provides strong fault containment for historically bug-prone code. This is furthered by the intentional simplicity of each component, which drastically minimised the debugging effort required for this thesis and will ease any future work beyond the scope of this project. This simplicity also enables future formal verification, which will only strengthen the security of seL4-based networking systems.

This thesis delved into the ways in which system designers can impose diverse policies through varied, plug-compatible multiplexer implementations, configuration of client-facing queues, adjustment of client scheduling parameters, and strategic pinning of components across cores. The example systems we evaluated provide a strong starting point when tailoring these parameters to configure a system optimally for its particular use case.

System designers should also consider the performance trade-offs when contemplating a single or multi-core system. Our evaluation unveiled substantial overheads associated with the SMP configuration of seL4, a factor that may be acceptable for compute-intensive applications. Despite this, we demonstrated that the design can be arbitrarily distributed across cores to cater to systems when a single CPU is insufficient.

Furthermore, we conducted a security analysis to assess potential vulnerabilities in our design when faced with an untrusted client application. In response, we introduced an additional component to safeguard the subsystem from potential compromise. Despite

the introduced overheads, this component can be easily configured to suit a particular system's threat scenario and performance requirements.

In conclusion, our evaluation and extension of the seL4 Device Driver Framework debunks the performance myths surrounding microkernel-based designs, demonstrating performance surpassing traditional monolithic setups. Significantly, such a design also provides stronger security guarrantees while proving itself adaptable for varying use cases.

8.1 Future Work

The seL4 Device Driver Framework is not yet a fully functional networking framework as there are a number of features not yet supported. Firstly, our evaluation was confined to UDP only. TCP is a more complex protocol, requiring outgoing packets to be acknowledged by the receiver. This means the IP stack must temporarily store buffers until an acknowledgement message is received to enable missed packets to be resent. As such, lwIP does not flexibly support zero-copy TCP and changes to the IP stack would be required to properly support this if lwIP is to be used with our framework. Additionally, to support TCP, further consideration must also be taken for client initiated TCP transmit using the sDDF as the client must poll incoming queues to receive acknowledgement packets. This thesis has omitted a deeper evaluation of TCP and left it for future work.

Our performance comparisons against Linux systems were limited to the synchronous socket API. Linux also supports a number of asynchronous designs, as discussed in Chapter 3, which would provide an interesting comparison against our work. Obtaining these would require porting the framework to an x86 architecture with a 10Gbps NIC.

This thesis was confined to static architectures. The sDDF aims to support minimal dynamicism by enabling the stopping and restarting of components as well dynamically changing policies. For example, this thesis could be extended to permit a trusted component to detect a misbehaving client and restart the client application. Multiplexer components could be swapped out at run time to enable swappable policy enforcement, however the framework as it stands does not support this and further work is required in measuring the performance cost of such a feature.

Additionally, our multiplexers were implemented at layer one, though multiplexing at layer two or three may be more practical for some systems as it would remove the onus on the system designer to set up hardware and protocol addresses for all clients, as well as the need for the ARP component. A layer two implementation would require

a separate, trusted DHCP client that can resolve DHCP for all client applications and thus ensure no address collision on the network. A layer three implementation would mean each client uses the same protocol and shares an IP address, but is confined to a set of ports. A separate DHCP client would still be required to assign and manage the shared IP address.

Our evaluation of multi-core systems revealed significant overhead in the SMP configuration of seL4. This warrants a thorough investigation to properly understand and optimise these overheads.

Finally, although our design goals enable our solution to be formally verified, this was left out of scope of this thesis. The TCB is significantly smaller in our design compared to a monolithic system, but the microkernel, our driver, multiplexers and copy components are all trusted. seL4 is already formally verified, however, a bug in one of our other trusted components could potentially compromise the system. The simplicity of each of these components means they are accessible to automatic verification methods. Furthermore, the communication protocols between components means a missed notification could deadlock the entire system. Fortunately, these protocols can be verified for the absence of such a case using model checking.

Bibliography

[Axboe, 2019] Axboe, J. (2019). Efficient IO with io_uring. `https://kernel.dk/io_uring.pdf`.

[Dunkels, 2001] Dunkels, A. (2001). Minimal TCP/IP implementation with proxy support. Technical Report T2001-20, SICS. `http://www.sics.se/~adam/thesis.pdf`.

[Emmerich et al., 2018] Emmerich, P., Pudelko, M., Bauer, S., Huber, S., Zwickl, T., and Carle, G. (2018). User space network drivers. In *Proceedings of the Applied Networking Research Workshop*, page 91–93.

[Ganapathy et al., 2008] Ganapathy, V., Renzelmann, M. J., Balakrishnan, A., Swift, M. M., and Jha, S. (2008). The design and implementation of microdrivers. In *Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIII)*, pages 168–178, Seattle, WA, US.

[Haecki et al., 2019] Haecki, R., Humbel, L., Achermann, R., Cock, D., Schwyn, D., and Roscoe, T. (2019). CleanQ: a lightweight, uniform, formally specified interface for intra-machine data transfer. In *ArXiv*, volume abs/1911.08773.

[Heiser et al., 2022] Heiser, G., Parker, L., Chubb, P., Velickovic, I., and Leslie, B. (2022). Can we put the "S" into IoT? In *IEEE World Forum on Internet of Things*, Yokohama, JP.

[Ispoglou et al., 2018] Ispoglou, K. K., AlBassam, B., Jaeger, T., and Payer, M. (2018). Block oriented programming: Automating data-only attacks. In *ACM Conference on Computer and Communications Security*, pages 1868–1882, Toronto, ON, Canada.

[Klein et al., 2014] Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., and Heiser, G. (2014). Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70.

[Koch, 2006] Koch, H.-J. (2006). The userspace I/O how to. `https://www.kernel.org/doc/html/v4.12/driver-api/uio-howto.html`. Accessed: 2023-03-22.

[Kuznetsov et al., 2014] Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R., and Song, D. (2014). Code-Pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, USA. USENIX Association.

[Leslie et al., 2005] Leslie, B., Chubb, P., FitzRoy-Dale, N., Götz, S., Gray, C., Macpherson, L., Potts, D., Shen, Y. R., Elphinstone, K., and Heiser, G. (2005). User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664.

[Linux Foundation, 2020] Linux Foundation (2020). Myth-busting DPDK in 2020. `https://www.dpdk.org/about/news/`.

[Linux Kernel Organization, Inc., 2023] Linux Kernel Organization, Inc. (2023). The Linux Kernel Archives. `https://www.kernel.org/`. Accessed: 2023-04-12.

[Linux manual page, 2023] Linux manual page (2023). socket(2). `https://man7.org/linux/man-pages/man2/socket.2.html`. Accessed: 2023-04-04.

[LKDDb, 2023] LKDDb (2023). Linux Kernel Driver DataBase. `https://cateee.net/lkddb/`. Accessed: 2023-03-20.

[Marty et al., 2019] Marty, M., de Kruijf, M., Adriaens, J., Alfeld, C., Bauer, S., Contavalli, C., Dalton, M., Dukkipati, N., Evans, W. C., Gribble, S., Kidd, N., Kononov, R., Kumar, G., Mauer, C., Musick, E., Olson, L., Rubow, E., Ryan, M., Springborn, K., Turner, P., Valancius, V., Wang, X., and Vahdat, A. (2019). Snap: a microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, Huntington, Ont, CA.

[Mi et al., 2019] Mi, Z., Li, D., Yang, Z., Wang, X., and Chen, H. (2019). SkyBridge: Fast and secure inter-process communication for microkernels. In *Proceedings of the 14th EuroSys Conference*, Dresden, DE.

[MITRE Corporation, 2023] MITRE Corporation (2023). Linux ≫ linux kernel: Security vulnerabilities. `https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33`. Accessed: 2023-03-20.

[Narayanan et al., 2019] Narayanan, V., Balasubramanian, A., Jacobsen, C., Spall, S., Bauer, S., and Quigley, M. (2019). LXDs: Towards isolation of kernel subsystems. In *USENIX Annual Technical Conference*.

[Narayanan et al., 2020] Narayanan, V., Huang, Y., Tan, G., Jaeger, T., and Burtsev, A. (2020). Lightweight kernel isolation with virtualization and VM functions. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM.

[Parker, 2022] Parker, L. (2022). The seL4 Device Driver Framework (sDDF). Talk at the seL4 Summit.

[Ren et al., 2019] Ren, X. J., Rodrigues, K., Chen, L., Vega, C., Stumm, M., and Yuan, D. (2019). An analysis of performance evolution of Linux's core operations. In *ACM Symposium on Operating Systems Principles*, Huntsville, Ont, CA. ACM.

[Rizzo, 2012] Rizzo, L. (2012). netmap: a novel framework for fast packet I/O. In *USENIX Security Symposium*, pages 101–112.

[seL4 Foundation, 2023] seL4 Foundation (2023). Performance. `https://sel4.systems/About/Performance/home.pml`.

[Swift et al., 2002] Swift, M. M., Marting, S., Levy, H. M., and Eggers, S. G. (2002). Nooks: An architecture for reliable device drivers. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, pages 101–107, St Emilion, FR.

[The Kernel Development Community, 2006] The Kernel Development Community (2006). VFIO "Virtual Function I/O". `https://www.kernel.org/doc/html/next/driver-api/vfio.html`. Accessed: 2023-03-22.

[Wang et al., 2022] Wang, J., Behrens, D., Fu, M., Oberhauser, L., Oberhauser, J., Lei, J., Chen, G., Härtig, H., and Chen, H. (2022). BBQ: A block-based bounded queue for exchanging data and profiling. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*.