School of Computer Science & Engineering
Trustworthy Systems Group

sDDF Design

Design, Implementation and Evaluation of the seL4 Device Driver Framework

Lucy Parker, Peter Chubb, Gernot Heiser

Release 0.2, October 2022

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

1 Aims of the sDDF

The 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 cards and USB);
- be robust against defined threats;
- extend to a device virtualisation framework (sDVF) for sharing devices between virtual machines and native components on seL4;
- 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 needed functionality, such as device discovery, device initialisation and mapping of device virtual memory for future work. Also, the framework is presently somewhat network-centric, with block devices to be added next.

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 server;
- 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 driver;
- compromise a trusted server;

- 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).

3 Device Model

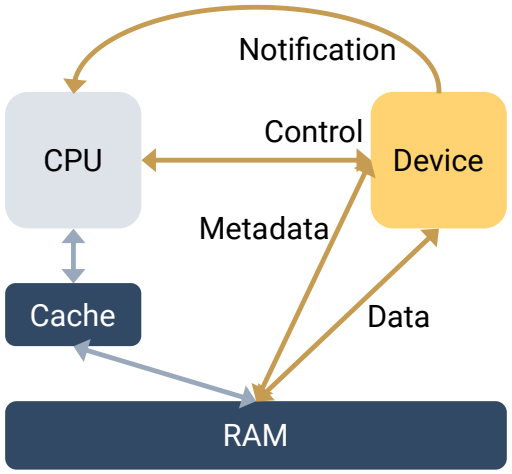

Figure 1: Device model.

We view a device as a state machine with four interfaces, indicated in Figure 1:

the control interface specifies how the 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 interfaces 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 eg. 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 manangement 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 differ 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 IO 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.

- Addresses used for DMA are either not mapped (i.e., they are physical addresses), or are mapped via a different MMU, the IOMMU, to 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 trigged by access) of device registers. It also specifies some ordering conditions on accesses (reads or writes). It may specify timing condition 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 (output completed).

A sequence of state transitions may move the device to a state where it can provide **input** to DMA buffers specified by the metadata interface, which in turn was referenced by the control interface. The commencement of input is to software 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 (data available).

Once an interrupt has triggered, the hardware disables ("masks") the interrupt (and lower-priority ones) from re-occurring. The kernel immediately acknowledges and masks that specific interrupt (unmasking all others). 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.

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, it 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 (eg. 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.

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.

4 Driver and Driver Interface Design

Our driver model strongly reflects the aim of separation of concerns: *the sole the 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 server-driver interface specification for each device class (network, USB, storage, etc). That interface is designed to support a lightweight, low-overhead translation, at least for performance-sensitive (high-throughput) devices.

4.1 Overview

In our design we utilise the flexibility afforded by a clean-slate design, where we can control the server-driver interface as well as the structure of the drivers. 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 (and hopefully verify them).

Our design is based on our prior work [Leslie et al., 2005], which established that, using a zero-copy transport layer with shared ring buffers 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.

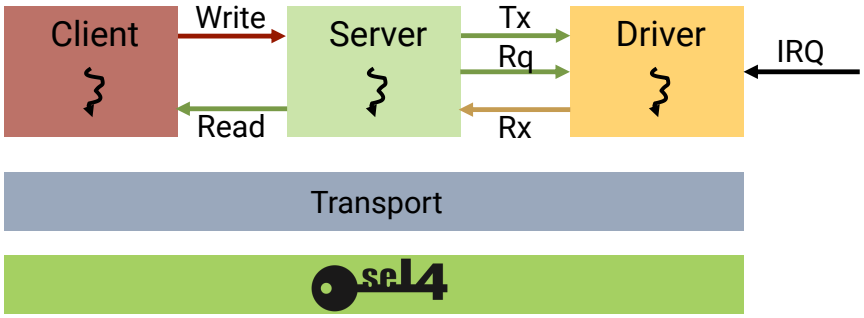

Figure 2: Structure of I/O systems on seL4.

Figure 2 shows the logical structure of I/O subsystems on seL4. Each box represents a user-level process that is encapsulated by seL4 and only able to communicate via defined interfaces. There are typically three components (although the “Server” component may in practice be broken down further):

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

I/O server (S): Implements a higher-level OS abstraction (socket, file, ...) that makes I/O data accessible to clients, or may just be a simple multiplexer that distributes input data to different clients and combines output streams from clients;

Client (C): Is an application (web server, user program) that uses the abstraction provided by the server to perform I/O, or has the abstraction implemented in a library and interfaces to the multiplexer.

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 server-driver interface traditionally uses somewhat different terminology for different device classes (e.g. network vs storage). Here we use the terminology used for network devices, referring to the output operation as *transmit* (Tx) and the input operation as *receive*

(Rx). Besides the Tx and Rx channels, there is also a *request* (Rq) channel, which lets the server request specific data from the device.

Each of the above processes (i) consists of an seL4 *address space* (VSpace) with access rights defined by its 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 $C_i \leq T_i$.

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

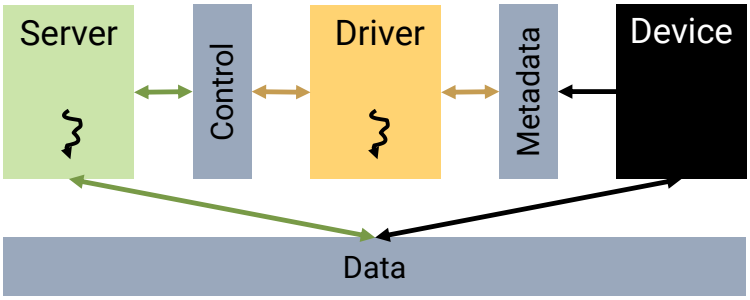

Figure 3: Memory regions shared between server, driver and device.

Focusing for now on the driver interfaces, there are three memory regions, as shown in Figure 3:

1. a ***control region*** shared between driver and server,
2. a ***metadata region*** shared between device and driver (DMA),
3. a ***data region*** region shared between device and server (DMA).

Note that the driver never needs to access the actual data transferred between the server and device, so the data region need not be mapped into the driver's address space, reducing the trust required in the driver. An implication of this model is that cache management (flushing, invalidating) on the data region is the server's responsibility, not the driver's.

This is presently a simplified model that works for network devices, but is insufficient for zero-copy I/O to high-performance block devices (such as NVMe or persistent RAM). Support for these device classes requires the server to dynamically set up mappings of DMA buffers. This will be added in the future.

In the present model, the *control region* is further split into a Tx and a Rx path, resulting in two logically separate control sub-regions (although server and driver require R/W access to both, so there is no difference in protection state between the sub-regions).

The data region can be considered as an array of I/O buffers, where each buffer can be in one of the following states:

1. **source-owned and idle:** the buffer is referenced by the *control region*, ready to be collected by the source, it contains no useful data;
2. **source-owned and in use:** the buffer is not referenced by the *control region* (but referenced by source-internal data structures), it is waiting for or is in the process of being filled with data;
3. **destination-owned and idle:** the buffer is referenced by the *control region*, ready to be collected by the destination, it contains valid data;
4. **destination-owned and in use:** the buffer is not referenced by the *control region* (but referenced by destination-internal data structures), it may be waiting for or in the process of being consumed by the destination.

Here "source" stands for the component producing data: for output that would be the server for the server-driver control region, or the client for the client-server control region, and the opposite for the "destination". *Mutatis mutandis* for input.

Note that while the driver holds pointers to buffers in the data region, it does not actually have the rights to access those buffers; all it can do is pass the pointers between the server and the device.

Within the driver framework, a particular DMA buffer in the data region is only ever used for either output (i.e. filled by the client, potentially modified by the server, and consumed by the device, and then handed back as empty to the client) or input (opposite data flow), it never changes direction. While the client may change a buffer's direction (eg. receiving a packet from the network, changing some of its content and sending it back to the network), this happens outside the framework.

4.2.2 Control region

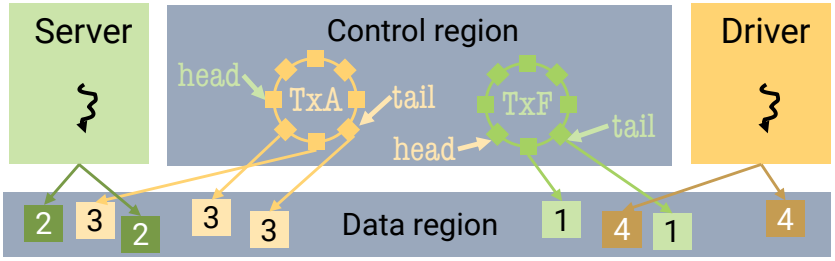

Figure 4: Server-driver transport layer for output, showing the *control* and *data regions*. The numbers in the buffers in the data region indicate the buffer state.

Figure 4 shows the structure of the transport layer, which uses *single-producer, single-consumer, lockless bounded queues* implemented as ring buffers. For simplicity, the figure only shows the *control region* of the server-to-driver output channel, i.e. the Tx control path of Figure 2, and the output data region. The other control paths have their own *control regions* and share the *data region*.

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

For a device capable of both input and output, there are corresponding structures for the Rx channel, *receive available* (`RxA`) of valid input data to be consumed, and *receive free* (`RxF` of free buffers ready to be re-used). Furthermore, there are corresponding interfaces between server and client.

```
#define RING_SIZE (1<<RING_LOG_SIZE)
struct buffer_descr {
    void   *address;
    size_t length;
}
struct ring_buffer {
  uint32_t head;
  uint32_t tail;
  struct buffer_descr buffer[RING_SIZE];
}
```

Listing 1: Control region ring-buffer data structures.

Listing 1 shows the ring-buffer data structures. Each ring buffer has a separate `head` and `tail` pointer, ring-buffer entries between those pointers are valid in that they point to a buffer in the respective data area.

Each ring buffer has exactly one *producer* (here: the server) and one consumer (here: the driver). Only the producer updates the head, and only the consumer updates the tail.

The producer hands ownership of a DMA buffer to the consumer by inserting the buffer into the *available* ring, from where the consumer collects it. When done with a buffer, the consumer hands it back to the producer by inserting it into the *free* ring, from where the producer can collect it.

Specifically, as shown in Listing 2, the producer enqueues a data buffer at the head by checking there is at least one available entry, inserting the new buffer there, and incrementing the head. Just before updating the `head` pointer, the producer issues a *memory write barrier* to ensure that no writes are re-orded by the compiler or the processor across this point. Dequeueing data buffers from the tail is analogous, as per Listing 2.

Lock-free updates to these data structures are possible by utilising the processor's property that reads and writes of small integers are atomic. The obvious data race between consumer and producer is benign, thanks to the code ensuring there is at least one unused buffer between head and tail. The memory barrier is sufficient to ensure consistency.

4.2.3 Metadata region

The *metadata region* plays the same role in the driver-device interface as the *control region* does in the server-driver interface.

A core difference between the two regions is that the *control region* is a standard shared-memory region (shared between two processes running on the CPU), while the *metadata region* is a DMA region (shared between the driver process on the CPU and the device hardware). The main practical difference is that cache management differs, as DMA bypasses the cache.

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

Listing 2: Control-region ring-buffer management.

```
#define HW_RING_SIZE (1<<HW_RING_LOG_SIZE)
struct HW_buf_descr {
    void      *address;
    size_t    length;
    status_t  valid;
}
struct HW_ring {
  uint32_t head;
  uint32_t tail;
  struct HW_buf_descr buffer[HW_RING_SIZE];
}
```

Listing 3: Network device ring-buffer data structures.

As it interfaces with the device hardware, the *metadata region*'s data structures are defined by the device interface protocol (see Section 3.2). However, for network devices they generally use similar ring-buffer structures as the *control region*, although there is one a single ring each for transmitting and receiving; we'll call those `HW_Tx` and `HW_Rx`, respectively. Listing 3 shows a representative example, which we'll assume for the following discussion, noting that details may differ between devices.

Instead of separate buffer-available and buffer-free rings, HW uses the `status` field of each ring-buffer entry. On output the device only processes entries marked as `ready`, and, once processed, sets the `status` to `free` or an `error` condition. In 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 ring buffer; this is provided to the device via a control register. The `head` and `tail` pointers are pure software constructs.

```
1  bool HW_full (struct HW_ring *ring) {
2    return (ring->head - ring->tail + 1) % HW_RING_SIZE) == 0;
3  }
4  HW_enqueue (struct HW_buf_descr *buf, struct HW_ring *ring, status_t status) {
5    assert (!HW_full(ring));
6    ring->buffer[ring->head % HW_RING_SIZE] = buf;
7    barrier();
8    buf->status = status;
9    ring->head++
10 }
11 struct HW_buf_descr *HW_dequeue (struct HW_ring *ring, status_t status) {
12   buf = ring->buffer[ring->tail % HW_RING_SIZE];
13   if (buf->status != status) {
14     return NULL;
15   }
16   barrier();
17   ring->tail++;
18   return buf;
19 }
```

Listing 4: Device ring-buffer management (simplified).

Listing 4 shows how software manages those ring buffers. For simplicity, this pseudocode assumes that the hardware processes Tx buffers in ring order.

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 server (*transmit available*, T_a);
- data requests from the server (*data requested*, R_q);
- receive-ready notifications from the server (*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 or device interface. This model has the benefits highlighted by Ryzhyk et al.: 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 server to ensure timely handling of interrupts as well as immediate response to server 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 so there is no need for flow control.

4.3.2 Server

The server, like the driver, is event-based, responding to:

- *write requests* from the client;
- *read requests* from the client;
- data-available notifications from the driver (*buffer available*, $\mathrm{B_a}$);
- write completed notifications from the driver (*buffer free*, $\mathrm{B_f}$).

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

The server is responsible for cache maintenance on the *data region* (unless it is only a multiplexer, in which case the client is responsible).

4.3.3 Client

The client-server I/O interface depends on the nature of the server, whether it is a simple multiplexer, a network stack or a file system, details will be specified later.

In general it is similarly structured, and just as asynchronous, as the server-driver interface, in order to maximize concurrency between computation and I/O. It is also a non-copying interface to minimise overheads. *This is an explicit and intended departure from the inherently inefficient Posix interface.*

There are clearly scenarios where the client is event-driven, e.g. a web server, but there are others where this may not be the appropriate model, e.g. where the client performs some complex computation to which I/O is more incidental yet latency-sensitive (e.g. checkpointing state). However, these scenarios still have in common that the client is interested in seeing the I/O completed as quickly as possible, as it either cannot continue until the I/O is completed, or continues with lower quality of service (risk of data loss). Furthermore, in cases where computation can overlap with I/O (mostly storage), the CPU usage from I/O is likely to be comparatively small.

As discussed in Section 4.3.2 and Section 4.3.3, we have a priority assignment of $P_C < P_S < P_D$.[2] This configuration is consistent with monolithic systems, where the OS generally runs at higher (effective) priority than apps.

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

4.4 Synchronisation

4.4.1 Server-driver Interface

There are two ways to implement this model: as *active* or *passive driver threads*. 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.

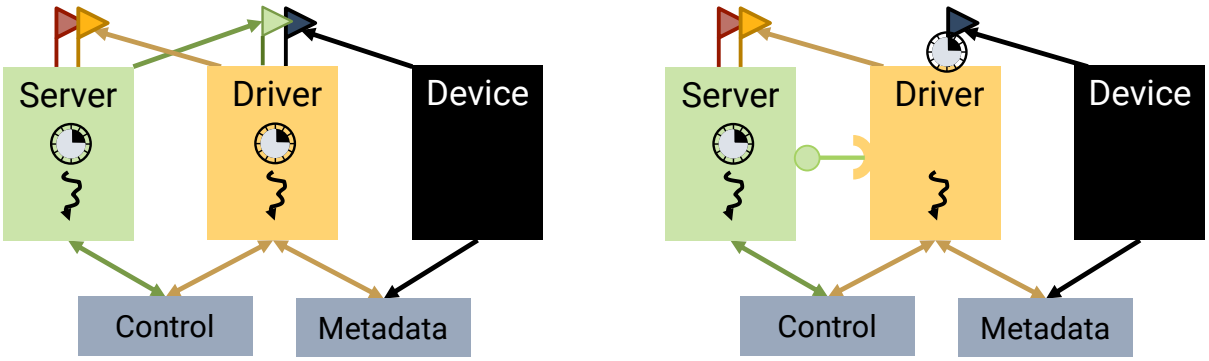

Figure 5: Active (left) vs passive (right) driver-thread model.

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 *control* and *metadata regions*).

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

The server performs its T_a, R_q and R_f operations by leaving an opcode in a specific location in the *control region* and signalling the driver's *notification*. The driver uses the badge to distinguish signals from the server and the device.

Similarly, the server has a *notification* for which the driver has a badged capability (and the server's client may hold a differently badged capability to the same *notification*). The driver uses this *notification*, together with an opcode stored in control memory to perform the B_a and B_f operations.

Note that, as the driver runs at higher priority than the server, the signals from the server to the driver are effectively synchronous, resulting in an immediate context switch to the driver (unless the driver is out of budget).

Listing 5 shows simplified pseudocode for an Ethernet driver. A more optimised version will, after processing notifications from the server, also check for pending IRQs from the device (which may have arrived while the driver was running on behalf of the server) and process these. Similar, in a multicore system, server notifications may have arrived while processing IRQs and should be checked prior to returning.

Figure 6 shows the mode switches for this model compared to Linux. The diagram shows a total of 10 kernel entries for a complete packet round trip. However, the not-full signal is only used when the transmit acknowledgement creates a free slot in a previously full ring buffer and is therefore not used under low load. (Similarly, the not-empty signal is not used if there

```
handle_irq() {
  while (event = clear_hw_events()) {
    signal = NULL;
    if (event & Tc) {
      was_empty = empty(TxF);
      while (!full(TxF) && buf = HW_dequeue(HW_Tx))
        enqueue(packet, TxF);
      if (was_empty && !empty(TxF))
        Signal(server, not_full);
    }
    if (event & Ra) {
      was_empty = empty(RxA);
      while (!full(RxA) && buf = HW_dequeue(HW_Rx))
        enqueue(buf, RxA); /* process input */
      if (was_empty && !empty(RxA))
        Signal(server, not_empty);
      while (!full(HW_Rx) && buf=dequeue(RxF))
        HW_enqueue(buf, HW_Rx);  /* return free Rx buffers */
    }
    if (event & error)
      fail;
  }
}

main()
  initialise();
  notif=init_done;
  while(true) {
    event = Signal_and_Wait(notif);
    notif = NULL;
    if (event & IRQ) {
      handle_irq();
      notif = ack;
      continue;
    }
    if (event & Ta) {
      was_full = full(TxA);
      while (!full(HW_Tx) && buf=dequeue(TxA))
        HW_enqueue(buf, HW_Tx);
      if (was_full && ! full(TxA))
        notif = server;
    }
    if (event & Rf)
      while (!full(HW_Rx) && buf=dequeue(RxF))
        HW_enqueue(buf, HW_Rx);  /* return free Rx buffers */
  }
}
```

Listing 5: Ethernet driver pseudocode (active-thread model).

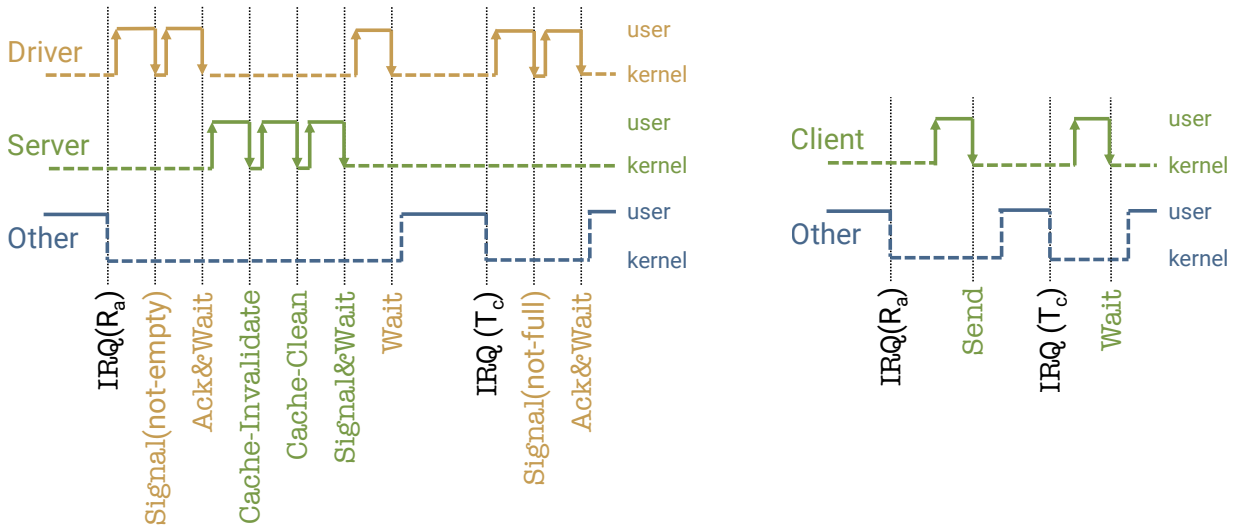

Figure 6: Mode switches for the active driver-thread model (left) compared to Linux (right) for a packet round trip. For horizontal lines, solid indicates execution, dashed indicates suspension (blocked or preempted). For vertical lines, arrows indicate synchronous mode switches (system calls and returns) while dashed lines indicate asynchronous switches (interrupts or scheduling). “Other” refers to a lower-priority background activity.

are already unprocessed packets in the receive ring.) Under low load the full packet round trip therefore requires 9 kernel entries.

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 a context switch to the driver and back, doubling the number of system calls. Furthermore, the usermode seL4 driver requires system calls to acknowledge IRQs. In addition, the cache management done by the server 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 coalescing (deferring the T_c interrupt for some amount of time or until a certain number of transmits have been completed) will in average reduce the number of kernel entries per round-trip by two for the seL4 system and by one for Linux.

Flow control is achieved by three means:

1. limiting the size of the receive ring buffer in the *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 ring buffer in the *control region* (which rate-limits the server’s output requests) – this is unlikely to achieve much, given that the driver runs at higher priority than the server;
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

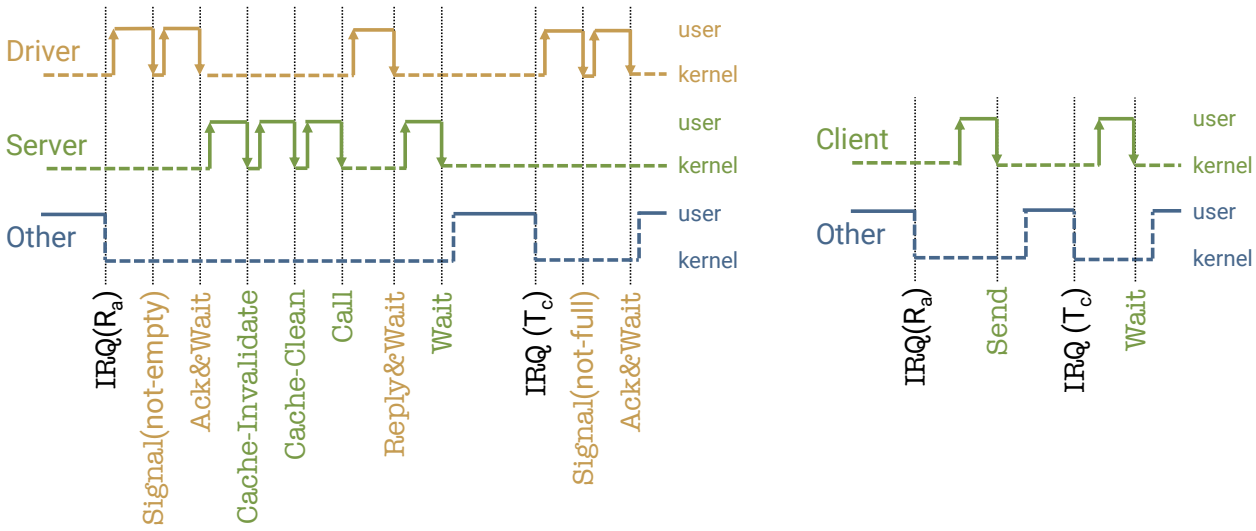

Figure 7: Mode switches for the passive driver-thread model (left) compared to Linux (right) for a packet round trip. Explanation as for Figure 6.

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 server 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. Similarly, more client requests may arrive during driver execution, if the driver's budget was exhausted during the processing of an event.

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 server, making the client→server invocation synchronous, this naturally leads to the server-side interface being an *endpoint*, which is invoked by the server as a protected procedure call, passing the server'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 B_a and B_f 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.

Figure 7 shows the mode switches for the this model. Compared to the active-thread model, there is one more mode switch (total of 10 under low load), as the driver needs to return control to the server after acting on the server's behalf to put a packet on the wire. This is traded off against the server's `call()` to the passive driver thread (without switching scheduling contexts) being inherently cheaper than the server's `signal()` to the higher-priority active driver thread, which requires a change in scheduling context.

Flow control is achieved by three means:

1. limiting the size of the receive ring buffer in the *metadata region*;
2. limiting the size of the send ring buffer in the *control region* (same caveat applies as for the active driver thread);
3. limiting the budget of the driver's *notification* (which implicitly limites the interrupt rate without limiting the driver's ability to respond to server 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 client requests execute on the client's budget. But note that, even if serving a server request, the driver needs to poll for IRQs before returning to the client, so it may handle interrupts on the server's SC.

However, the server 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 server while the driver is waiting for a budget replenishment reduces the server's ability to operate concurrently.

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

Multi-threaded driver? It should be possible to split the driver into two separate threads, one interfacing to the server (i.e. handling T_a, R_q and R_f events) and one interfacing to the device (handling IRQs). This would allow serving both types of requests concurrently by running the two threads on different cores (and *only* makes sense for this case).

The idea of a dual-threaded 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, as the two threads do mostly not access the same data structures, and where they do, there is already concurrency between software and device. Specifically, only the server-side thread accesses the `TxA` ring of Figure 4, while only the device-side thread accesses the `TxF` and `RxA` rings.

The only competing access to a server-side ring is for returning free buffers from `RxF` to the hardware ring. However, we note that it is sufficient to do this in the server-side code (Line 44 of Listing 5) – the identical code in the IRQ handler (Line 17 of Listing 5) is a performance optimisation, not a functional requirement. The performance benefit of this optimisation in the multithreaded case is not clear without detailled evaluation, but we note that, if needed, the loop at Line 17 could be replaced by

```
if (!full(HW_Rx) && !empty(RxF))
  Signal(driver_s);
```

where `driver_s` is the Notification of the server-side driver thread. The race here is benign.

Similar arguments apply to the input control data structures.

The model also rules out further optimisations for reducing the number of kernel entries: The driver may check for pending interrupts after processing server requests, or check for pending server requests after processing interrupts (such new server requests cannot happen on a single-core configuration). However, this is unlikely to be an issue where I/O load is high enough to justify the use of multiple cores for the driver.

4.4.2 Client-server interface

Similar considerations apply to the server, which could be an active or passive thread. As it executes at a lower priority than the driver, its driver-side interface must be a *notification*. As it executes at a higher priority than the client, the client-side interface could be either a *notification* or an *endpoint*, with the same considerations applying as for the server-side driver interface.

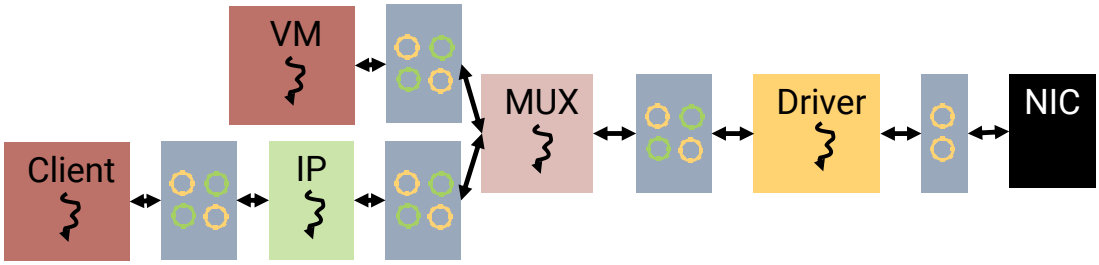

Figure 8: Modularised design of network-device sharing architecture.

If the server is really a multiplexer, it would use the same interface on the driver as on the client side, interposing transparently. Figure 8 show an example with a multiplexer sharing a NIC between a native client with its private IP stack and a virtual machine, with all interfaces using similar lock-free, bounded, single-consumer, single-producer data structures.

The control data structures for Tx and Rx flows are completely separated. Consequently, the server could consist of two separate threads (in fact, address spaces), which increases modularity but also allows exploiting more parallelism, by allocating the threads to different cores. Such a design may be particularly attractive for servers that are multiplexers. They may apply different policies for the Tx and Rx flows (e.g. copying Tx data to reduce trust in the driver).

4.4.3 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, and they wait for timer signals.

4.4.4 Discussion

There are a number of open questions in this design.

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 server 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 budged replenishment, it blocks the server, 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 server's core. Which model is better likely depends on the application scenario.

C1.6: The server may want to use a watchdog to prevent indefinitely blocking on an untrusted driver. As long as the server only synchronises with *notification*s, the watchdog timeout can be signalled to the server's *notification*, terminating the wait. If the server invokes the driver by IPC, then the watchdog would have to explicitly reset the server to abort the IPC, a somewhat more complicated operation.[3]

Similar questions arise for the server thread, which may also be active or passive.

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

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 server?

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

C2.3: A file server, if it provides a high-level abstraction, may require more heavyweight processing and thus a larger budget and possibly period. In contrast, a server that is just a multiplexer should be able to run with a small budget.

Research Question 3.: *Under which circumstances does the two-thread driver model make sense?* Considerations:

C3.1: There is probably no point using two threads on the same core, as this has overheads and risks one thread starving the other. However, it may allow handling higher loads if the two threads run on separate cores.

C3.2: How does the performance compare to re-entrant and concurrent Linux drivers?

C3.3: Does the two-thread design lessen or worsen driver complexity or verifiability? Each individual thread should be somewhat simpler, but there is now more concurrency on accessing the hardware ring buffers, increasing the risk of misplaced or omitted barriers, ir incorrect ordering of updates.

Research Question 4.: *How does the two-thread design decision interact with active vs passive treads?* Considerations:

[3] 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.

C4.1: There is no inherent reason why both threads have to be of the same kind.

C4.2: For the device-side thread, the difference is mostly moot: it comes down whether to the SC is associated with the (active) thread or the *notification*, with no conceptual difference. In either case, the thread is pinned on a particular core. Rate-limiting is likely more straightforward if the budget is associated with the *notification*, resulting in a passive thread.

C4.3: For the server-side thread, the active configuration would execute on the server's core vs. being pinned on its own. (This probably makes little difference if the sever is just a multiplexer). As the thread is exclusively operating on behalf of the server, and an out-of-budget situation is unable to impact the device-side thread, making this thread passive seems the obvious choice, that also benefits from the more efficient SC-switch-free invocation.

We plan an extensive evaluation program over the next 12 months to settle the above questions.

5 Security Analysis

An implication of the threat model of Section 2 is that any component of the system must be verifiable. Another implication is that each component must assume that any other component it interfaces with is untrusted: the driver must not trust the server and vice versa, the client must not trust the server and vice versa.

5.1 Verifiability

No verification work has been performed yet on the sDDF and its components, but the design aims to minimise complexity, and leverage modularity. Here we explore what needs to be verified (eventually) to satisfy the requirements of the threat model.

5.1.1 Drivers and servers

The driver model is designed to simplify the driver dramatically (compared to e.g., Linux drivers). Drivers are simple state machines free from internal concurrency (but do access/modify data structures that are concurrently accesses/modifies by hardware). This seems to maximize the chances of verifying them, although verification requires not only formalising the driver framework but also the device interfaces.

Similar arguments apply to servers: they are also kept simple. Especially in the case of networking, the server would most likely not be a complete protocol stack, but just a simple multiplexer that interfaces multiple clients (each with their own IP stack library) to a single driver.

Trusted drivers need to interface to untrusted servers and vice versa, and servers also need to interface to untrusted clients. As these interfaces employ shared memory, each value accessed in the shared memory must be sanitised; especially all pointers in shared memory must be range-checked before de-referencing. Short of proving full functional correctness, sanitising may be verifiable using fully automated means, such as model checking.

If blocking on a response (Notification or IPC reply) from an untrusted component, a trusted component must employ a watchdog timer to prevent indefinite blocking.

Support for untrusted drivers and servers also means that the IOMMU must be programmed to limit DMA to valid *data* and *metadata regions*.

5.1.2 Driver framework

Simplifying the drivers by removing any internal concurrency potentially shifts some of the complexity into the driver framework itself, including concurrency. However, the advantage is that this complexity is encapsulated in a relatively small code base, and the verification needs to be done only once per device class (as opposed for each individual driver).

The sanitation requirements on drivers and servers apply to the framework implementation as well.

6 Implementation Status and Performance

6.1 Implementation status

The sDDF is implemented as described above, providing high-performance networking but not supporting any other device classes yet. Active as well as passive driver threads are implemented and tested, but the passive model is not yet heavily exercised.

The code is not yet officially released, but is accessible on github:
https://github.com/lucypa/sDDF.

6.2 Performance evaluation setup

We evaluate a simplified two-component system, where the server and client are merged into a single component, and only the driver is separate.

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 PHY. 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 simply echos the packets back. 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. Comparisons with Linux use a stock Debian Bullseye system on the same hardware.

6.3 Initial performance results

6.3.1 CAmkES

Figure 9 shows CAmkES [Kuz et al., 2007] network performance based on the new transport layer compared to the previous driver framework. Both configurations handle the applied load until they max out the CPU, and performance collapses at higher applied loads. The

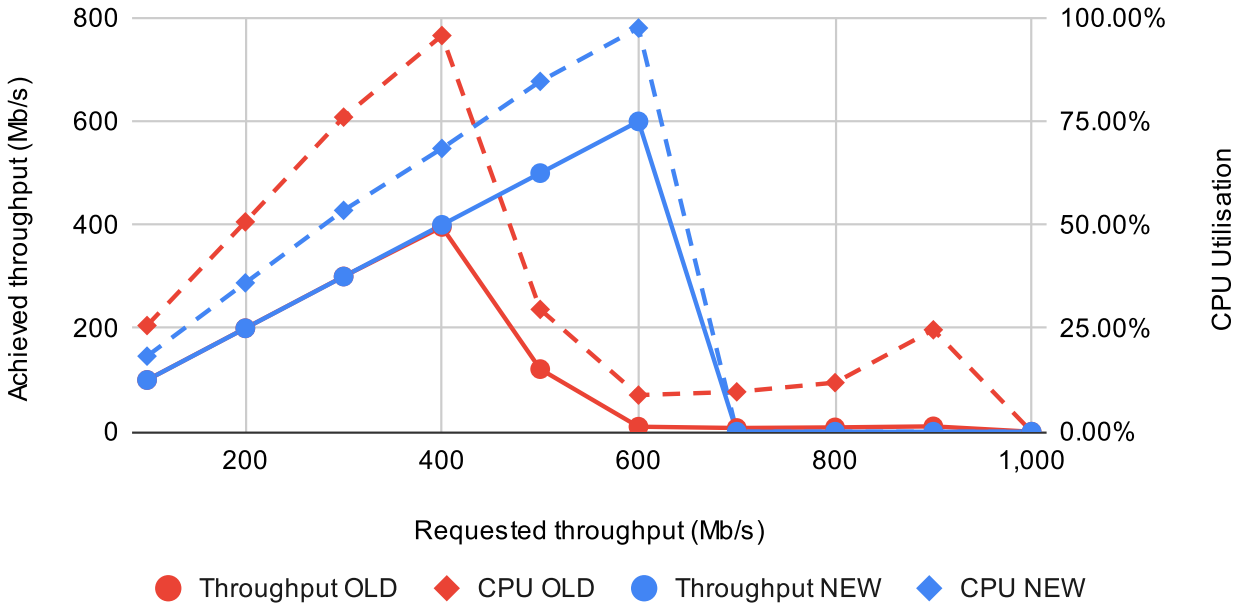

Figure 9: CAmkES networking performance, compared, comparing the new against the old transport layer.

performance collapse is a result of poor flow control, which we do not, investigate further at this time.

With the old transport, the CAmkES system saturates the CPU at 400 Mb/s, while with the new transport it reaches 600 Mb's, a reflection of the new transport reducing overheads by 1/3. This is an significant improvement, a result of the zero-copy transport which significantly reduces the number of instructions executed per packet.

6.3.2 seL4CP

Figure 10 compares CAmkES (same data as Figure 9) with seL4CP performance (both using the new transport). The seL4CP-based system uses an active driver thread with a period of 300 μs and a budget of 160 μs. It also uses a new fastpath for the `signal` system call when notifying a lower-priority thread.[4]

Unlike CAmkES, the seL4CP keeps up with the applied load though almost the full range, topping out at 958 Mb/s and 93% CPU. This is a direct result of the seL4CP configuration reducing per-packet cost by 25–31% compared to CAmkES, a reflection of the much leaner implementation of seL4CP. In total, per-packet cost with seL4CP is halved compared to CAmkES with the old transport.

Interestingly, all systems outperform Linux, which can only handle a maximum load of 300 Mb/s, where it maxes out the CPU (but avoids a performance collapse). This is despite

[4]At time of writing, the `signal` fastpath has not yet been merged into the mainline kernel, but that seems imminent, see https://github.com/seL4/seL4/pull/793.

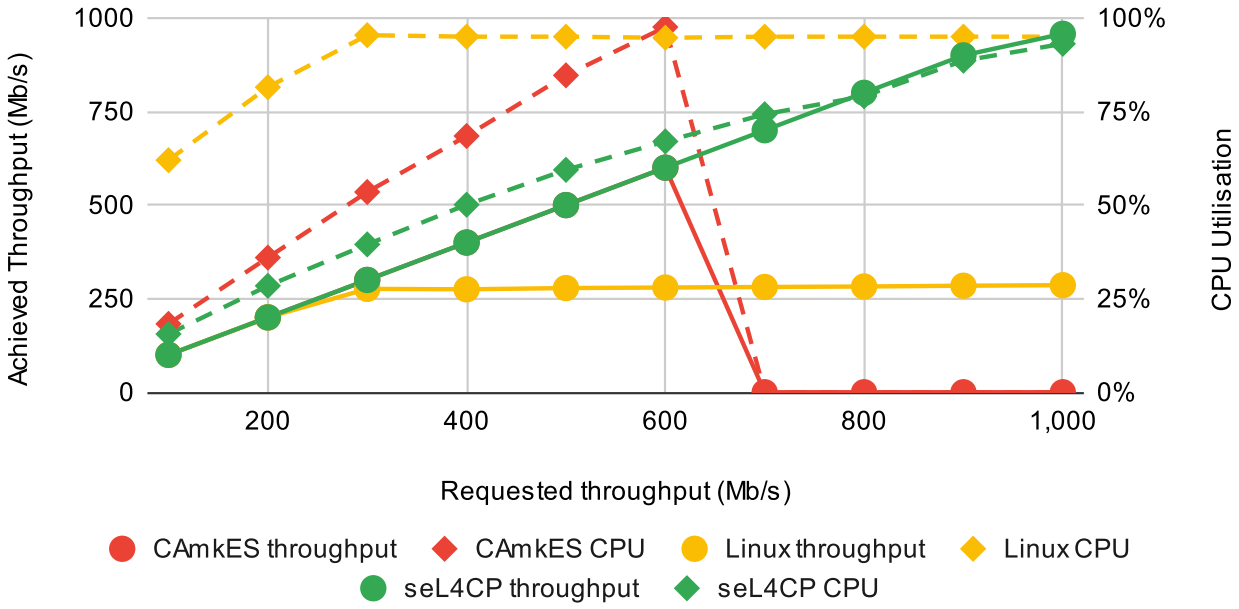

Figure 10: Performance compared to Linux, showing achieved throughpout and CPU utilisation against applied load.

our echo client performing an extra copy of the data received, to simulate Linux's copying Posix interface. That extra copy is only used for comparison reasons, the sDDFs asynchronous interface avoids copying.

6.4 Cost of modularity

To determine the performance cost of modularity (and hence the feasibility of strong separation of concerns and enabling verification) we extend the above 2-tier setup (driver plus client co-located with the IP stack) into a 3-tier one, resembling the structure of Figure 2.

However, to separate out the pure modularisation cost, we keep the client co-located with the server, and add instead a trivial forwarder between the driver and the client/stack.

Figure 11 shows the result. The extra module boundary adds about 2,000 cycles per packet, increasing CPU load by about 10%. This is still vastly better than any of the other configurations (including Linux).

6.5 Discussion

The framework clearly over-achieves our intermediate goal of "performance comparable to Linux", although further work should investigate why Linux performance is so poor.

The main observation so far is that the overall performance is good and clearly not degraded by our much simplified driver model. Furthermore, the cost of a module boundary is quite

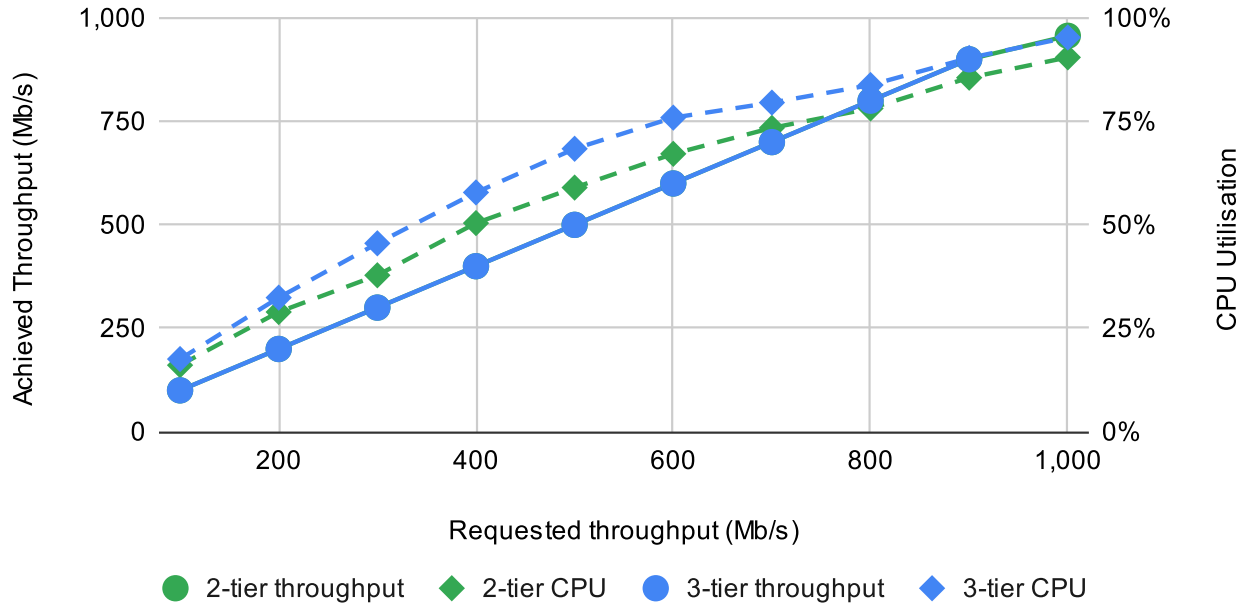

Figure 11: Performance of a two- vs a three-tier system.

moderate. This implies that a strong separation of concerns looks feasible, even if taken to the extreme of a five-tier extension of Figure 8:

1. device driver, responsible only for hardware abstraction;
2. multiplexer, responsbile only for merging/splitting data streams between multiple clients and a single device;
3. copier, responsible only for copying input data from the device-accessible shared memory region to a region accessible to the untrusted client;
4. server, responsible for implementing the OS abstraction; and
5. client.

Furthermore, there is still room for further performance improvement. Specifically, most seL4 system calls used do not yet have a fastpath, which would further shave off a few hundred cycles per system call. There may also be scope for optimising syscall functionality.

A number of other issues still need investigating:

- While we have not observed performance collapse in overload with the seL4CP, we have not yet stressed the system sufficiently be confident our flow control works well. The easiest test is to perform more work in the client to increase competition for CPU time. This includes investigating a more principled assignment of budgets and periods.
- We have not yet properly investigated the active vs passive driver-thread models, beyond having both work. The above measurements all use active threads.
- We have not yet investigated distributing the system over multiple cores. The relatively

fine-grained modularity, combined with the lock-free, asynchronous transport, lends itself to using multiple cores without requiring extra effort or even code, but we have not yet done that.

7 Conclusions

We have developed the first part of a low-overhead device driver framework for seL4, presently supporting 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 original CAmkES-based framework, halving CPU usage and clearly outperforming Linux, despite the higher number of system calls and context-switches required. Further improvements are possible with more optimised seL4 system calls and their usage.

Next steps, besides the near-term investigations discussed in Section 6.5, include:

- The performance investigations discussed in Section 6.5,
- integration with the seL4CP's virtualisation infrastructure,
- the extension to other device classes, specifically storage and USB.

References

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

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

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.

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.

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.

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.

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.