School of Computer Science & Engineering
Trustworthy Systems Group

sDDF Design

Design, Implementation and Evaluation of the seL4 Device Driver Framework

Gernot Heiser, Peter Chubb, Alex Brown, Julia Broady, Courtney Darville, Lucy Parker

gernot@unsw.edu.au

Release 0.6, 2025-03-21

Abstract

This is a work-in-progress report that documents the design of a high-performance device driver framework for seL4, including the structure and interfaces of compliant drivers, and presents some preliminary evaluation.

The document is intentionally explicit about the assumptions it makes on hardware (the *device model*) and the structure it prescribes on device drivers (the *driver model*). This is to facilitate exploring formal specification and, eventually, verification of device drivers.

Consequently, besides specifying how drivers and their interfaces are structured, the document also serves to define the context for the *Pancake* project that develops a programming language for verifiable device drivers. As such, the report serves as an informal interface document between the Pancake team and systems researchers (and thus explains many things systems people take for granted).

Contents

List of Figures v

List of Listings vi

Acronyms vii

1 Aims of the sDDF 1

2 Threat Model 2

3 Device Model 3
3.1 Device interfaces . . . 4
3.2 Device interface protocol . . . 5
3.3 Device operation . . . 5
3.4 Comments . . . 6

4 Drivers and Their Software Interface 8
4.1 Overview . . . 8
4.2 Transport layer . . . 9
4.2.1 Memory regions . . . 9
4.2.2 Data region . . . 11
4.2.3 Metadata regions . . . 12
4.2.4 Device metadata region . . . 14
4.3 Component models . . . 16
4.3.1 Driver . . . 16
4.3.2 Virtualiser . . . 16
4.3.3 Observations . . . 17
4.4 Synchronisation . . . 18
4.4.1 Active driver-thread model . . . 18
4.4.2 Passive driver-thread model . . . 21
4.4.3 Two-component driver . . . 22
4.4.4 Time-triggered architectures . . . 23
4.4.5 Discussion . . . 23

5 Device Classes 25
5.1 Network . . . 25
5.1.1 Main properties . . . 25
5.1.2 Ethernet . . . 26
5.1.3 Status . . . 26

5.2 Serial ports . . . 26
5.3 Serial busses . . . 27
5.4 Storage . . . 28
5.4.1 Main properties . . . 28
5.4.2 Details . . . 29
5.4.3 Status . . . 32
5.5 Sound . . . 32
5.5.1 Main properties . . . 32
5.5.2 Message Protocol . . . 33
5.5.3 Playback . . . 33
5.5.4 Recording . . . 34
5.5.5 Sound Virtualisers . . . 34
5.5.6 Status . . . 35
5.6 Display . . . 35
5.7 Camera . . . 35

6 Hotplugging **36**
6.1 Overview . . . 36
6.2 Requirements . . . 36
6.3 Design . . . 37
6.3.1 Requirement 1: Insertion and Ejection . . . 37
6.3.2 Requirement 2: Policy-free Virtualiser Remapping . . . 38
6.3.3 Requirement 3: ‘Safe’ Ejection . . . 39
6.4 Limitations . . . 40

7 Device Discovery **41**

8 Leveraging Linux **42**
8.1 Component development . . . 42
8.2 Legacy driver reuse . . . 44

9 Security Analysis **45**
9.1 Trusted components . . . 45
9.2 Verifying components . . . 45
9.3 Verifying component interactions . . . 46

10 Implementation Status **47**

11 Performance **48**
11.1 Performance evaluation setup . . . 48
11.2 Performance results . . . 49
11.2.1 Simplified system, CAmkES, Linux . . . 49
11.2.2 Cost of modularity and security . . . 50
11.2.3 Multicore . . . 52
11.3 Discussion . . . 53

12 Conclusions **55**

Bibliography **56**

A OS Abstraction **58**

A.1 Microkit Mapping 58

B Overview of Changes **59**
B.1 Changes Since Release 0.4 of 2024-03-27 59
B.1.1 OS abstraction 59
B.1.2 More device classes 59
B.2 Changes Since Release 0.2 of 2022-10-03 59
B.2.1 More device classes 59
B.2.2 Mandatory virtualisers, cache maintenance 59
B.2.3 Completely separated Tx and Rx paths 60
B.2.4 Clarified terminology 60
B.2.5 Linux-based component development and legacy re-use 60
B.2.6 Performance evaluation 60
B.2.7 Changes resulting from evaluation 60

List of Figures

3.1 Device model. 3

4.1 High-level structure of I/O systems on seL4, using network terminology. 8
4.2 Memory regions shared between software components and the device. 10
4.3 Tx virtualiser-driver transport layer, showing the *metadata* and *data regions*. . . . 12
4.4 Active (left) vs passive (right) driver-thread model. 18
4.5 Mode switches for the active driver-thread model compared to Linux. 19

5.1 Modularised design of a networking architecture. 25
5.2 Memory regions shared between software components and the device. 27
5.3 Storage device transport layer. 29
5.4 TX prebuffering 35

6.1 The high-level structure of how insertion events are relayed to clients, and how clients can request ejection, focussed on storage devices. 37
6.2 A structure showing how a controller can request device ejection. 39

8.1 Linux driver re-use through UIO. 43

11.1 Evaluation setup for sDDF (top) and Linux (bottom). 48
11.2 sDDF networking performance compared to other systems running single-core. . 49
11.3 Performance of differently modularised sDDF systems compared to Linux. 50
11.4 Comparing per-packet processing costs of the four sDDF systems. 51
11.5 Single-core sDDF performance with single- and multicore configurations. 51
11.6 Single- and two-core sDDF performance compared to 2-core Linux. 52

List of Listings

4.1 Control region queue data structures. 13
4.2 Control-region queue management. 14
4.3 Typical network device queue data structures. 15
4.4 Device queue management (simplified). 15
4.5 Ethernet driver pseudocode. 20
5.1 I2C token definitions. 28
5.2 I2C request and response queue entry data structure. 28
5.3 Storage request and response queue data structures. 31
5.4 Storage information region data structure. 32
5.5 Sound definitions 34
6.1 Virtualiser policy interface prototypes. 38
8.1 Example UIO stub driver command-line configurations. 42
8.2 Example UIO node in device tree. 42
8.3 Example of Linux usermode driver helper. 44

Acronyms

API application programming interface 19

CAmkES component architecture for microkernel-based embedded systems, a user-level framework for developing on top of seL4 49, 50, 53, 55

CD-ROM compact-disk read-only memory 30

CPU central processing unit 1, 3, 5, 14, 17, 21, 49, 50, 52, 53

CSpace capability address space 9

DMA direct memory access 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 17, 30, 33, 42, 45

I/O input/output 4, 5, 9, 11, 13, 16, 17, 19, 21, 26, 29, 30, 33, 44, 59, 60

I2C Inter-Integrated Circuit (serial communication bus) 27, 47

IOMMU input/output memory management unit 5, 16, 17, 45

IRQ interrupt request 6, 18, 19, 21, 22, 23, 24, 43

MMIO memory-mapped I/O 27

MMU memory management unit 4, 5

NIC network interface card 9, 14, 21, 22, 26, 46, 48, 50

OS operating system 1, 8, 9, 16, 17, 18, 58

PCM pulse-code modulation 32, 33, 34

PCM physical layer 48

PPC seL4 protected procedure call 27, 38

Rq request 16, 17, 27, 28, 29

Rs response 27, 28, 29

Rx receive iv, 9, 10, 11, 12, 14, 16, 17, 25, 26, 28, 52, 60

SC scheduling context 18, 21, 22, 23

sDDF seL4 Device Driver Framework v, 1, 8, 9, 19, 26, 36, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 58, 59

TCB trusted computing base 2

Tx transmit iv, 9, 10, 11, 12, 13, 14, 15, 16, 17, 19, 25, 26, 28, 52, 60

UIO Linux userspace I/O v, vi, 42, 43, 44, 60

VirtIO Linux I/O virtualisation framework 33

VM virtual machine 32, 42, 43, 44, 55

VM virtual-machine monitor 43, 44

VSpace virtual address space 9

Chapter 1

Aims of the sDDF

Every operating system (OS) needs device drivers, and OSes based on the seL4 microkernel [Klein et al., 2009; The seL4 Foundation, 2021] are no exception.

The seL4 Device Driver Framework (sDDF) provides libraries, interface specifications and tools for writing/porting device drivers to run as native, isolated components on seL4. Specifically, the sDDF design aims to:

- support a wide class of devices, including network, USB, graphics, storage, serial ports etc;
- achieve performance comparable to Linux in-kernel drivers, as measured by throughput, latency and CPU load, at least for latency-sensitive high-throughput devices (network interface cards and USB);
- be robust against defined threats;
- support sharing of devices between multiple clients, which might be seL4-native components or virtual machines;
- provide strong *separation of concerns*;
- eventually enable the formal verification of its components.

The sDDF assumes a general device model that should enable formal reasoning. It defines a device driver model that is appropriate for seL4 and aims to simplify driver implementation, eliminate common causes of driver bugs and aid formal verification. It is based on an asynchronous, zero-copy transport layer that minimises overheads while keeping driver interfaces simple.

For the time being we focus on defining a driver model and its control and data interfaces to the driver's clients, leaving other required functionality, such as device discovery and device initialisation for future work.

Chapter 2

Threat Model

The ultimate aim is a system whose whole *trusted computing base* (TCB) is verified to be *trustworthy*. This requires the TCB to be minimised. An implication is that some drivers will be part of the TCB and thus trusted, while others (the majority) of drivers are not. It also means that the driver framework itself is trusted.

We assume that an attacker is able to compromise and fully control any component that is not part of the TCB. Specifically this means that **the attacker may:**

- execute arbitrary instructions in the address space of any untrusted driver;
- execute arbitrary instructions in the address space of any untrusted client;
- arbitrarily change the data on untrusted external media (network or physically insecure storage).

However, **the attacker cannot:**

- compromise a trusted component (driver, virtualiser);
- compromise a trusted client;
- compromise the driver framework itself.

We assume that trusted clients or servers dealing with untrusted entities use encryption protocols, such as TLS, to ensure confidentiality and integrity of data, even if using untrusted components. If a client only employs trusted components, the system must guarantee confidentiality, integrity and availability.

The threat model implies that it must be feasible to formally verify all trusted components (eventually).

Chapter 3

Device Model

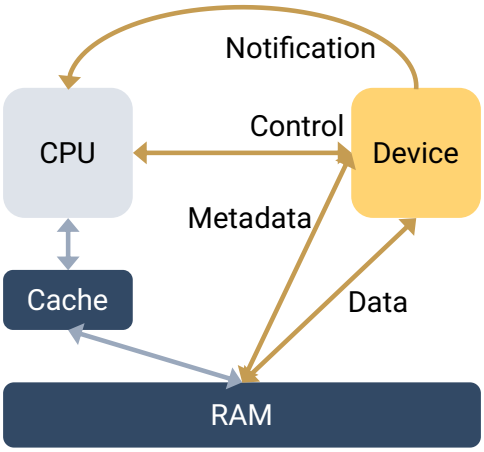

Figure 3.1: Device model.

We view a device as a state machine (implemented in hardware) with four interfaces, indicated in Figure 3.1:

the control interface specifies how the software running on the CPU, i.e. the *device driver*, issues commands to the device, and how the device reports status information back to the driver;

the notification interface lets the device alert the driver to a state change;

the data interface transports data between software and the device;

the metadata interface specifies data locations and is referenced by the control interface.

The notification interface is one-way (device to driver) while the control interface is two-way. The data interface can be one-way: for a pure input device (microphones, cameras, read-only storage media), data only travels from the device to the driver, while for a pure output device (audio output, display), data only travels from the driver to the device; other devices (network, storage) have two-way data interfaces. The metadata interface is two-way, as for both input and output channels it is updated by both sides.

The model is presently somewhat simplified, in that it does not take into account hot-plugging/unplugging (important e.g. for USB devices), nor the possibility of separate DMA controllers.

3.1 Device interfaces

The **control interface** is memory-mapped: The driver accesses the interface by reading from or writing to specific physical addresses, *device registers*, which are mapped (by the memory management unit, MMU) into the driver's virtual address space.[1]

These device registers do not *behave like regular memory!* Specifically:

- a write to a device register followed by a read from the same register might *not* return the value written (and may result in an error condition);
- the sequence of reads or writes matters: the outcome may change if register accesses are reordered;
- the granularity of device register accesses matters: a byte-sized write to a register address followed by three byte-sized writes to subsequent addresses might not have the same effect as a 32-bit write to the same address.
- multiple reads or writes to the *same* address are all significant, and can have a user-visible effect, even if the values read/written are identical.

Among others, these properties require that device registers are mapped uncached, and that software declares them `volatile`. The correct granularity and sequence of device register accesses are defined by the device interface protocol.

The **notification interface** is an interrupt (which the seL4 kernel converts into signalling a *notification* object). It instructs the driver to use the control interface to determine what is being notified. Interrupts can signal a number of conditions:

- **transmit completed:** the device has completed an output (data write) operation and will no longer access the relevant output data;
- **data available:** the device has completed an input (data read) operation, will no longer access the respective buffer and software can now safely access the data;
- **other state changes:** the device state has changed in some other way, which includes such events as disk spin up completed, network cable inserted, firmware download completed;
- **error:** some failure occurred (which includes a device losing power or being disconnected).

The **data** and **metadata interfaces** use actual memory (called *direct memory access*, DMA). This means that the device accesses the memory concurrently with software, similar to a separate processor core. This has a number of implications:

- DMA generally bypasses the cache. While some processors (specifically the x86 architecture) hides this fact by ensuring the cache remains coherent with DMA memory, others (specifically Arm) do not, requiring the use of memory barriers for consistency, and the use of cache-management operations (cache flush or invalidate).
- Software must not read DMA memory while an input is in progress, and must not write DMA memory while an output is in progress. In fact, because reads and writes affect

[1] On the x86 architecture some legacy devices, specifically serial ports, are not memory mapped and require I/O port instructions to control. These devices process small amounts of data (have no data interface) and are not performance-critical. They will be handled in an ad-hoc manner.

the cache, it is advisable not to perform any reads or writes to a DMA buffer during I/O to that buffer.

- Speculative reads (aka. pre-fetching) on DMA memory may happen while the device is writing input data, requiring extra cache management on input buffers.
- Addresses used for DMA are either not mapped (i.e., they are physical addresses), or are mapped via a different MMU, the IOMMU, from the addresses used by the CPU.
- For the *metadata region*, explicit cache management is usually not worthwhile, meaning that the CPU should map this memory *uncached* (unless the hardware maintains cache coherency, as on x86).

Simple devices (serial port, timers) do not usually use DMA. We can treat them as the special case of an empty data and metadata interface.

3.2 Device interface protocol

Software access to device registers (writes and potentially reads) trigger state transitions in the device; a device state transition may result in a software-visible change in a device register.

The device interface protocol specifies the addresses, sizes and semantics (i.e. state transitions triggered by access) of device registers. It also specifies some ordering conditions on accesses (reads or writes). It may specify timing conditions on accesses.

Examples of timing conditions are:

- access B must happen no earlier than x microseconds after access A;
- after access A, the driver must poll (read) register b until it is non-zero, before performing access C.

For the case that the protocol specifies minimal delays between accesses, we can assume that there are only a small number of such delay values, which are calibrated at device setup time and are abstracted by delay functions defined outside the driver proper.

3.3 Device operation

A sequence of state transitions may move the device into a state where it performs **output**, by reading DMA memory. Specifically it will follow references, supplied by the control interface, to descriptors in DMA memory (the metadata interface), and from there to the actual data (data interface). Addresses for DMA have to be translated to the device address space before being given to the device. The device may use physical addresses or its own virtual address space (mapped by the IOMMU).

The completion of the output operation results in a state transition that triggers the notification interface (i.e. an interrupt); the driver then needs to use the control interface to determine the reason for the interrupt (i.e. output completed).

A sequence of state transitions may move the device to a state where it can provide **input** to DMA buffers provided by the driver via the metadata interface, which in turn was referenced by the control interface. To software, the commencement of input is non-deterministic (it is

determined by the environment). The completion of input results in a state transition that triggers the notification interface, the driver then needs to use the control interface to determine the reason for the interrupt (i.e. data available).

Once an interrupt has triggered, the hardware disables the interrupt (and lower-priority ones), preventing it from re-occurring if it is a level-triggered interrupt. The kernel immediately masks that specific interrupt (preventing it from triggering again) and then acknowledges it (which re-enables all the lower priority ones). When the driver *acknowledges* the interrupt to the kernel, the kernel unmasks the interrupt, after which the driver can wait for the next interrupt.

A single interrupt may signal multiple completions (this is called *interrupt coalescing*). The driver needs to process all pending completions prior to acknowledging the interrupt. Depending on the device, the driver may also have to clear interrupt conditions in the device registers to prevent the interrupt re-triggering.

Completions can continue to occur while interrupts are masked, which implies that upon acknowledging, the interrupt may immediately trigger again, resulting in the driver immediately receiving a new notification as soon as it finished processing the present one. In order to minimise the number of kernel entries, the driver should, before acknowledging the IRQ, use the device's control interface to check for pending interrupts. After processing all pending events, the driver clears the device's event register. (Note: this is a performance optimisation, not a correctness issue.)

If interrupts are masked for too long, or software is too slow to process input data, the device will run out of free DMA buffers for depositing input data; in this case data will be lost (the device drops packets). This creates an automatic flow control of inputs, and can be used to prevent overloading the system (called *rate limiting* the device).

Network packets can be lost for external reasons too, e.g. in a router between source and destination, or by interference on a wireless link. Higher-level network protocols (e.g. TCP) handle packet loss using acknowledgement and re-transmission.

We ignore error conditions for now, but note that dropping of packets is not considered an error condition.

Devices that can be plugged into or removed from a running system, such as USB devices or SD cards, add further complications which we ignore for now. Support for "hotplugging" will be covered in Chapter 6.

3.4 Comments

The device interface protocol is defined by the hardware, and is thus not under our control. To enable formal reasoning about driver correctness, the device protocol will have to be formalised.

Unfortunately, these protocols are usually specified in manufacturers' data sheets that are highly informal, typically vague, and frequently wrong. Errors in data sheets arise from device implementation bugs as well a manufacturer-internal miscommunication between hardware designers and documenters. Even worse, there are many cases where not even a data sheet is available, and the interface is reverse-engineered from a Linux driver implementation.

Buggy device interface specs will inherently lead to buggy drivers. Unfortunately, this is not something we can address for now, we are at the mercy of what is available. However, it is a

problem *every* driver developer faces, whether or not they employ formal reasoning.

But we do have the opportunity to at least eliminate all other driver bugs, which are the majority [Ryzhyk et al., 2009]. And should we succeed in verifying realistic drivers, we can offer a good value proposition to hardware manufacturers, and may be able to tackle the specification problem in partnership with device IP producers.

Chapter 4

Drivers and Their Software Interface

Our driver model strongly reflects the aim of separation of concerns: *the sole purpose of a device driver is hardware abstraction*. The driver translates a hardware-defined and -specific device protocol into a hardware-independent (but OS-specific) *device-class protocol*. In other words, there is a single software-side driver interface specification for each device class (network, USB, storage, etc). This interface is designed to support a lightweight, low-overhead translation, at least for performance-sensitive (high-throughput) devices.

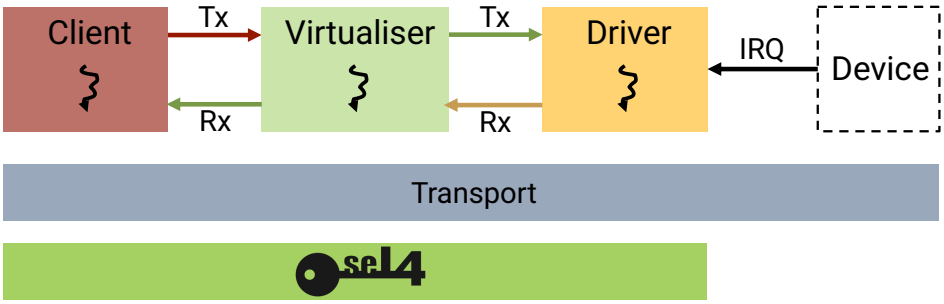

Figure 4.1: High-level structure of I/O systems on seL4, using network terminology. Coloured boxes are software components, grey indicates a shared memory region.

4.1 Overview

In our design we utilise the flexibility afforded by a clean-slate design, where we can control the structure of the drivers as well as their interface to the rest of the system's software. Specifically we use this flexibility to define a structure that eliminates unnecessary complexity in driver implementations and thus makes it easier to write performant yet correct drivers (with the intent to verify them eventually).

Our design is based on our prior work [Leslie et al., 2005], which established that, using a zero-copy transport layer with shared queues and asynchronous communication, usermode network drivers can deliver performance competitive with in-kernel drivers. It also incorporates what we have learned from the later Dingo work [Ryzhyk et al., 2009], which proposed a driver model for Linux that would eliminate many common driver faults.

Figure 4.1 shows the high-level logical structure of device interfaces in the sDDF. Each coloured box represents a user-level process that is encapsulated by seL4 and only able to

communicate via defined interfaces. The high-level view has three components:

Device driver: Interfaces with the device hardware (NIC, flash, ...), receives interrupts (as seL4 *notifications*), and transmits raw data between the device and the client;

Client: Is the producer/consumer of the I/O data handled by the device. In many cases the "Client" is actually an OS server implementing a higher-level abstraction for applications, but this is not relevant to the sDDF.

Virtualiser: Allows a single device to be shared between one or more Clients. Where there are multiple clients, the Virtualiser presents the Client with an illusion of exclusively owning the device. Each control path has its own virtualiser (see Section 4.2.1).

The components are connected by simple control and notification interfaces. The control interfaces refer to a metadata and data interface that is provided by the *transport layer*. The Driver- and Client-side interfaces of the Virtualiser look almost the same, except the Driver-side uses I/O-space addresses, whereas the Client-side uses offsets into the Data region.

The Client-Driver interface traditionally uses somewhat different terminology for different device classes (e.g. network vs storage). For now we use the terminology used for network devices, referring to the output operation as *transmit* (Tx) and the input operation as *receive* (Rx). We will discuss interfaces for different device classes in Chapter 5.

Each of the above processes (i) consists of (at least) one seL4 *virtual address space* (VSpace) with access rights defined by its *capability space* (CSpace), a thread, and scheduling parameters. The latter consist of a priority, P_i, and a (possibly null) scheduling context, S_i. A scheduling context has two relevant parameters, a *budget*, C_i and a *period*, T_i, where $C_i \leq T_i$.

For the rest of this document, unless explicitly stated otherwise, we will use the term Client to refer to the whichever software is interfacing to the driver via the virtualiser, keeping in mind that in a particular configuration this could be a component implementing an OS abstraction, a copier, or a directly-connected application.

In fact, this reflects a flexibility enabled by our design based on strict separation of concerns: components such as copiers can be transparently added or removed depending on the requirements of the specific system. Similarly, issues such as mapping of virtual (Client) addresses to device I/O addresses, or dynamic re-mapping of DMA regions, are of no concern to the Driver.

4.2 Transport layer

The transport layer consists of a number of shared memory regions, data structures in those regions, and access protocols. It is designed to minimise software overheads by minimising (ideally eliminating) copying of data.

4.2.1 Memory regions

There are three types of memory regions, as shown in Figure 4.2:

1. a ***device control region*** (Dev Ctrl) shared between driver and device

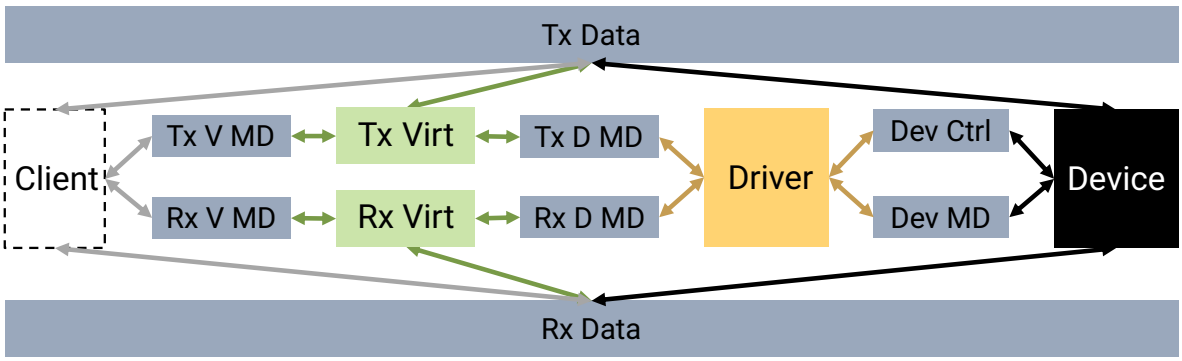

Figure 4.2: Memory regions (grey) shared between software components and the device. “MD” represents metadata regions of the device (Dev), driver (D) and virtualisers (Virt).

2. multiple ***metadata regions*** (MD) shared between driver and device, and similar regions shared between components
3. a ***data region*** shared between client, virtualisers and device.

Note that while the driver holds *references* into the data region, it only passes these between the virtualiser and the device; the driver never needs to access the actual data transferred. The data region is therefore not mapped into the driver’s address space, reducing the trust required in the driver. The driver only needs access to the metadata region.

Also, for devices (e.g. networking) having separate Tx and Rx control paths, the data region will generally be split into separate Tx and Rx regions, as indicated in Figure 4.2.

Driver-device interface

The *device control region* is a set of memory-mapped device registers used to change and inquire the device state; they constitute the primary mechanism for software to interact with hardware. The set of device registers, and thus the size of this region, is hardware-defined, and typically fits on a single page. Multiple devices may share a single page of device registers, in which case the drivers must take care not to interfere with each other.

On architectures that do not ensure cache consistency with DMA, we map the device control region *uncached* in the driver’s address space. This avoids any need for cache management by the driver (without significant performance penalty).

For simple devices, such as serial ports, the device control region constitutes the complete hardware-software interface. Interaction happens by the driver reading or writing individual registers (which can be byte- or word-size).

For devices processing bulk data, there is a *metadata region* (Dev MD), which is of flexible size but usually small (few pages), located in RAM. It contains data structures referencing the data buffers (which transfer the actual input or output data). Typically these are circular queues (“ring buffers”). While the format of those data structures is hardware-defined (part of the device protocol), their size is usually software-defined. The driver uses the device control interface to inform the device of the size and location of the metadata structures.

References to the metadata region (whether passed through the device-control interface or internal pointers in the metadata region) are *I/O space addresses*.

The device accesses the metadata region by DMA. The region is therefore also *mapped uncached* in the driver's address space, and no cache management is required by the driver.

Virtualiser-driver interface

The driver and virtualiser interface via a separate, shared metadata region. To distinguish it from other MD regions, we refer to this as the *driver metadata* (D MD) region. For device classes featuring multiple (e.g. Tx and Rx) virtualisers, there is a separate metadata region for each virtualiser (Tx D MD and Rx D MD).

As a general rule, the driver metadata regions are regular memory, i.e. *mapped cached* in the driver (as well as the virtualisers). References into the data region from the metadata region use *I/O space addresses*.

For drivers that do not do DMA (e.g. our current serial drivers), the metadata region contains the data to be transferred; they do not have a separate data region.

Client-virtualiser interface

This interface, the *virtualiser metadata region*, again consists of one region per virtualiser. Obviously, each client has its own virtualiser metadata region(s). The region is regular memory (mapped cached) and looks identical to the driver metadata region, *except* that references into the data region are *offsets relative to the start of the data region*. Generally, each client will have its own data region(s). It is part of the virtualiser's job to translate the offsets into the per-client data regions into *I/O space addresses* for the driver, adding the offset to the corresponding base *I/O space address* of each client's data region. At least for now we require client data regions to be contiguous area in the physical address space.

4.2.2 Data region

The layout of the data region(s) depend on the device class. In general it contains a set of I/O buffers that are referenced by the various metadata regions. In many cases (e.g. networking) the data region is simply an array of equal-sized I/O buffers.

For some device classes (e.g. networking) the data region can be split into separate regions for input and output.

As mentioned above, references to the data region on the client side of the virtualiser are represented as offsets relative to the start of the region, while the driver side uses I/O-space addresses. Translating between those references is one of the duties of the virtualisers.

In general, each I/O buffer is in one of the following states:

1. **source-owned and in use:** the buffer is not referenced by a *metadata region* (but referenced by source-internal data structures), it is waiting for, or is in the process of, being filled with data;
2. **destination-owned and idle:** the buffer is referenced by a *metadata region*, ready to be collected by the destination, it *contains valid data*;
3. **destination-owned and in use:** the buffer is not referenced by a *metadata region* (but referenced by destination-internal data structures), it may be waiting for or in the process of being used by the destination;

4. **source-owned and idle:** the buffer is referenced by a *metadata region*, ready to be collected by the source, it *contains no useful data*.

Here “source” stands for the component that produces data: For output that would be the component to the left (client side) of the region as per Figure 4.1, while “destination” would be the component to the right (device side) of the region. For input, source refers to the device-side and destination to the client-side component.

Ownership is the exclusive right to access a buffer, as well as the descriptor that references it in its respective metadata region. In other words, if a buffer is owned by a component, no other component may access the buffer nor the queue entry that references the buffer.

For device classes that have separate input and output data regions, a particular DMA buffer is only ever used for either output or input: An output buffer is filled by the client, consumed by the device, and then handed back as free to the client. Similarly, an input buffer is filled by the device, consumed by the client, and eventually returned as free to the device.

While the client may change a buffer’s direction (e.g. receiving a packet from the network, changing some of its content and sending it back to the network), this happens outside the framework (and may not be advisable as it will significantly complicate buffer management).

4.2.3 Metadata regions

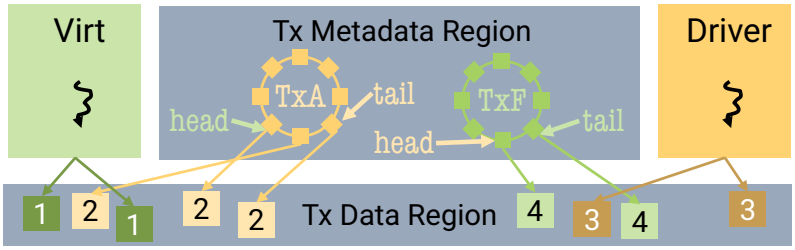

Figure 4.3: Tx virtualiser-driver transport layer, showing the *metadata* and *data regions*. The numbers in the buffers in the data region indicate the buffer state as defined in Section 4.2.2.

Figure 4.3 shows the structure of the transport layer interfacing the virtualiser and the driver, again using a Tx path as an example.

The metadata region uses *lockless, bounded, single-producer, single-consumer queues*. These are implemented as what is commonly referred to as “ring buffers”. To avoid confusion, we will not use this terminology, and only refer to them as “queues”. We reserve the term “buffer” for the DMA-able data buffers.

The Tx metadata structure consists of two fixed-size queues, *transmit active* (`TxA`) of output data to be handed to the device, and *transmit free* (`TxF`) of buffers ready to be re-used. The `TxA` queue references all buffers that are destination-owned and idle, the `TxF` queue references all buffers that are source-owned and idle.

The Rx path has an equivalent structure, with two queues: *receive active* (`RxA`) of valid input data to be consumed, and *receive free* (`RxF`) of free buffers ready to be re-used).

For a device using a request interface, there are *request active* (`RqA`) and *request free* (`RqF`) queues.

While Figure 4.3 shows a driver metadata region, the virtualiser metadata region has the same structure, the only difference being that data region references use offsets from the start of the data area, instead of I/O addresses.

```
1  #define QUEUE_SIZE (1<<QUEUE_LOG_SIZE)
2  struct buffer_descr {
3    uint64_t io_or_offset;
4    uint16_t length;
5  }
6  struct queue {
7    uint16_t head;
8    uint16_t tail;
9    uint32_t consumer_signalled;
10   struct buffer_descr buffers[QUEUE_SIZE];
11 }
```

Listing 4.1: Control region queue data structures.

Listing 4.1 shows the queue data structures. Each queue has a separate `head` and `tail` index, queue entries between those indices are valid in that they contain an offset to a buffer in the respective data area.

Each queue has exactly one *producer* and one *consumer*. Only the producer updates the tail, and only the consumer updates the head.

Tracing the path of a Tx buffer from virtualiser to driver, the virtualiser (producer) hands *ownership* of a DMA buffer to the driver (consumer) by inserting the buffer into the *active* queue (changing its state from ① to ②), from where the driver collects it (changing state from ② to ③). When done with a buffer, the driver (who for this queue is the producer) hands it back to the virtualiser (now the consumer) by inserting it into the *free* queue (③ to ④), from where the virtualiser can collect it (④ to ①).

Specifically, as shown in Listing 4.2, the producer enqueues a data buffer at the tail by checking there is at least one unused entry, inserting the new buffer there, and incrementing the tail pointer. Just before updating the `tail` pointer, the producer issues a *memory write barrier* to ensure that no writes are re-ordered by the compiler or the processor across this point. Dequeuing data buffers from the head is analogous, as per Listing 4.2.

Inserting a reference to a buffer into one of the queues changes ownership (see the definition of buffer states in Section 4.2.2). The inserting component loses the right to access the buffer, *as well as the descriptor referencing it*. The precise point of ownership transfer is Line 11 in Listing 4.2: Incrementing the queue's tail pointer by the producer makes the entry visible to the queue's consumer, which has then acquired ownership.

We can observe that ownership of data buffers is *defined* by where they are referenced: ownership of a queue entry implies ownership of the buffer it references. An interesting consequence is that for an active Tx buffer, ownership passes from the client to the virtualiser, from there to the driver and from there to the device. So, while the driver cannot access the buffer (it is not mapped in the driver's address space) it still *owns* it, for the sole purpose of being able to pass it to the device. Hence, ownership is a necessary but not sufficient condition for being able to access data.

```
bool empty (struct queue *queue) {
  return !((queue->tail - queue->head) % QUEUE_SIZE);
}
bool full (struct queue *queue) {
  return !((queue->tail + 1 - queue->head) % QUEUE_SIZE);
}
int enqueue(struct queue *queue, struct buffer_descr buffer) {
  if (full(queue)) return -1;
  queue->buffers[queue->tail] = buffer;
  memory_release();
  queue->tail++;
  return 0;
}
int dequeue(struct queue *queue, struct buffer_descr *buffer) {
  if (empty(queue)) return -1;
  *buffer = queue->buffers[queue->head];
  memory_release();
  queue->head++;
  return 0;
}
```

Listing 4.2: Control-region queue management.

A consequence is that a buffer can at any time be referenced by at most one queue – this is a core integrity condition of the scheme.

Lock-free updates to these data structures are possible by using the processor's property that *reads and writes of small integers are atomic.* The obvious data race between consumer and producer is benign. The memory barrier is sufficient to ensure consistency.

4.2.4 Device metadata region

The *device metadata region* plays a similar role in the driver-device interface as the *driver metadata region* does in the virtualiser-driver interface.

A core difference between the two regions is that while the *driver metadata region* is a standard shared-memory region (shared between two processes running on the CPU), the *device metadata region* is a DMA region (shared between the driver process on the CPU and the device hardware).

The main practical difference is that DMA bypasses the cache. On architectures that do not guarantee consistency between DMA and caches (anything but Intel), this requires explicit steps to ensure consistency. We prescribe that on such architectures *the device metadata region is mapped uncached in the driver's address space*.

As it interfaces with the device hardware, the *device metadata region*'s data structures are defined by the device interface protocol (see Section 3.2). Network devices (NICs) generally use similar queue structures as what we specified for the *driver metadata region* in Section 4.2.3 although there is usually one single queue each for transmitting and receiving; we will call those `HW_Tx` and `HW_Rx`, respectively. Listing 4.3 shows a representative example,

```
#define HW_QUEUE_SIZE (1<<HW_QUEUE_LOG_SIZE)
struct HW_buf_descr {
  uintptr_t io_address;
  size_t    length;
  status_t  status;
}
struct HW_queue {
  uint32_t head;
  uint32_t tail;
  struct HW_buf_descr buffers[HW_QUEUE_SIZE];
  struct buffer_descr buffer_desc[HW_QUEUE_SIZE];
}
```

Listing 4.3: Typical network device queue data structures.

which we will assume for the following discussion, noting that details may differ between devices.

Instead of separate *active* and *free* queues, hardware (HW) uses the `status` field of each queue entry. On output the device processes only entries marked as `ready`, and, once processed, sets the `status` to `free` or an `error` condition. On input, the device uses entries marked as `free` and, once data is delivered, changes the `status` to `ready` (or `error`). Note that the device only knows about the location of the memory area that contains the queue; this is provided to the device via a control register. The `head` and `tail` pointers are pure software constructs.

```
bool HW_full (struct HW_queue *queue) {
  return (queue->head - queue->tail + 1) % HW_QUEUE_SIZE) == 0;
}
int HW_enqueue (struct HW_queue *queue, struct HW_buf_descr HW_buf_descr) {
  if (HW_full(queue)) return -1;
  queue->buffers[queue->tail] = HW_buf_descr;
  memory_release();
  queue->tail = (queue->tail + 1) % HW_QUEUE_SIZE;
  return 0;
}
int HW_dequeue (struct HW_queue *queue, struct HW_buf_descr *HW_buf_descr) {
  if (queue->buffers[queue->head].status == READY) return -1;
  *HW_buf_descr = queue->buffers[queue->head];
  memory_release();
  queue->head = (queue->head + 1) % HW_QUEUE_SIZE;
  return 0;
}
```

Listing 4.4: Device queue management (simplified).

Listing 4.4 shows how the driver manages those queues. For simplicity, this pseudocode assumes that the hardware processes Tx buffers in queue order (which may not be the case in reality).

4.3 Component models

4.3.1 Driver

The driver is event-driven, in line with conclusions from our earlier work [Ryzhyk et al., 2009]: It acts in response to

- transmit requests from the virtualiser (*transmit active*, T_a);
- data requests from the virtualiser (*data requested*, R_q);
- receive-ready notifications from the virtualiser (*receive buffers free*, R_f);
- data-available interrupts from the device (*receive available*, R_a);
- completion interrupts from the device (*transmit completed*, T_c).

Notwithstanding some differences in terminology, this model is a refinement of earlier work, where we argued for *active* device drivers [Ryzhyk et al., 2010] (in the sense of having their own thread of execution): The driver operates single-threaded in its own address space, handling requests from either the OS (via the virtualiser) or device interface. This model has the benefits highlighted by Ryzhyk et al. [2010]: The driver is free from most concurrency issues and does not need to concern itself with client state that is not explicit in the request to be processed.

The driver thread runs at higher *priority* than the virtualiser to ensure timely handling of interrupts as well as immediate response to virtualiser requests. Flow control (described below) prevents the driver from monopolising the processor in the case of high incoming network load. In the storage case, all reads are triggered (and buffers provided) by the client (via the virtualiser) so there is no need for flow control.

4.3.2 Virtualiser

The virtualiser (Virt) is responsible for presenting a physical device as a virtual device to one or more clients. Each control path (Rx/Tx/Rq) has its own virtualiser.

Virtualisation includes translating between (Client) virtual addresses and I/O space addresses seen by the device. I/O space addresses may be physical addresses, or may be translated by an IOMMU. The translation also includes a change of representation: the client-side interface of the virtualiser uses offsets relative to the start of a memory region as memory references, while the driver side uses actual addresses.

If an IOMMU is used, I/O-space mappings may be static (defined at build time) or dynamic (e.g. to dynamically make client-provided I/O buffers available to the device). If dynamic IOMMU mappings are used, managing the IOMMU is the responsibility of the virtualisers. Static mappings generally imply the need for Copier components to keep client address spaces separated, and also imply trusting the Driver as well as the device hardware.

Virtualisers generally require access to the *data region*, so it is mapped in their address spaces. There are two reasons:

1. an Rx Virt needs access to packet headers to determine the destination (i.e. the client which should receive the data);

2. On architectures that do not guarantee coherence between caches and DMA accesses, the virtualisers need to do cache management on the data buffers, which (on most architectures) requires the buffers to be mapped in its address space.

On architectures that require cache management, this must happen *whenever a buffer is handed to the device* (via the driver). Specifically, the virtualiser must *invalidate* input (i.e. *free*) buffers before handing them to the driver, and must *flush* output (i.e. *active*) buffers before handing them to the driver. Speculative reads on input buffers may happen while the I/O is in progress, requiring the virtualiser to *invalidate* input (i.e. *active*) buffers again when receiving them from the device [Rutland, 2016, p. 33].

A virtualiser that has multiple clients presents each client with the illusion of exclusively owning the device. This requires partitioning (for storage) or multiplexing (for network-like devices). Details are device-specific and will generally be based on a policy.

For network devices, the Rx Virt generally does not require a specific run-time policy, other than the use-case independent policy of dropping incoming packets if the RxA queue is full. (But note that the size of the Rx queues in the virtualiser metadata region limits the number of Rx buffers a client can have, and thus their size enforces a policy on the amount of Rx traffic processed for the particular client.)

A network Tx Virt imposes some form of traffic-shaping policy, such as priority, round-robin or bandwidth limits. Furthermore, in the Tx path, an I/O buffer is permanently associated with a particular client: once the data is transmitted, the virtualiser will return the buffer to the client that had originally supplied it.

- *TxA not empty* notifications (i.e. data available) from the client;
- *TxF not empty* notifications (i.e. returning free buffers) from the driver.

The Tx Virt needs to transmit requests as quickly as possible, both to provide the best service to the client as well as to return free buffers. It therefore should run at a priority higher than the clients. Flow control prevents the virtualiser (as well as the driver) from monopolising the CPU.

Further optimisations of this signalling protocol should be possible and will be explored in the future.

4.3.3 Observations

As mentioned earlier, IOMMU management and cache maintenance is no concern of the driver, it is the responsibility of the virtualiser. Other than that, a virtualiser does little more than moving buffer references between queues. A Tx or Rq Virt will generally implement some traffic-shaping policy (while an Rx Virt is usually policy-free). In order to keep the implementation simple, this policy should be tailored to the specific use case. If a different policy is needed, the Virt should be replaced with one implementing the new policy, rather than making it more complex.

Combining the discussion of priorities in Section 4.3.1 and Section 4.3.2, we have a priority assignment of $P_C < P_V < P_D$. This configuration is consistent with monolithic systems, where the OS generally runs at higher (effective) priority than apps. Also mentioned earlier, what we refer to as the “client” may in fact be a pipeline of components, possibly comprising copiers

and a server providing an OS abstraction. The principle of priorities increasing along the pipeline from client to driver should also hold if the client is in fact a pipeline of components.[1]

Priorities may not mean much in a multicore system, where each pair of communicating components may be located on different cores. Hence, as in any parallel system, components *must not make assumptions on relative execution order!* Nevertheless, to maximise location transparency, the proposed priority assignment should be maintained even in a multicore scenario. This will be essential if core assignments change dynamically, e.g. if an energy-management policy consolidates components in order to off-line cores.

4.4 Synchronisation

There are two ways to implement this model: as *active* or as *passive driver threads* (Figure 4.4). Each has some advantages and it is a-priori not clear which one is better. We need a thorough evaluation to settle on the preferred implementation.

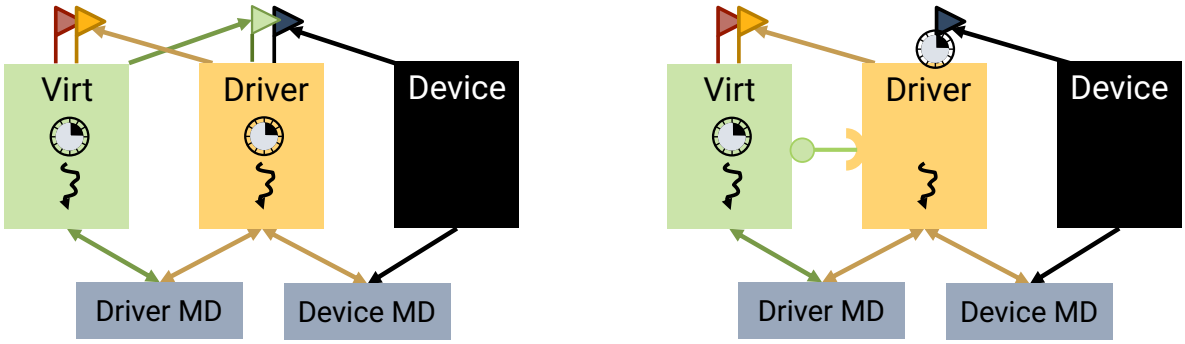

Figure 4.4: Active (left) vs passive (right) driver-thread model.

4.4.1 Active driver-thread model

In this model, the driver thread is *active* in seL4 MCS terminology, meaning it has its own *scheduling context* (SC). All its interfaces are semaphores represented by seL4 *notifications* (besides the shared memory *metadata regions*).

More specifically, the driver and virtualiser each have a *notification*. The virtualiser holds a badged *send* capability for the driver's *notification*, the badge identifies the virtualiser to the driver. The IRQ is represented by a different badge representing the device.

The virtualiser performs its T_a, R_q and R_f operations by enqueuing descriptors in the *driver metadata region* and signalling the driver's *notification*. The driver uses the badge to distinguish signals from the virtualiser and the device.

Similarly, the virtualiser has a *notification* for which the driver has a badged capability (and the virtualiser's client holds a differently badged capability to the same *notification*). The driver uses this *notification*, together with a status indicator in the HW queue, to perform the R_a and T_c operations.

Remember that the driver runs at higher priority than the virtualiser. Therefore, if both components run on the same core, the signals from the virtualiser to the driver are effectively

[1] This priority assignment is the opposite of the original device driver framework and addresses some of the latter's performance problems.

synchronous, resulting in an immediate context switch to the driver (unless the driver is out of budget).

Listing 4.5 shows *simplified* pseudocode for an Ethernet driver. A more optimised version will, after processing notifications from the virtualiser, also check for pending IRQs from the device (which may have arrived while the driver was running on behalf of the virtualisers) and process these. Similarly, in a multicore system, virtualiser notifications may have arrived while processing IRQs and should be checked prior to returning. The driver will also, upon processing the Tx-completion IRQ, check whether there is work in the TxA queue.

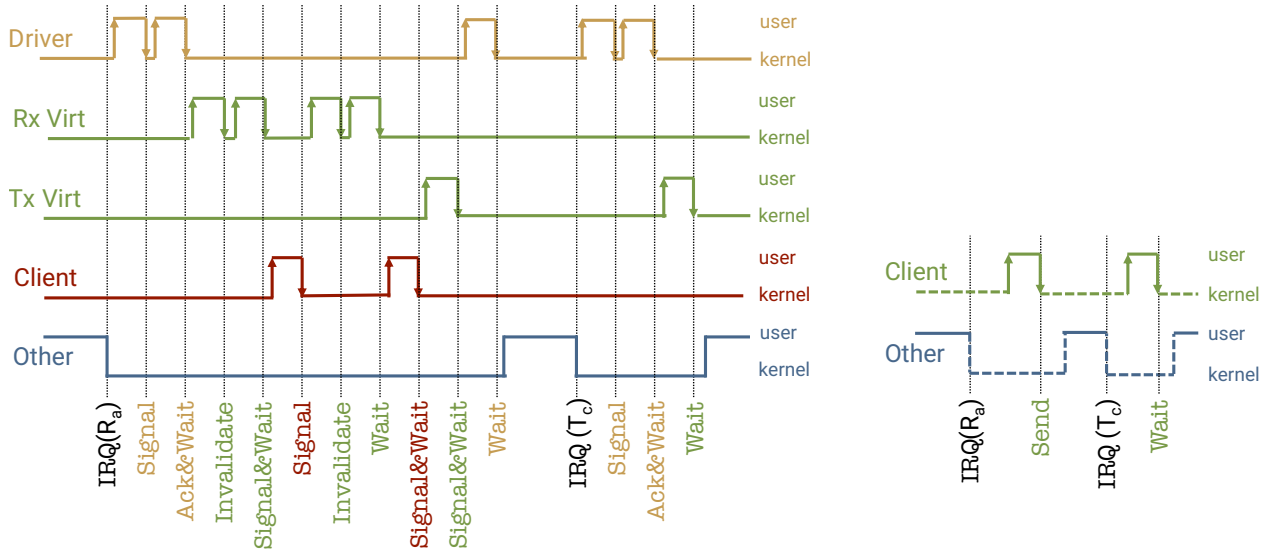

Figure 4.5: Mode switches for the active driver-thread model (left) compared to Linux (right) for a packet round trip in a single-core setup. For horizontal lines, solid indicates execution, dashed indicates suspension (blocked or preempted). For vertical lines, arrows indicate synchronous mode switches (system calls and returns) while dashed lines indicate asynchronous switches (interrupts or scheduling). “Other” refers to a lower-priority background activity.

Figure 4.5 shows the mode switches for this model compared to Linux. The diagram shows a total of 15 kernel entries for a complete packet round trip. However, if the system is under high load, it is less likely that all signals will be required by the driver, as it is more likely that the virtualiser is already awake and processing these queues, which is indicated to the Driver by a flag in the respective queues.

In contrast, the Linux system only has 4 kernel entries. The difference represents the inherent overhead of the microkernel-based design: On Linux, each client operation is a single system call (total 2), while in seL4 this corresponds to context switches to the virtualisers and the driver and back, significantly increasing the number of system calls. Furthermore, the usermode seL4 driver requires system calls to acknowledge IRQs. In addition, the cache management done by the virtualiser also requires system calls, as these operations are privileged on the Arm architecture (on RISC-V they can be delegated to usermode code, reducing the required system calls). Linux’s synchronous I/O API also avoids one kernel entry by forcing the client to block until the send is completed, while the sDDF API is asynchronous (to enable execution concurrent to I/O without forcing a multi-threaded design).

Note that interrupt batching, which occurs automatically if further interrupts arrive while the driver is processing packets with IRQs disabled, will on average reduce the number of kernel entries per round-trip by two for the seL4 system and by one for Linux.

```
handle_irq() {
  while (event = clear_hw_events()) {
    if (event & Tc) {
      enqueued = false;
      while (!full(\gls{tx}F) && buf = HW_dequeue(HW_\gls{tx})) {
        enqueue(buf, \gls{tx}F);
        enqueued = true;
      }
      if (enqueued && require_signal(\gls{tx}F)) signal |= \gls{tx}F;
    }
    if (event & Ra) {
      enqueued = false;
      while (!full(\gls{rx}A) && buf = HW_dequeue(HW_\gls{rx})) {
        enqueue(buf, \gls{rx}A); /* process input */
        enqueued = true;
      }
      if (enqueued && require_signal(\gls{rx}A)) signal |= \gls{rx}A;
      while (!full(HW_\gls{rx}) && buf=dequeue(\gls{rx}F)) {
        HW_enqueue(buf, HW_\gls{rx});  /* return free \gls{rx} buffers */
      }
    }
    if (event & error) {
          fail;
    }
  }
}
main() {
  initialise();
  signal = init_done;
  while (true) {
    event = signal_and_wait(signal);
    if (event & IRQ) {
      hand_irq();
      signal |= ack;
    }
    if (event & Ta) {
      while (!full(HW_\gls{tx}) && buf=dequeue(TxA)) {
        HW_enqueue(buf, HW_\gls{tx});
      }
    }
    if (event & Rf) {
      while (!full(HW_\gls{rx}) && buf=dequeue(RxF)) {
        HW_enqueue(buf, HW_\gls{rx});  /* return free Rxbuffers */
      }
    }
  }
}
```

Listing 4.5: Ethernet driver pseudocode.

Flow control is achieved by three means:

1. limiting the size of the receive queue in the *device metadata region* (which rate-limits interrupts by forcing the device to drop packets and thus limiting the input packet rate);
2. limiting the size of the send queue in the *driver metadata region* (which rate-limits the virtualiser's output requests) – this is unlikely to achieve much, given that the driver runs at higher priority than the virtualiser;
3. limiting the driver thread's budget (which limits the amount of work it can do in a period, irrespective whether the work is on behalf of the client or to service IRQs).

For a NIC, the *period* of the driver's SC should be the (desired) minimal inter-arrival rate of packets. The *budget* is part of flow control. Its choice depends on a number of factors, in particular the relative speeds of CPU and NIC: If the CPU is fast enough to process packets at line speed, the budget should be large enough to process one incoming and one outgoing packet. If the CPU is not fast enough, the budget may need to be reduced to allow the back-end to keep up with I/O.

If the driver's budget is depleted, the driver is suspended by the kernel until its budget is replenished at the end of the period. This suspends processing IRQs as well as virtualiser requests.

Note that on each invocation, whether on an IRQ or a client request, the driver needs to check for further events that may have happened while it was processing the current event: IRQs may arrive at any time and the driver must poll for pending interrupts before returning (resulting in batching). Similarly, more client requests may arrive during driver execution, if the driver's budget was exhausted during the processing of an event or the virtualiser runs on a different core than the driver.

More batching can be forced by using the NIC's *IRQ coalescing* feature, i.e. deferring the T_c interrupt for some amount of time or until a certain number of transmits have been completed. This is unlikely to provide significant benefit, at least for the 1 Gb/s (or less) NICs typically used in embedded systems. It might be useful for faster NICs.

4.4.2 Passive driver-thread model

In this model, the driver thread is *passive* in seL4 MCS terminology, meaning it does not have its own SC and can only run on a borrowed SC. As it runs at higher priority than the virtualiser, this makes the client→virtualiser invocation synchronous if both components share a core. This naturally leads to the virtualiser-side interface being an *endpoint*, which is invoked by the virtualiser as a *protected procedure call*, passing the virtualiser's SC along for the driver to execute. Thus, T_a, R_q and R_f are invocations of the driver's *endpoint* (with the opcode passed as an argument), while R_a and T_c are returns from the *endpoint* invocation.

As the driver does not have its own SC, its *notification* (which delivers the IRQ from the device) must be active, i.e. have an SC that is lent to the driver's thread.

The number of mode switches in this model is similar to the active driver model (possibly one higher). There is a performance advantage, as a protected procedure call to a passive PD is faster than signalling a notification to a higher-priority PD. A potential drawback is that this forces the driver and virtualiser to be co-located on the same core (and the virtualiser with its client, if it is also passive). The performance trade-offs are not obvious and need evaluation.

Flow control is achieved by three means:

1. limiting the size of the receive queue in the *metadata region*;
2. limiting the size of the send queue in the *control region* (same caveat applies as for the active driver thread);
3. limiting the budget of the driver's *notification* (which implicitly limits the interrupt rate without limiting the driver's ability to respond to virtualiser requests, but see below for more detailed discussion).

For a NIC, the period of the *Notification's* SC should be the (desired) minimal inter-arrival rate of packets. The budget primarily needs to allow for IRQ handling, as requests from the virtualiser execute on the latter's budget. But note that, even if serving a virtualiser request, the driver needs to poll for IRQs before returning, so it may handle interrupts on the virtualiser's SC.

However, the virtualiser is blocked during that time, unless the driver has a timeout exception that allows it to hand control back to the client if the budget runs out. This would, however, complicate driver logic (including re-introducing a degree of concurrency). On the other hand, blocking the virtualiser while the driver is waiting for a budget replenishment reduces the virtualiser's ability to operate concurrently.

For the R_a IRQ, the budget must be sufficient to enqueue the packet(s), notify the virtualiser, and acknowledge the IRQ. Similar, the T_c IRQ needs to have sufficient budget to update queues, notify the virtualiser and acknowledge the IRQ.

Note that IRQ coalescing leads to multiple packets processed for a single R_a interrupt, while IRQ masking leads to multiple buffers freed for a single T_c interrupt. The budget must allow for that.

4.4.3 Two-component driver

It is possible to split the driver into two separate components, one interfacing to the virtualisers (i.e. handling T_a, R_q and R_f events) and one interfacing to the device (handling IRQs). This allows serving both types of requests concurrently by running the two PDs on different cores (and *only* makes sense for this case).

The idea of a two-component driver may look like heresy in light of the discussions of Section 4.3.1, specifically the stated goal of avoiding error-prone concurrency control inside the driver. However, it is actually a straightforward extension and does *not* require extra concurrency control inside the driver. The two driver components do not access the same software-provided data structures. Specifically, only the virtualiser-interfacing component accesses the `TxA` queue of Figure 4.3, while only the device-interfacing component accesses the `TxF` and `RxA` queues.

The only potentially competing access to a virtualiser-side queue would be for returning free buffers from `RxF` to the hardware queue. In order to avoid explicit concurrency control and maintain the single-producer, single-consumer property of the queues, we need to restrict accessing this queue to the virtualiser-side component (Line 43 of Listing 4.5) – the identical code in the IRQ handler (Line 19 of Listing 4.5) is a performance optimisation, not a functional requirement.

The performance benefit of this optimisation in the multi-component case is not obvious without detailed evaluation. An evaluation performed by Parker [2023, Sect. 7.4.3] showed no performance benefit (in fact, a slight performance degradation) and we therefore do not investigate this model further for now.

4.4.4 Time-triggered architectures

The model readily adapts to *synchronous* (aka. *time triggered*) architectures [Kopetz, 2003], where each component executes at a pre-determined time point, irrespective of interrupts. In this case, none of the components signal notifications, synchronisation is exclusively via timer signals. If a component receives a signal, it processes its queues and then waits for the next signal.

4.4.5 Discussion

The design currently has two major open questions that need more analysis.

Research Question 1.: *Active or passive driver threads?* Considerations:

C1.1: The active *notification* for receiving IRQs arguably allows the budget to be adjusted to the driver's needs, but the need for checking for new work before returning reduces the benefit.

C1.2: An *endpoint* results in the virtualiser being charged for the time the driver consumes on its behalf. However, this benefit is limited, as the IRQ processing is still not accounted to a specific client.

C1.3: Invoking a passive thread through an *endpoint* avoids switching the scheduling context, while signalling a higher-priority thread forces a switch of SC, incurring higher cost. However, the active-thread design comes a the cost of one extra system call.

C1.4: If the passive driver thread is blocked on budget replenishment, it blocks the virtualiser, reducing the benefits of using the budget for flow control.

C1.5: On multicore, the *notification* forces the driver to run on a specific core, while the *endpoint* forces it to run on the virtualiser's core. Which model is better likely depends on the application scenario.

C1.6: The virtualiser may want to use a watchdog to prevent indefinitely blocking on an untrusted driver. As long as the virtualiser only synchronises with *notification*s, the watchdog timeout can be signalled to the virtualiser's *notification*, terminating the wait. If the virtualiser invokes the driver by IPC, then the watchdog would have to explicitly reset the virtualiser to abort the IPC, a somewhat more complicated operation.[2]

Similar questions arise for virtualiser threads, which may also be active or passive.

Research Question 2.: *What are appropriate budgets for the driver (and virtualiser), and how are they determined?* Considerations:

[2]The simpler aborting logic could be extended to IPC via a kernel change recently discussed: allowing specified signals to abort an IPC. The full implications of this change are not yet fully understood, so this change may or may not happen.

C2.1: How important is the driver / IRQ *notification* budget for rate-limiting the driver? Is limiting buffer size sufficient, in which case the driver could simply be given a full budget, removing the issue of the suspended (passive) driver blocking the virtualiser?

C2.2: A virtualiser probably still only has small budgets, and a period matching that of the driver.

Chapter 5

Device Classes

Having described the general model of device drivers and their interfaces, we will now refine those to specific device classes.

5.1 Network

5.1.1 Main properties

Network devices are characterised by output being client-initiated while input is spontaneous. While in practice inputs may result from some client action (e.g. a request to a web or file server), this is the result of higher level protocols that are invisible to the driver (and device).

As such, network drivers do not have a request interface, only a transmit and a receive interface. A typical network structure is shown in Figure 5.1.

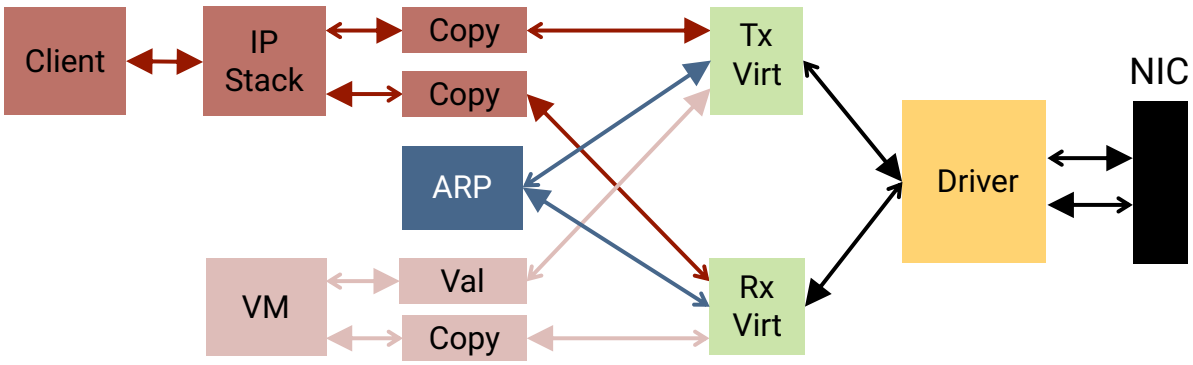

Figure 5.1: Modularised design of a networking architecture, showing copier (Copy) and Validator (Val) components for isolating untrusted clients, as well as an ARP component that handles broadcast requests.

Network devices have been our running example throughout Chapter 3 and Chapter 4, so much relevant information has already been covered. Here we summarise the main characteristics:

- The Tx and Rx paths are separated, each with its own virtualiser, data region and a metadata region between any two components, as shown in Figure 4.2.

- Each of these interfaces consists of two queues. The *active* queue (TxA/RxA) points to valid data buffers which the producer (Driver on input, Client on output) hands to the consumer. The *free* queue (TxF/RxF) points to buffers that are returned by the consumer to the producer for re-use, see Figure 4.3.
- Each I/O buffer has at any time a defined *owner*, and is in one of the four states introduced in Section 4.2.2:
    1. source-owned and in use
    2. destination-owned and idle
    3. destination-owned and in use
    4. source-owned and idle.
- The states are implicitly encoded in where the buffer is referenced, see Figure 4.3. Consequently, buffers change state and owners by being moved on and off the queues by either the producer or the consumer, see Section 4.2.3 for details.
- The driver's Rx and Tx interfaces are entirely separate, in that the driver will only move buffers between queues of the same interface, not across interfaces.

5.1.2 Ethernet

Ethernet represents an important sub-class of network devices. It represents a well standardised category, with common properties such as frame size. Specifically, Ethernet devices have a standard transfer granularity, called a *frame*, of 1.5 kB. Many Ethernet devices also support larger frames, called *jumbo frames*, but we do not consider them here.

Server platforms frequently feature *self-virtualising NICs*, with multiple interfaces to the same underlying device. These enable secure pass-through access of the same device to multiple virtual machines. Using this feature generally involves extra work by the hypervisor to handle broadcast traffic. For now we do not plan to support this feature.

5.1.3 Status

The specifications for Ethernet are tested and evaluated, they are **stable**.

5.2 Serial ports

The sDDF covers only asynchronous serial I/O at the moment, as synchronous serial is not anywhere near as common.

Asynchronous serial I/O is very similar to networking, in that output is client-initiated, and input is spontaneous. At the hardware level, communicating terminals need to agree on bit-rate, number of bits per character, and whether a parity bit is included.

Serial tends to be much lower bandwidth (most serial ports have a maximum bit rate around 4Mb/s; the POSIX standard only mentions up to 32768b/s.

As such the serial framework merges the data being transferred into the same metadata region as the queue head and tail.

Note: More **details to come**.

5.3 Serial busses

SPI

Details to come.

I2C host

The I2C device class is a multi-client serial bus that allows clients to initiate requests (i.e. acting as controllers) to a device as well as receiving a response (i.e. acting as targets).

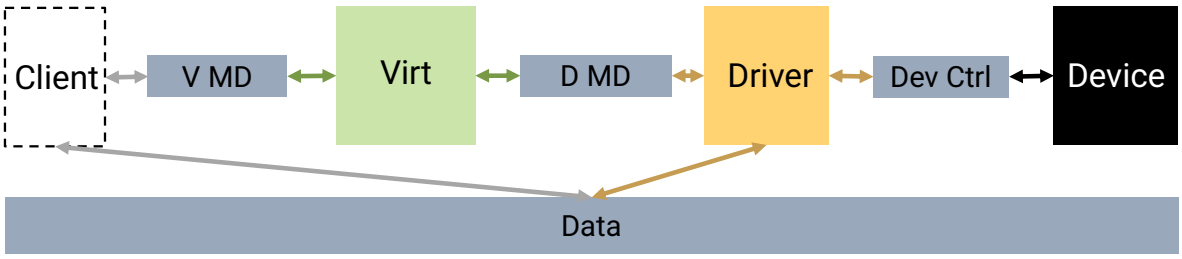

Figure 5.2: Memory regions (grey) shared between software components and the device. “MD” represents metadata regions of the driver (D) and virtualisers (V).

The I2C subsystem has each client’s data region mapped directly into the driver’s address space. Each Virtualiser corresponds to exactly one driver which corresponds to exactly one I2C bus line. The Virtualiser checks for authentic requests before it maps client-specific offsets into a driver-specific offset and vice versa. It also demultiplexes the responses from the driver to correct client. Each MD region contains only a Rq and a Rs queue.

I2C interactions happen in terms of *tokens* (1 token = 1 byte), see Listing 5.1 for their definitions. Intially a client will create a single request in the form of a request buffer containing tokens and payload data which once enqueued travels only along the Rq interfaces between Client, Virtualiser and Driver.

Note: a single I2C transaction can only be for one address.

The Driver will parse the request and potentially divide it into smaller segments, each of which will be loaded into MMIO device registers on an interface of an explicitly, initially configured I2C bus line. If the request is a READ, for each segment of the request a corresponding response will be loaded into the MMIO device registers by the device. The driver uses this data to build a response which re-uses the request buffer. Once the request has been completely loaded through the registers and converted into a response, the resulting response buffer will be enqueued by the Driver into the Rs queue. This response now travels only along the Rs interfaces between Driver, Virtualiser and Client.

Listing 5.2 shows the format of the entries in the Rq and Rs queues.

The Virtualiser provides a PPC interface for a Client to claim a free address on the I2C bus line.

Status The basic design is **preliminary and subject to change**.

```
enum i2c_token {
  /* START: Begin a transfer. Causes master device to capture bus. */
  I2C_TOKEN_START = 0x1,
  /* ADDRESS WRITE: Address target.
  The byte immediately following this token is an integer (N) length of the
  succeeding write. Max size 255. The next N bytes are the payload. */
  I2C_TOKEN_ADDR_WRITE = 0x2,
  /* ADDRESS READ: Address target.
  The byte immediately following this token is an integer (N) length of the
  desired read. Max size 255. */
  I2C_TOKEN_ADDR_READ = 0x3,
  /* CONTINUING READ: same as ADDRESS READ, but doesn't end the read operation.
  Allows chaining of multiple reads for arbitrarily long read operations.
  A multi-read chain should consist of several READCs terminated by a
  final READ. */
  I2C_TOKEN_ADDR_READC = 0x4,
  /* STOP: Used to send the STOP condition on the bus to end a transaction.
  Causes master to release the bus. */
  I2C_TOKEN_STOP = 0x5,
};
```

Listing 5.1: I2C token definitions.

```
struct i2c_queue_entry {
  size_t offset;
  unsigned int len;
  size_t bus_address;
};
```

Listing 5.2: I2C request and response queue entry data structure.

USB

Details to come.

5.4 Storage

5.4.1 Main properties

Storage devices deliver input upon specific requests rather than sporadically. As such they have *request* (Rq) and *response* (Rs) interfaces instead of the Tx and Rx interfaces that network devices use. Unlike network devices there is no separation into two control and data paths. Specifically:

- There is a *single data region*.
- There is a *single virtualiser*.
- There is a *single driver metadata region* that contains two queues: the Rq and Rs queues, as shown in Figure 5.3.

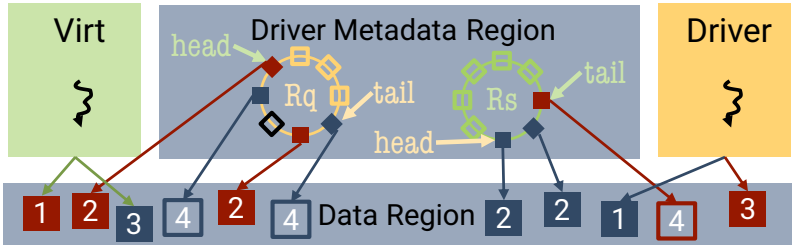

Figure 5.3: Storage device transport layer, showing the *control* and *data regions*. The numbers in the buffers in the data region indicate the buffer state, as defined in Section 4.2.2, where output buffers are red, input buffers blue. The black, empty node in the Rq queue indicates a *barrier* request.

- The virtualiser uses the Rq queue to pass request descriptors to the Driver. Each descriptor holds either a **read**, a **write** or a **barrier** request, where read or write requests reference I/O buffers.
- The Driver dequeues a request from the Rq queue and (logically) hands it to the device.
- Buffer states are as defined in Section 4.2.2.
- Unlike for network devices, the queues reference a mixture of input and output buffers in arbitrary order. Inserting into the queue changes a buffer's state and owner. Removing from the queue changes the buffer's state but not the owner.
- Specifically, the Rq queue passes buffer ownership from the virtualiser to the device. Until a buffer is returned in the Rs queue, the virtualiser (or its client) must not access it. Buffers referenced by **write** requests will not be changed by the device. Buffers referenced by **read** requests will be filled by the device. Buffer state for *read* request becomes *source-owned and idle* (no useful data in the buffer); for a *write* request it becomes *destination-owned and idle* (contains valid data).
- Likewise, the Rs queue passes buffer ownership from the device to the virtualiser (and on to the client). Inserting a response into the Rs queue indicates that the device will not access the buffer referenced in the response. The buffer state is *destination-owned and idle* for both read and write requests.
- As the device may reorder requests, the Driver may return buffers to the virtualiser (via the Rs queue) in an order different from the order they were inserted into in the Rq queue.
- The **barrier** request can be used to limit re-ordering: All requests before the barrier must be completed by the Driver/device before any of the requests after the barrier are initiated. The barrier request only ever exists in the Rq queue and references no buffer.

5.4.2 Details

Storage devices transfer data at the granularity of a *block*. In traditional rotating disks, this is the *sector*, typically 512 B (or a small multiple). These days, storage devices are large and a small amount of internal fragmentation is of no relevance. Furthermore, the setup time for a small I/O operation dominates the actual transfer time. As such, we see no benefit of supporting such small transfer sizes, and instead specify the size of transfers (and thus blocks) as a multiple of the platform's base page size (usually 4 KiB).

Where a device has an *optimal transfer size* that is larger than the page size, we use that as the block size.

Note we do not plan to support legacy devices such as CD-ROM.

Our model of a storage device is thus based on the following properties:

- The device has a *block size* that is a small integer, specifying the size in multiples of the base page size (usually the page size is 4 KiB and the block size is one).
- Large sequential (in DMA space) transfers tend to have a performance advantage, so the client (file server) should be able to issue requests larger than the block size.
- The device may support a scatterlist of DMA blocks for output and this may be faster than a sequence of individual output requests. Hence the client should be able to issue a batch of requests at once.
- The device can queue many read requests internally (typically a small power of two, $2^6 \cdots 2^8$).
- Read requests can complete out-of-order (the device may re-order requests to increase spatial locality).
- The device may cache write requests and perform them out-of-order (necessitating the use of barriers to maintain write coherence).
- Many (but not all) storage devices are slow, so caching and asynchronous operation is important.
- Solid-state storage devices wear out (they typically have limited write cycles). The controller transparently re-maps bad blocks, maintaining the illusion of a contiguous I/O space. It can be queried in order to estimate lifetime. Many storage devices use SMART [Wikipedia] for this. We plan to support a SMART interface in the future.

This leads to the set of request and response structures in Listing 5.3.

These are similar to the queues for the Ethernet devices. The main difference is the addition of a `plugged` boolean as an optimisation. When `plugged`, the driver is meant not to handle this or any subsequent requests. This allows the client (filesystem) to queue many requests, up to the size of the queue, before releasing the driver to queue them to the device. The device then has a queue of requests it can reorder for optimum device throughput (making use of a scatterlist if supported by the device). The client needs to notify the driver when it removes the plug, in the same way it notifies the driver when adding a request to an initially empty request queue.

The `id` field is to aid in matching requests and responses; it is set by the client, maintained by the virtualiser, and copied into the response by the Driver. This field is not strictly needed, as the request and response can be matched on `address`, but may simplify the client logic.

The `success_count` field in the response structure indicates the number of blocks successfully transferred. The `status` indicates the status of the first failing transfer, or `SUCCESS` if all requested blocks were transferred correctly. Thus a request that asked for 128 blocks to be transferred, where there was a seek error on the 67th, would return `success_count`=66, and `status=SEEK_ERR0R`.

In addition to the queues and shared-memory areas, there is a shared *information page* that contains the information a client might want about a storage device, and information for

```
#define QUEUE_SIZE (1<<QUEUE_LOG_SIZE)
enum request_code {
  READ_BLOCKS,
  WRITE_BLOCKS,
  BARRIER,
  FLUSH
};
struct blk_request {
   enum request_code storage_command;
   uint32_t block_number;
   uint16_t count;
   void *address;
   int32_t id;
};
struct blk_response {
   enum {
      SUCCESS,
      /* various error codes */
   } result;
   uint16_t count;
   uint16_t success_count;
   void *address;
   int32_t id;
};
struct blk_req_queue {
  uint32_t head;
  uint32_t tail;
  bool     plugged;
  struct request buffer[QUEUE_SIZE];
}
struct blk_response_queue {
  uint32_t head;
  uint32_t tail;
  struct response buffer[QUEUE_SIZE];
}
```

Listing 5.3: Storage request and response queue data structures.

monitoring the device. The exact layout and content is still to be determined, for now we use the fields in Listing 5.4.

The only non-obvious field is `ready`: It is initially `FALSE`. The driver switches it to `TRUE` when the data in `struct storage_info` is valid, and the driver is ready to accept requests. This allows disk drives to spin up, and the driver to query information about the attached storage, before file systems can access the device.

The virtualiser presents a very similar interface. We have two virtualisers designed, and one implemented at present.

One virtualiser reads the block device's partition table, and presents each partition on the

```
struct storage_info {
    char serial_number[64];
    bool read_only;
    bool ready;
    uint16_t blocksize;
    uint16_t queue_depth;
    uint16_t cylinders, heads, blocks; /* geometry to guide FS layout */
    uint64_t size; /* number of blocksize units */
};
```

Listing 5.4: Storage information region data structure.

device to a separate client.

The other lives inside a virtualiser/driver VM, and presents a single file on the driver's filesystem as a virtual block device to a client.

5.4.3 Status

The basic design is **stable**, but details of specifications are **subject to change**.

5.5 Sound

5.5.1 Main properties

Sound devices expose a series of *streams* each of which either *play* or *record* (capture) audio. This audio is transmitted through buffers of pulse-code modulation (PCM) frames, where each frame contains a snapshot of the amplitude of each audio channel at a particular point in time.

The driver exposes information about each stream to clients through shared memory. For each stream, the driver specifies supported formats, rates, the stream direction (playback or capture), and how many channels the stream supports. The driver signals to the client that all streams are ready by setting the shared `ready` flag to true. Until then, clients must assume no streams are available (usually busy waiting).

Similar to storage drivers, sound drivers communicate with clients through two pairs of request / response queues: one for commands and one for PCM transfer. A stream has the following life cycle:

1. Take – take ownership of the stream, specifying the number of channels, format and rate.
2. Prepare – allocate resources for playback.

   Client sends buffers to driver for pre-buffering.
3. Start – start playback or recording.

   Driver begins responding to buffers.
4. Stop – stop playback or recording.

5. Release – free stream resources.

5.5.2 Message Protocol

The format of commands and PCM requests is shown in Listing 5.5. The type of a command is specified by its `code` field, and `stream_id` denotes the stream the command refers to. The field `cookie` is copied over to the corresponding response so the original message can be identified. Commands are sent in the `cmd_req` queue, and responses are received in the `cmd_res` queue. The field `set_params` contains stream parameters for a *take* request, and `status` is set for responses to signal the result of a request.

To both play and record PCM data, the client sends PCM request through the `pcm_req` queue. The fields `addr` and `len` refer to a shared PCM data buffer. On playback, this buffer will contain PCM frames to play and on recording this buffer will be filled by the device. Once the driver has finished with the buffer, it will respond on the `pcm_res` queue updating the `status` and `latency_bytes`. This process differs slightly for playback and recording.

The sound data region's ownership model is identical to Storage (Section 5.4.1), where playback is equivalent to writing and recording is equivalent to reading.

5.5.3 Playback

A minimal sound system consists of a *client*, *virtualiser*, and *driver* component, where the driver communicates to the *device* via DMA. To play audio, the client must first send *take* and *prepare* commands specifying the direction as `SOUND_D_OUTPUT` and setting stream parameters. Then, $n > 0$ PCM buffers must be pre-filled to `pcm_req` to fill the device's internal buffer, followed by a *start* command. Buffers are only responded to after they have been played. This means the ith pre-filled buffer will only be responded to after i buffers have been consumed. After the stream has been started, the client must ***replay*** (send again) these n pre-filled buffers. The ith replayed buffer will not be responded to until $n + i$ buffers have been played.

An example is shown in Figure 5.4 with $n = 5$ pre-filled buffers. The first three pre-fill buffers fill up the device's internal buffer, so buffers 4 & 5 are silently ignored. After *start* is sent, the driver will initiate playback and will respond to the ith pre-filled buffer after i buffers have been played. While this is happening, the client *replays* buffers 1-5; buffers 1-3 are silently ignored and buffers 4-5 are sent to the device. There are several reasons for this pre-buffering behaviour.

- Pre-buffering reduces the latency of playback by starting the stream immediately as *start* is sent.
- The driver can choose to not support pre-buffering by simply ignoring all buffers sent before *start*, transparent to the client.
- This behaviour is required for compatibility with Linux's I/O virtualisation framework (VirtIO) sound implementation.

After all PCM buffers have been sent, the client sends a *stop* command followed by a *release* command to release stream resources.

```
typedef enum {
  SOUND_CMD_TAKE,
  SOUND_CMD_PREPARE,
  SOUND_CMD_RELEASE,
  SOUND_CMD_START,
  SOUND_CMD_STOP,
} sound_cmd_code_t;

typedef struct sound_pcm_set_params {
  uint8_t channels;
  uint8_t format; // SOUND_PCM_FMT_*
  uint8_t rate;   // SOUND_PCM_RATE_*
} sound_pcm_set_params_t;

typedef struct sound_cmd {
  sound_cmd_code_t code;
  uint32_t cookie;
  uint32_t stream_id;
  union {
      sound_pcm_set_params_t set_params; // Set on TAKE request
      sound_status_t status; // Set on all responses
  };
} sound_cmd_t;

typedef struct sound_pcm {
  uint32_t cookie;
  uint32_t stream_id;
  uintptr_t io_or_offset;
  unsigned int len;
  // Only used in responses.
  sound_status_t status; // SOUND_S_*
  uint32_t latency_bytes; // play/record latency in bytes.
} sound_pcm_t;
```

Listing 5.5: Sound definitions

5.5.4 Recording

Recording is almost identical to playback, however there is no “replay" behaviour. The client first takes and prepares the stream specifying `SOUND_D_INPUT`. The client then pre-sends empty buffers in `pcm_req` before starting the stream. On stream start, the driver will begin to respond to these with recorded PCM data. The client must continue to send empty buffers for the driver to fill. The client then sends a *stop* command and the driver fulfils the remaining requests.

5.5.5 Sound Virtualisers

The protocol is policy agnostic, however it has been designed to facilitate various virtualiser implementations. A simple virtualiser could use the *take* and *release* commands to allow

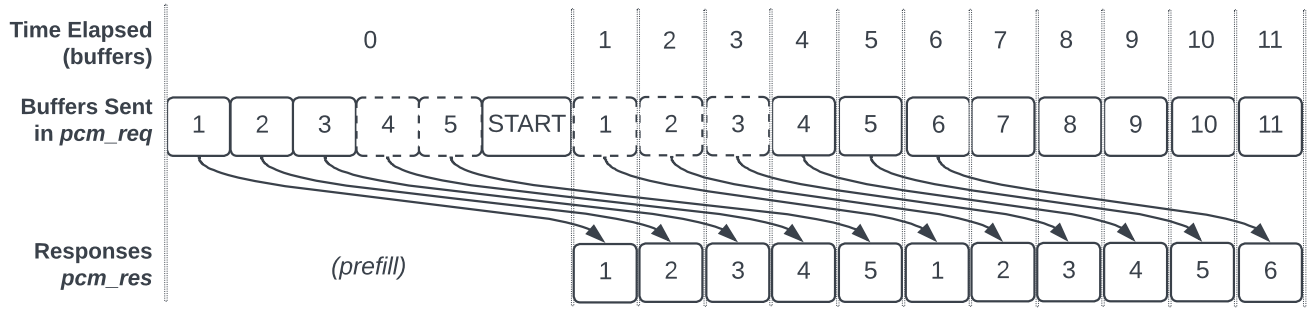

Figure 5.4: TX prebuffering

clients to have temporary exclusive ownership over a stream. If another client tries to take a stream while it is in use, the request will fail. Alternatively, a virtualiser could allow multiple clients to access a stream at the same time by *mixing* (combining) the audio signals.

5.5.6 Status

This device-class specification is **subject to change**.

5.6 Display

Details to come.

5.7 Camera

Details to come.

Chapter 6

Hotplugging

6.1 Overview

Some devices are not present for the lifetime of a system, even in statically-architected systems such as those supported by the seL4 Microkit. Specifically, USB and PCIe and other device classes have support for “hotplugging”, where the physical devices can be removed or inserted whilst the system is running. sDDF needs a framework that is device-class independent for handling insertion and ejection events, so that drivers and clients are able to function without data loss, crashes, or locking up the system.

Some common use cases for hotplugging include:

- Inserting or removing a USB keyboard/mouse to create user input.
- Updating configuration files by transferring them to/from an external storage medium.
- Replacing a failed NVMe SSD drive (which might involve a change in partition UUIDs or potential system data loss).
- Updating a web server’s hosted files without redeploying the entire system.
- Turning off WiFi to conserve energy, and re-enabling it on demand (with the consequence of having to reinitialise network stacks).

The sDDF, as presented so far, lacks sufficient mechanisms for supporting the above use cases. In the next section we will outline specific requirements for hotplugging, followed in Section 6.3 by a design that satisfies them.

6.2 Requirements

1. There needs to be a way for driver and system to communicate insertion/ejection events:
    - The driver needs a way to indicate to the system when a device has been inserted/ejected.
    - Clients need a way to request for the device to be ejected, so the driver can power down (etc). Not every client is trusted, so we need a way to restrict these requests to certain clients.

2. The block virtualiser needs to be able to redo its partition-to-client mapping, as the partition layout may have changed when switching devices.
    - Other device classes would need to do similar; e.g. USB Keyboards.
    - This behaviour is specific to a system, so should not require modifying existing code.
3. Where multiple clients access a single device through a virtualiser, there should be a way to negotiate a “safe” ejection, where clients can save any active state and pause further activity.
    - A system-specific policy must be able limit such pausing to prevent denial-of-service attacks from untrusted clients.

6.3 Design

We now show how these requirements are satisfied in sDDF. Our initial focus will be on storage devices.

6.3.1 Requirement 1: Insertion and Ejection

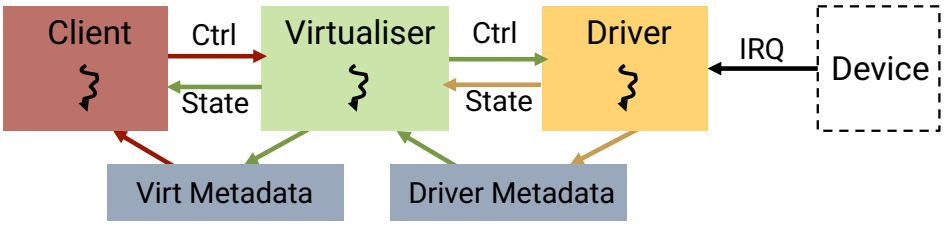

Figure 6.1: The high-level structure of how insertion events are relayed to clients, and how clients can request ejection, focussed on storage devices.

Driver Events

- The existing block queues are not suited for these, as the driver can only send responses to requests. So in the case of an insertion or ejection, there is no way to notify clients until a request is sent, which may never happen.
- The block storage info shared page, with its ready flag, is almost good, but we cannot expect the virtualiser and clients to poll continually on the ready flag.
- Hence we add a separate notification channel, referred to as the *Block State Notification*, as this allows the driver to notify the virtualiser on changes to the ready state.
    - The *ready* flag semantically corresponds to a readiness to accept block queue events.
    - This is strictly speaking, different to an inserted/ejected state, but “Inserted but not ready” is not useful.
- The virtualiser is responsible for broadcasting the driver’s ready flag to each client’s ready flag.

  - As the *Block State Notification* indicates a change, and the system is asynchronous, even if the client and/or virtualiser only sees a movement from **Ready** to **Ready** it must be treated as **Ready** → **Not Ready** → **Ready**.
  - When the driver becomes **Ready**, the virtualiser should update clients to **Ready** at the point that it has performed any required initialisations.
- We currently implement the driver such that if the device is physically injected or ejected, the driver begins any initialisation or deinitialisation steps. We do not foresee the need to prevent "auto"-insertion, downstream clients such as filesystems can implement their own policies regarding this, but they still need to know the device exists (and what it is).

Client Requests

- In contrast to driver events, we could extent the block request queues with `INSERT` and `EJECT` request codes (as this flows in the correct direction). This implementation was trialled, however, we found that this complicated both the block virtualiser and the SD card driver. The existing code generally had an initialisation phase, followed by a "handling clients" phase. Having to mix the two made everything more difficult. Furthermore, if hotplugging was added for other device classes, this solution wouldn't necessarily be viable — as a contrived example, the serial queues don't support different request codes. This also places a requirement on the block virtualiser to correctly implement a security policy.
- We instead perform PPC using a small control interface (see: `include/sddf/hotplug/control.h`). Clients can request insertion (`REQ_INSERT`) or ejection (`REQ_EJECT`) and the driver can respond indicating success (`RESP_OK` or `RESP_NOOP`) or failure (`RESP_ERROR` an insert/eject is in progress, or `RESP_NO_DEV` if there is no device).
  - For the block driver, we re-use the *Block State Notification* channel. We rely on restricting the channel capabilities so that only certain clients are able to perform PPCs; thereby reusing existing security mechanisms.
  - For other device classes, we can re-use an appropriate channel or add a dedicated hotplug control PPC.

6.3.2 Requirement 2: Policy-free Virtualiser Remapping

We extend the virtualiser with a policy interface (`blk/components/virt.h`), with the following prototypes:

```
1 int get_drv_block_number(uint32_t cli_block_number, uint16_t cli_count,
2                          int cli_id, uint32_t *drv_block_number);
3
4 bool policy_init(void);
5
6 void policy_reset(void);
```

Listing 6.1: Virtualiser policy interface prototypes.

The block virtualiser will call `policy_init()` until it returns true for any block queue notifications; at which point it can set the client's ready flags to true. The `get_drv_block_number()` function is called on every client request, and allows the policy component to map the client requests to device blocks (or deny them entirely, if invalid).

By abstracting these interfaces out, different policies can be implemented, without needing to change the virtualiser code itself.

The `policy_reset()` function is called to reset the policy state, and allow `policy_init()` to be called again. This enables us to reinitialise the policy again — for instance, the exact partition layout may have changed, and will need to be reassigned to clients.

We have migrated the existing static MBR mapping to use this interface. For other device classes, we could use a similar approach if necessary.

6.3.3 Requirement 3: 'Safe' Ejection

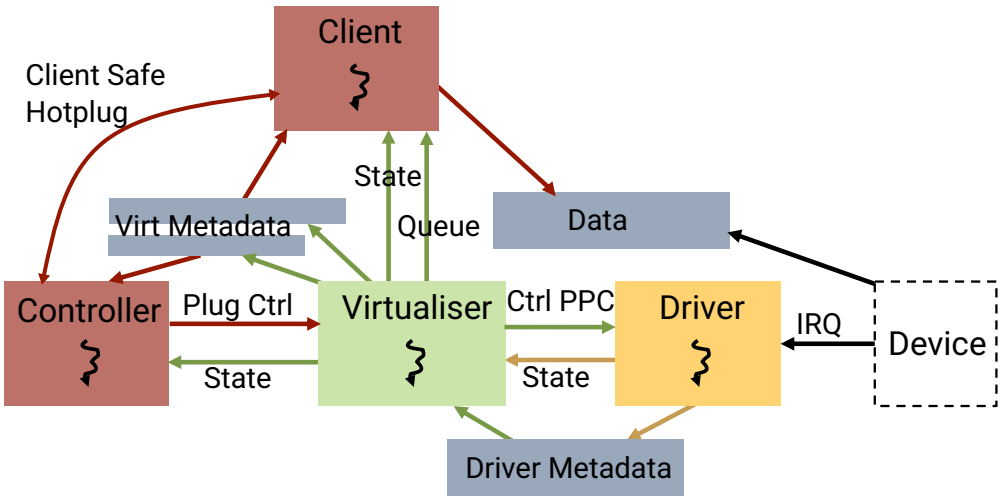

Figure 6.2: A structure showing how a controller can request device ejection.

We assume that we have a specific privileged client that we designate as the 'controller client' that acts as the human interface to the system. We broadly have two approaches that we could take:

1. The controller client talks to the virtualiser, which notifies all of the downstream clients (who have to notify any further downstream clients, in the case of a filesystem client for a block virtualiser). The clients can then safely stop, and notify back up the chain to the virtualiser, who can then request the driver to eject.
2. The controller client talks directly to all the leaf clients to safely stop, and these clients respond when complete.

The advantage of (1) is that we re-use the existing virtualiser infrastructure, and so don't need to map extra pages into the clients. However, we can't effectively re-use the existing virtualiser. We would have to re-use the *Block State Notification* to notify clients, who then, in the case of a filesystem, need to notify their clients is some other way. If it's another device class, we wouldn't be able to use this, and would need to re-use their queues (if they have a compatible queue) or define some other method. Furthermore, when the clients need to talk the virtualiser (or other clients, again) using the queues doesn't make sense, so we need to use ppcalls again.

In contrast, (2) imposes the exact same burden on clients — being notified and responding - but does not require modifications to the virtualiser and intermediary clients. In addition, policies regarding timeouts waiting for 'ready for ejection' are implemented within the controller client, not the virtualiser, which is simpler. The only distinction between a 'safe' ejection and an unsafe one is in the clients, from the virtualiser and driver's perspective it is just an ejection.

We implement option (2) with a thin wrapper over notifications in `include/sddf/hotplug/clients.h`. A shared page for each client is mapped R/W in the controller, and R/O in each client, containing a boolean representing the pending eject state. The controller can call `hotplug_set_pending_eject()` to notify the clients of an ejection request, who respond with a ppcall back in `hotplug_ready_for_eject()`. In our sample implementation, the controller waits for all clients to respond, but this policy could easily be changed.

6.4 Limitations

- Race conditions with block queues during a quick eject and insert request. Requests sent for the old device could end up being sent for the new one, from the perspective of the virtualiser and driver. Each block request needs to be tagged with the device ID or a generation number so old requests can be invalidated.
- Initialisation failures are not communicated to the clients.

Chapter 7

Device Discovery

There is **preliminary** work on extracting device parameters from the Linux *device tree* specification as part of configuration tool for the seL4 Microkit [Trustworthy Systems Group, 2023]. This is not yet in a halfway stable state. We will describe this in a later version.

Currently, it is expected that the system designer knows all the devices that will be available and used, and can choose components and assign memory manually to use all the devices that are needed for the system being built.

Chapter 8

Leveraging Linux

Note: This chapter is **preliminary**.

8.1 Component development

The Linux *userspace I/O* (UIO) mechanism [Koch, 2016], designed to support usermode drivers, allows mapping physical memory regions uncached into Linux user space.

We can use this to develop sDDF virtualisers and device drivers in Linux user space, on a Linux system running inside a *virtual machine* (VM) on top of seL4. This allows us to use the normal Linux development tools for component development. While not suitable for performance analysis and tuning, this can simplify and accelerate development.

```
chosen {
    bootargs = "... uio_pdrv_genirq.of_id=sddf_framebuffer ...";
};
```

Listing 8.1: Example UIO stub driver command-line configurations.

UIO works with a stub driver in the Linux kernel. There are two generic drivers provided in the mainline Linux source tree, `uio_pdrv_genirq` and `uio_dmem_genirq`. `uio_pdrv_genirq` is used when the device is not a bus-mastering DMA controller; `uio_dmem_genirq` when the device can do DMA. These drivers do not specify which devices they are compatible with; they can be bound to a particular device in various ways. We usually bind at boot-time using a Linux kernel command line argument as shown in Listing 8.1.

```
uio@0x30000000 {
    compatible = "sddf_framebuffer","uio";
    reg = <0x00 0x30000000 0x00 0x2000000>;
    interrupts = <GIC_SPI 18 IRQ_TYPE_EDGE_RISING>;
};
```

Listing 8.2: Example UIO node in device tree.

To use UIO, a node describing the interface is added to the device tree handed to the virtual machine. For details, see the device-tree specifications in the Linux source tree, Listing 8.2 shows an example.

You can choose the `compatible` string to be whatever does not clash with a real device. The `reg` parameter needs to give the addresses of the regions. When running sDDF (and the development VM) on top of the seL4 Microkit [Trustworthy Systems Group, 2023], this will need to match the address of the memory region specified in the Microkit's *system description file* (SDF), to ensure that Linux and the sDDF component under development see these regions at the same address.

The above example specifies a single 32 MiB region at (virtual-machine) physical address 0x30000000. Likewise, the `IRQ` parameter gives the IRQ controller, IRQ number (relative to the first interrupt on that controller) and the type. Again this must be different from any other interrupt in the device tree.

The virtual-machine monitor (VM) maps each interrupt onto a separate Microkit *channel*. If such a channel is connected to the VM, a notification on that channel (resulting from an interrupt or a notification from another component) will result in injecting the corresponding virtual interrupt into the VM. When the VM re-enables that (virtual) interrupt, this triggers a notification to the Microkit protection domain at the other end of the channel, or an interrupt acknowledgement operation (if the notification was from a real interrupt).

A usermode program running in the VM can then use `mmap()` calls on `/sys/class/uio`N`/maps/map`n`/addr` to access metadata and data regions, `read()` or `poll()` system calls on `/dev/uio`N to wait for notifications, and `write()` on the UIO device file to generate notifications. Here, N identifies the UIO device (of which there can be several), while n refers to the UIO-mapped region of the device (of which there can be multiple as well).

Most of this is abstracted into a (still under development) `libuio` library, making it possible to write components that are relatively easy to port between Linux user space and native Microkit.

In addition, one can prevent the Linux in-kernel driver from binding to a particular device, by changing its `compatible` string in the device tree. One can then bind it to a UIO stub driver and write a user-mode sDDF-style driver that runs in Linux userspace on the guest.

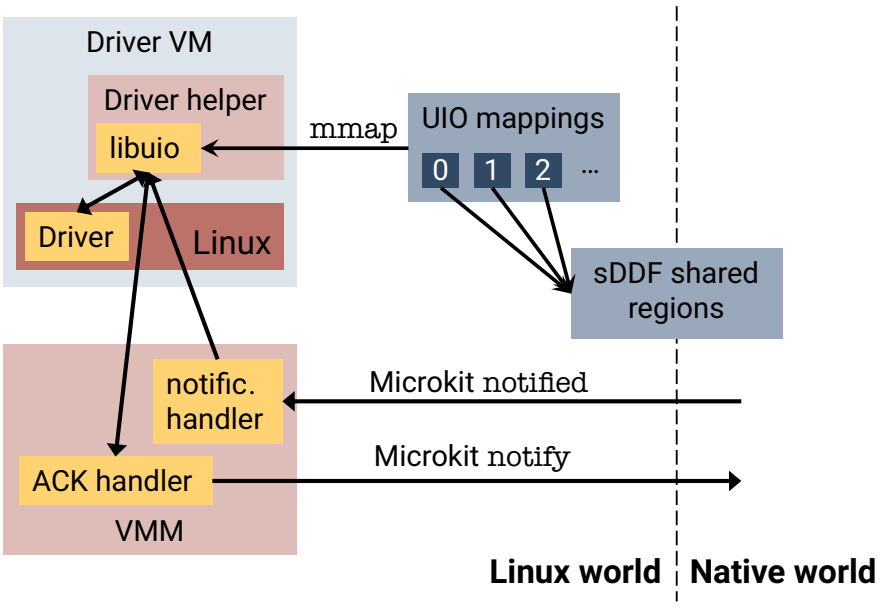

Figure 8.1: Linux driver re-use through UIO.

8.2 Legacy driver reuse

Instead of developing drivers in Linux, we can use the same setup for re-using existing Linux drivers in the sDDF. A virtualiser connects to a “driver PD” that is in reality a Linux VM containing the driver and a userspace program, the *driver helper*, that interacts with the sDDF using the above UIO mechanisms, as indicated in Figure 8.1.

```
notified(int channel) {
  needs_notify = false;
  while (dequeue_request(request_queue, &request)) {
    switch (request.storage_command) {
      case READ_BLOCKS:
        llseek(fd, BLOCKSIZE*request.block_number, SEEK_SET)
        count = read(fd, adjust_address(request.address),
                        BLOCKSIZE*request.block_count);
        response.count = request.block_count;
        response.success = count/BLOCK_SIZE;
        response.result = count == BLOCKSIZE * request.block_count
                          ? SUCCESS : map_error(errno);
        enqueue_response(response_queue, &response)
        needs_notify = true;
        break;
        ...
    }
  }
  if (needs_notify)
    notify(channel);
}
```

Listing 8.3: Example of a Linux usermode driver helper for a block device accessed via a regular Linux file (simplified).

The driver helper forwards sDDF requests to the Linux in-kernel driver using standard Linux I/O mechanisms. This can be as simple as reading block-sized chunks from a Linux block device, or even a regular file, using normal Linux `read` or `write` calls, as shown in Listing 8.3. The `notified()` function is invoked by `libuio` when the notification channel is signalled. The `notify()` function writes to the appropriate `/dev/uio` special file to cause a notification by the VM on the corresponding channel.

The virtualiser code is unaware of whether the driver is native or a driver VM, the only difference is that the driver metadata region must be configured as *uncached* in the Microkit SDF.

Obviously this approach will not result in high-performance drivers, nor can Linux drivers be trusted, but most drivers are not particularly performance-sensitive, and trust can be minimised by encrypting data that is sent to the device (reducing the potential attacks to denial of service). These limitations are frequently acceptable, particularly if this allows obtaining complex drivers for free.

In such a case, the Linux instance should be reduced to the bare minimum to support the driver in question, as demonstrated many years ago by LeVasseur et al. [2004].

Chapter 9

Security Analysis

Note: This chapter is **preliminary**.

An implication of the threat model of Chapter 2 is that any trusted component of the system must be verifiable. Another implication is that components that interface with untrusted components must sanitise their inputs.

The design of our queues means that pointers into those queues are automatically sanitised (the index is taken modulo the queue size). Similarly, pointers into the data and metadata regions on the client-side of the virtualiser are all offsets into the respective region and only need a check that they do not exceed the region size. These requirements should be easy to verify.

9.1 Trusted components

For now we assume all the components of the sDDF to be trusted, specifically:

- drivers are trusted
- virtualisers are trusted
- copiers are trusted.

In addition, the device hardware is (currently) trusted.

Untrusted device drivers or devices should be achievable, provided that:

- all data handled by the device is encrypted and signed (using higher-level protocols);
- denial of service by the device or driver is not considered a threat;
- the device is not DMA-capable, or its access is restricted by suitable IOMMU mappings (managed by the virtualiser);
- the virtualiser sanitises all references received from the client.

9.2 Verifying components

The driver model is designed to simplify drivers dramatically (compared with e.g. Linux drivers). Drivers are simple state machines free from internal concurrency (but do

access/modify data structures that are concurrently accessed/modified by hardware). In fact, experience with Ethernet drivers shows that they are so simple that little debugging effort is required to achieve correct functionality [Parker, 2023].

The simplicity should make it finally feasible to formally verify drivers for non-trivial real-world devices, such as NICs, although verification requires not only formalising the driver framework but also the device interfaces.

Similar arguments apply to virtualisers: they are also kept simple (even simpler than drivers) and should be verifiable.

The use of virtualisers allows simplifying the IP stack as well, as it only handles packets for a single client and does not have to deal with broadcast requests. Such a simple IP stack may be verifiable as well, and there is work attempting exactly this [Rollins, 2023]. A verified IP stack may eliminate the need for copiers.

When blocking on a response (Notification or IPC reply) from an untrusted component, a trusted component may have to employ a watchdog timer to prevent indefinite blocking, depending on its purpose.

9.3 Verifying component interactions

Simplifying drivers and virtualisers by removing any internal concurrency potentially shifts some of the complexity into the driver framework itself, including concurrency. In fact, our experience shows that the dominating debugging task is not of the driver/virtualiser functionality, but the inter-component signalling protocol. Achieving deadlock freedom is not difficult in itself, but achieving it without having components perform unnecessary signals resulting in other components being awoken without having useful work to perform is the challenge. Optimisations to the simplest policy of "signal each time work has been done" can easily lead to unforeseen deadlocks or delays in processing further down the line, and without the use of an automated tool it can be difficult to calculate by hand whether potential optimisations could lead to a stagnation in the system.

Thus, this is an ideal use case for model checking, and we have developed simplified abstract models of our networking components in order to verify that each optimisation to our signalling protocol cannot lead our system into deadlock. As the signalling protocol is updated during development, we are able to update our abstract models accordingly and rerun them through the verification tool to ensure that we have not inadvertently introduced any issues.

An advantage of the sDDF model is that this complexity is encapsulated in a relatively small code base, and the verification needs to be done only once per device class (and not for each individual driver).

The sanitation requirements on drivers and virtualisers apply to the framework implementation as well, and should be verified.

Chapter 10

Implementation Status

The sDDF is implemented as described above on top of the seL4 Microkit [Trustworthy Systems Group, 2023]. The Microkit simplifies the implementation (among others) by moving the handler loop (see Listing 4.5) into the framework, so the driver and virtualiser implementations mostly consist of the notification handler functions.

The Microkit implementation presently provides high-performance networking (see Chapter 11). There are a implementations for a number of other device classes (serial, I2C, storage, graphics, audio) which are still in a preliminary state. The framework is presently based on active virtualiser and driver components.

The code accessible on GitHub under a BSD license: https://github.com/au-ts/\gls{sddf}.

Chapter 11

Performance

11.1 Performance evaluation setup

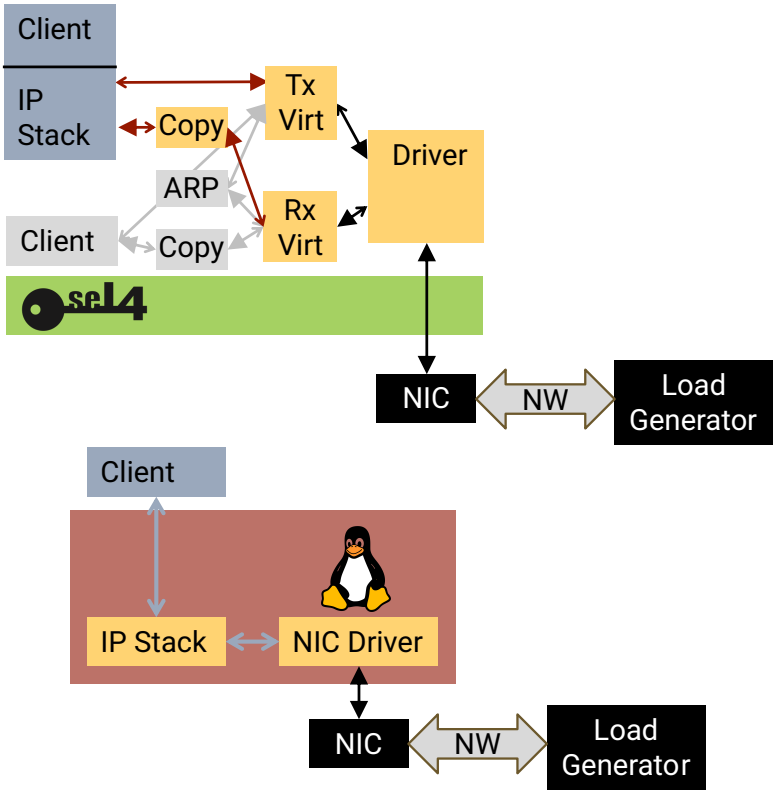

Figure 11.1: Evaluation setup for sDDF (top) and Linux (bottom).

We evaluate sDDF networking performance against the older CAmkES-based driver framework, as well as against Linux.

The evaluation platform is an Arm Cortex-A53 processor based on the i.MX8M Mini microarchitecture running at 1.8GHz, equipped with a Gigabit Ethernet NIC using a 10/100/1000 Atheros AR8031 PCM. We run *ipbench* [Wienand and Macpherson, 2004] for distributed load generation from four client machines, each sending UDP packets to the seL4-based target machine, which sends them through an IP stack (*lwIP* [Dunkels, 2001]) to

a co-located client, which returns the packets, slightly modified according to the ipbench protocol. The load generators count the successful replies to determine the achieved throughput and latency, while a low-priority thread on the target machine measures idle time to determine the CPU load imposed by the test.

Figure 11.1 shows the setup. The sDDF system is based on the seL4 Microkit [Trustworthy Systems Group, 2023]. Linux benchmarks use a Linux 6.1.1 kernel system on the same hardware.

11.2 Performance results

11.2.1 Simplified system, CAmkES, Linux

We initially evaluate a simplified two-component system, where the client, IP stack and cache management are merged into a single component (no device sharing supported), and only the driver is separate. This setup, while not very realistic, is directly comparable to the older driver framework based on CAmkES [Kuz et al., 2007]. We also convert the CAmkES framework to our transport layer and run it in the same, two-component configuration.

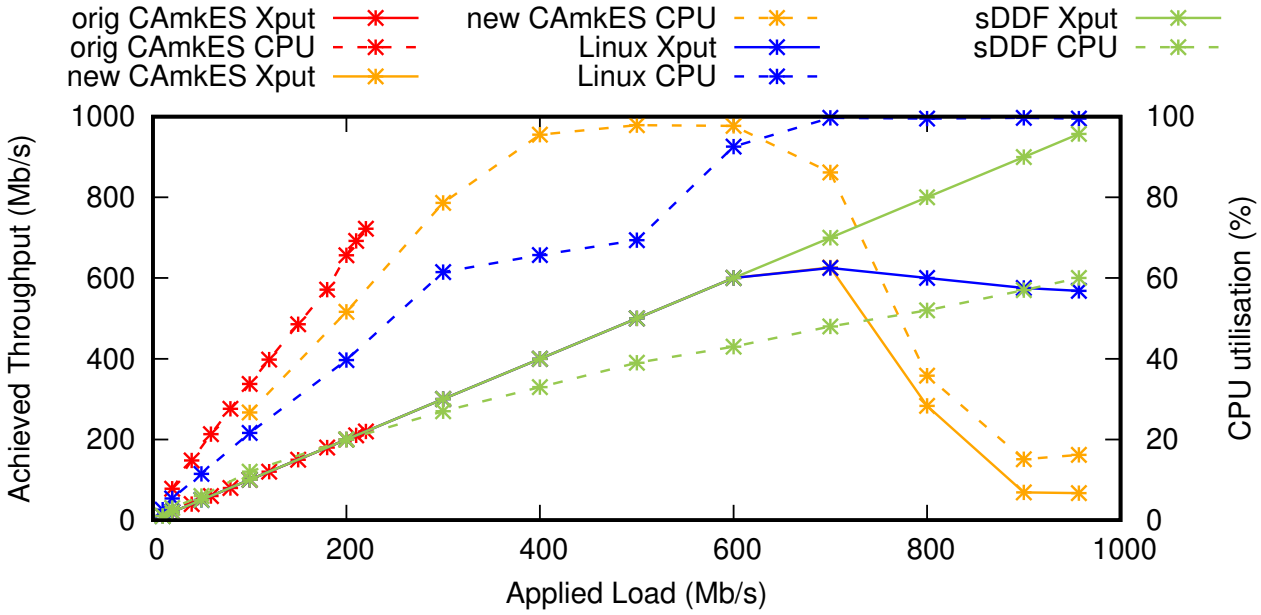

Figure 11.2: sDDF networking performance, compared to original CAmkES, CAmkES with the new transport layer, and Linux. All benchmarks use a single core.

Figure 11.2 shows the results. We can see that the original CAmkES system already reaches 66% CPU utilisation at 200 Mb/s applied load, and can be expected to saturate the CPU at around 350 Mb/s.[1]

CAmkES adapted to the new transport layer ("new CAmkES") uses somewhat less CPU (52% vs. 66% at 200Mb/s, a 20% improvement) but maxes out the CPU above a load of 400 Mb/s. Achieved throughput scales with applied load to 600 Mb/s, and then collapses when applied load exceeds 700 Mb/s. Performance collapse under overload is not untypical in network systems, and generally results from latency blowout.

[1]The measurements on "original CAmkES" (in fact after a fair amount of performance tuning) were done in 2021 and we only have records for the low-load cases.

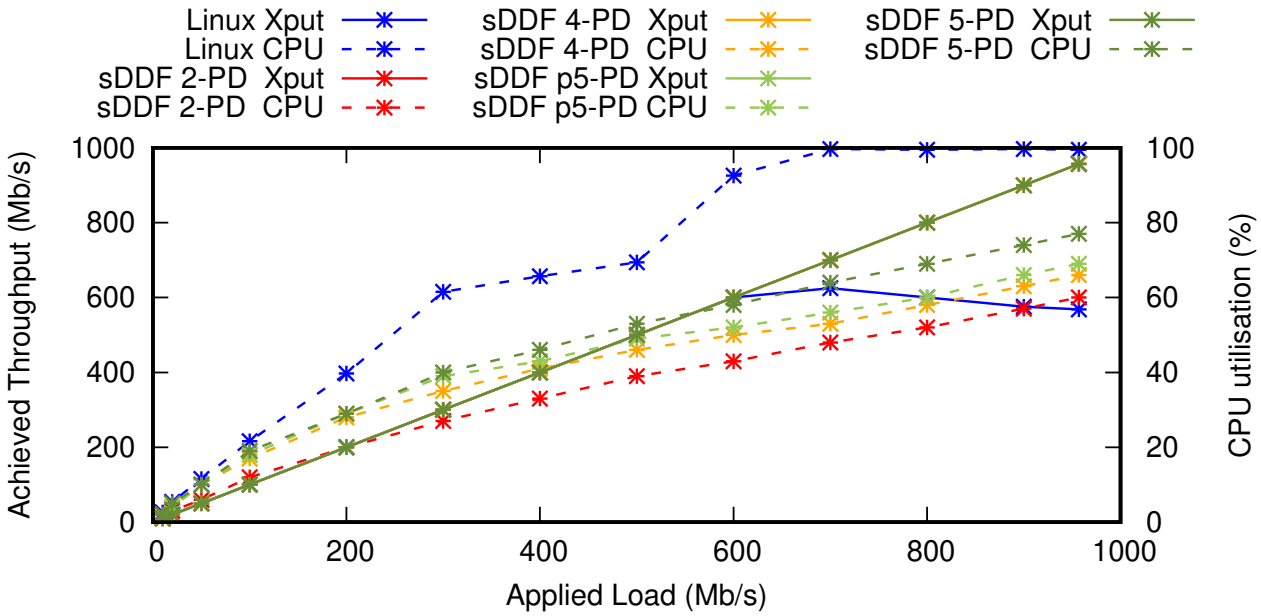

Figure 11.3: Performance of a two-, four-, pseudo-five- and a standard five-PD sDDF system compared to Linux, all on a single core.

Linux behaves somewhat better, also scaling just over 600 Mb/s, when it saturates the CPU. While nowhere near as extreme as the performance collapse of CAmkES, Linux throughput still degrades somewhat under overload.

In contrast, the sDDF system scales linearly to the wire speed.[2] It does *not* max out the CPU, but handles the full applied load with only 60% CPU utilisation, leaving plenty of performance head space. At all load levels it uses far less CPU than Linux. At low load (100 Mb/s), the sDDF CPU usage is 12%, beating all the other systems by a factor of 1.9 or more. The ratio becomes smaller at higher load due to batching.

11.2.2 Cost of modularity and security

Now we compare the simplified two-PD system to the more realistic one of Figure 11.1, where five PDs (out of a total of 8) are involved in handling each packet, *and* incoming data is copied to separate client address spaces. For more analysis, we present also two intermediate systems: a four-PD system, which has the Copier component removed, and a pseudo-five-PD system (labelled “p5-PD”) where instead of a copier we have a dummy component that just forwards packets without doing any meaningful processing.

Figure 11.3 shows the results. We can see that the three extra context switches plus one data copy increase CPU usage noticeably, from 12% to 19% at 100 Mb/s, and from 60% to 77% at high load, a relative increase of 64% at low load and 27% at high load (again, the effects of overheads are lessened by batching). Yet, this highly modular setup still clearly outperforms Linux by more than a factor 1.5 (low load).

The intermediate systems show that the extra data copy plays a similar role as the extra context switches, and they all add low to moderate overheads. This becomes clearer in Figure 11.4, which compares per-packet processing costs. It clearly shows that the difference

[2]Note that a Gigabit Ethernet NIC can handle about 950 Mb/s of payload with standard 1.5 KiB Ethernet frames, the rest of the bandwidth is lost to Ethernet headers and inter-packet gaps.

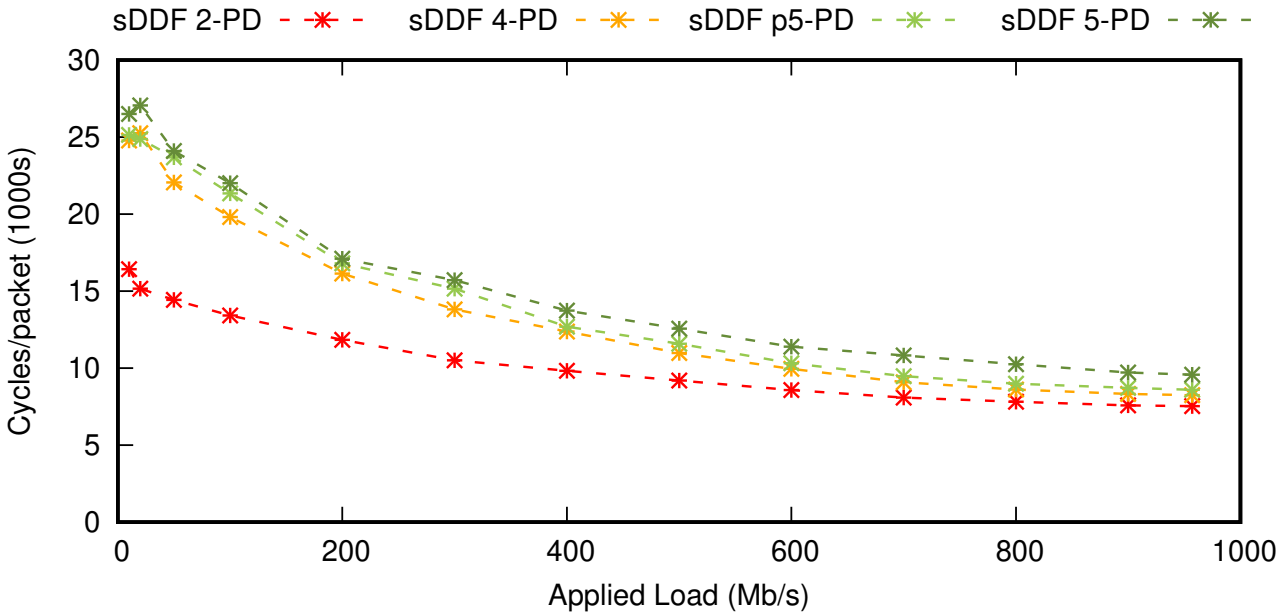

Figure 11.4: Comparing per-packet processing costs of the four sDDF systems.

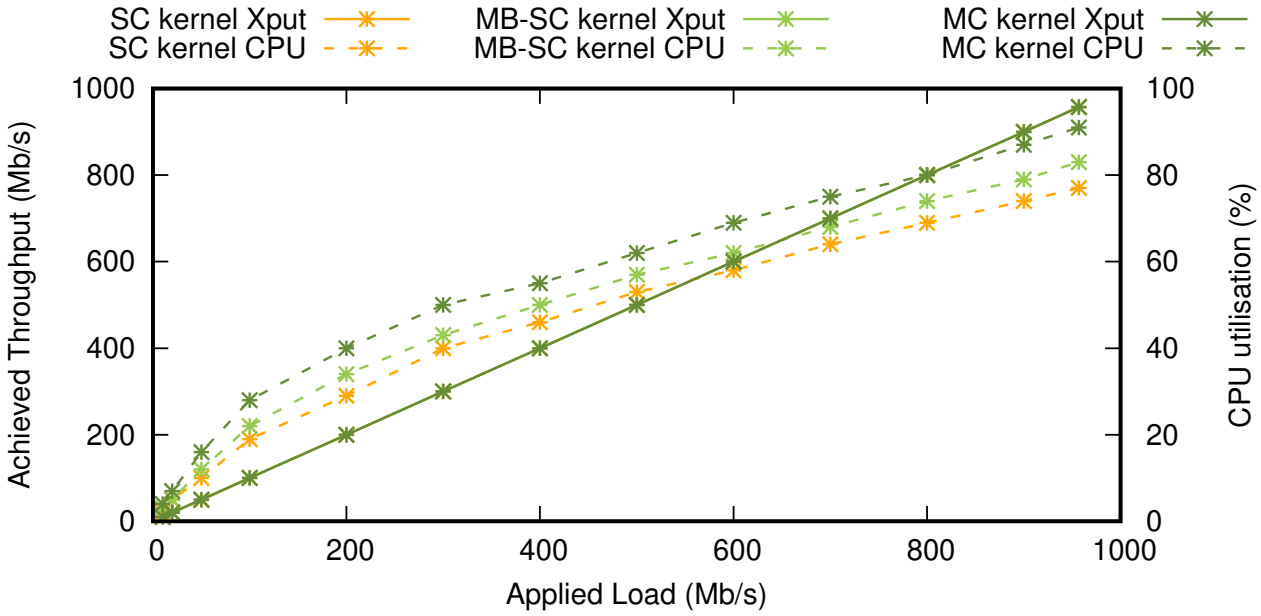

Figure 11.5: Single-core sDDF performance using single core seL4 kernel with and without memory barriers enabled vs multicore seL4 kernels.

between the 4-, pseudo-5- and 5-PD systems is fairly small: the extra PD (without copying) increases per-packet processing cost by 1.5k cycles at 100 Mb/s and 350 cycles at wire speed, a relative increase of 8% and 4% respectively. In comparison, copying adds 700 cycles at low and 1,000 cycles at high load, a relative increase of 3% and 11% respectively.[3]

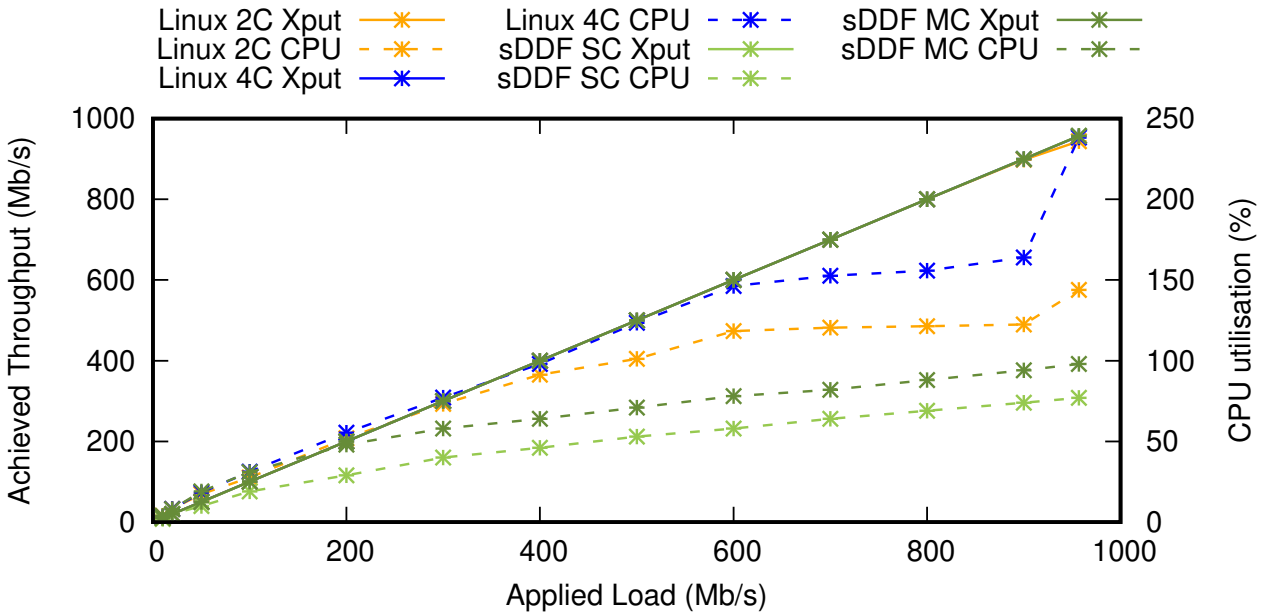

Figure 11.6: Performance of sDDF running single-core (sDDF SC) vs. distributed over two cores (sDDF MC) compared to Linux running on two (Linux 2C) or four (Linux 4C) cores.

11.2.3 Multicore

All the above benchmarks were run on a single core, using an system that is optimised for that case: the seL4 kernel compiled with multicore support disabled, and no memory barriers in the sDDF queue libraries.

Enabling multicore increases the baseline system-call cost of seL4 by about 50%, basically the cost of acquiring and releasing the kernel lock. Additionally, supporting a multicore system requires memory barriers when enqueuing or dequeuing entries in the shared queues, to ensure they work correctly of the other PD accessing the queue runs on a separate core.

The CPU load figures in Figure 11.5 show that both requirements add similar overheads. Here we compare the strict single-core configuration used so far (“SC kernel”) with a version that adds the memory barriers but still uses the single-core kernel (“MB-SC kernel”), and additionally the same sDDF version running on seL4 with multicore enabled (“MC kernel”). Adding memory barriers adds about 15% overheads at low load and 8% at high load, while the total cost of enabling multicore is almost 50% at low load and almost 20% at high load.

From now on we use the kernel and sDDF version that fully supports multicore. We compare multicore-enabled sDDF running on a single core with sDDF with its components distributed over two cores: the driver and Tx virtualiser run on one core, and the other components, Client, Copier and Rx virtualiser, on the other. We compare to two Linux configurations, one restricted to two cores and the other giving it free choice of all four cores.c

The results are in Figure 11.6. Having 2 cores at its disposal, Linux just manages to handle the applied load (as indicated by latencies, which increase significantly at the highest applied load). Its CPU usage is well above that of the sDDF system. On four cores, Linux shows significantly higher overall CPU load, while still barely coping with the network load (indicated by the CPU usage blowing out at 950 Mb/s, and also reflected in a latency blowout).

[3]One would expect the absolute per-packet copying cost to be independent of load. We did not investigate this further, but suspect caching effects.

In contrast, the sDDF system still has significantly lower CPU utilisation than Linux, total CPU load staying (just) below 100% of one core even at the highest network load. The total CPU load of the two-core configuration is 65% higher (31% vs 19%) than the single-core version at low load and 30% higher (98% vs 77%) at high load. This is due to cross-core notifications being more expensive than intra-core notifications.

The overall CPU load of an sDDF system will obviously be sensitive to the allocation of PDs to cores, which, in turn, depends on other considerations, such as maximising cache locality or freeing up cores for other activities. The optimal configuration will therefore be dependent on the use case. The multicore-enabled configuration provides full location transparency to PDs.

11.3 Discussion

The framework, running on the Microkit, clearly over-achieves our goal of “performance comparable to Linux”, outperforming Linux by a significant margin under all configurations evaluated. Performance is vastly better than that of the CAmkES system adapted to our transport layer (which is already a significant improvement over the performance of the previous CAmkES-based driver framework).

The performance comparison to Linux is especially encouraging, given that the Linux IP stack is supposedly well optimised (especially for multicore use), while we are using the simple lwIP [Dunkels, 2001]. We find that even on multicore (the scenario most favourable to Linux) our per-packet costs are significantly lower.

Parker [Parker, 2023] conducted a more comprehensive performance analysis, including highly asymmetric (send-mostly or receive-mostly) traffic, multiple active clients with traffic shaping, more multicore configurations, and forcing the system to be overloaded. The evaluation confirmed that the sDDF/Microkit system shows no performance collapse under any overload conditions evaluated, unlike what we observed in Section 11.2.1 for the CAmkES system.

The evaluation, specifically the results shown in Section 11.2.2, provide strong justification for our highly modular design: The cost of a PD switch is small compared to the base processing cost.

This confirms the original hypothesis driving the sDDF design: Context-switching costs (of the order of 400–500 cycles for the single-core kernel on our Armv8 platform, about 50% higher for the multicore kernel, and still higher for cross-core notifications) are not a major factor. In fact, we measure handling costs of less than 50,000 cycles per packet on sDDF under any conditions, and no more than about 12,000 cycles under high network load. Hence, the pure kernel overhead of adding an extra component (two system calls) to the packet-processing pipeline adds no more than about 15% (usually much less) to the handling cost.

As such, our main take-away is that the fine-grained modularisation, resulting from the strict separation of concerns, works and adds far less overhead than the complexity of the Linux system. The reason is two-fold:

1. as stated, context-switching costs in seL4 and the Microkit are small enough to not significantly impact performance, and
2. the sDDF model encourages implementation simplicity that can *reduce* overheads (which is the reason sDDF outperforms Linux).

The latter has further, highly encouraging implications: We can keep individual component implementations very simple, without sacrificing performance. This simplicity not only greatly reduces debugging efforts, it should also make automatic verification techniques applicable. The experience with verifying the Microkit [Paturel et al., 2023] gives us confidence that this should be achievable.

The evaluation of the multicore configurations are also highly encouraging: It shows that the sDDF naturally distributes across cores. While all PDs are strictly sequential (resulting in decreased code complexity), the design makes them location-transparent, meaning they can be arbitrarily allocated to cores in order to balance load. The evaluation confirms the feasibility of this design choice of keeping components sequential and instead allowing arbitrary concurrency *between* components.

Further performance improvements seem possible with improvements to the kernel itself. Candidates are fast-pathing notifications to higher-priority threads, and taking notifications out of the kernel lock.

Chapter 12

Conclusions

We have developed a low-overhead device driver framework for seL4, sDDF, implemented it on the seL4 Microkit, and comprehensively evaluated it on network devices. It is based on a highly efficient, asynchronous zero-copy transport layer using lock-free single-producer, single-consumer, bounded queues.

Measured performance shows a vast improvement over the CAmkES-based framework, as well as significantly better performance than Linux, despite the higher number of system calls and context-switches required. In particular we find that a highly componentised sDDF system easily distributes across cores, while each individual component being strictly sequential, reducing its complexity.

So far we have only examined the performance of the network system. We have started applying the same ideas to other device classes, especially storage. Other device classes are generally much less sensitive to system performance than networking, so we do net expect any surprises there.

We are also working on integrating sDDF with the Microkit's virtualisation infrastructure, which will support VMs as clients as well as drivers and virtualisers. On the one hand, this provides a convenient way of developing sDDF components as Linux programs that can later be converted into native sDDF PDs. On the other hand, this allows re-use of existing Linux drivers in the form of a driver-VM.

Obviously, such legacy drivers are in no way trustworthy. Furthermore, virtualisation overheads are expected to dominate sDDF overheads, so these configurations are important for functionality (especially the ability to re-use unmodified Linux drivers) but will not be suitable for performance-sensitive devices.

Bibliography

Adam Dunkels. Minimal TCP/IP implementation with proxy support. Technical Report T2001-20, SICS, February 2001. http://www.sics.se/~adam/thesis.pdf. 48, 53

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, October 2009. ACM. URL https://trustworthy.systems/publications/nicta_full_text/1852.pdf. 1

Hans-Jürgen Koch. The userspace I/O HOWTO, 2016. URL https://www.kernel.org/doc/html/latest/driver-api/uio-howto.html. Retrieved 2024-2-27. 42

Hermann Kopetz. The time-triggered architecture. *Proceedings of the IEEE*, 91:112–126, 2003. 23

Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. CAmkES: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software Special Edition on Component-Based Software Engineering of Trustworthy Embedded Systems*, 80(5): 687–699, May 2007. URL http://dx.doi.org/10.1016/j.jss.2006.08.039. 49

Ben Leslie, Peter Chubb, Nicholas FitzRoy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting (Rita) Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, September 2005. URL https://trustworthy.systems/publications/papers/Leslie_CFGGMPSEH_05.pdf. 8

Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, San Francisco, CA, US, December 2004. 44

Lucy Parker. High-performance networking on seL4. BSc(Hons) thesis, School of Computer Science and Engineering, Sydney, Australia, November 2023. 23, 46, 53

Mathieu Paturel, Isitha Subasinghe, and Gernot Heiser. First steps in verifying the seL4 Core Platform. In *Asia-Pacific Workshop on Systems (APSys)*, Seoul, KR, August 2023. ACM. URL https://trustworthy.systems/publications/papers/Paturel_SH_23.pdf. 54

Wyeth Greenlaw Rollins. Toward a verified, minimal IPv6 network stack implementation, September 2023. URL https://sel4.systems/Foundation/Summit/2023/abstracts2023#a-toward-verified. Talk at the seL4 Summit. 46

Mark Rutland. Stale data, or how we (mis-)manage modern caches, 2016. URL http://events17.linuxfoundation.org/sites/events/files/slides/slides_17.pdf. 17

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *EuroSys Conference*, pages 275–288, Nuremberg, DE, April 2009. URL https://trustworthy.systems/publications/nicta_full_text/1527.pdf. 7, 8, 16

Leonid Ryzhyk, Yanjin Zhu, and Gernot Heiser. The case for active device drivers. In *Asia-Pacific Workshop on Systems (APSys)*, pages 25–30, New Delhi, India, August 2010. URL https://trustworthy.systems/publications/nicta_full_text/3681.pdf. 16

The seL4 Foundation. The seL4 microkernel, 2021. URL https://sel4.systems/. 1

Trustworthy Systems Group. The seL4 Microkit, October 2023. URL https://trustworthy.systems/projects/microkit/. 41, 43, 47, 49

Ian Wienand and Luke Macpherson. ipbench: A framework for distributed network benchmarking. In *Conference for Unix, Linux and Open Source Professionals (AUUG)*, pages 163–170, Melbourne, Australia, September 2004. URL https://trustworthy.systems/publications/papers/Wienand_Macpherson_04.pdf. 48

Wikipedia. Self-monitoring, analysis and reporting technology, 2004. URL https://en.wikipedia.org/wiki/Self-Monitoring,_Analysis_and_Reporting_Technology. 30

Appendix A

OS Abstraction

The sDDF is meant to be usable by all seL4-based OSes. As such it needs an abstraction layer for invoking underlying OS services. This abstraction layer consists of a number of (inlineable) functions that map the mechanisms required by the sDDF to the underlying OS.

A.1 Microkit Mapping

Appendix B

Overview of Changes

B.1 Changes Since Release 0.4 of 2024-03-27

B.1.1 OS abstraction

We made the sDDF largely independent of the underlying OS by providing an OS abstraction layer (Appendix A). We specify the mapping to the Microkit in Section A.1.

B.1.2 More device classes

We added the Section 5.3 class.

B.2 Changes Since Release 0.2 of 2022-10-03

The document has undergone major revision and restructure, with changes in many places. We summarise the main changes.

B.2.1 More device classes

While networking is still the running example of the bulk of the document, we tried to point out where other device classes may differ.

We moved device-specifics into a separate chapter, provide details of some device classes, and added place-holders for others.

B.2.2 Mandatory virtualisers, cache maintenance

We generalised the concept of multiplexers to virtualisers, as they are also responsible for translating between client virtual and I/O addresses. Consequently, virtualisers are now mandatory components (required even if a device only has a single client).

We include a discussion of cache maintenance, and specify that this is the responsibility of the virtualisers.

B.2.3 Completely separated Tx and Rx paths

We require that for device classes providing spontaneous input (e.g. networks) the Tx and Rx paths are completely separated. This means separate Tx/Rx virtualisers, and separate Tx/Rx data and metadata regions.

B.2.4 Clarified terminology

We removed the term “ring buffers” as this caused confusion with data buffers. Instead we consistently use the term “queue” for the metadata structures and reserve “buffer” for the actual I/O-data locations.

We now require the transmit, receive, request metadata regions as well as the virtualisers (formerly: multiplexers) to be disjoint (before this was considered an option).

We removed the concept of a “Server” in the driver model, there are only drivers, virtualisers and “clients” (the latter possibly consisting of a pipeline of components).

We use the term *metadata region* for all software-defined regions shared between components (other than the data regions). We explicitly distinguish the device’s *control region* (i.e. the memory-mapped device registers) from its metadata region.

We clarified (and fixed!) the concept of *ownership* of I/O data and metadata, to aid verification.

B.2.5 Linux-based component development and legacy re-use

We added Chapter 8 which describes how to leverage UIO to do driver development under Linux, and integrate legacy Linux drivers.

B.2.6 Performance evaluation

We extended and updated the performance evaluation, including extending to multicore scenarios.

We furthermore found that the Linux distribution we used to benchmark against had serious performance deficiencies which do not exist in the mainline kernel, so we re-ran Linux evaluations with a version built from source.

We extended the performance discussion as a result.

In general, performance is significantly improved compared to the original release, and keeps out-performing Linux.

B.2.7 Changes resulting from evaluation

We changed/refined details of the Ethernet driver interface as the result of evaluating implementation practicalities and performance.

We also reported on other implementation/evaluation experience, including debugging and the use of model checking of the signalling protocols.