Trustworthy Systems Group
School of Computer Science & Engineering

Faculty of Engineering

The University of New South Wales

Evaluating queuing performance in the seL4 Device Driver Framework

by

Lesley Rossouw

Thesis submitted as a requirement for the degree of

Bachelor of Computer Engineering

Submitted:	2025-08-27	Student ID:	z5220299
Supervisor:	Prof. Gernot Heiser	Topic ID:	999

Abstract

The seL4 Device Driver Framework is a high-performance driver framework for the seL4 microkernel, facilitating best-in-class driver performance. It is not clear that the current design, policy set or parameters are ideal under all conditions however, owing to complex dynamic properties. In order to evaluate the suitability of the design and to enable future analysis and optimisation, this thesis presents a discrete-event simulator for the seL4 Device Driver Framework and a suite of analysis tools to accompany it. We provide an efficient, flexible and accurate simulation capable of predicting the dynamics of given system with high accuracy. The analysis tools enable exploration of every event in the system to facilitate in-depth analysis of system configurations and policy sets. We use the combined simulation and analysis suite to explore extreme cases for the sDDF to evaluate its robustness in a variety of cases. We also use the simulation to construct empirical models of the behaviour of the network driver class and to predict optimal parameters that are capable of exceeding all current performance expectations.

Acknowledgements

Thank you to everybody at Trustworthy Systems for all their work on the sDDF before me, their analysis, model checking and benchmarking was the foundation for the work in this thesis.

Thank you to Nick Sims for kindly offering to proof read my thesis, and thank you to Alex Long for kindly bringing me a coffee at a critical juncture.

Contents

List of Figures

List of Listings

Acronyms

- CPU: central processing unit
- FIFO: first-in, first-out
- IPC: inter-process communication
- NIC: network interface card
- PD: protection domain
- PPC: protected procedure call
- RX: receive
- sDDF: seL4 Device Driver Framework
- TCB: trusted computing base
- TLB: translation lookaside buffer
- TX: transmit

Chapter 1

Introduction

The seL4 Device Driver Framework (sDDF) [Heiser et al., 2024] is a high-performance driver model for seL4-based systems, such as LionsOS and Djwarula. In controlled benchmark environments, the sDDF outperforms Linux and other microkernels, but it is not clear that the expected performance or efficiency will be maintained in all cases.

Cases with unusual traffic distributions, large numbers of clients and malicious clients are unexplored in general for the sDDF. sDDF systems can be parameterised in a number of ways and it is currently unclear how parameters should be set to achieve stable performance for a given set of clients.

This thesis seeks to shed light on these uncertain situations and generate an empirical analysis of the performance characteristics of the sDDF using a bespoke discrete event simulator, pysddfsim.

1.1 Driver data flow performance

In an operating system, the flow of data between the kernel, drivers, hardware and user programs is an important design consideration. In microkernel systems like the seL4 Microkit, these principles still apply albeit with everything in user space.

Drivers and their supporting infrastructure are particularly relevant since they provide resources used by the whole system and are a potential bottleneck for all user applications dependent on their associated hardware [Yu et al., 2014]. Operating systems such as Linux provide a device model which is geared towards building data paths which are isolated from other system resources. Linux also provides various interfaces to communicate with user-space applications [Linux Kernel Development Community, 2024].

In a microkernel system, there is typically no kernel-supplied abstraction for driver data flow [Song and Kim, 2023]. In static systems such as those built with the seL4 Microkit, these design decisions must be made explicitly by the system designer using a framework such as the sDDF. As a result, the design of the system must be balanced carefully to provide competitive performance [Parker, 2024], but this can only be done precisely if the behaviour of the system can be anticipated and tuned correspondingly. Performing such analysis empirically is very expensive and can greatly lengthen development cycles.

When optimally configured, sDDF drivers built with the seL4 Microkit can exceed the performance of monolithic kernels such as Linux [Parker, 2024].

1.2 Why is the sDDF hard to analyse?

For the purposes of data flow performance, sDDF systems can be thought of as a network of programs such that neighbours are connected to each other. Each program serves a distinct purpose in the driver stack and passes only the data that is required to neighbouring programs.

This network is interconnected with queues containing a certain number of slots for data. Each program in the network can signal other programs to indicate that data is ready or required over the queue. The queuing structures are trivial to analyse in themselves; they are simple first-in-first-out buffers with fixed capacity. Service times effectively decrease with throughput as more data can be processed per process wakeup and there is a fixed delay to switch processes. In Kendall's notation [Kendall, 1953], such a queue can be described as ($Q = M/G(\lambda)/1/K/K/FIFO$); that is, a first-in-first-out queue with random arrivals, a service time which is a function of the arrival rate, K entries and K buffers. The relationship between individual nodes is easy to understand. Nodes can share data with their neighbours over the queue and signal them to indicate data is ready.

Data does not flow through the network continuously. Programs can only put data into the queues or remove data from the queues when they are selected to run by the seL4 scheduler. Data only moves when one node has been scheduled to supply data and another node has been scheduled to accept it . The system rapidly evolves in complexity once scheduling is considered as processes are assigned priorities to govern their order of execution and signals between processes will change the run order.

Signals are implemented by the seL4 Microkit as an inter-process communication (IPC) method. Whenever a process is signalled, it will be scheduled to run as soon as possible [Heiser et al., 2022]. In the event that a lower-priority process signals a higher-priority process, it will be immediately pre-empted. The consequence of this is that the order of execution has a severe data dependency, and it is very difficult to anticipate fine-grained behaviour.

This is significant because it introduces instability into the system, except where the design is balanced to prevent it. Drivers trigger IPC as a result of data flow, but now data flow triggers IPC, triggering further data flow. Change to the graph has a cascading effect on the total data flow and order of execution of the whole system!

Multiprocessor systems add further challenges. Inter-core notifications add an effectively random source of IPC from the perspective of the recipient node in addition to the typical challenges of intra-core communication.

This means that it is practically impossible to perform transient analysis of the sDDF or any other seL4 Microkit system due to the runtime dynamism of all systems of non-trivial scale. It follows that it is similarly challenging to design a stable system or even judge if a given system is stable. These characteristics make the sDDF into a ”non-classical” or ”non-stationary” queuing network.

1.3 Analysis of non-classical queuing networks

Analysing non-classical queuing networks generally falls into one of two categories: simulation-based methods [Carnevali et al., 2022] or analytical methods to approximate the classical behaviour of the non-classical system [Hirayama, 2009]. Simulation carries numerous benefits for dynamic analysis since it allows the system to be excited with an arbitrary load and analysed continuously to observe patterns of instability that may occur before the system reaches a steady state.

Useful tools for stochastic analysis via simulation include stochastic-time Petri nets [Glynn and Haas, 2012], a mathematical modelling tool for dynamic systems. These are a useful element for describing simulation elements in isolation.

These approaches will be explored in conjunction with an overview of classical queuing theory in Chapter 3.

1.4 Problem statement

To effectively analyse the sDDF, we must be able to judge whether a particular system is stable and how different parameters contribute to its dynamic properties. With such analysis available, systems can be evaluated and an empirical model can be derived to reason about general performance of sDDF systems.

This thesis presents a simulation-based approach to analyse arbitrary seL4 Microkit systems with a particular focus on the sDDF. We present an efficient, extensible and flexible Python-based simulation engine which is capable of generating fine-grained execution traces in Chapter. We present a suite of analysis tools which can parse execution traces and derive critical performance details about the system down to the scale of tens of clock cycles. Finally, we perform an in-depth analysis of the sDDF to evaluate its performance under extreme conditions, using simulation results supported by real-world benchmarks.

The principal goals of this thesis are to answer the following research questions:

1. Is the core design of the sDDF able to maintain high performance in extreme cases such as malicious clients, large numbers of clients or unusual traffic patterns?
2. How can we analyse any given sDDF system to draw conclusions about its stability and the validity of chosen parameters?
3. What improvements can be made to the sDDF to further improve its trustworthiness and performance?
4. Can a tool be devised for in-depth analysis of future sDDF protocols and device classes such that any possible known issues can be avoided?

Chapter 2 will introduce the seL4 Microkit, seL4 Device Driver Framework and seL4 itself in addition to a variety of important formal theory of queuing networks. The simulation tool, pysddfsim, will be introduced in Chapter 4. We will model the sDDF network device class in Chapter 5. Finally, Chapter 6 will present a variety of experiments and results to understand the limits of the sDDF.

Chapter 2

Background

Performance analysis of a scheduler-driven, feedback-free, multi-processing queuing network is an unusual problem and little work has been done to address it. We approach it by utilising concepts from the disciplines of queuing theory, statistics, discrete event simulation and managerial science for analogous dynamic systems.

2.1 Queues

As a system composed of entities connected with queues, a natural starting point to conceptualise the sDDF is from the perspective of queuing theory.

A queue is a structure for administering a service repeatedly to a number of *customers*. Customers enter the queue and await their turn to be serviced by a *server*. There are some number of servers c who can work at the same time, each serving one customer and taking some amount of time to service them defined by the *service time distribution* S. Customers who are not being served will wait in the queue until they are selected to be serviced according to the *queuing discipline* D, assumed to be "first-in, first out" (FIFO) unless otherwise stated. Customers will arrive and enter the queue according to the *arrival time distribution* A. Queues may have a *capacity* K - if the number of customers awaiting service is equal to the capacity, new arrivals are discarded and cannot enter the queue. Optionally queues may have a population size N, the number of entities that are available to enter the queue [Kendall, 1953].

A given queue can be described in term of Kendall's notation as $Q = A/S/c/K/N/D$, commonly shortened to $Q = A/S/c$, highlighting the arrival process, service process and number of servers as the more important properties in most queues [Kendall, 1953]. In the simplified case it is assumed that $K = N = \infty$ and D is first-in, first-out. Note that A and S will be written as a shorthand name of the distribution, e.g. M for Markovian, D for degenerate, etc.

While the language used to formally define queues is specific to customer service, it's obvious how we can map these terms to other disciplines. For example, a queue of network packets awaiting intake by a network stack has a flow of customers (packets), some number of servers (worker threads handling packets) and some fixed capacity of the queue which is serviced according to a FIFO or priority-queue queuing discipline.

Sets of queues are *stochastic* systems, their behaviour is defined by some set of random processes. This is a critical property which facilitates analysis methods of traditional queuing theory - if we can identify the statistical distributions that describe a queue, we can predict its behaviour with relative accuracy by interpreting the queuing system as an *embedded Markov chain* which is parameterised by the distribution of service and arrival times [Bhat, 2015c; Kendall, 1953].

2.2 Markov Chains and the Embedded Markov chain

A Markov chain is some discrete sequence of states, each with some independent probability of a change to a subsequent state. The Markov property (”Markovian”) refers to the ”memoryless” property of such a system - that is, the current state of a Markovian system contains all information defining the possible future states of the system [Wu, 2024a].

to interpret a queue as a Markov chain, we must ensure that both the arrival and service time distributions are Markovian. This is to say, we can ”embed” the statistical model of the two defining distributions into a Markov chain to create a single equivalent model [Bhat, 2015a]. Such a Markov chain can be conceptualised as a *birth-death process*; a system where states are sequential and can only increment or decrement by one [Bhat, 2015c].

Intuitively, we can understand the chain as some list of states each representing a number of customers in the queue. There is one state for every possible level of capacity, and each state K may only transition to the state $(K + 1)$ with one more customer or the state $(K - 1)$ with one less customer. These transitions correspond to the probability of an arrival or a customer being serviced, as defined by the arrival and service time distributions. Figure 2.1 depicts a birth-death process as a Markov chain, with probabilities of incrementation from state n as λ_n or decrementation from state n as μ_n.

2.3 Non-stationary queues

Non-stationary or non-classical queues are queues with behaviour that cannot be adequately analysed by means of classical queuing theory or possibly with behaviour that is not describable in terms of Kendall's notation; that is, their distributions of arrival times or service times are non-Markovian [Miller, 1959; Rolski, 1989]. Transient analysis

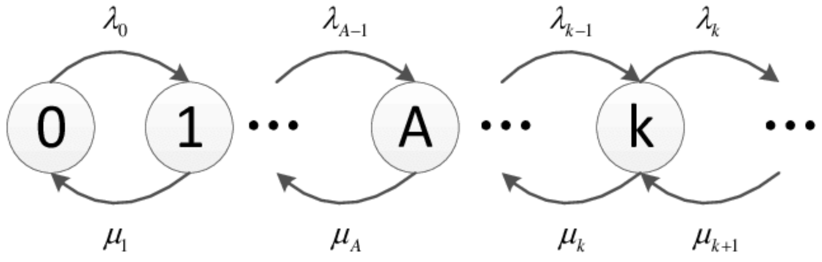


Figure 2.1: A Markov chain representing a birth-death process [Fu et al., 2015]

and reinterpretation of non-stationary queues as stationary ones is possible in limited cases, but this is not true in general [Daw et al., 2022; Konrad and Liu, 2024].

A non-stationary queue can be understood as a birth-death process Markov chain wherein the probabilities of the queue size increasing or decreasing are not static somehow - i.e. the arrival and/or distribution times are no longer memoryless (non-Markovian).

2.4 Queuing networks

A queuing network is an arrangement of several queues such that when a customer exits a queue, it will soon enter another queue or leave the network. queuing networks are referred to as *closed* if customers may not enter or leave the system, or *open* otherwise. If an open network has an identical total arrival and departure rate, it can be approximated as closed since the number of customers doesn't change. Analysing queuing networks is complex, but similarly to individual Markovian queues it is practical to statistically analyse a network of Markovian queues statistically [Bhat, 2015b].

Non-stationary queuing networks cannot simply be described and generally are analysed by means of simulation [Wagner, 2017].

2.5 Simulation

Individual queues and queuing networks are both discrete systems and can be simulated to produce reasonably accurate approximations of their behaviour. This holds irrespective of if they are classical or not [?]. A stochastic simulation of behaviour by generating random variables matching the properties of random properties of that system and observing the discrete state changes over time. [Wu, 2024b].

For example: a stochastic simulation of an $M/M/1$ queue (Markovian arrival and services times, one server) would define states $[s_0, s_K]$ and define two possible events; the arrival and departure of a customer. As time passes we decide if events occur by

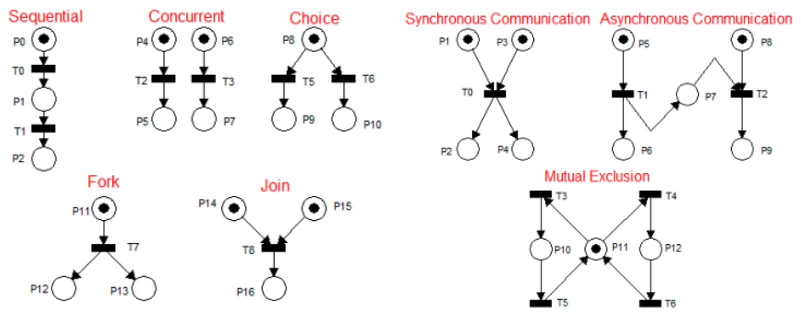


Figure 2.2: Various Petri nets describing dynamic relationships.

evaluating some random variable for every unit of time to simulate the queue servicing packets. Each random variable is derived from the associated distribution - in this case, a Markovian (exponential) random variable for both.

The exact construction of a simulation is heavily dependent on the number and behaviour of states in a system as well as the conditions upon which states may change.

2.6 Stochastic-time Petri Nets

With the challenge of complex dynamic systems, a model of elements in the system is necessary. A stochastic-time Petri Net (STPN) is one modelling formalism suitable for this purpose.

Petri nets enable descriptions of systems with complex dynamic behaviour such as concurrency, blocking and synchronisation. They are designed to encapsulate the full set of possible behaviours of the system into a single unified description which can then be reasoned about to infer every possible chain of actions [Balbo, 2001]. Figure 2.2 [Gehlot and Nigro, 2010] depicts various Petri nets showing complex relationships. A circle depicts a *place*, a bar depicts a *transition* and an arc depicts a relationship between elements.

An important metric of a STPN is its markings. A marking refers to the distribution of resources and state throughout the net. This is best understood intuitively. In the case of a queuing network, the marking would describe the distribution of customers in queues at any given time [Glynn and Haas, 2012].

A *timed Petri net* is an extension to this concept for dynamic systems with discrete events which includes a notion of time — i.e. systems such as queues which can be interpretted as continuous time Markov chains. This enables Petri nets to reflect atomicity, race conditions and more as depicted in Figure 2.2 [Gehlot and Nigro, 2010]. *Stochastic-time Petri nets* are a final extension upon this concept which can represent both state transitions which take take time and are immediate [Balbo, 2001].

For the scope of this report, Petri nets simply should be understood as a "language" to represent dynamic systems.

2.7 seL4

seL4 is a security-focused, capability-based L4 microkernel. It is formally verified for both functional correctness and enforcement of security properties [Klein et al., 2014] while also achieving exceptional performance [Parker, 2024]. As a microkernel, it has a minimal feature set and does not offer many features that are present in a typical kernel, instead delegating all work that can be done outside of the kernel to user-level programs.

A core feature of seL4 is its capability-based access control system. Capabilities (caps) are abstract unforgeable structures that reference objects with a set of fine-grained access rights [Dennis and Horn, 1966]. seL4 capabilities are communicable objects which reference a kernel object and provide the right to interact with it. Userspace programs cannot interact with any object without a capability and capabilities are the primary mechanism used by the kernel to guarantee isolation. Capabilities are used to grant access to system memory and other hardware interfaces, inter-process communication objects, the right to execution time and more.

seL4 uses a message-passing interface for inter-thread communication and interacting with the kernel [Heiser, Gernot, 2025]. Messages are sent by a process via *invoking* a capability. Capabilities are invoked by using one of the three primary system calls:

- `seL4_Send()` sends a message via a capability.
- `seL4_Recv()` blocks the thread until a message arrives via a capability.
- `seL4_Yield()` surrenders the remaining timeslice of the thread, putting it to sleep.

seL4 has a *mixed-criticality system* (MCS) kernel variant improving temporal integrity to allow seL4 to support real-time applications [seL4 Foundation, 2024b]. This thesis only considers the MCS kernel as other variants are not typically used with the sDDF.

A distinguishing feature of MCS seL4 is the MCS scheduler. On multiprocessor systems, the MCS scheduler runs separately on each processor (core) in the system, managing the processes assigned to it. Processes have two properties to define their execution characteristics: budget and period. A process can run for *budget* microseconds before it runs out of time and goes to sleep, and the budget is refilled once every *period*. The period, budget and right to execute are encapsulated in a `scheduling context`, a kernel object prescribing the right to use CPU time. The highest-priority, non-blocked thread on a given core with available budget is selected to run at any given time, and if multiple equal-priority threads are runnable they execute ”round-robin” [seL4 Foundation, 2024a].

In addition to a scheduling context, there are several other types of kernel object typically held by a thread. A `VSpace` is a kernel object representing a virtual address space, and a capability to the VSpace represents the right to use it. A `CSpace` (capability space) is a kernel object containing the capabilities a thread has access to. There are two classes of kernel object to enable inter-process communication: `Notifications` and `Endpoints`.

A `Notification` is logically an array of binary semaphores for signalling. A process sends a signal to a `Notification` by `seL4_Send`-ing a word with bits corresponding to the semaphores to notify, and a waiting process can call `seL4_Recv` to block on some set of semaphores until they are signalled. When a notification is signalled and a waiting thread is unblocked, the scheduler will attempt to make it run immediately. If a signalled thread is of higher priority than the signalling thread, it will be scheduled immediately. Otherwise, the thread will move to the front of its round-robin queue irrespective of if it will run next [Zupancic, Indan, 2022].

An `Endpoint` is a synchronous IPC mechanism, allowing messages to be passed with data attached. Unlike notifications, the sender will block upon calling `seL4_Send` on the endpoint capability. The receiver will also block upon calling `seL4_Recv` to indicate readiness to receive the message. Once both threads have blocked, a rendezvous is achieved and the message is passed and both threads are unblocked. Each endpoint capability can specify it as send-only, receive-only or bidirectional.

2.8 The seL4 Microkit

The seL4 Microkit is an operating system framework intended to simplify the construction of seL4-based systems. It enables the composition of systems with a static architecture using a set of simplified abstractions over seL4 [Heiser et al., 2022]. The Microkit runtime hides the low-level details of seL4 threads and instead offers a single abstraction for userspace programs: the *Protection Domain.*

A protection domain (PD) is an event-driven process template where events are triggered by IPC. When implementing a program as a PD, there are four entrypoint functions that may be defined:

- `init()` - initialisation logic, called at system startup.
- `notified()` - primary event loop logic. Called every time this PDs notification object is signalled. The notified entrypoint calls `seL4_Recv()` on the PDs notification every time it exits, thereby waiting for the next signal to arrive and running again when the thread is unblocked by the kernel.

- `protected()` - called whenever this PDs endpoint is invoked by a sender. We refer to an endpoint invocation as a *Protected Procedure Call* (PPC).
- `fault()` - called if a child PD of the current PD has a kernel fault. Out of scope for this thesis.

Protection domains are implemented using a library, `libmicrokit`, and this code is used as an interface to the primary Microkit tool. The Microkit tool is a program which compiles a *system description file* and PD source code to generate a single operating system image which can be booted for a target platform.

The Microkit provides several abstractions for building event-driven systems [seL4 Foundation, 2024b]. The significant structural abstractions are as follows:

- **System** - a description of a whole Microkit system for a specific platform.
- **Protection Domain (PD)** - the principle runtime abstraction for the Microkit. Analogous to a UNIX process, containing a virtual memory space and scheduling context for a single user-space program.
- **Protected Procedure call (PPC)** - a protected function call supported by an seL4 `Endpoint`. Allows transmission of up to 64 bytes of data and always executes in the context of the providing domain. PPCs may only occur from a lower-priority PD to a higher-priority PD.
- **Interrupt** - A method for hardware interrupt delivery similar to notifications and delivered to the 'notified' endpoint.
- **Channel** - A structure for connecting two PDs for communication by PPC or notifications. Used to generate `Endpoint` and `Notification` kernel objects and capabilities which are automatically mapped into PDs. Interrupts also receive a channel number.
- **Notification** - a semaphore-like IPC mechanism that wraps around seL4 `Notification` objects. Microkit notifications present only a single semaphore of the kernel notification, such that the channel number of Microkit signal is the index of the semaphore inside the seL4 `Notification`. This allows one notification object to be used to communicate with many PDs. Microkit only supports one-to-one signals, i.e. one PD cannot signal multiple others.

A Microkit system description file (SDF) will define all components present, including all user applications, drivers, system utilities, etc.

2.9 The seL4 Device Driver Framework

The seL4 Device Driver Framework (sDDF) is a driver model for seL4 systems designed to provide high-performance, secure drivers [Parker, 2024].

The sDDF is a paradigm for driver design systems for general seL4 systems, however we will only discuss the sDDF implemented with the seL4 Microkit. It prescribes a method of writing high-performance drivers and includes a library of driver layers which can be used with arbitrary seL4 Microkit systems. The sDDF driver model proposes that systems are designed minimally, with each driver consisting of a minimum three PDs:

- **Driver** - PD responsible for interfacing with device. Interrupts and hardware memory mappings from device are mapped to the driver.
- **Virtualiser** - PD responsible for moderating access to the services of the driver and for virtualising I/O addresses for DMA devices. The virtualiser provides security control and multiplexing between multiple clients while abstracting client details away from the driver.
- **Clients(s)** - some number of PDs which require access to the device managed by the driver, communicating with the virtualiser.

The collection of trusted PDs that interface with the clients are referred to as a **device class** within the sDDF. Several device classes are currently supported, notably:

- Network (ethernet),
- Serial (UART),
- Timer (system peripheral timer),
- I2C (Inter-integrated circuit protocol),
- Blk (block devices such as SD cards),
- SPI (Serial peripheral interface protocol),
- GPIO and pinmux (device IO multiplexing),
- Driver VMs (reuse of Linux drivers within virtual machines)

Several shared memory regions are mapped between each PD in the sDDF to enable communication. At a minimum, there are two regions:

- **Data region** - a large shared memory region subdivided into many smaller buffers containing data to move between the PDs.

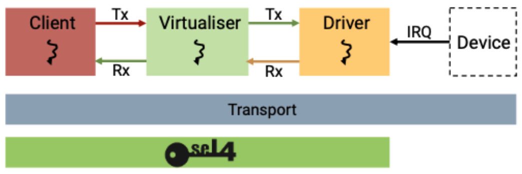


Figure 2.3: The seL4 Device Driver Framework architecture [Heiser et al., 2024]

- **Control region** - a shared memory region containing datastructures for managing the buffers of the data region. Typically, this is implemented as lockless, single-producer-single-consumer queues.

Figure 2.3 depicts this structure, with the transport box depicting the underlying shared memory regions and arrows representing control connections. This structure is principally for describing driver classes that interact with DMA however, and non-DMA device classes have conceptually similar shared regions that pass raw data instead of IO addresses.

Logically, this structure connects the PDs with shared FIFO buffers. These are $Q = M/M/1$ queues. Practically, the actual processing properties of the PDs on either side of the queue ensure it is non-Markovian however. seL4 notifications provide the primary means of synchronisation and signalling, wherein PDs will notify each other to indicate readiness to receive data or availability of data to be consumed [Parker, 2024].

For some device classes, the queues implement an additional signalling flag to prevent oversignalling. In these classes, each queue has a flag to indicate ”consumer requires signal”, and if this flag is not set the producing PD will not send a signal. This allows PDs to only allow themselves to be signalled and therefore rescheduled when required [seL4 Foundation, 2024b].

2.9.1 Network device class

The network device class in the sDDF is the most developed device class and offers best in class performance, beating Linux in synthetic benchmarks [Parker, 2024]. Although general in theory, the class currently supports wired Ethernet connections only.

The networking layer introduces additional elements from the baseline sDDF design:

- **ARP client** - optional client for handling Address Resolution Protocol (ARP) packets to the system. Only eligible for use if ARP broadcasts are the only type of broadcast packet that will be received by the system.
- **RX virtualiser** - instead of a single virtualiser, the network class implements a distinct RX copier which multiplexes packets based on their MAC address. The

RX virtualiser also performs reference counting for broadcast packets if the ARP component is not present.

- **TX virtualiser** - a virtualiser which forwards client-supplied buffers to the driver for transmission.
- **RX copier** - simple trusted program which ensures clients are isolated on the receive path by copying the contents of IO buffers to an isolated queue for the client. There is one copier per receiving client, ensuring that clients cannot interfere with one another as they consume packets. There are a finite number of IO buffers that can be used by the driver stack, and copying them to a separate queue ensures that an untrusted component cannot starve the system of buffers.

Networking is the most performance-sensitive driver class currently supported. to move multiple gigabits of data per second, there is very little room for error as the stack must be able to supply data at the maximum rate while leaving sufficient CPU time for clients to consume data. Owing to its importance and sensitivity, the network driver class will be the primary focus for this thesis.

The network device class requires that there are N RX I/O buffers (i.e. single DMA spaces for a packet) in the system and that for any queue Q_n is sized such that $\texttt{sizeof}(Q_n) \leq N$. This is an important property as it should guarantee that queues are never overfilled and prevents latency problems as the driver-virt queues are always in motion. TX buffers are also finite but they are considered to be "owned" by particular clients. The TX virt returns empty TX buffers to their associated client by using its IO address as an index. In conjunction with the TX virt being of a higher priority, transmitting clients can possibly deny service to the RX subsystem, and we explore this problem in Section 6.2.

To interface with the network driver system, each client must implement its own internet protocol (IP) stack. Consequently, there are many independent untrusted IP stacks in the system. Conceptually, the network driver subsystem works more like a virtual Ethernet switch than a traditional monolithic kernel networking subsystem, which typically include a trusted IP stack [Linux Kernel Development Community]. The untrusted IP stacks are a possible vector for attacking the entire networking subsystem as clients can send any bytes to the network, including flow control packets and broadcasts.

Although the additional components in the networking design improve trustworthiness and provide better isolation, it is not clear that optimal performance can be maintained in all cases. In particular, there is a risk of resonances developing in the data flow between clients. A resonance is a phenomenon wherein data flow will begin to oscillate as a function of certain hidden parameters of the system such as the lengths of queues or frequency of scheduler invocations. This is critically important because resonances effectively cause a constant swing between the stack being overloaded and underloaded, reducing performance and possibly increasing CPU utilisation as elements such as the virtualisers wake more often. We explore this problem in Section 6.1.

The signalling protocol is currently verified to be free of deadlock, undersignalling and oversignalling under normal operation, but it is not clear that it is able to function appropriately if a malicious client creates stressful traffic or abuses the signalling mechanisms. We investigate malicious usage of the signalling protocol in Section 6.3.1.

The networking protocol is currently known to perform well under a symmetric transmit (TX) and receive (RX) load. The receive and transmit paths are asymmetrical with respect to how they execute however, and this may cause problems. We explore alternate traffic patterns and their impact on the sDDF in Section 6.2.

2.10 Discrete Event Simulation

Discrete event simulation (DES) is a class of paradigms for system modelling predicated upon describing the system in terms of discrete events occuring over time [Gordon, 1961]. There are a variety of paradigms that may be classified as DES, however the "process-oriented" or "processing network" paradigm is of particular interest for the scope of this thesis. This paradigm is described as "(p) a narrative of entities flowing through a system" such that each arrival, exit or movement of an entity is an event [Wagner, 2017]. The occurrence of these events is based upon the sequence of previous events and possibly a stochastic function.

Within the process-oriented paradigm, the system is described as a series of processing nodes (workers) which process *work objects* (packets) via *activities* (procedures that take time to handle a work object). Processing nodes are connected via *queues*, such that nodes can pass work objects to other nodes as needed [Wagner, 2019]. All timeliness in the system is defined by activities - i.e. time advances by jumping to the next activity that should start at any given point in time. The output of the simulation is the final state of the system.

Another important variation of DES is the "event-oriented" paradigm. The event-oriented paradigm instead models the system in terms of a list of events that will happen. As events occur, they may alter the future schedule of events as state changes (e.g. in a networking system, a packet arriving will cause the driver to awaken). Time is advanced by jumping to the next most recent event in the schedule at any given moment. The output of the simulation is the sequence of events that executed over time (thereby defining the final state).

A third, less-applicable variation of DES is the "activity-oriented" paradigm. The activity-oriented paradigm functions similarly to the event-oriented paradigm fundamentally, however instead of time being advanced in terms of a pending event in an event schedule, time is always advanced in fixed timesteps. The output of the simulation is typically the final state of the system [Van Tendeloo et al., 2024].

2.11 SQLite3

SQLite3 is an embedded database implementing the SQL language in the form of a C library [SQLite project, 2025]. SQLite3 databases are simple binary blobs which can be used entirely in-memory or within a filesystem file, and has Python3 bindings for easy use from Python.

SQLite3 is used in this thesis as a data store.

2.12 SimPy

SimPy is a Python library implementing a framework for discrete event simulation [Team Simpy]. It supplies an asynchronous event dispatcher and a variety of tools to compose process-driven simulations using Python generator functions.

Simpy has a variety of features for representing resources, dataflow and more, but no features other than the event dispatcher for processes will be used for the scope of this thesis.

Chapter 3

Related work

to find a suitable approach to analyse the dynamic behaviour of the sDDF, we must start by considering previous work in analysis of non-classical queuing systems. A microkit or sDDF system is a closed queuing network of non-stationary FIFO queues but has complex dynamics throughout, preventing it from being easily classed.

Due to the unusual characteristics of the sDDF, we will explore previous work covering general analysis of complex dynamic systems with unusual properties rather than trying to isolate research to a specific problem class. As a reminder, most methods applicable to STPNs can be applied directly to a queuing network as SPTNs represent the superset of discrete dynamic systems.

3.1 ORIS: quantitative modelling of STPNs

ORIS is a toolkit for modelling and analysis of concurrent dynamic systems with non-Markovian running times [Carnevali et al., 2022]. ORIS specifically seeks to enable modelling of stochastic systems with multiple concurrent timers - e.g. a computer system with multiple scheduler instances on different cores. The authors provide an extensive GUI and toolkit which allows the specification of a system in terms of an SPTN, effectively making ORIS a general-purpose dynamic system analysis tool with SPTNs as a specification tool as opposed to a Petri net evaluator.

A system is specified to ORIS entirely in terms of an SPTN. The user must specify some set of system *transitions* which represent activities and *places* representing possible states and *tokens* representing an activation of some state. Transitions must be assigned some time to fire τ as either "immediate" or some statistical distribution.

To represent conditional execution, *enabling functions* can be applied to transitions based on their input tokens. Similarly, *update functions* can be defined to model behaviours like resetting state or modelling priority in conditionals.

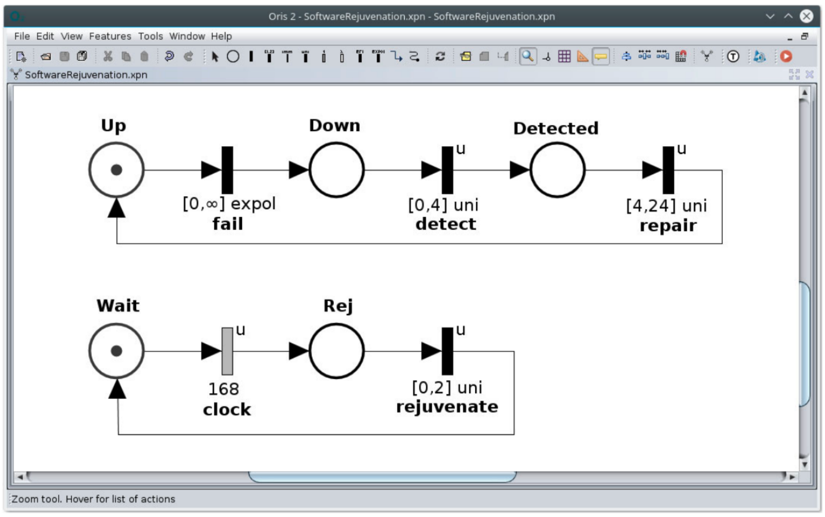

Oris 2 - SoftwareRejuvenation.xpn - SoftwareRejuvenation.xpn
File Edit View Features Tools Window Help
SoftwareRejuvenation.xpn
Up
Down
Detected
[0,∞] expol
fail
u
[0,4] uni
detect
u
[4,24] uni
repair
Wait
Rej
u
168
clock
u
[0,2] uni
rejuvenate
Zoom tool. Hover for list of actions

Figure 3.1: The ORIS graphical Petri net editor

The simulation and evaluation features of ORIS require that reward and stop conditions are defined for the network. Rewards are an expression of some list of marking values describing a system state. For example, when modelling a queuing network a reward might be defined at the state where the network has a uniform distribution of customers across all nodes. Rewards are evaluated in a continuous marking process - the reward is satisfied when the marking vector discovered at runtime matches the reward mark vector. Stop conditions are defined by transforming certain system states into a so-called *absorbing state*. In an absorbing state, the system will sojourn infinitely - i.e. it will become blocked forever.

The combination of reward and stop conditions enables evaluation of very complex states by identifying time spent in states of interest while indicating undesired state transitions.

Finally, to simulate the network and analyse it, ORIS contains several analysis engines for different classes of problems. All engines accept the same model specifications with the same reward and stop conditions, enabling reuse of models with different engines.

Unfortunately, ORIS provides an abstraction that is wholly unfriendly to the problem of analysing the sDDF. The sDDF doesn't aim to achieve uniform packet distributions or service rates, and instead seeks properties which cannot easily be described by reward conditions in ORIS.

Additionally, ORIS does not present a friendly modelling formalism for the sDDF - if this tool was used for the scope of this thesis, it would create a final product which is extremely unfriendly to future use and provides dubious utility owing to how abstractly ORIS the system it models.

In summary, ORIS provides an extensive toolkit for simulating continuous-time discrete systems. It provides many useful ideas, especially in the use of SPTNs as a modelling tool and generalised specification of reward and stop conditions that can work on many systems. It is ultimately inapplicable for this thesis due to the strict focus on Petri nets as a modelling formalism.

3.2 Stochastic time Petri nets: time process modelling in Modellica

Lask demonstrate the modelling of a variety of hospital processes using STPNs using the modelling language Modellica [Lask, 2017]. The authors begin by linking the operation of a Petri net to real-world effects. Petri nets are described as consisting of a graph of weighted *places* and *transitions*. Places are connected to transitions by a weighted *arc*.

The weighting of a node is the concept of *marking* as introduced in Section 2.6. That is, there are some number of tokens at the place. Arcs are only weighted when they originate from a transition and terminate at a place. The set of arcs connected to a place or transition are referred to as the input of that place or transition, and similarly the set of arcs leading out from a place or transition are the outputs. A transition is considered active if the markings (node weight) of all input places to a transition are greater than the arc weight.

Markings may change when an active transition *fires*, i.e. becomes active. Firing results in all tokens of the input being reduced by the arc weight of the firing transition and the tokens of the output of the transition increasing by the same amount.

This paper discusses a variety of concepts that can be layered upon regular Petri nets to enhance their ability to match real world systems, notably:

- Time events - adding additional conditions to transitions to force them to fire at a discrete time relative to the system time.
- Time tacts - adding repeating time events according to some periodicity, starting at some *tact-start* and repeating upon the *tact-interval*.
- Transition delay - adding some constant time delay between the transition activation and firing.
- Fire duration - adding some fixed time a transition firing takes to occur, i.e. the time before inputs and outputs are altered.
- Immediate firing - firings that are decoupled from time, occurring instantly.
- Continuous firing - transfer of tokens as part of a firing occurs continuously over some period of time instead of discretely.

This is to say, the authors discuss the construction of a timed Petri net and link each concept to a real world phenomenon in the hospital system. Time events are used to

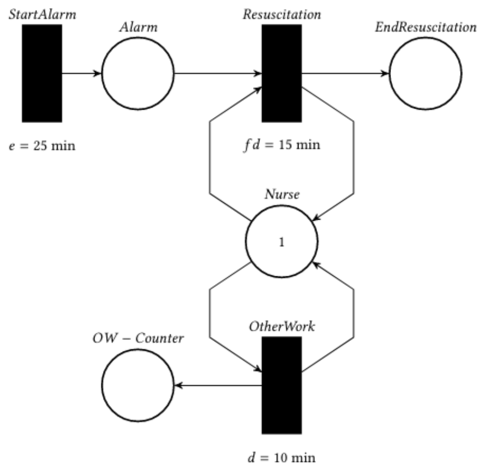


Figure 3.2: A timed Petri net depicting a nurse's workflow with an urgent interruption

describe the vacation schedule of a staff member. Time tacts are used to represent a shift model which is tolerant of interruptions. Delays are used to represent surgical preparation times. Continuous firing is used to model the process of paperwork being completed while other interruptions are possible. Finally, firing duration is used to represent a nurse's schedule always halting for performing a resuscitation while time events and durations model other activities.

The authors note that immediate firings introduce inaccuracy in simulation and subsequently recommend their usage is avoided.

The connection between STPNs and timed Petri nets is explained as a simple replacement of the exact time values of the timed model with statistical distributions taken over some random variable.

In summary, this paper presents an excellent summary of the process of mapping real-world systems to a STPN representation. Simulation tools are described and recommendations are made for which to use for a given need, making this paper a good starting point for dynamic modelling with Petri nets. The methods described can be applied to similar languages, such as the SimScape language which is supported in Mathworks Simulink. This work faces a similar problem to ORIS in that it ultimately doesn't provide a strong mapping to the sDDF. Petri nets are poorly suited to constructing a system which behaves procedurally within a single process, such as PDs executing serially on a core as they are selected by the scheduler. This paper provides useful context to consider when designing a tool to analyse the sDDF however, particularly with respect to how the combination of events with a variety of possible entity activation types.

3.3 Real-time estimations for waiting-time distribution in time-varying queues

Konrad and Liu present methods for analysis of non-stationary queues such that effective waiting times can be calculated [Konrad and Liu, 2024]. The authors also supply simulation-based experiments to validate their methods. The scope of this paper is primarily concerned with real queuing systems such as those in hospitals, and this paper correspondingly describes queuing systems in terms of staffing. The paper also assumes multi-server queues in all cases.

The distribution of waiting times for customers is a critical metric because most common service-level quality of service metrics used in industry can be derived from it. Consequently, if the waiting time can be inferred, so can other metrics such as the probability of delay and probability of abandonment.

The paper present several ideas for how to do this. The first is the so-called *myopic method*. Prediction of waiting time is nontrivial for non-stationary systems, hence the myopic method simply interprets a non-stationary system as a stationary one by assuming that the cause of non-stationary behaviour ceases at steady state.

The authors experimentally demonstrate that the myopic prediction varies from the truth significantly by method of a *Monte-Carlo simulation*. A Monte-Carlo simulation performs random sampling of some random variables repeatedly to determine deterministic problems. The results show that this myopic method deviates from the ground truth almost immediately and is of little practical use.

The authors then explore analytical methods for analysing waiting time under constant and time-varying staffing - we shall only focus on the constant case under the *potential waiting time* (PWT) waiting time definition.

The authors supply a heuristic method for potential waiting time approximation - assuming the queue is operating such that all positions are full, a new customer entering the queue is *tagged* and tracked. By writing an expression for the rate of queue position change for each position, the authors derive a transition-rate matrix and predict the waiting time using the method of Latouche and Ramaswami [Latouche and Ramaswami, 1999].

The concept of potential waiting time is a useful takeaway from this work for analysing the sDDF. Cores in the sDDF lack the properties of the queuing networks in the paper, but this work shows the value of a constant abstraction of complex time-varying processes. Potential waiting time can be applied conceptually at a finer grain in the sDDF, such as to describe the black-box behaviour inside of a PD between queue operations.

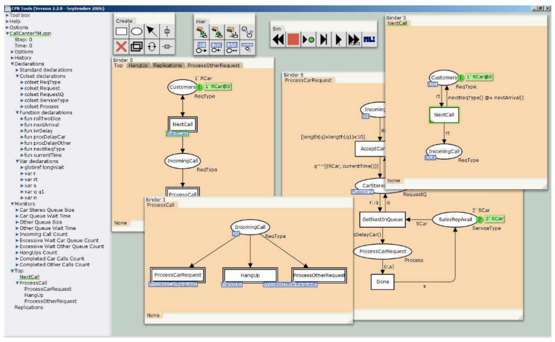


Figure 3.3: A CPN as seen in the CPN GUI

3.4 Systems modelling and simulation using Coloured Petri Nets

Coloured Petri Nets (CPN) is a systems modelling language which combines a Petri net simulator with the programming language Standard ML [Jensen et al., 2007]. The CPN tooling includes simulation and analysis tools as a GUI suite.

CPN is distinct from other SPTN runtimes in that it directly integrates a high-level programming language for describing data model. CPN is intended for the broader discipline of *systems engineering*, permitting more approachable general systems modelling instead of simple queuing networks or systems of Petri nets.

Gehlot and Nigro introduce a variety of methods for mapping Petri net systems to a CPN model [Gehlot and Nigro, 2010]. Unlike regular Petri net models, in CPNs the tokens in each place are visible and have a value. This allows tokens to carry information in addition to simply exciting state changes in the network. Tokens also may have a *colour set*, effectively a datatype. Every place and transition has an associated colour set and tokens of the matching colour flow through the network. Tokens are used to represent entities and their attributes in the simulation.

Simulations are constructed under CPN by defining an input state (i.e. some set of valued input markings) and the output is evaluated in terms of the output markings of the system. A set of *statistics collectors* are supplied, able to continuously collect data as well as defining breakpoints. This is a very useful characteristic as it allows for the simulation data to encapsulate the state of the system at every point in time.

The total effect of adding colouring, typing and stateful tokens is that CPNs more closely resemble a flowchart than a traditional Petri net. Figure 3.3 depicts the structure of a CPN as seen inside the GUI.

CPNs are a much more powerful language than similar options such as ORIS [Carnevali et al., 2022] and present a significantly more compelling option for modelling the sDDF than STPNs. The availability of Standard ML for describing behaviour of CPNs is also very valuable for creating a reusable model of the sDDF that can be more easily modified in the future when compared with pure Petri net representations. Variable and visible tokens with CPNs are a significant strength for modelling the internal behaviour of PDs.

CPNs are capable of describing all the behaviour of the sDDF in a cohesive way and the integrated performance analysis is ideal for our purposes. Ultimately however, it has a severe falling in that it is a complex formalism that may become troublesome for future use by end-users. Ideally, our model of the sDDF should be easy to both operate and extend without requiring significant retraining, and this presents some problems. If the graphical modeller of CPNs could be replaced with a domain-specific procedural specification language, CPNs would be ideal as we ultimately seek to model procedurally defined systems! PDs running on a core cannot be elegantly implemented as a concurrent CPN graph. The features of the CPN analysis suite such as the visualiser and simulator are very useful ideas for a bespoke solution for the sDDF however and this work is a valuable influence.

3.5 Using domain-specific languages for modelling and simulation

Miller et al. examine an alternate approach to simulation: using object-oriented programming languages to define *domain-specific languages* (DSL) for simulation [Miller et al., 2010]. The authors present ScalaTion, a Scala-based DSL to demonstrate this principle.

The authors contend that a middle ground can be achieved between the traditional dichotomy of usage of a general-purpose language or a simulation-specific language such as SLAM. The authors present previous work as a dicotomy between simulation programming languages (SPLs) and general-purpose languages (GPLs) for simulation work with usage decided according to various trade-offs between them. A DSL is an alternative; i.e. a general-purpose language which is utilised to build a sublanguage for the domain. Internally-defined DSLs are the focus of this paper; i.e. the DSL is implemented in the language as opposed to being transpiled to the target language.

Object-oriented, functional programming languages such as OCaml are the suggested target for the process of DSL creation due to their rich syntax. The advantage of a DSL is the capability for greater expressiveness towards a target domain than can be achieved by either a GPL or an SPL.

ScalaTion is a general-purpose DSL for modelling arbitrary dynamic systems, having similar features to CPN [Jensen et al., 2007] but with a procedural code interface rather than a graphical Petri net editor. ScalaTion implements an event-based programming model to implement simulations.

Usage of domain-specific languages or language subsets is a very useful premise for simulating the sDDF, especially given that this thesis seeks to create a reusable and extensible analysis suite which can be used without knowledge of the simulation. The abstractions of the seL4 Microkit such as PDs, cores and queues can be trivially mapped into a library of classes in an object-oriented language, and this can be used to drive a bespoke simulation engine. A correctly-implemented domain-specific ”sublanguage” would provide an ideal solution for the sDDF!

The parleance of this paper is confusing for the scope of systems work however and for the remainder of this paper I will refer to a general purpose language with a ”sublanguage” for modelling simply as a ”domain-specific model” within that general-purpose language.

3.6 Discrete Event Simulation of Aircraft Sorite Generation on an Aircraft Carrier

Yoon et al. approach a highly domain-specific analysis problem with discrete event simulation (DES) [Yoon et al., 2023]. This work concerns itself with finding and optimising the sortie generation rate (SGR) of combat aircraft launched from a naval vessel, i.e. the rate at which an aircraft carrier can sustainably launch, recover, resupply and relaunch the aircraft on-board.

This work distinguishes itself from work looking at the SGR of ground-based airbases by including the notion of spatial limits, as the finite amount of deck room and limited number of aircraft elevators and resupply stations drastically change the scope of the analysis. The authors correlate the problem of SGR analysis of an aircraft carrier to classical ”machine shop” problem which forms the basis of many types of optimisation methods in the manufacturing industry. This paper seeks to extend DES methods typically used in manufacturing to suit this use-case.

The aircraft carrier is modelled as a coloured graph where each vertex represents a space for an aircraft (sized to fit the largest on-board) and an edge represents a path an aircraft may take to move from one space to another. The colouring of nodes represents their purpose, as seen in fig 3.4.

The authors implement a complex, custom simulation and analysis environment which enables users to input system specifications, generate a system, run the simulation and present results. The authors use SimPy, a Python library which appears in various works referenced in this thesis.

The input layer of the system accepts a specification from the end user, which is used in conjunction with the ”simulation component” layer to compose the primitives of the system. The simulation layer is a library of Python classes that model all processes, events and activities in the simulation - this can be considered a ”domain-specific sublanguage” as described in section 3.5 in the work of [Miller et al., 2010]. The ”modeller component” assembles the specification in conjunction with the simulation classes to create an executable SimPy simulation.

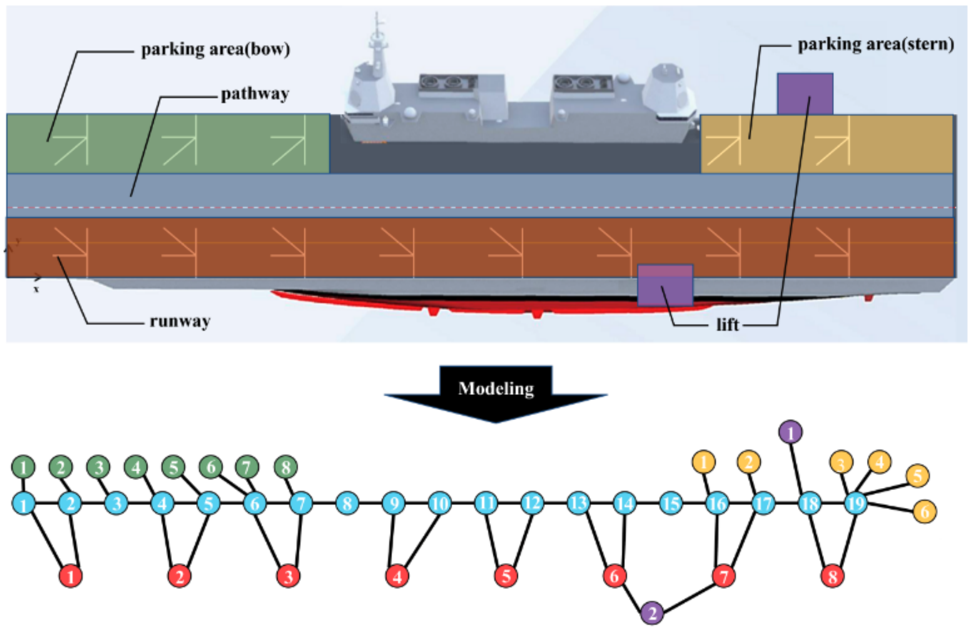


Figure 3.4: Yoon et al. model an aircraft carrier as a graph with nodes of finite capacity

The simulation is executed as a ”future event list” within SimPy, i.e. using it as a simple asynchronous event dispatcher. A separate ”data component” defines future actions based upon a supplied policy specification which are appended to the SimPy future event list. Within the simulation, a specific ”monitor class” is attached to parts of the graph and stores events such as aircraft passing it. These events are stored and used by the ”reporter component” to perform analysis on the simulation at termination. The analysis results of the simulation are relatively simple, effectively being just a set of success rates for how often sorties were able to launch according to the desired SGP.

This paper presents a startlingly apt simulation model for application to the sDDF - similarly to an aircraft carrier, the sDDF is a closed system where entities flow between a various nodes through fixed paths and are limited by the space available across the graph. With a simple change in language, this becomes apparent - consider the graph in figure 3.4 as a queuing network. All nodes of a single colour can be considered spaces within a single **server** (a PD) where aircraft (packets) can be serviced concurrently, but aircraft can only move to the next server serially by by entering a queue! The lack of any queuing theory or queues in the original work reveal a variety of extremely useful ideas for application to the sDDF, and this is ultimately unsurprising given that we have thoroughly established the inadequacy of most querying theory approaches for the problem.

The sDDF diverges in that packets flowing out of a PD can only follow a single path and always join the end of a FIFO queue, instead of entering a position in the queue related to their time of arrival in the aircraft carrier case.

The usage of SimPy as an event dispatcher combined with a domain-specific simulation

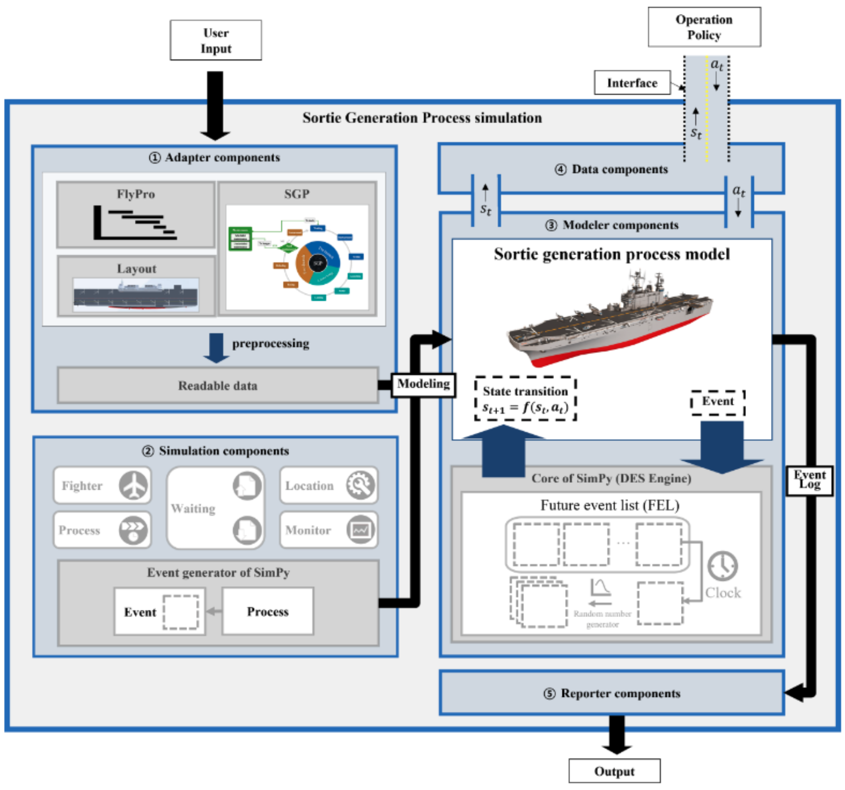


Figure 3.5: Architecture of the SGP simulator

sublanguage is directly applicable to the sDDF and offers substantial benefits over all other work reviewed. The internal DSL combined with an abstract "adaptor component" for untrained users to supply parameters allows this work to be used to analyse any system that can be represented by the adaptor without any code changes. Using "reporter components" which emit recorded events which are analysed at the termination of the simulation is also a useful concept for simulating the sDDF, as the flow of packets between PDs with respect to time can be trivially described in terms of events and also produces a complete description of the system behaviour.

In summary, this paper presents a well-considered solution for simulating and analysing a domain specific system by end-users. It provides many important ideas that can be directly reused for analysing the sDDF.

3.7 Comprehending Performance from Real-World Execution Traces: A Device Driver Case

Yu et al. propose a methodology for correlating execution traces of computer systems with their impact on performance [Yu et al., 2014]. An execution trace is a series of recorded *events* that occur as a computer executes tasks, described by the authors in the terms of a schema referred to as Trace Stream (TS).

A trace stream bins events into one of four categories:

1. Running: a sample of CPU usage for a thread.
2. Wait: emitted when a thread is blocked or goes to sleep.
3. Unwait: a thread awakens due to being unblocked or signalled by another thread.
4. Hardware service: a timestamp and duration of time where the thread is blocked by interacting with hardware.

This schema is suitable for evaluating the performance of threads as they interact with each other, particularly in the case of drivers and other operating system services. These events clearly demarcate the amount of work threads can perform (running), how frequently they are interrupted (wait), when they resume (unwait) and how non-software visible effects (hardware service) contribute.

The authors suggest a two-stage analysis approach for reconstructing performance impacts of particular threads on other threads:

- **Impact analysis**: measure the performance impact of a single thread or component.

- **Causality analysis**: Use the results of impact analysis to determine how blocking or CPU contention cause threads to affect other threads.

The authors perform impact analysis by the mode of *wait graphs* and other complex formalisms for describing concurrent behaviour. Using these formalisms and reasoning about the TS schema, they establish several fundamental metrics which should be used to inform further analysis:

- **Total duration**: total time spent by each thread inclusive of waiting.
- **Total wait duration**: total time spent blocked or waiting by each thread.
- **Total running duration**: Total running time of threads exclusive of wait time.
- **Total distinct-wait duration**: The total lenth of all distinct waits, i.e. ignoring recurring waits or finding the impact of specific event irrespective of traffic.

Using these metrics, the authors establish metrics for the impact of each duration on total execution time. The authors use these metrics in combination with a *contrast data mining* approach to correlate the impact of components on each other.

The value of this paper lies in the scoping of which events are required to reconstruct the performance impact of components, that is, the list of event types and how to quantify them. The concurrent semantics and device interactions of the sDDF are drastically simpler than the systems modelled in this work, and as a result we can elect to use simpler methods for direct analysis of the sDDF. A wait graph is simply not valuable for the sDDF given that PDs pass data asynchronously in the cases we wish to examine.

This work informs us that ”running”, ”wait” and ”unwait” events are sufficient for us to reconstruct the performance characteristics of a single executing program, and important property to understand for the goals of this thesis.

Chapter 4

Design: pysddfsim

4.1 Overview

As there is no tool that is suitable for modelling sDDF that can maintain the necessary degree of user-friendliness, we introduce pysddfsim. Pysddfsim is a hybrid discrete-event simulator with domain-specific modelling abstractions for easy extensibility. The tool is broken into five independent parts:

- **Simulator and library**: custom discrete event simulation implementation which uses SimPy as an asynchronous event dispatcher. Includes a library of many domain-specific elements: PDs, cores, scheduler implementations, device models and a variety of PDs mirroring parts of the sDDF network device class. We use the simulator and library to predict the behaviour of the sDDF.
- **Analytics engine**: SQLite3-based event collection interface. Items in the simulation optionally declare **analytics hooks** to automatically generate events when notable circumstances occur. E.g. queue X has buffer Y inserted by PD Z at time T. The analytics engine can be disabled or removed with no impact to the simulator. Emits an sqlite3 database dump at simulation exit.
- **Analysis engine**: Tool for parsing the output of the analytics database dump. Uses execution trace analysis methods to reconstruct practically all important information about the simulation such as throughputs, execution order of PDs, pre-emptions of PDs and more.
- **Analysis GUI tool**: Tool which wraps the analysis engine and creates a graphical user interface using the Python `matplotlib` library. Provides graphical representations of dozens of important heuristics about system performance, with windows for inspection of individual PDs and queues.
- **Network test runner**: Another tool which automatically composes pysddfsim systems based on a user specification at the commandline. Users are offered 29

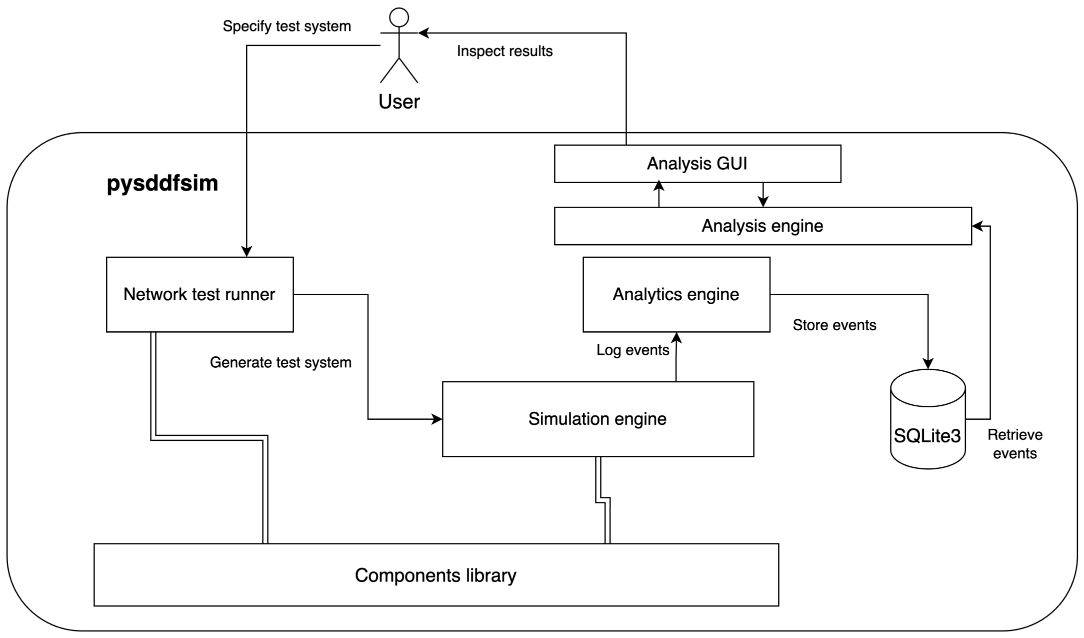


Figure 4.1: Architecture of pysddfsim

arguments which can be mixed and matched to create thousands of distinct test configurations. Used to generate **all analysis results in this thesis**.

We can use pysddfssim to analyse the efficacy of the sDDF protocol under a variety of extreme conditions that are difficult to represent or instrument on a real system. We are able to inspect every packet movement and PD interaction at a fine granularity without limits such as sampling bias. Unlike a real computer, we define systems in pysddfsim in terms of how PDs interact emit signals and interact with queues and define the delays between these operations with approximate values. We neither need or are able to perfectly predict performance for the scope of this project, and that frees us to focus on building sophisticated instrumentation to evaluate the efficacy of system operation.

Pysddfsim lets us discover the full set of queuing parameters for any given sDDF system such as the service and waiting times. We are also able to correlate the numerical relationship between periodic operations such as signals, PD wakeups and packet arrivals with achieved performance via Fourier analysis.

The test runner analysis GUI provide a user-friendly abstraction layer which we may give to anybody familiar with the sDDF for future usage.

Owing to the complexity of the tool, this chapter will be structured discursively. We will first introduce the motivation and theory behind the simulation as designed before proceeding to describe the design of each component.

4.2 Motivation

While there are a variety of possible methods to model the seL4 Device Driver Framework as a queuing network, there are no suitable queuing-based formalisms. It simply is not worth the effort for us to formalise a particular device class if that formalism cannot be easily extended as the sDDF is developed. Further, lacking the capability to model other device classes fails to satisfy our goal of creating an easily reusable tool.

Owing to the complexity of the behaviour of the sDDF, an optimal solution to the problem of analysing the sDDF is one which can easily be modified to evaluate new systems, policies or driver classes in the future. Various works on SimPy-based discrete event simulations provide useful ideas, especially the work of [Yoon et al., 2023] which presents a compelling model for a reusable, end-user-friendly simulation and analysis suite.

There is little former work that exploits a critical property of the sDDF: it is a purely software-defined system! We do not need the complex parallelism that can be represented with queuing formalisms. The only parallel behaviour that we must describe is the interaction of cores in a multiprocessor system and devices such as the network interface. Similarly, the continuous independent behaviour of individual tokens (aircraft, nurses) in the various DES simulations we explored are unneeded for the sDDF.

PDs run one at a time on each core, and we can describe their interactions procedurally instead of in terms of stochastic structures. A procedural model of the behaviour of each PD is significantly easier to construct and extend than any formalism because the original system we are modelling is also procedurally defined! Instead of modelling individual PDs as simulation processes, a more sensible abstraction is to model entire cores as single processes which internally implement PDs as serially-executing subprocesses driven by a scheduler.

Thus, instead of modelling the sDDF as a queuing network, we can instead construct a bespoke simulation of the seL4 Microkit by observing the flow of information through sDDF systems — the internal behaviour of PDs at any one point in time is irrelevant to the overall behaviour of the system. Instead of trying to simulate every PD cycle-by-cycle, we can instead reduce them to a black box with the following measurable datapoints:

1. When and how PDs interact with queues.
2. When and how PDs signal each other.
3. How long PDs take between signalling or queue interaction events.
4. When PDs decide to go to sleep or otherwise yield their execution time.

As described in chapter 2, all PD behaviour is heavily data-dependent. The above datapoints precisely describe that data dependency as well as behaving as an execution trace which can be used for impact analysis [Yu et al., 2014]! With such a trace, we can deduce performance characteristics, diagnose signalling problems and observe the full set of possible system-level behaviour. We can unite the concept of the execution trace with the "recorder component" event emissions by Yoon et al. to have PDs, queues and cores emit traceable events that describe PD behaviour as well as the fine-grained flow of data through the system. This is to say that the key to analysing the sDDF is to completely abstract away the statistical and queuing domain as we simply do not need it — we are able to model the effects that decide the stochastic behaviour directly instead.

Ideally, the analysis framework should be usable by anybody who works with the sDDF. In this interest, we can create a domain-specific modelling interface for users as informed by Yoon et al. and Miller et al., wherein bespoke objects (PDs, queues, cores) can be created using procedural code which resembles the original sDDF interface.

The intersection of an event-driven data model with a bespoke process-driven modelling style is the foundational behind pysddfsim. Unlike the work of Yoon et al., we entirely abandon stochastic modelling and analysis and only introduce sources of randomness via devices which supply or source data from the system and to model cache and other memory delays. We can achieve this simplification because the sDDF is a system with complex **data-dependent** behaviour which appears as stochastic behaviour at runtime as opposed to having any processes truly decided by a random process.

We can perform event-based data collection by automatically storing events as they happen; e.g. when the abstract model of a PD signals another PD, an event is transparently emitted and stored by the pysddfsim library functions.

4.3 The sDDF dataflow model

We must first explore the conceptual basis for modelling the sDDF as an event-driven system with a discrete perspective of time to coherently describe the design of pysddfsim. We can only reach the final model with the context of the dramatically different previous iterations.

4.3.1 Model of protection domains

The aforementioned observable actions of PDs share a critical property: they are all single events that occur at a point in time. We can consider a PD as a state machine which is stimulated by IPC and contains some invisible internal states encapsulating business logic. Each time a PD changes states, we can observe some event from the perspective of the system. Within those invisible internal states, the PD takes some amount of time to process data or await overheads from performing IPC. We can combine the notion of the PD as a procedural process with the PD as an event-driven state machine to construct a black-box model.

Under this model, we can consider the PD to be an iterator emitting (`event`, `timestamp`) tuples. The `timestamp` defines the delay before the iterator may yield again and the `event` is an optional event to be observed by the universe. We can represent externally-invisible states by emitting a timestamp without an event to create a delay. We can add these delays to model critical sections such that the PD can be pre-empted like a real concurrent program.

If we inspect PDs from the perspective of data flowing through the system, PDs only spend time waiting (performing processing, etc.), taking or putting items in attached queues, signalling other PDs or yielding and going to sleep. We can't observe any other behaviour. This is a very important observation as this allows us to advance time on the order of hundreds or thousands of cycles per event instead of simulating the effect of every line of source code. Every time a tuple is yielded by the iterator, all relevant business logic between the last yield and the current yield is executed as a batch and the simulation time then advances by the time specified. **This forms the basis for time in the discrete simulation**. That is, we track all timestamps in terms of CPU cycles and the time between events is defined by whichever scheduled PD is ready to emit an event at a given time.

4.3.2 Cores and scheduling

With an abstract representation of PDs established, we now must consider how PDs should execute in tandem and how they should interact with the simulated scheduler. The natural abstraction to represent a set of PDs executing serially is a **logical processor** with containing an instance of the MCS scheduler which we will refer to as a **Core**.

Each core is a bespoke process in the simulation, i.e. a single thread of execution in parallel with other processes. We can consider the core as a simple container for PDs and a scheduler which runs continually runs whichever PD is selected by the scheduler. PDs execute by returning an (`event`, `next timestamp`) tuple which the core will advance the simulation for. If the PD has less available budget than the next time it specifies, we advance time for the maximum amount the budget permits. Every time a core returns from the time advance, it charges the PDs budget by the time that passed.

Since processes run separately and PDs may interact across cores, we must consider the event in which an interrupt or a cross-core notification arrives. When this situation arises, the core will preempt the running PD, the sim will cancel the time advance specified by the PD the core will charge however much time passed from the PDs budget before it was preempted.

We can implement queues as by transposing their source code. They have no behaviour other than reacting to PD-emitted events such as insertions, removals or updates to signalling flags so transposition is a trivial process for us. We only require a single addition — an event must be emitted each time a PD puts or removes an item from the queue.

We can consider drivers as a special case of PDs — instead of only interfacing with other PDs they also interact with a **device**. Devices are **independently running** processes in the simulation which behave as sources or sinks for data and periodically deliver interrupts to alter control flow.

We can model devices as free-running processes separate from any core such that they present a simple interface to the driver via interrupts and simulated memory-mapped IO (MMIO). Similarly to PDs, we can treat devices as black boxes that emit events and delays to advance simulation time.

With this concept of simplified PDs, abstract devices and cores, we can construct a simulated model of the system.

4.4 Design: simulation engine

The simulation is the heart of pysddfsim and is solely responsible for implementing the abstract model we have composed and also encapsulates a library of pre-made components for extending the simulation. The analytics engine sits atop the simulator and is closely coupled to it, but is designed to have an agnostic interface that permits reuse. The analysis engine, analysis GUI and network test runner tool are all effectively applications built atop the analytics engine and simulator.

4.4.1 Concept

To make sense of the final design of the simulation engine we can consider the iterative development process leading to the final design.

The original design of the tool was focused on the simulator and aimed to model sDDF systems as stochastic-time Petri nets with a high-level modelling abstraction supplied via Mathworks Simulink GUI modelling. Data collection and analysis was supplied by tools inbuilt to Simulink. A full version of this tool was developed, but faced a show stopper at the half-way point of the thesis due to shortcomings in the Simulink simulation engine.

Instead of Simulink, the second iteration of the tool moved to SimPy and Python. The first Python iteration had some shortfalls, notably that it used a purely activity-driven simulation model as described in Section 2.10 — i.e. simulating every clock cycle in the system. PDs didn't implement logic at a clock cycle granularity, but rather scheduled **timeouts** repeatedly to model execution time while the core advanced time one cycle at a time. Internal PD logic was abstracted into a list of serially executing **step functions**: procedures which were executed back to back and encapsulated business logic and a stochastic modelling function that would emit delays based on the particular step function. Step functions could only proceed serially or reset to the 0'th index. This design was both slow and inflexible however, finally leading us to the third and final iteration of simulation engine.

With the final design, we move to a hybrid process-driven / event-based simulation instead of an activity-driven one. Under this model, all cores and devices in the system are free-running processes which advance time using a **future event list** via SimPy which affords us drastically more flexibility.

We abstract cores to consist of three components:

1. **Process**: main logic for the core. Delivers events to SimPy, both from PDs and for other events such as handling the core going idle. Responsible for advancing time.
2. **CoreDispatch**: abstraction layer that encapsulates the management of the core's PD list and handles cross-core IPC.
3. **Scheduler**: mirror of the seL4 MCS scheduler. Selects the next runnable PD for use by the main core process and ensures the current runnable PD is never overcharged.

To support the change to a process-driven simulation, PDs needed to be completely rewritten. Instead of the previous "stepfunc" model, we model PDs as the hybrid process design of Section 4.3.1. We can implement the abstract iterator of the high-level design as a single **generator** function for all business logic and delays. A generator is a function which progressively returns single values while retaining an internal context, an ideal abstraction for representing business logic which can be interrupted at any time.

Within the generator, we implement all business logic such as queue interactions, performing calculations and more. This logic is interspersed with 'yield' statements to emit **delays** in cycles. Instead of yielding events, we instead send them directly to the analytics engine as the generator runs without coupling those events to yields. This model empowers us to write a simple representation of any business logic while also explicitly informing the simulation of the time of the next event for the future event list! This allows the simulation to advance time in units of hundreds or thousands of cycles at a time.

We implement devices processes as generators identically. Devices execute in their own time and yield from a business logic generator.

The set of all devices, cores and PDs are encapsulated in a single `Simulation` object which exposes a method which is invoked to trigger execution.

The remaining subsections will discuss the design of each component in detail from the bottom-up, as this is the best way to understand how components interact.

4.4.2 Protection domains (FlexPD)

The template class for implementing PDs in pysddfsim is called `FlexPD`. It is designed as an abstract base class; it cannot be used on its own and must instead be inherited by a child class in an implementation. The FlexPD superclass contains all bookkeeping required by the simulation and includes a set of methods used by the core to interact with the PD which allows us to model PD implementations without duplicating simulation logic. To describe the business logic, we use two important abstract methods are from the superclass: `pd_init` and `pd_notified`. These allow us to model the initialisation and notified entrypoints of real PDs respectively. Child classes override these methods to define business logic.

There are five notable methods in the superclass:

1. `run()`- primary generator for PD behaviour. This implements an infinite loop which mirrors real Microkit PDs, i.e. it blocks on the signal handler and once signalled will yield from the `pd_notified()` function and store the next delay as the next timestep. The run function mirrors the semantics of real notification objects; i.e. if the PD is notified while it is not waiting, the notification object will update and the PD will immediately re-enter the notified function once it finishes work instead of blocking on the notification again.
2. `next_step_len()` - used by the Core to append to the future events list if this PD is runnable. This function returns the next step delay most recently yielded by `run()` for the Core or whatever fraction of it remains if the last delay was partially completed.
3. `charge_budget()` - the interface for the Core to control the PD. Once a delay specified by `next_step_len()` passes in simulation time or the PD is preempted by a signal, the Core calls this method to subtract the time that was used. If the current timestep is completed, this method transparently calls the
4. `run()` - generator containing all business logic and periodically yielding delays in cycles. The business logic can yield zero to indicate it has gone to sleep.
5. `signal()` - causes the PD's notification object to be triggered. The PD superclass transparently escalated all signal calls to the `CoreDispatch` of its core to handle any change to scheduling that may result.

These five methods above and the two user-defined functions define the entire runtime behaviour of PDs in general. After the PD is selected to run, the core tries to advance the simulation to the `next_step_len()`. If it succeeds without interruption, `charge_budget()` gets the next delay and advances the business logic, or otherwise partially decrements the next step length and it will attempt to complete it next time the PD is runnable.

The two subclass-supplied methods are simple. The `pd_notified()` method must implement a generator which yields delays in cycles to drive `run()`, and the `pd_init()` method is run at system startup and allows PDs to set up initial queue state, etc. Both of these functions are designed to support transposition of C code into the Python model directly, such that the programmer defines all yield points where a change is visible to other PDs.

4.4.3 Scheduler

The scheduler in pysddfsim is an abstraction of the seL4 MCS scheduler which implements logic to choose runnable PDs based on their priority and budget, logic to define the impact of signalling a PD on the scheduling order and a replenishment handler which is called periodically by the core to refill PD budgets. Instead of statically implementing the exact behaviour of the MCS scheduler, pysddfsim implements an abstract base class for the scheduler which can be used to implement different variations of scheduler behaviour, e.g. for modelling alternate kernel versions or experimenting with possible changes in scheduler design.

The scheduler abstract base class defines three methods that must be child classes:

1. `get_next()` - return the next runnable PD. In the case of the regular MCS scheduler, this is the highest priority runnable PD which has budget available.
2. `handle_replenish()` - kicks the scheduler to process the replenishment list and refill the budgets of PDs. Called repeatedly by cores.
3. `register_pds()` - used by the core to supply the list of PDs to the scheduler. Prompts the scheduler to prepare internal structures to accommodate the set of PDs and perform any necessary initialisation.

Two scheduler implementations are supplied in the pysddfsim library:

1. `MiniMCSScheduler` - Python mirror of the seL4 MCS scheduler for the seL4 Microkit. Assumes PDs only have a maximum of two refills for round-robin and one for sporadic to match the constraints on PDs in the Microkit.
2. `MiniMCSFairNotify` - Altered implementation to mitigate a bug in the seL4 scheduler which can be abused for unfair service. Used to mitigate a severe problem in section 6.3.1.

4.4.4 Logical processors (Core)

As described in the introduction to the simulator, the `Core` class is a container class which acts as a single process in the simulation and contains a list of PDs and encapsulates an instance of the scheduler. `Core` class is the only class which is able to control the flow of time for PDs.

Cores are created with a defined clock frequency and a type definition for the scheduler to be used, allowing individual cores to have different clock speeds and providing an easy interface to swap scheduler implementations. PDs are not supplied to the core immediately and are instead incrementally added with the `add_pd()` method to support flexible initialisation of PDs as this is necessary to support multi-core systems.

The `run()` method of each core is a generator which is loaded as a SimPy process. The generator is an infinite loop where each iteration corresponds to a selection and execution of a PD (i.e. select future event time and charge budget). Each loop begins by calling `scheduler.handle_replenish()` to update the scheduler state before getting the next runnable PD. The next event time is acquired by calling `pd.next_step_len()` and the function yields a SimPy timeout for that duration to wait until then. Finally, upon control returning to `run()` the core will subtract the amount of time that passed between the last timeout and the current time from the PD's budget via `pd.charge_budget()`.

There is a problem however: a timeout will cause the core to go to sleep until the time it specifies. If an interrupt or cross-core notification arrives, the core needs to be able to cancel the current execution. This problem is handled by the `CoreDispatch` subcomponent of each core - all signals are caught by the dispatcher which checks if the signal would alter the current runnable PD. If it does, the current timeout the core is waiting on is cancelled and the PD's timestep is only charged for the amount of time that elapsed before the signal or IRQ arrived.

The core tracks the previous PD that ran a timestep and in the event that the PD changes, a **context switch** occurs. Each context switch causes a timeout for a delay that is specified by the platform the system emulates. In the event that there is no runnable PD, the core will **idle**. An idle core schedules long timeouts repeatedly until the scheduler refills an eligible PD or an interrupt arrives to awaken a PD.

4.4.5 Devices and the SystemIRQTable

Devices in the system are modelled as bespoke processes that function similarly to PDs, albeit without a Core to govern their execution. Each device is a child class of `Device`, an abstract superclass which implements a set of features defining the interface between the device and the rest of the simulation. Devices have an execution state: running or halted. As long as the device is in a running state, the `run()` generator will repeatedly yield delays while executing some business logic.

Communication between PDs and Devices is managed by a separate abstraction: the `SystemIRQTable`. For a given system, a shared `SystemIRQTable` is available for drivers to access to devices and to allow devices to attach to their IRQ handler functions. The table lists all devices by type and provides a slot for **interrupt callback** for each device.

When a device is instantiated, it gets a handle to the table and registers itself. All devices have an `interrupt()` method which will attempt to fire the callback handler that is associated with it. Driver PDs must have logic in their `pd_init()` method to associate their interrupt handler method with the target device. The table is symmetric and allows either the PD or the Device to register first.

We transferred between PDs and Devices via a bespoke mechanism defined by the device, meaning that driver PDs necessarily must implement device-specific logic to retrieve data. The `SystemIRQTable` provides a handle to the device with an associated IRQ handler to mimic memory-mapped IO, allowing the PD to directly access internal booking structures such as hardware ring buffers. We can use the pysddfsim `DMARing` and `DMABuf` classes to represent these bookkeeping structures for DMA devices such as network interface cards.

The `DMARing` class is a simple queue-like structure that contains a list of free and active buffers, mirroring the aforementioned hardware ring buffers which keep descriptors (pointers) to their buffers in the ring. We model DMA descriptors with the `DMABuf` class, a simple data container class which includes a simulated ”offset” for mirroring DMA abstractions in business logic.

4.4.6 Queues (NetQueue)

Queues are the primary mode of dataflow in the sDDF and consequently require a first-class representation in pysddfsim. We implement queues via the `NetQueue` class, a functional mirror of the namesake queues in the sDDF. Instead of using traditional ring buffers internally, Python lists are used to implement the internal structure instead. Each NetQueue class encapsulates two queues internally, one for unused (free) buffers and another for used (active) buffers.

The `NetQueue` implements the signalling flags of the sDDF network class to allow PDs to indicate when they require signalling. They have an extra role in pysddfsim however: **queues implement seL4 Microkit channels** for most PD-to-PD communication.

Each `NetQueue` has a `signal_other_user()` method which transparently sends a signal to the PD on the other side of the queue such that each PD instance doesn't need a handle to its neighbours for dataflow. This method is a generator that yields a platform-specific delay to signal the other PD, which allows PDs to `yield from` this method to perform all actions needed for signalling inclusive of yielding for the scheduler. This drastically simplifies system composition as PDs almost never need any information about their neighbours and always have a consistent signalling protocol.

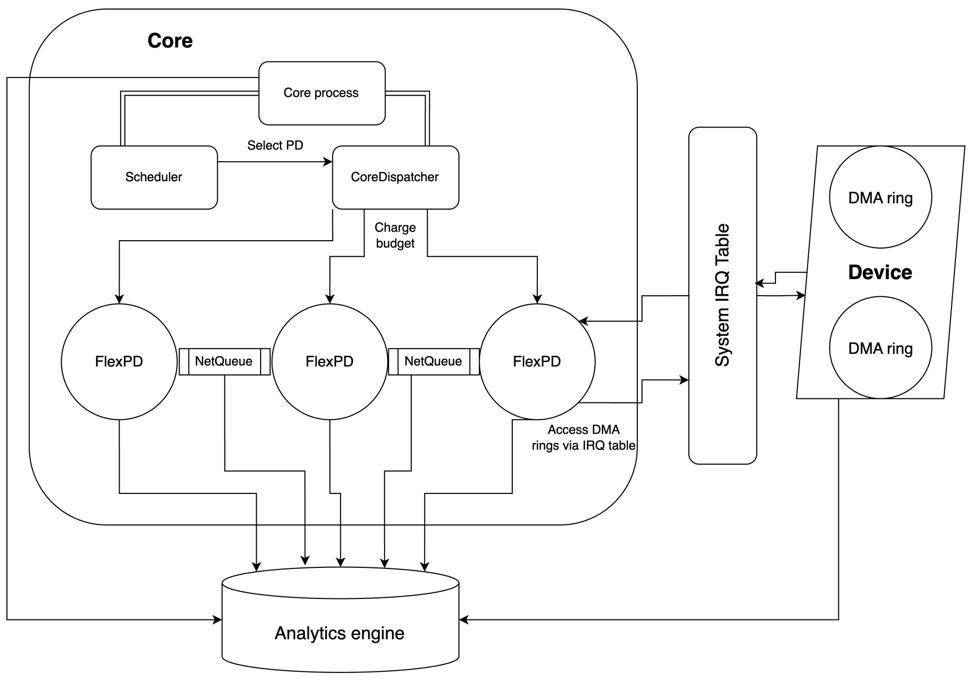


Figure 4.2: A simple system with one core in the simulation engine

We can store any Python object in a NetQueue, allowing the class to be reused for non-networking device classes.

4.4.7 Assembling a core

Now that we have introduced cores, FlexPDs, devices, the IRQ table and more, we can put together an image of a system in pysddfsim to clarify the design thus far. A core contains some number of PDs which communicate via queues. The scheduler decides which PD on the core will run based on its available budget and other scheduling parameters.

The core process prompts the core dispatcher to run a PD for the remaining time in its current timestep and charges the budget thereafter.

As PDs run, they may signal each other to alter control flow. All signals are caught by the scheduler and the execution order is updated. Devices exist outside the core as standalone processes and deliver interrupts via the `SystemIRQTable`. Interrupts are delivered similarly to interrupts, but they have the capability of interrupting the core while it is midway through runnning a PD. PDs can exchange data with devices by retrieving a handle to a `DMARing` from the device via the `SystemIRQTable`.

This architecture is depicted in Figure 4.2.

4.4.8 Overhead models and system generator

To model any system, there are several important runtime features that must be set such as the delays for context switches or cache overhead. pysddfsim implements an abstraction to represent these fixed costs called the `OverheadProfile`.

The `OverheadProfile` is a globally-scoped object similar to the `SystemIRQTable` that defines a series of methods that yield randomised delays for different operations. These methods are:

1. **Signal delay**: time to perform an seL4 notification.
2. **PPC delay** time to `seL4_Send()` and `seL4_Recv()` a protected procedure call for use with passive PDs.
3. **Recv delay**. Time to perform a single `seL4_Recv`.
4. **IRQ delay**. Time taken for the seL4 kernel to accept an interrupt and forward it to userland.
5. **Metadata interaction delay**: represents the delays that are likely to occur while interacting with shared memory such as cache misses. Internally models the probability of cache hierarchy misses with different variable costs.
6. **DMA interaction delay**: represents possible costs for interacting with DMA buffers.

We can set four of the six parameters accurately using `seL4bench`[seL4 Foundation, 2025], a tool for benchmarking seL4 systems. The standard MCS benchmark set will return results that indicate the signal delay, times to send and reply to PPCs, `seL4_ReplyRecv()` delay and the IRQ delay. Listing 4.1 shows the parameter set used for the i.MX8MM single-board computer.

```
$ ../init-build.sh -DPLATFORM=imx8mm-evk  -DFASTPATH=TRUE -
    DHARDWARE=TRUE -DFAULT=TRUE -DAARCH64=TRUE -DMCS=TRUE
```

Listing 4.1: seL4bench invocation for the i.MX8MM

The remaining parameters must be tuned manually and we will explore this process in Section 5.2. These are system dependent values that are largely a byproduct of cache behaviour of real systems and utilise a simple cache simulation to simulate cache hit probabilities increasing as a PD runs.

We supply two overhead profiles in pysddfsim:

1. `ARMv8iMX8`: modelled on NXP iMX8m series boards.
2. `x86Makatea`: generic model of x86 systems.

In order to compose systems, end-users can invoke the `sDDFSystemGenerator` module to access preconfigured test platforms and create an instance of a device driver connected to a list of PDs, including a device. Platforms are sets of parameters such as clock speeds, device configurations and an `OverheadProfile` which collectively model a real computer system, encapsulated in the `sDDFPlatform` class. Platforms define methods to generate devices that match their configuration.

Currently the `sDDFSystemGenerator` only supports generating systems mirroring the network device class as no other classes are available in the library.

4.4.9 The Simulation class and using the simulation engine

To run a simulation, the end user must define a *test program*. We do this by writing a declarative Python script which instantiates any PDs required and adding them to a core, possibly also invoking the `sDDFSystemGenerator` to create a premade driver class. Given the list of cores containing PDs and a list of devices, an instance of `sDDFSim()` can be created to execute the simulation. The `sDDFSim` class is a wrapper which completes the initialisation of all cores and devices and then starts execution. A simple test program in Listing 4.2 highlights the simplicity of generating and running a system.

4.5 Design: Analytics engine

The analytics engine is a thin SQLite3 database wrapper with a schema which defines all possible events in pysddfsim. The schema (`schema.sql`) is the primary component of the analytics engine.

The schema must encode all relevant information about a particular run for analysis such that no assumptions must be made later for analysis. As a result, there are top level tables for:

1. PDs: store name, scheduling parameters, type and associated core.
2. Cores: store frequency and ID.
3. Devices: store name, target RX and TX speeds.
4. Queues: stores PDs on either side of the queue and total capacity.

At initialisation we prompt the simulation engine to register all PDs, cores and devices to the top level tables. As entities are registers, they are assigned a database ID. We use this ID to identify all events associated with the corresponding entity. There can be any number of event types for a given entity type which are stored in child tables such that the primary key of the child tables is the event ID of the top-level table. For

```python
# Create top-level simpy environment
env = simpy.Environment()
init_analytics(env, "path/to/analytics.sqlite3")
timer = TimerDriver(env)

# Create some clients and their associated queues
rx_q = NetQueue(256)
tx_q = NetQueue(256)
echo = EchoServer(rx_q, tx_q, timer)
# Client definition: define queues, network address, copier=
  True
client_def = (echo, rx_q, tx_q, "172.16.1.1", True)

plat = sDDFPlatform_imx8mm(packet_sz=1472, rx_throughput
  =1000000,                                nic_traffic_gen=
  NICTrafficExponential, env=env)

# Create network driver class and NIC - set copier and
  driver queue size to 256
(pd_list, nic) = sDDFSystemGenerator(plat, [client_def],
  256,256)
# Set up core and add PDs
c = Core(env, clk_freq=plat.clk_freq)
for pd in pd_list:
    c.add_pd(pd)

# Make simulation and run
RUNTIME_CYCLES = 5 * 1e9
sim = SDDFSim(env, plat.clk_freq, [c], [nic]).simulate(
  RUNTIME_CYCLES)
```

Listing 4.2: Simple pysddfsim test program with a network echo server client modelling the NXP i.MX8MM single-board computer

example, the queue events table might have child tables for queue insertions and queue removals. This is illustrated in Listing 4.3 and Listing 4.4, detailing the parent and several child tables for PDs.

Each entity type has a top-level events table which associates a timestamped event with the ID of the entity.

We expose the database to the simulation via the `DatabaseAnalytics` object, a singleton class that wraps a database connection. A handle to the `DatabaseAnalytics` is given to all PDs, cores and devices for use via a variety of logging methods.

In most cases, we do not need to interact with the analytics engine directly as a majority of operations are automatically logged by classes in the pysddfsim library. For example, queues automatically log all insertions and removals, cores automatically log all PD budget charges and context switches, and PDs automatically log all sleeps and signals.

```sql
create table if not exists PDevents(
    id integer primary key autoincrement,
    pd_id integer not null,
    timestamp integer not null,
    foreign key (pd_id) references PDs(id)
);
```

Listing 4.3: Top-level events table for PDs

```sql
-- PD events --
-- Event: a PD was signalled
create table if not exists PDSignalledEvents(
    event_id integer primary key,
    sender integer, -- Null if signalled by a device

    foreign key (event_id) references PDevents(id),
    foreign key (sender) references PDs(id)
);

-- Event: a PD has gone to sleep due to running out of work
create table if not exists PDSleepEvents(
    event_id integer primary key,

    foreign key (event_id) references PDevents(id)
);
```

Listing 4.4: Several child event tables for PDs

We only need to call the analytics methods for operations that track implementation-specific behaviour, e.g. when a client PD consumes an ethernet buffer or when the NIC generates a new packet.

Whenever the analytics engine is initialiased with a particular database file, it will always overwrite it to prevent integrity errors. One file always corresponds to a single simulation. To improve performance, we do not allow the database to insert indices at runtime as it operates as a strictly write-only system while the simulation runs.

4.6 Design: Analysis engine and GUI

The analytics engine and the database dumps it emits are of little value on their own, we need something to parse the data. The analysis engine solves this problem for us: it is an independent Python module which parses database dumps and supplies a set of methods to generate inferences about the data within.

The analysis module is called `sDDFAnalysis` and contains 28 methods which we can use to perform inferences. The most important of these methods are listed below:

1. `create_performance_indices()` - generates database indices for all important operations to accelerate further other analysis. Called automatically when loading a database dump.
2. `infer_tx_paths()` - traverses the list of queues and PDs in the database to find all data transmission paths. Transmission paths in the sDDF are always an inverted tree where data flows from clients towards the virt. This function returns a separate list of PDs and queues for each branch of that tree to enable analysis of every path and intersection.
3. `infer_rx_paths()` - functions nearly identically to `infer_tx_paths`, but traverses the RX paths instead. The RX paths form a (non-inverted) tree where data flows from the driver to clients.
4. `calculate_pd_throughput` - find the RX and TX throughput of a single PD.
5. `calculate_total_throughput` - calculate the average RX and TX throughput of the entire system by finding the virtualisers in the RX and TX paths.
6. `get_total_pd_runtime` - find the total execution time of a PD by summing all intervals that start with the core context switching to the target PD and end with a context switch away from that PD or the PD going to sleep.
7. `get_mean_latencies` - find the average delay for a packet to traverse the RX and/or TX paths of the system.

We can classify the computations performed by the analysis into one of three broad categories:

1. Simple database queries to gather entities or count events corresponding to certain PDs or queues.
2. Interval counting queries which perform tasks such as counting all intervals between PDs being awoken and going to sleep to infer execution time.
3. Simple graph traversals to find paths data flows through or to identify which PDs are bottlenecks.

The principle function of the analysis module is to provide a set of complex inference methods for use in other programs to build more targeted heuristics, such as the analysis GUI we introduce in the following section.

4.7 Analysis GUI

We can use the analysis module directly as a part of the pysddfsim analysis GUI. Implemented with the `matplotlib` library, the GUI provides a complex graphical breakdown

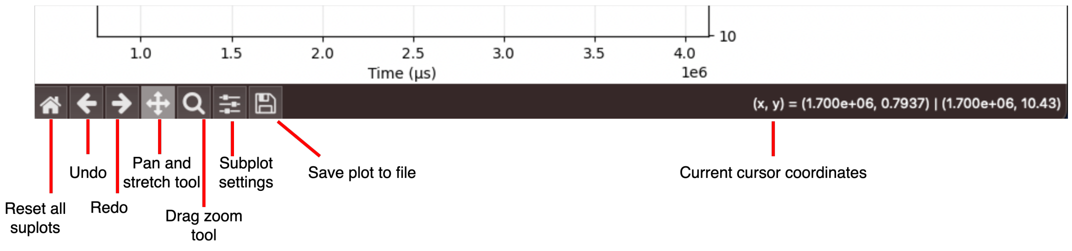


Figure 4.3: matplotlib controls at the bottom of each pysddfsim window

of an analytics database dump. As we analyse a given system the GUI is the primary means for analysis contained within pysddfsim, although other tools can be built easily using `sDDFAnalysis`.

The GUI is divided into six separate pages which open in separate windows:

1. **Overview page** — displays a breakdown of total system behaviour. Includes a visual rendering of the system as the analysis engine interpreted the database for reference and "sanity checking" the system configuration.
2. **Protection domains** — a listing of bar graphs showing important statistics about individual PDs for quick comparison. Includes links to individual PD pages for every PD in the system.
3. **Queues** — listing of all queues and graphical depiction of average queue state with a simple state checker.
4. **Frequency domain** — listing of PD throughputs in the frequency domain with analysis tools.
5. **Detailed throughput** — interactive full-screen version of the throughput and latency graphs on the overview page with with additional exploratory tools.

Since we use a `matplotlib` interface, we can interact with all subplots (individual graphs) by zooming in, panning and resizing or save the current view to a file using the controls bar at the bottom of each window as shown in Figure 4.3.

The GUI can be invoked using `run_analyser.py` in pysddfsim as follows:

```
$ python run_analyser.py --interactive path/to/dump.sqlite3
```

4.7.1 Overview page

When we invoke the analysis GUI, the first page we see is the overview shown in Figure 4.4. If we wish to make fast decisions about a simulation run, the overview page helps us to do this by listing all important performance characteristics. The name of the database dump is displayed as the title of the overview page for convenience.

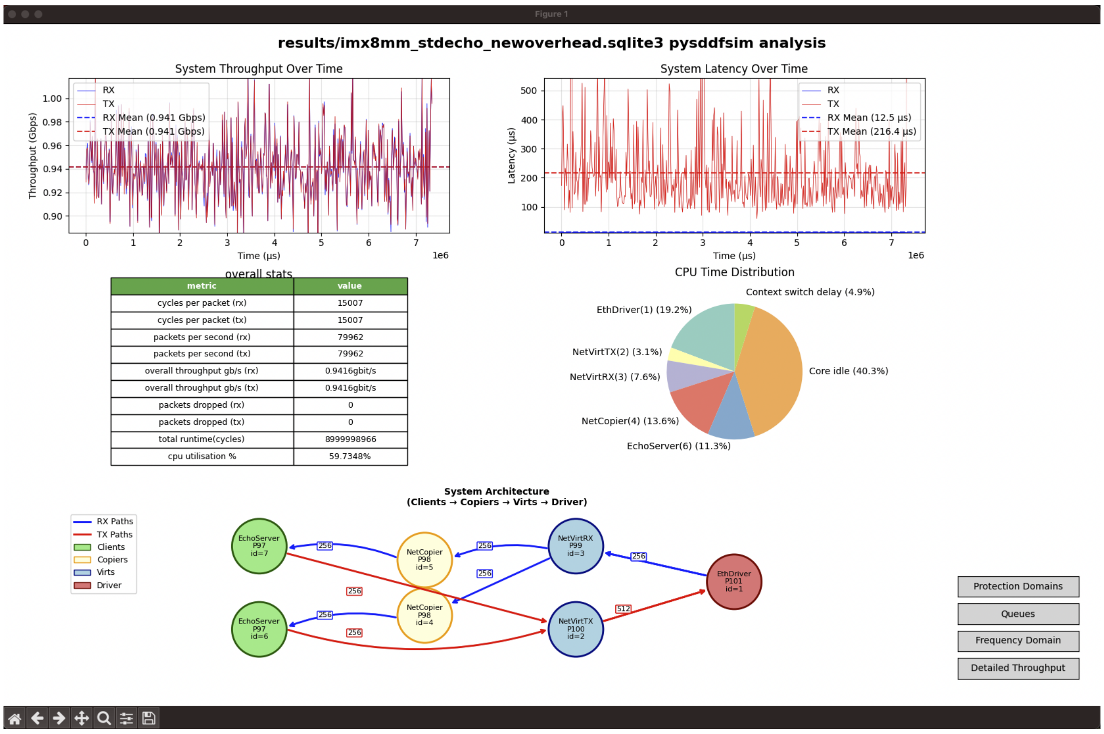
Clockwise from the top left, the elements are:

- **System throughput over time graph**. Shows the total RX and TX throughput of the system over time with mean lines. The throughput is evaluated as a series of averages over relatively large intervals to allow us to read it easily. The raw data is far too noisy to understand without zooming in. We can use the pan and zoom tools to inspect the graph closely. The system throughput is evaluated by counting packets passing through the virtualisers.
- **System latency over time graph**. A similar line graph to the throughput, but showing the RX and TX latency over time. Latency is evaluated by tracking the timestamp between received packets arriving at the NIC and being consumed by a client, or by between the emission of a packet by a client and transmission by the NIC.
- **CPU time distribution pie graph**. An overview of how our simulation used CPU time. Context switch delay represents kernel time when PDs are not executing while idle time represents time where the core had no work to do. PDs are labelled by their type name and their ID.
- **Subpage stack**. A series of buttons to navigate to the subpages of the GUI.
- **System architecture diagram**. A rendering of the system based upon the contents of the PDs and queues table which is ordered based on the RX and TX paths inferred by the analysis engine. We can use this view to make sense of various other parts of the GUI as it provides an easy way to map PDs to their IDs. RX queues are displayed in blue while TX queues are displayed in red, and all queues are annotated with their size.
- **Statistics table**. A simple table of important facts about the execution such as the cycles per packet, number of packets dropped and runtime.

4.7.2 Protection domains page and per-PD page

To compare the high-level behaviour of our PDs, we can use the protection domains page to get a summary of important metrics about the PDs on the system. Figure 4.5 shows the structure of the page, with six subplots showing bar plots of PD metrics by individual PD. One button is rendered at the bottom of the page per PD which will open a per-PD view. The metrics shown in the PDs overview are:

- Signals sent and received per PD.
- Packets dropped per PD.



overall stats

metric	value
cycles per packet (rx)	15007
cycles per packet (tx)	15007
packets per second (rx)	79962
packets per second (tx)	79962
overall throughput gb/s (rx)	0.9416gbit/s
overall throughput gb/s (tx)	0.9416gbit/s
packets dropped (rx)	0
packets dropped (tx)	0
total runtime(cycles)	8999998966
cpu utilisation %	59.7348%

Figure 4.4: The analysis GUI overview page

- Total runtime per PD in cycles.
- Total number of times each PD went to sleep (exited `notified()`).
- RX and TX throughput per PD.
- Count of PDs being pre-empted and pre-empting others.

The per-PD view for each PD shows a simple table values for each PD:

- period,
- budget,
- core number,
- runtime,
- times selected to run,
- number of sleeps,
- CPU utilisation,
- packet service time in μs and cycles,
- RX producer and/or TX consumer (next PD in RX/TX path),

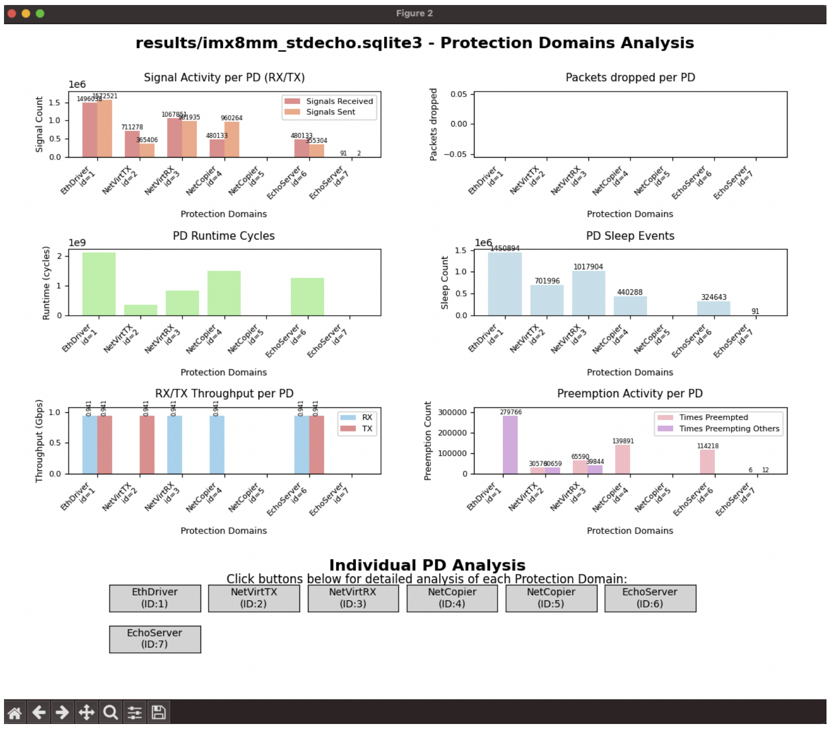


Figure 4.5: The analysis GUI protection domains page

4.7.3 Queues page

The queues page allows us to inspect the properties of all queues in the system while also providing a summary of the observed queuing properties of each. The interface is broken into a set of bar and whisker graphs indicating the average queue fullness and a table of statistics as shown in Figure 4.6. The queue fullness whisker graph shows the average, 2nd and 98th percentile fullness levels of the queue. Each bar is colour coded to indicate whether the queue is a part of the RX or TX path and an overload threshold line is rendered for convenience.

The statistics table contains the majority of information we are interested in while analysing a system. The fields are:

1. **Queue ID**. Database ID for the queue.
2. **Producer and consumer PDs**. PDs that insert or remove active buffers from the queue. Each PD is listed with its typename and database ID so that it can be identified with the system architecture diagram on the overview page.
3. **Capacity**. Number of buffers this queue can contain.
4. **Direction indicator**. Indicates whether this queue is on the RX or TX path. Rows are also colour coded based on direction.

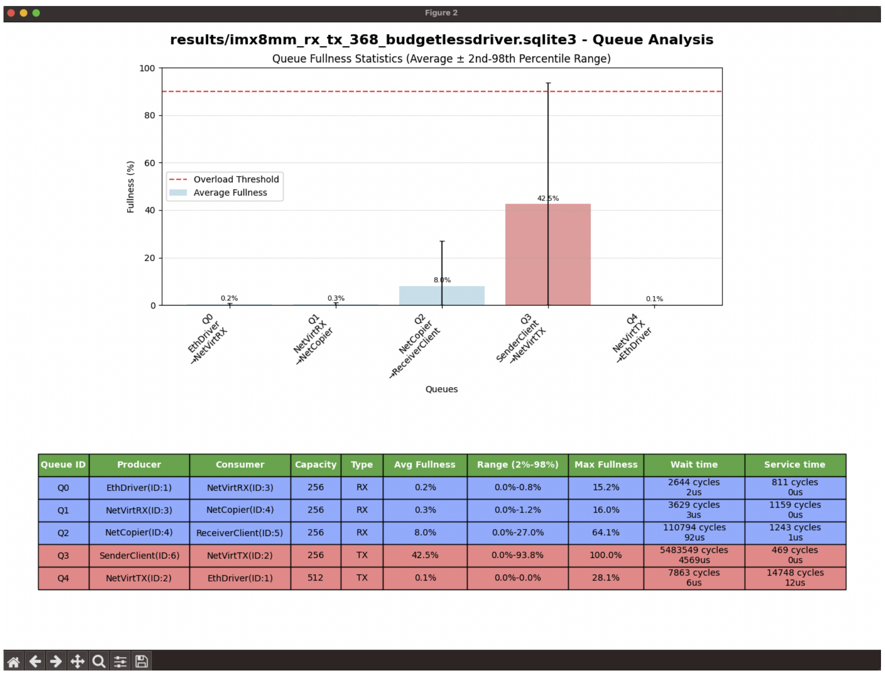


Figure 2
results/imx8mm_rx_tx_368_budgetlessdriver.sqlite3 - Queue Analysis
Queue Fullness Statistics (Average ± 2nd-98th Percentile Range)
Fullness (%)
Overload Threshold
Average Fullness
0.2%
0.3%
8.0%
42.5%
0.1%
Q0 EthDriver →NetVirtRX
Q1 NetVirtRX →NetCopier
Q2 NetCopier →ReceiverClient
Q3 SenderClient →NetVirtTX
Q4 NetVirtTX →EthDriver
Queues

Queue ID	Producer	Consumer	Capacity	Type	Avg Fullness	Range (2%-98%)	Max Fullness	Wait time	Service time
Q0	EthDriver(ID:1)	NetVirtRX(ID:3)	256	RX	0.2%	0.0%-0.8%	15.2%	2644 cycles 2us	811 cycles 0us
Q1	NetVirtRX(ID:3)	NetCopier(ID:4)	256	RX	0.3%	0.0%-1.2%	16.0%	3629 cycles 3us	1159 cycles 0us
Q2	NetCopier(ID:4)	ReceiverClient(ID:5)	256	RX	8.0%	0.0%-27.0%	64.1%	110794 cycles 92us	1243 cycles 1us
Q3	SenderClient(ID:6)	NetVirtTX(ID:2)	256	TX	42.5%	0.0%-93.8%	100.0%	5483549 cycles 4569us	469 cycles 0us
Q4	NetVirtTX(ID:2)	EthDriver(ID:1)	512	TX	0.1%	0.0%-0.0%	28.1%	7863 cycles 6us	14748 cycles 12us

Figure 4.6: The analysis GUI queues page showing a system with an overloaded queue

5. **Fullness statistics**. Mean, 2nd percentile, 98th percentile and maximum achieved fullness levels.

6. **Queuing parameters**. queuing-theory description of each queue showing the wait time and service time for the queue. Both variables are shown in cycles and μs.

We calculate fullness by collecting the set of all queue insertions and the set of all queue removals. From these, we effectively recreate the queue state by accumulating insertions and deletions over time while storing the fullness level at every step. The list of fullness levels is then used to determine all the statistics shown.

The queuing parameters tell us the waiting and service time of each queue. These factors are useful in their own right as they explicitly indicate the rate of data flow through each queue and also may be useful for future approximate application of queuing theory.

We calculate the service time by determining the average period of queue departure as it is equal to the mean service time. Wait time is more difficult as it is defined as the average time between a packet entering the queue and leaving the queue. We find the wait time by collecting all queue insertion and removal events and finding the delay between every matching pair.

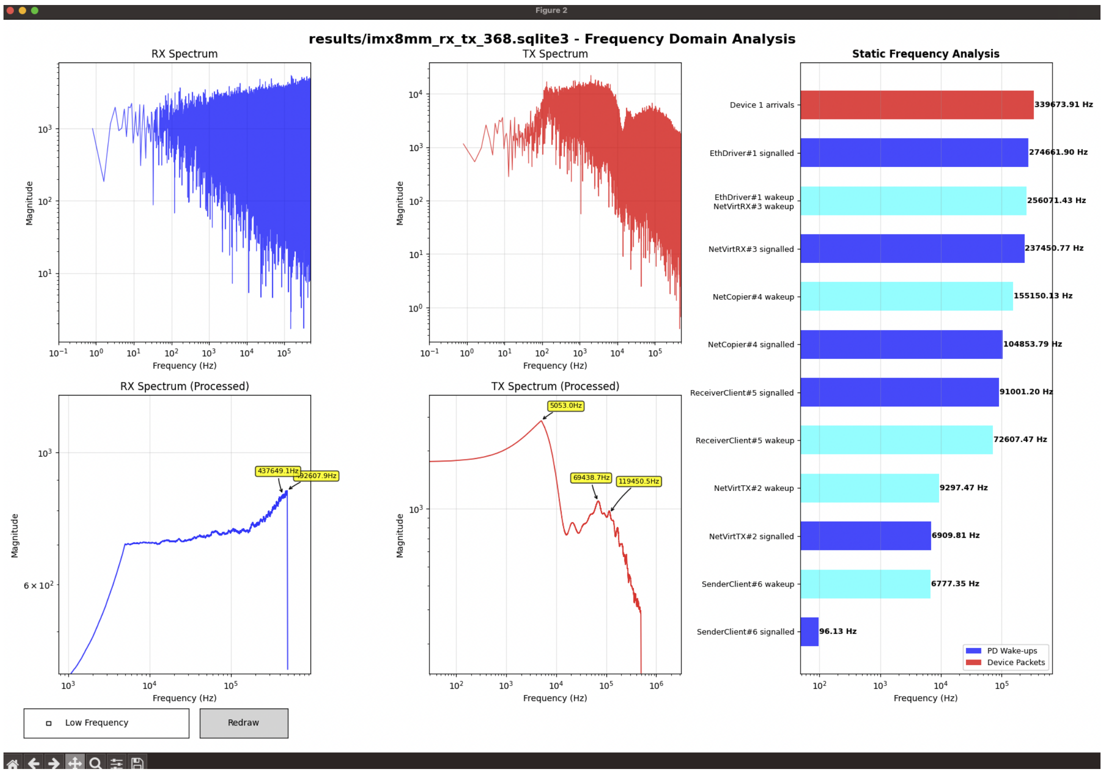


Figure 4.7: Frequency domain GUI page

4.7.4 Frequency domain page

The frequency domain page allows us to inspect the frequency components which compose the RX and TX throughput we observe. This tool allows us to identify any unusually strong periodic components to our output which we can use to diagnose performance issues or reason about how each PD impacts the throughput. The page is divided into five subplots as shown in Figure 4.7:

1. **Raw RX and TX spectrum**. The output of a discrete Fourier transform of the RX and TX spectrum as two separate graphs. These plots are only lightly filtered for legibility.
2. **Processed RX and TX spectrum**. Heavily processed spectrum with noise removal and peak detection. Significant peaks are labelled automatically.
3. **Static frequency pane**. A listing of the average frequencies of various important properties about the PDs in the system. Contains rate of packet arrivals from NIC, signal frequency of all PDs and wakeup frequency of all PDs.

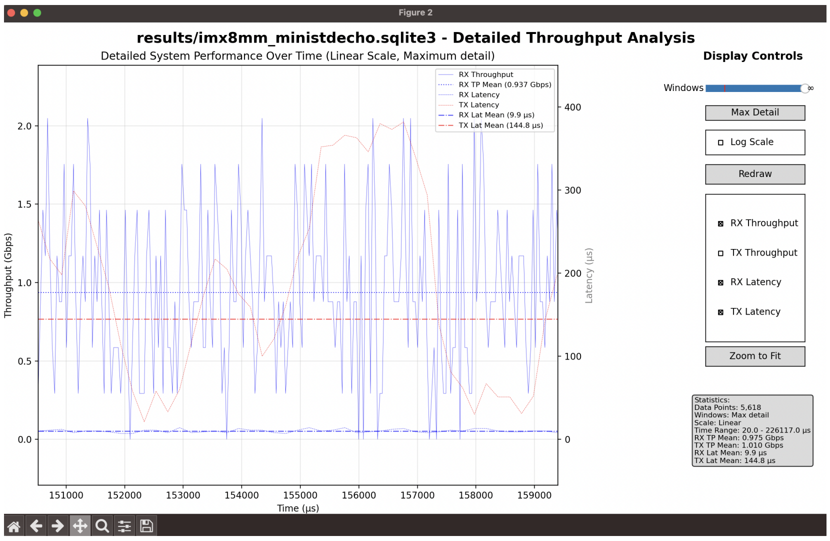
In order to allow us to inspect a wide bandwidth accurately, the GUI has a ”low frequency” mode which can be triggered via the checkbox in the bottom left. While in low frequency mode, throughput samples are averaged in groups of sixteen to emphasise the lower frequencies. This is necessary to sufficiently lower the noise floor of the lower frequency ranges and to prevent the high frequencies from skewing the peak detection.

In either mode, the rendered graphs are bounded by the Nyquist frequency of the collected data. The Nyquist frequency is a concept derived from the Nyquist-Shannon theorem, describing the sampling rate required to accurately reconstruct a periodic signal [Shannon, 1949]. We cannot usefully derive signals of a frequency exceeding one half of the sampling rate of our input data as there is insufficient information. The low frequency mode in the GUI should be disabled when inspecting higher frequency ranges at it caps the Nyquist frequency and hides higher peaks from the spectrum.

This page is intended to be used to correlate notable actions like PD wakeups or signals to spikes in the throughput as this can decisively indicate limiting factors. We will use this GUI for this purpose in Section 4.3.1.

To generate the processed views, we must perform several operations on the throughput data:

1. Crop data to remove any inconsistent samples at simulation exit.
2. Remove linear trend from data to simplify peak detection by removing any constant components in the frequency range. In terms of electrical engineering this can be described as removing the DC component.
3. Apply the Hann window function [Weisstein, Eric W, 2025]. Our input data is digital and in the form of "blocks" extracted from the system behaviour, but we are evaluating the continuous frequency by looking at the average of each block. "Sharp" edges on these blocks can cause spectral leakage as the boundaries between blocks contain some data from their neighbours. The Hann function smooths away the edge of each block to avoid this by taking three averages at points across the window and using them to emphasise the centre frequencies of the window.
4. Perform Fourier transform to convert our preprocessed data into the frequency domain.
5. Signal postprocessing. Several small steps to remove irrelevant data such as negative frequencies and limit the frequency range to the Nyquist frequency to remove data that is not helpful for analysing a system trace. Negative frequencies correspond to imaginary components that we can never correlate to real signals in the sDDF. Frequencies higher than the Nyquist frequency cannot be used to reconstruct behaviour as the Nyquist frequency is the threshold for accurate data reconstruction.
6. Noise removal. Remove the lower 75th percentile of magnitude data to hide insignificant frequencies.



```
Statistics:
Data Points: 5,618
Windows: Max detail
Scale: Linear
Time Range: 20.0 - 226117.0 µs
RX TP Mean: 0.975 Gbps
TX TP Mean: 1.010 Gbps
RX Lat Mean: 9.9 µs
TX Lat Mean: 144.8 µs
```

Figure 4.8: The GUI detailed throughput view

Once we have processed the signal, we can finally apply peak detection by searching for local maxima in the output data.

4.7.5 Detailed throughput page

Although the overview page provides us with graphs of throughput and latency, the small graphs are impractical for close inspection. The detailed throughput page in Figure 4.8 solves this problem by presenting a full-window viewer for both throughput and latency with controls for sampling precision and data scale.

The window consists of a full-sized line graph viewport and a controls pane on the right-hand side. The controls pane allows the user to toggle certain signals on or off, change to a log scale and to adjust the sampling precision. The sampling precision can be set to a finite value using the slider or set to infinity by clicking the "max detail" button, thereby using all frequencies up to the Nyquist frequency. Larger windows are useful for inspecting the system at an overview as they hide noise, while smaller windows allow inspection of fine-grained results.

In order to use the graph viewport in a useful way we must use the `matplotlib` pan and zoom tools to move through the data.

4.8 Design: Network test runner

A chief goal of pysddfsim is that we wish to allow end users to use the tool without any knowledge of the tool itself. The network test runner is a designed with this purpose in mind; it allows end users to specify complex tests with commandline arguments. The tool is supplied as `echo_analysis.py` within the pysddfsim source code.

We can use the test runner to compose and execute a test of the network device class with 38 parameters. Most parameters can be "mixed and matched" to allow composition of arbitrary tests. Throughout the later sections of this document, all tests are executed using this tool and invocation commands are supplied.

The test runner itself does little except parse the arguments supplied by the user. The tool has three mandatory arguments:

- Number of cycles to run the simulation for.
- Outfile file name.
- Test platform. Currently support "imx8mm" and "makatea".

The remaining flags define which PDs should be present on the system, how the network interface should generate traffic for the test PDs, and how all clients and the driver class should be configured. As we compose a test, we can draw from a library of sample PDs. Any PDs that generate TX traffic have their target throughput specified, and any PDs that receive RX traffic must specify what fraction of incoming packets belong to them. All throughputs can be specified with an SI postfix for convenience, e.g. "300M" for 300 Mb/sec.

Test clients are defined positionally and RX distributions are assigned with the `rx_throughput_dist` argument which assigns a distribution to PDs based on their position. E.g. if `client_echo` is defined before two instances of `client_rxonly`, their RX throughputs are defined in the same order.

Two special test flags are available for our usage:

1. Throughput sweep: run the target system with a series of different RX throughputs specified in a list and generate a line graph of the achieved TX throughput and CPU utilisation for each level.
2. Echo server count sweep: run a generic system with different numbers of echo server clients specified in a list. Generates a line graph of achieved TX throughput and CPU utilisation for each number of clients.

The sweep tests use a very significant amount of disk space! Each test run of non-trivial size generates a database dump file that is several gigabytes in size. These tests automatically use a scratch location on disk using the Python `tempfile` library, but system stability may still be impacted if pysddfsim uses all disk space on your system.

4.9 Limitations

Pysddfsim is very powerful but faces several limitations owing to the infancy of the tooling and the nature of the simulation engine.

4.9.1 Simulator accuracy limitations

The simulation engine is principally a simulation of dataflow through an sDDF graph which is enabled by a simulacrum of the seL4 Microkit. As a result, we can expect the simulation to accurately recreate the queue interactions and dynamics, but we cannot realistically predict the exact behaviour of real systems.

The simulation engine doesn't attempt to model a majority of the internal behaviour of PDs, devices or the system as a whole. As we have established in Section 4.4, we simply have no need to do so. Considerations such as the memory hierarchy, microarchitectural effects and thermal constraints are not directly modelled whatsoever and as a result we simply cannot expect to precisely predict how a real system will perform with an identical configuration.

This limits the overall accuracy of our system, particularly with respect to how we can mirror system behaviour as system load increases.

4.9.2 Simulation performance limitations

Currently, performance issues are the most significant limitation on pysddfsim. The core simulation engine is designed to efficiently run in a single thread such that we can launch many threads at once to execute many tests concurrently but single-thread performance is not optimal.

The analytics engine currently performs a certain degree of unnecessary work. The database schema is designed permissively to provide us with maximum flexibility at analysis time, but several events can be considered as superfluous. For example, we explicitly record every time a client "emits" or "consumes" a packet to indicate the terminal ends of the data path, but we could find the equivalent data heuristically using queue insertion operations such that we follow one buffer as it moves from an active to a free queue. Every event takes a nontrivial amount of time to add to the database which cannot be used by the simulation!

Superfluous events serve a function however: they allow us to more efficiently run the analyser by avoiding extremely complicated database queries or heuristic calculations. Minimising them where possible is ideal, but a balance must be struck to ensure we end up with a usable analysis process. The performance concerns could be alleviated by using multiprocessing, possibly by extracting the database to a separate process and sharing write requests over asynchronous queues.

4.9.3 File size of database dumps

The output files created by pysddfsim are very large, often on the order of gigabytes for runs that last minutes. This is unavoidable to a degree, execution traces are inherently very large. We could possibly improve this problem in the future by using a more compact data representation of some type, e.g. by using a different storage library or refactoring the database events tables to remove the 1-to-1 relationship between parent event tables and specific events.

4.9.4 GUI performance

The analysis GUI is a very simple `matplotlib` program internally. The database, rendering and analysis logic all execute in a single thread and this causes significant delays when opening large database dumps and navigating to certain views (particularly the PDs page).

We can improve this problem by refactoring the GUI into three threads:

1. GUI renderer and interaction thread.
2. Database thread.
3. Analysis thread(s).

Such a division can allow us to decouple the GUI entirely from storage and computation to improve responsiveness. Moving compute operations to different threads can possibly enable parallel computation of certain expensive metrics such as waiting times or latency.

Chapter 5

Implementation

With pysddfsim introduced, we can now proceed to apply the tool for analysis of the dynamic behaviour of the sDDF itself. The network device class will be the focus for this analysis as it is the most performance sensitive and best studied class at the time of writing. It also holds several properties that make it especially interesting for the purpose of analysis:

- The network class has far more PDs internally than any other extant class, with a minimum of a driver, two virtualisers, (possibly) an ARP and one trusted copier per client. Having more members in the flow network increases complexity of trivial analysis and many unusual edge cases are possible.
- Unlike other device classes, the network class is often limited by CPU utilisation. This makes it more valuable for optimisation than other classes who do not face any major bottlenecks in general operation.
- Zero-copy passing of DMA buffers down the stack exposes the stack to a variety of new problems such as buffer exhaustion or CPU bottlenecks in copiers.

5.1 Modelling networking

The first step to analyse the device class is to build a model of it for the pysddfsim library. We can construct all components of our device class using the primary abstractions of pysddfsim with no additional bespoke tooling required.

The network device class consists of the following components:

- **Network device**. A network interface card (NIC) is the focus of the networking stack as the interface between the network and software. The NIC is a direct memory access (DMA) device which fills shared memory buffers with data and provides handles to these buffers to the driver. Similarly, the driver provides DMA buffers filled with data to transmit.

- **Driver**. A program responsible for *driving* the network device — that is, controlling the software interface to the NIC while inserting and extracting data.
- **Receive virtualiser** (RX virt). A PD connected to the driver with a queue such that the driver delivers all received packets to the RX virt. Responsible for dispensing incoming packets to clients based on their MAC address and returning used buffers to the driver and managing the cache.
- **Transmit virtualiser** (TX virt). A PD similar to the receive virtualiser, but responsible for forwarding packets sent by clients to the driver and returning empty transmit buffers back to clients.
- **Receive copier** (RX copier). A trusted PD which mediates the receive process from the RX virtualiser. There are a finite number of DMA buffers in the system and the copier is responsible for extracting data from DMA buffers and returning them to the virtualiser as fast as possible to prevent starvation. The copied buffers are placed in a queue to the client. If the queue between the copier and client is full, the copier will silently drop incoming packets and return the buffers. There is one copier per client.
- **Clients**. Clients are generic PDs which integrate an internet protocol (IP) stack locally. Typically, the IP stack in clients is LWIP, an open-source IP stack [Free Software Foundation, 2002].

We can begin the process of transposing the class by inspecting the original C source code for each PD. With the exception of the driver and clients, no behaviour other than trivial business logic and queue interactions are necessary. Clients present some challenges as LWIP is very large and cannot trivially be mirrored in pysddfsim, but we can evade this problem by approximating the outward-facing behaviour of the IP stack. The driver requires a handle to the driver and the capability to interact with the `SystemIRQTable` to interface with the NIC correctly.

5.1.1 Modelling the NIC

The network interface card (NIC) is one of the most important components of the pysddfsim network device class. It is the principle data source and sink, and without it we cannot really model the network device class. The pysddfsim `NIC()` class is subclass of `Device` and is designed for maximum generality. Since we don't have any real network to interface with, the `NIC` contains a traffic generator to simulate a network connection. It is configured with two unique configuration classes for constructing tests: the `NICConfig` and `NICTrafficGenerator`. The `NICConfig` defines hardware parameters such as the size of the NIC internal ring buffers or interrupt emission behaviour as well as defining target operating speeds and a range of MAC/IP addresses for assigning to generated packets. The `NICTrafficGenerator` is an class type defining a pattern of packet arrivals given a target speed, defaulting to `NICTrafficExponential` which generates packets with a long-tailed exponential distribution.

To mimic the memory-mapped interface of real NICs, the `NIC` uses `DMARing` from the pysddfsim library. We add a separate RX and TX ring. Unlike PDs, the `NIC` doesn't take buffers from a queue. We simply assume that the driver supplies free RX or active TX buffers and the `NIC` will do nothing if there are no buffers available.

As it is a pysddfsim `Device`, we implement a majority of the runtime business logic for the `NIC` in a `run()` generator method. Unlike a `PD`, there is no notion of sleep or scheduling. Instead, `run()` implements an infinite loop that will produce and consume data unless we set it to halt via the `Device` superclass method.

We can define three throughput parameters via the `NICConfig`:

1. **Packet size**. Defines size of a packet in bytes.
2. **Device throughput**. Defines the maximum rate the NIC can send and receive packets at symmetrically.
3. **Target RX throughput**. Defines the target rate to generate incoming packets.

The `run()` generator is a simple process which repeatedly sets timeouts using the `NICTrafficGenerator`. We repeatedly draw delays from the traffic generator `rx_next_delay` and `tx_next_delay` methods and send the `NIC` to sleep until the most recent of the two delays. The RX and TX work in the NIC are effectively isolated and do not affect each other — we can make the `NIC` transmit and receive simultaneously at the maximum device speed.

Every time we reach the `rx_next_delay` the `NIC` "receives" a new packet with random contents and assigns a MAC/IP address to it based upon the address distribution in the `NICConfig` if there is a free DMA descriptor in the RX `DMARing`. We can specify the rate of received packets for every eligible client in the system this way. When we reach the `tx_next_delay`, the `NIC` will "send" a packet by destroying its contents and returning the DMA descriptor to the free TX ring to be taken by the driver.

The NIC implements three possible interrupt delivery schemes depending on the `NICConfig` to model different devices:

1. **Coalesce IRQ**. Send an IRQ once for every N packets.
2. **Throttle IRQ**. Send an IRQ whenever a packet is ready at most once every N microseconds.
3. **Watermark IRQ**. Send an IRQ whenever the `DMARing` has N or more packets.

We can use any of these schemes for both RX and TX with different coalesce, throttle, or watermark values in each direction. Interrupts are delivered by firing the interrupt handler function that has been associated with the `NIC` via the `SystemIRQTable`.

The `NIC` integrates with the analytics engine by logging all of the packets it sends and receives. We can use these events as the basis of latency and throughout calculations.

5.1.2 Modelling the Ethernet driver

The driver is the first PD in the network device class and is responsible for interacting with our `NIC`. The driver is implemented in the `EthDriver` class as a child of the `FlexPD` base class. To convert the original sDDF ethernet driver to a pysddfsim representation, we can directly transpose the C business logic while inserting delays.

We will use the sDDF i.MX8MM ethernet driver as our baseline design to transpose as it is the most refined driver currently available. Like all PDs, the driver has a `init()` and `notified` function which defines initisation and primary business logic respectively. We can omit all i.MX8MM-specific code such as configuring the NIC or interrupt handling and instead simply focus on the queue interactions and points where the driver performs MMIO with the NIC.

The `init()` function only performs two notable tasks for our purposes: it turns on the NIC and sets up the queues to the virtualisers. In our `EthDriver`, we can replace the NIC initialisation with a call to the `SystemIRQTable` to get a handle to the device and register a new interrupt handler method. Using the device handle, we call `NIC.enable()` to set it to the running state and define an interrupt callback which will mask interrupts and signal the PD. to set up the queues, we call the matching pysddfsim library calls on handles to the queues supplied in the class `__init__()` method, including generating the initial set of receive DMA buffers and putting them in the queue.

The `notified` function has is called whenever the PD received a signal and supplies the source of the signal. It is a simple loop which will perform one of three actions depending on the signal source:

1. If the signal originated from an IRQ, call `handle_irq()` and then acknowledge the IRQ to unmask it.
2. If the RX virt signalled us, call `rx_provide()`.
3. If the TX virt signalled us, call `tx_provide()`.

We can create methods that mirror `rx_provide()` and `tx_provide()` in our `EthDriver` by simply defining them as separate methods that also behave as generators. The Python `yield from` keyword allows one generator to yield from another generator until the child generator is out of work to do, at which point the outer generator will continue. We can directly transpose `notified()`, requiring only a single line of extra code to handle the possibility of an interrupt arriving while we have one outsanding. Listing 5.2 shows the original source code in C while Listing 5.1 shows the transposed logic.

```python
def pd_notified(self, channel):
    if self.outstanding_irq or channel == None:
        yield from self.handle_irq()
        self.irq.ack_unmask()
        self.outstanding_irq = False
    elif channel == self.virt_q_rx.consumer:
        yield from self.rx_provide()
    elif channel == self.virt_q_tx.producer:
        yield from self.tx_provide()
```

Listing 5.1: The ethernet driver notified() method in Python

```c
void notified(microkit_channel ch)
{
    if (ch == device_resources.irqs[0].id) {
        handle_irq();
        microkit_deferred_irq_ack(ch);
    } else if (ch == config.virt_rx.id) {
        rx_provide();
    } else if (ch == config.virt_tx.id) {
        tx_provide();
    } else {
        sddf_dprintf("ETH|LOG: received notification on
    unexpected channel: %u\n", ch);
    }
}
```

Listing 5.2: The ethernet driver notified() method in pysddfsim

As we model the methods that `notified()` yields from, we must consider where delays should be inserted. For example: the `rx_provide()` function is responsible for extracting all available RX buffers from the NIC and placing them into the virtualiser queue. Per our abstract model in Section 4.4.2, we should insert yield points with delays at every point where the driver interacts with a device, queue or other PD.

For the driver, we can derive all of our delays directly from the `OverheadModel` for the current system. For each queue interaction, we yield `queue_interaction_delay()` and each time we interact with the DMA buffers in the NIC, we yield `dma_interaction_delay(data_size)`.

To model MMIO interactions with the device, we subsitute code performing the MMIO operations with an equivalent operation on the NIC `DMARing` structures and yield an interaction delay once again as shown in Listing 5.3.

We can simply transpose all remaining functions from the C code that interact with queues or the device MMIO to complete our model of the driver.

```python
# Insert empty bufs from rx_virt.free until we cannot fit more
while (not self.ring_rx.free_is_full()) and (not self.
    virt_q_rx.free_empty()):
    buf = self.virt_q_rx.take_free()
    yield self.queue_interaction_delay()
    buf.clear()
    # Insert into NIC
    self.ring_rx.return_free(buf)
    yield dma_interaction_delay(self.packet_sz)
```

Listing 5.3: Excerpt of rx provide showing MMIO interaction and delay yields

5.1.3 Modelling virtualisers

To model the virtualisers we follow a transposition process similarly to the driver. There is a single point of divergence however: the driver's performance is principally bounded by IO and syscalls, while the virtualisers have meaningful delays from computation as it decides how to send packets to the correct clients and performs operations like cache cleans.

For example: the RX virtualiser performs a linear lookup to match the MAC address of incoming packets against a map of addresses to client queues. Considering that our business logic has no inherent cost, we can implement the lookup as a check against a hashmap in Python and yield a static delay that is proportional to the number of clients as this will amortise to match the real computational time.

5.1.4 Modelling the timer driver and protected procedure calls (PPCs)

Thus far, we have scarcely mentioned passive PDs or protected procedure calls (PPCs) within the scope of pysddfsim as they are mostly uninvolved in the queuing process and are only a minor component of the network device class.

Let us revisit PPCs and passive PDs before we proceed to model them. PPCs are effectively cross-PD function calls delivered by the seL4 kernel over an `Endpoint`. In the sDDF, PPCs are only used to interface with passive drivers and for minor configuration tasks at initialisation that are irrelevant for the scope of this thesis. PPCs to passive drivers have a unique property: they occur within the scheduling context of the calling PD. I.e. instead of invoking the scheduler, the passive driver will run using the budget of the calling PD.

The `FlexPD` abstraction assumes that PDs have some kind of active thread of execution and subsequently is not an ideal tool to model such drivers. We instead model passive drivers as a bespoke class type which offers a `ppc()` generator method which can be directly called by PDs. When a PD performs a `yield from` the passive driver, the passive driver performs whatever function is required and yields a delay which reflects the cost of performing the PPC and the time the passive driver spent executing.

The real sDDF timer driver is a very simple program which accepts a requested delay over PPC and notifies the client after that amount of time has passed. We can easily implement this by creating a temporary SimPy process for each PPC that yields a timeout for the specified delay and signals the caller PD when it exits, and then deletes itself.

5.1.5 Modelling copiers and LWIP clients

The copiers and clients are distinct from the other PDs in the device class as there may be many of them. We can easily model copiers with a similar method to the virtualisers in that we simply copy the queue-relevant business logic and insert delays proportional to what would be encountered in the real system. The `memcpy()` call in the copier is the only significant source of overhead in the copier, and we can model this trivially by assuming a 64-bit word size and a two cycle cost to copy a word before memory costs. Thus, the delay we yield is $2 \times \frac{\text{size}_{\text{packet}}}{8}$ plus memory delays from the `OverheadProfile`, given that packet sizes are in bytes and eight bytes are contained within a 64-bit word.

Clients pose a greater challenge: we need to model at least one accurate client PD for the purpose of establishing accuracy. The best target for this is the "echo server" test client, a program which accepts incoming packets and resends them to their sender. To model this program, we need to model the LWIP IP stack in the client. LWIP does not have an obvious order of execution, it operates as a free-running thread within the client that is driven by a continuation-based concurrency model. It periodically receives notifications from a timer driver which prompts the IP stack to process the list of packets it has available.

Free buffers only return to the queue once LWIP processes them, and LWIP does not do this as soon as they arrive. Similarly, when the client code attempts to resend a packet, it is stored in an internal list to be sent when LWIP is ready.

An important observation is that over the hundreds of thousands of packets the client will receive in a simulation run, the exact time when many of them move is mostly inconsequential to the driver stack itself. As long as the number of packets entering and leaving the client over an interval is correct, any error will amortise away. Thus, we can once again loosely approximate this behaviour for a sufficient model of LWIP.

To model the client, we add three LWIP-specific methods:

1. `lwip_process_rx`. Called whenever the PD is signalled by the copier and is responsible for ingesting packets and simulating the delays of LWIP processing them. Adds packets directly to the send list instead of handing them off to a receive callback as in the real client.

2. `transmit`. Called whenever the PD is signalled by the TX virt, prompting all packets LWIP needs to send to be dispatched.
3. `lwip_maybe_notify`. Called at the end RX and TX operations to decide if the RX or TX path should be signalled.

Whenever a timer notification arrives, the client simulates an arbitrarily-sized book-keeping delay. We don't need to add any additional logic as the seL4 Microkit signalling behaviour will cause the PD to be scheduled to be awoken and complete any outstanding work.

5.2 Tuning and accuracy

With a model composed, we must now establish that it is a sufficiently accurate for our purposes before we can proceed to analysis. This is nontrivial however — pysddfsim exists specifically to circumvent the difficulty in instrumenting real sDDF systems and as a result we are primarily limited to "black box" style testing. In previous work, the sDDF networking device class has been tested using an "echo server," a simple level 4 network program which accepts packets and redirects them back to their sender.

These echo tests supply us with valuable information through specialised testing programs which are able to record approximate CPU utilisation by each PD as well as throughput and latency. We can easily mirror these tests by transposing the echo server client into a `FlexPD` and using the pysddfsim analysis engine to gather results.

We trivially transpose the echo server using the method described in Section 5.1.5. Excluding the business logic used in the C code for recording results and performing other operations that are invisible to pysddfsim, the echo server is just a PD which stores incoming packets in a list and resends them when it can.

There is an immediate problem that we must face however: it is extremely difficult to reason about the correct magnitudes of delays within PDs from the real program. We cannot simply reason about the code as the compiled binary is likely to perform differently due to optimisation. We cannot disassemble the compiled program as a reference either, as we cannot easily determine runtime without predicting a variety of microarchitectural effects that are not visible to us from a software perspective.

Fundamentally however, this is not a problem for the scope of pysddfsim. To restate our goals, we wish to:

1. Analyse the efficacy of the signalling protocol and,
2. observe the relationship between top-level parameters and various effects visible at runtime.

That is to say, we are not seeking to predict the true performance of real systems! We just need our model of performance to be "good enough" such that it will predict the relative order of operations in a given sDDF system.

In order to do so, we can use a trial and error approach to tune our client PDs and `OverheadProfile` to match the effects visible to real echo server benchmarks. If we are able to set up a small set of delays within PDs that amortise to a facsimile of the real system, we have reached an adequate level of accuracy.

5.2.1 Qualifying accuracy

As we have established from Section 4.4.8, there are several variables in the overhead profile which we cannot gather from seL4bench. These are:

1. **Metadata interaction delay**: represents the delays that are likely to occur while interacting with shared memory such as cache misses. Internally models the probability of cache hierarchy misses with different variable costs.
2. **DMA interaction delay**: represents possible costs for interacting with DMA buffers.

In addition to these properties, there are always some delays that are a result of business logic. There is no easy way for us to determine these, even when using decompiled code as a reference. Fundamentally, the delays we add to pysddfsim to model the time business logic takes to execute are not strictly related to the number of instructions executed by a real system. This is because we are unable to model the microarchitecture of the target machine in pysddfsim in any general way. Branch prediction, instruction caching, variable length instructions and a gamut of similar hardware features are invisible in the current pysddfsim abstraction.

Fortunately, we do not need to accurately model any such delays because the amortised behaviour of sDDF systems is dominated by the costs of seL4 syscalls. We already have approximate measurements of these from seL4bench which are adequate for our purposes. Consequently, we just need to add delays in the correct place with approximately the correct magnitude such that our simulated system will scale similarly to a real system that we model. We can do so by inspecting the C source code and approximating notable delays, especially those that have dependencies on input data from queues or occur in loops.

Once we have inserted our arbitrary delays, we can tune them by comparison to a real system. The sDDF has been extensively benchmarked with the echo server for various research projects on many boards and we will rely on these results to inform our tuning process. These benchmarks are conducted using `ipbench` [Wienand, Ian and Luke Macpherson, 2004], a bespoke network testing tool for experimental operating systems. The full set of results can be found in Appendex A. These benchmark results in combination with the output of the echo server in benchmark mode give us the following critical metrics at a variety of test throughputs for several target boards, notably:

1. **Target benchmark throughput**. Total throughput of incoming packets that network load generators will attempt to reach.
2. **Received throughput**. Actual throughput achieved by load generators and received by test target.
3. **Send throughput**. Throughput achieved by echo server and returned to load generators.
4. **Total CPU utilisation**. Fraction of time core spent busy.
5. **Kernel CPU utilisation**. Represented in benchmarks as kernel cycles, but we can convert this to a utilisation by dividing it with the total number of cycles. Principally indicates the cost of context switches and system calls.
6. **Total cycles per packet**. Mean number of CPU cycles required to process one packet by the whole system.
7. **Total CPU utilisation by PD**. Fraction of time CPU time used by the driver, virtualisers, copiers and clients.

If we are able to make pysddfsim match these metrics to a reasonable degree at various speeds, we know that our system is doing the same amount of work per unit of data in each PD and must model delay times to a reasonable degree of accuracy!

A heuristic "guess and check" method is the best approach we have access to without requiring extremely complex analysis of the runtime characteristics of real systems. Such complex analysis would require system profiling hardware and other resources that are not available for the scope of this thesis.

As we perform our tuning, it is critically important that we never introduce arbitrary new delays or change the frequency of any extant delays. We can only alter the magnitude of our arbitrarily selected delays.

5.2.2 Tuning the i.MX8MM OverheadProfile

The i.MX8MM is a single-board computer that was the primary sDDF benchmark target for several years. It is reasonably fast, doesn't face memory bandwidth limitations and thus is an ideal target for pysddfgen as the magnitude of unrepresentable delays is smaller than some other boards. For example, the ODROID C4 is another frequently tested board which faces memory bandwidth issues which are difficult to represent as they reflect as large periods of idle time.

The most important characteristic we wish to match is the CPU utilisation scaling of the system with respect to the incoming workload. If we can successfully match the amount of time taken at various throughputs, it stands to reason that we are mirroring the cost per packet and cost per batch reasonably well as two variables quickly diverge if they aren't balanced.

We can begin our tuning process by running seL4test to populate most of our overhead profile. The next step is to run an echo-server example with 1472 byte packets and a a range of target speeds up to $1Gb/s$. We use the largest packet size as this will emphasise the internal delays in PDs while minimising noise due to high packet volumes. In order to build this for the real i.MX8MM, we can simply run the default configuration of the echo example:

```
    $ make MICROKIT_BOARD=imx8mm_evk MICROKIT_SDK=/home/
lesleyr/lib/microkit-sdk-2.0.1 MICROKIT_CONFIG=benchmark
```

Listing 5.4: Building the echo example

We can then flash this image to an i.MX8MM board and run a test using ipbench:

```
    ipbench \
        --debug \
        --test latency \
        --test-args socktype=udp,bps=125000000,\
        size=1472,warmup=10,cooldown=10,samples=50000 \
        --port 8036 \
        --client vb02 \
        --client vb03 \
        --client vb04 \
        --client vb05 \
        --test-target 172.16.1.145 \
        --test-port 1235 \
        --target-test cpu_target_lukem \
        --target-test-hostname 172.16.1.145 \
        --target-test-port 1236
```

Listing 5.5: Executing an ipbench test using tester systems vb01-vb05

We get most of our results as a CSV output from ipbench, and the remainder are printed over the serial console by the echo server itself. A notable result is the graph of CPU time used by PDs and the idle CPU time, rendered in Figure 5.1.

We repeat this testing process in 100Mb/s intervals up to 1Gb/s.

With our results collected, the next step is to run pysddfsim with our arbitrary values set. We immediately can observe a problem: seL4test only returns the time taken for a `seL4_RecvReply()`, not a raw `seL4_Recv()`! The time to do a `seL4_RecvReply()` is useful for modelling PPCs, but it is far too slow for modelling the delay for a PD to wake on an activated notification object. As a result, we approximate the time of a `seL4_Recv()` to be one third of a `seL4_RecvReply()`.

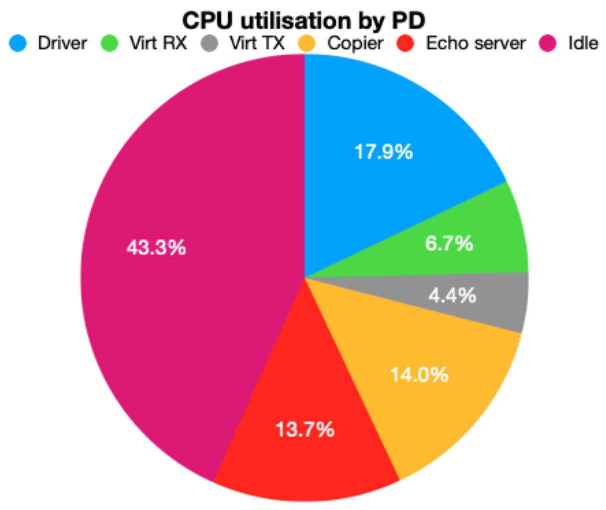


Figure 5.1: ipbench results: CPU utilisation at 1Gb/s with 1472 byte packets

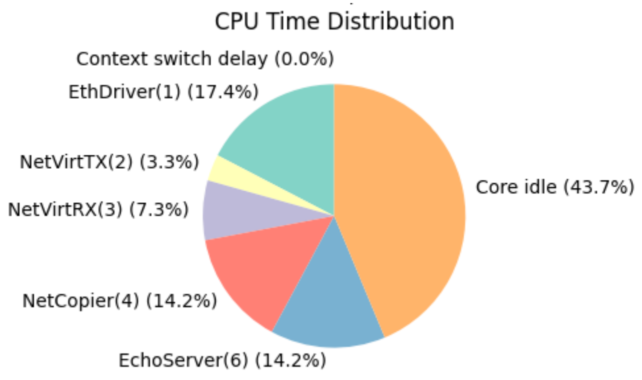


Figure 5.2: pysddfsim initial results: CPU utilisation

We then run our test using the command in Listing 5.6.

```
$ python echo_analysis.py 300000000 results/imx8mm_stdecho.
    sqlite3 imx8mm --protocolbudget 20000 --driverbudget 100 --
    driverperiod 400 --clientbudget 20000 --client_echo 2 --
    client_rx_dist 1 0 --interactive
```

Listing 5.6: Running an echo system mirroring the sDDF echo example

The results show that our baseline arbitrary values are fairly close: this is expected since we have already correctly configured the costs of signalling and syscalls. We achieve a near-identical throughput and our distribution of CPU times is very close, as shown in Figure 5.2. There are some inaccuracies however: the RX virt, copier and client are slightly too slow, the TX virt and driver are slightly too fast. This suggests that we must reduce the cost of interacting with the shared memory regions from `OverheadProfile.dma_interaction_delay` to accelerate the slower PDs.

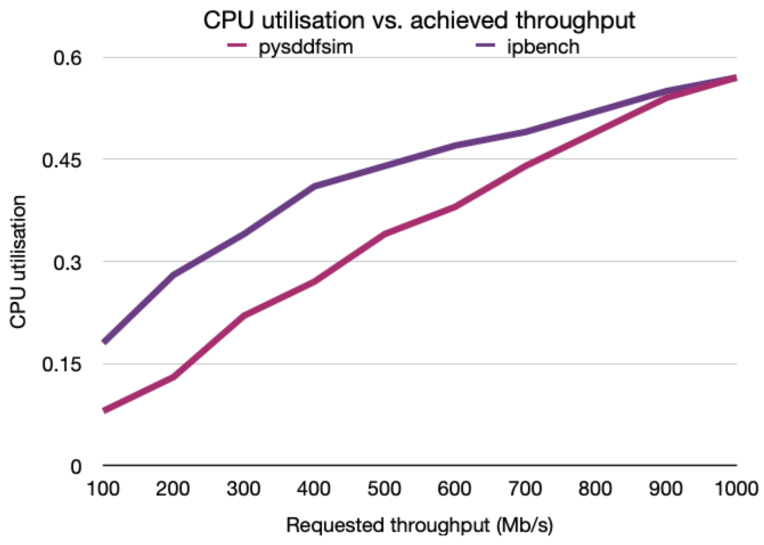


Figure 5.3: Comparison of CPU utilisation between ipbench and pysddfsim

We can take a much more efficient approach to tuning than using a single test run. The network test runner has a throughput sweep function which we can use to quickly iterate tests with multiple throughputs. A throughput sweep can be invoked as shown in Listing 5.7, yielding a GUI dashboard showing the throughput of each run and the distribution of PD CPU time for each as shown in Figure 5.2.

```
$ python echo_analysis.py 250000000 \
    results/imx8mm_accswp.sqlite3 imx8mm\
    --protocolbudget 20000 --driverbudget 100 --driverperiod
   400\
    --clientbudget 20000 --client_echo 2 --client_rx_dist 1 0\
    --throughputsweep 250m 500m 750m 1g
```

Listing 5.7: Running a throughput sweep with pysddfsim for overhead tuning

Performing sweeps allows us to evaluate the impact of overhead changes across the spectrum.

5.2.3 Evaluation of accuracy

Performing a throughput sweep with increments 100Mb/s up to 1000Mb/s, we are able to match the scaling curve of the real system to a degree as in Figure 5.4.

We are not able to mirror an efficiency drop in the real system in the middle of the throughput range. Figure 5.3 shows a divergence in CPU utilisation in the middle of the range as pysddfsim overperforms. Evaluating the output from ipbench reveals the reason: there is a large spike in TLB and cache misses in the middle of the range as shown in Figure 5.5. The current overhead model effectively is a simulation of the

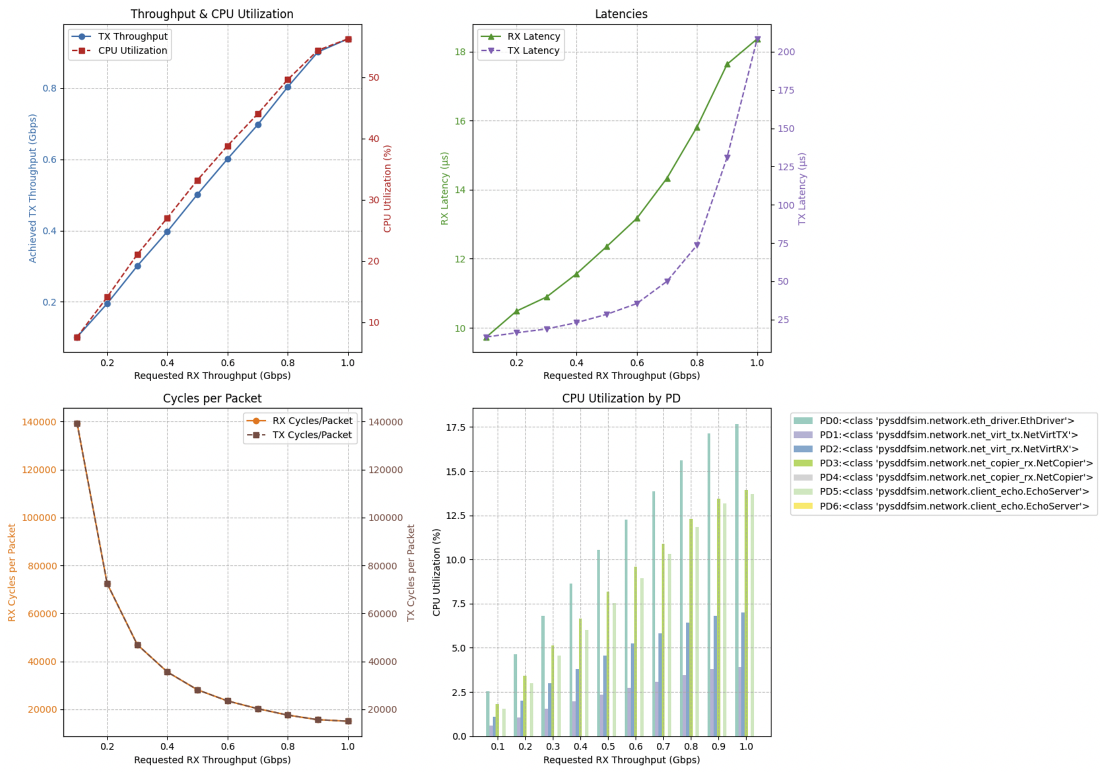


Figure 5.4: Network tester throughput sweep GUI output showing results of tuning

D-cache only and we fail to capture the I-cache and I-TLB having a decaying miss rate with throughput. In the real system, this is likely a byproduct of larger batch sizes. As PDs run longer and perform more work per run, their code remains resident in cache longer and an efficiency gain can be observed.

Ultimately, this is not a significant problem for the scope of this thesis. It simply means that our bound of accuracy is limited to a specific frequency range. In Section 7.4 we discuss possible future methods to improve the pysddfsim overhead model such as adding a simulated notion of instructions. Such a change would not require significant reworking of pysddfsim, but time constraints made it impractical to implement such a model for the scope of this thesis given that it would not meaningfully assist us in the analysis we wish to perform.

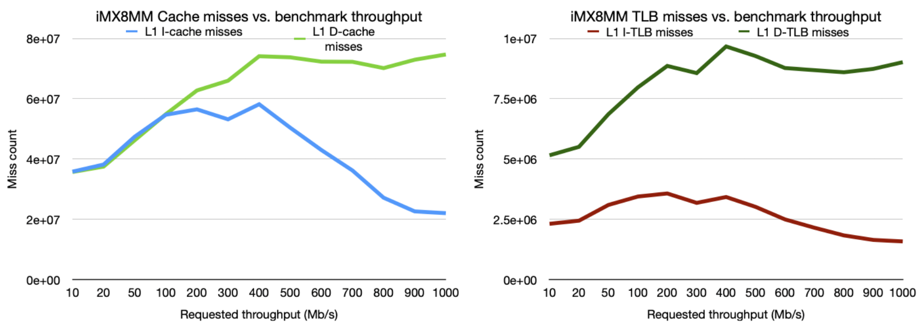


Figure 5.5: i.MX8MM cache and TLB misses against requested benchmark throughput

Chapter 6

Evaluation

In this chapter, we will apply our model of the network driver protocol to explore a variety of behavioural aspects of the sDDF. We will first evaluate how design parameters such as queue sizes impact performance, then delve into the impact of traffic patterns and malicious clients.

6.1 sDDF Parameters

A primary goal for this thesis is to quantify the impact of the standard parameters available in all seL4 Microkit and seL4 Device Driver Framework systems. In current sDDF examples, the available parameters are set on an empirical basis. We can use pysddfsim to determine the impact of these parameters on any given system. We will find some efficient parameters as a part of this section, but determining true optimal values is out of the scope of this thesis.

These parameters are:

- **Queue size**. Sizing of queues within the driver stack and between the driver stack and clients.
- **Budgets and periods**. Typically, a budget and period of 20000 μs is set for most PDs but this value is set arbitrarily. The network driver is often tested with a period of 400 μs and a budget of 100 μs, and this is once again arbitrary.

All of these parameters have a direct impact on the execution order of the graph. Budgets and periods limit the fraction of time PDs can run for directly, while queue sizing places an upper bound on the amount of work a given PD can do at any one time. In both cases, it is difficult to anticipate the net effect of changing parameters. The most significant way these parameters can affect PDs is by changing the average

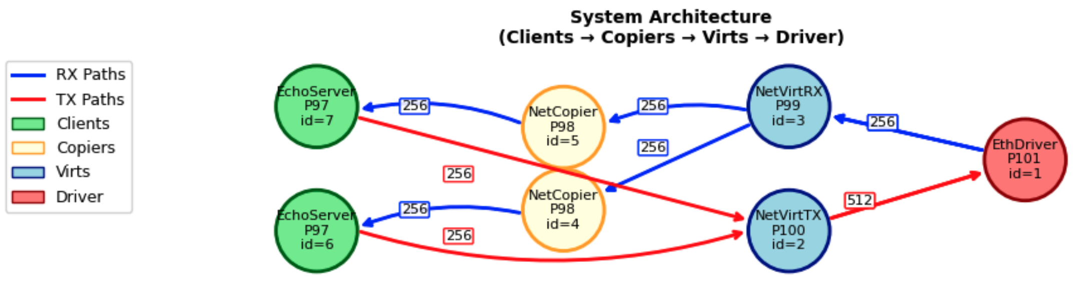


Figure 6.1: Architecture of the standard echo example from the pysddfsim GUI

batch size, the number of packets a given PD handles per wakeup. We ideally want to maximise batch size as the cost of waking up a PD due to signals or awaiting an `seL4_Recv()` is significantly larger than the cost of doing work for the trusted PDS! We cannot make the batch sizes too big however as latency can become excessive, adversely affecting protocols such as TCP.

6.1.1 Typical PD execution

To gain some control data, we can use the echo server examples from Section 5.2 and analyse the system via the queues view in the GUI and the execution statistics in the per-PD view.

We should note an important feature about the networking class before proceeding with our analysis: only the queues that contain DMA buffers threaten to violate non-interference between PDs. This is because of the finite number of RX DMA buffers and limited space in the TX virt—driver queue respectively. The queues between the client and the TX virt or the copier are generally ”safer” in that we can rest assured that they cannot do anything other than move data to or from a single client.

We can run our baseline test to mirror a very common system configuration: the i.MX8MM echo server test as used in Section 5.2. We can repeat the 1472 byte packet case at 1 Gb/s using the invocation in Listing 6.1, yielding a system with two echo servers but with only one server receiving packets. The architecture of the system is rendered by the pysddfsim GUI in Figure 6.1.

```
    $ python echo_analysis.py 300000000\
    results/imx8mm_stdecho.sqlite3 imx8mm\
    --protocolbudget 20000 --driverbudget 100\
    --driverperiod 400 --clientbudget 20000\
    --client_echo 2 --client_rx_dist 1 0 --interactive
```

Listing 6.1: Invoking the network tester for an echo server test

We use the `--interactive` flag to automatically open the analysis GUI as soon as the simulation completes.

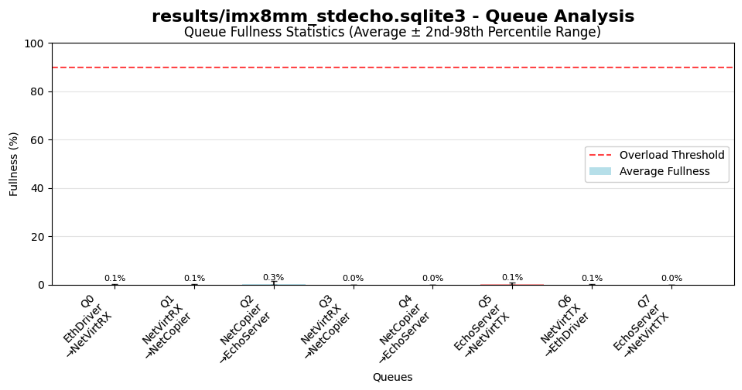
Navigating to the queues page in the GUI as shown in Figure 6.2, we can notice several critical features:

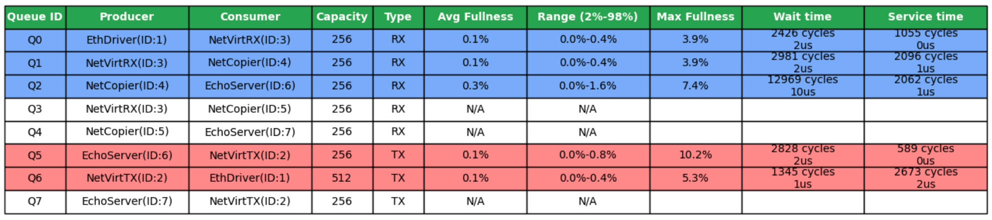
1. In a stable system, the average queue fullness of all queues is very close to zero as packets are moved out of the queues rapidly after entering the queues.
2. The RX queues rarely fill up as the RX path is able to process packets must faster than they arrive. We observe a mean fullness of 0.1% for the driver and virtualiser queues, or 2.56 packets with the 256 entry queues as configured. We attain a maximum fullness of 3.9% for these queues, corresponding to approximately 10 packets. We can reason that the maximum batch size this system achieves is ten packets within the driver stack.
3. The RX virtualiser-to-driver queue and copier-to-RX-virtualiser queues have identical fullness measurements, indicating that the RX virtualiser is able to move a whole batch to the copier every time it wakes.
4. The TX queues have a marginally higher maximum fullness of 10.2% and 5.3% for the client-to-virtualiser and virtualiser-to-driver queues respectively. As we can recall, the TX virtualiser queue is sized equal to fit all buffers from all clients, and in this example we have two clients. Thus the queue for the echo server receiving packets and the virtualiser queue both contain a maximum of 27 packets.
5. The larger TX burst size is expected given that clients only signal the virtualiser to transmit data once they have either run out of data to send or free packets. RX data arrives constantly, pre-empting the transmission process and ensuring that the client always has a larger number of buffers to transmit.

Graphing the service time of each queue in Figure 6.3, we observe that the virtualisers are significantly faster than the client, copier and driver. The service time of a queue is the average rate of packet removal by the queue consumer and is indicative of the speed at which the consumer can process data. Service time is a very useful metric as it encapsulates the average time to handle a packet inclusive of all delays from scheduling, preemptions and other system effects.

We can explain the lack of queue fullness in a fast-flowing example with some simple arithmetic: packets arrive with a period of $T_{\text{net}} = \frac{1}{(1.0\ \text{Gb} \div 8) \div 1472} = 11.8 \mu s$, but the total time to service a packet after it leaves the driver is equal to $((1044 + 2087 + 2063) \times \frac{1}{1.2\text{GHz}} = 4.3\ \mu s$.

Given that $4.3\ \mu s < 11.8\ \mu s$, it makes sense that queues are emptied extremely quickly. This begs the question however: why do we ever accumulate batches as large as 10 packets on the receive path if we consume packets faster than they arrive? The answer must be related to the driver as it is the only possible variable that may deliver batches. We will inspect this behaviour in Section 6.1.3.





Queue ID	Producer	Consumer	Capacity	Type	Avg Fullness	Range (2%-98%)	Max Fullness	Wait time	Service time
Q0	EthDriver(ID:1)	NetVirtRX(ID:3)	256	RX	0.1%	0.0%-0.4%	3.9%	2426 cycles 2us	1055 cycles 0us
Q1	NetVirtRX(ID:3)	NetCopier(ID:4)	256	RX	0.1%	0.0%-0.4%	3.9%	2981 cycles 2us	2096 cycles 1us
Q2	NetCopier(ID:4)	EchoServer(ID:6)	256	RX	0.3%	0.0%-1.6%	7.4%	12969 cycles 10us	2062 cycles 1us
Q3	NetVirtRX(ID:3)	NetCopier(ID:5)	256	RX	N/A	N/A			
Q4	NetCopier(ID:5)	EchoServer(ID:7)	256	RX	N/A	N/A			
Q5	EchoServer(ID:6)	NetVirtTX(ID:2)	256	TX	0.1%	0.0%-0.8%	10.2%	2828 cycles 2us	589 cycles 0us
Q6	NetVirtTX(ID:2)	EthDriver(ID:1)	512	TX	0.1%	0.0%-0.4%	5.3%	1345 cycles 1us	2673 cycles 2us
Q7	EchoServer(ID:7)	NetVirtTX(ID:2)	256	TX	N/A	N/A			

Figure 6.2: Queue fullness analysis from pysddfsim GUI Queues view

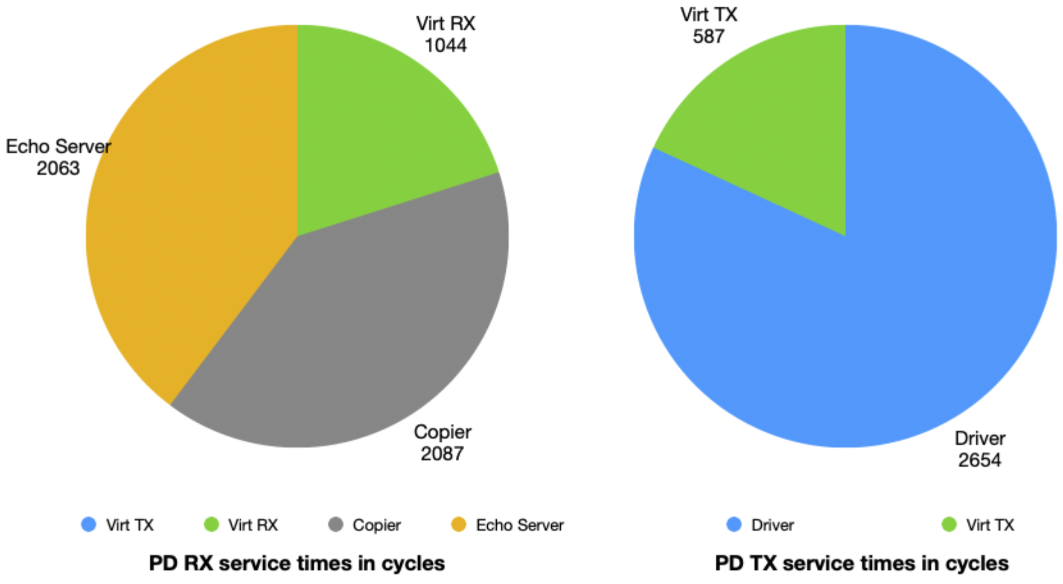


Figure 6.3: Service time by PD simulating the i.MX8MM echo server example

The case of a single echo server is far from reflective of the sDDF in general however! The client ideally will receive and resend only a small number of packets every wakeup and this prevents anything unexpected from happening.

Let us consider a transmit-only case with two clients of equal priority and the privileged PDs retaining their standard priority configuration such that the driver, TX virtualiser, RX virtualiser and copiers are in descending order of priority. These clients transmit at a high speed and busy wait in-between packets. We run this test on the i.MX8MM model once again. We can observe that the i.MX8MM has a NIC with ring buffers containing 128 packets, limiting the maximum number of packets the driver can take or put into the device at any one time.

The `SenderClient` class we use for this test is parameterised with a target speed. It will send packets whenever it has free buffers and will busy wait such that it will emit packets according to a specified rate. In this test, we do not want to overload the system by using all CPU time so we configure clients to transmit at 3 Gb/s such that they will not run out of CPU time. We should note that this is a relatively slow transmission rate relative to to the service time of the echo server.

We set up our test system with the following test runner invocation:

```
$ python echo_analysis.py 5000000000 results/imx8mm_2tx_fast.sqlite3\
 imx8mm --protocolbudget 20000 --driverbudget 100\
 --driverperiod 400 --clientbudget 20000\
 --client_txonly 3g 3g --interactive
```

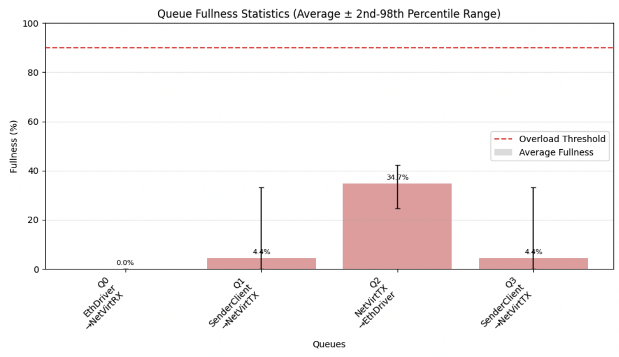
Once the simulation completes, we observe the following:

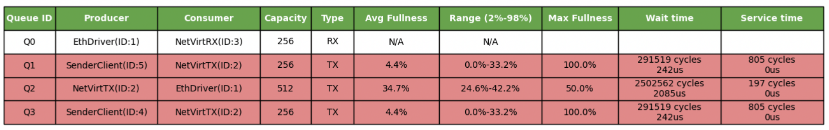
1. Clients produce at a high rate until their TX queue to the virtualiser is full or they are out of free buffers, at which point they signal the TX virt to pass the data on. TX queues in the system are exercised as the PDs infrequently signal the TX virt and all TX queues in the system are able to meaningfully fill up as we can see in Figure 6.4.

2. Although the senders have 256 free buffers to send, they are unable to use all of them or achieve a maximum of 100% queue fullness reliably as indicated by their 98th percentile result of 33.2% fullness. This corresponds to 84 packets and is substantially lower than the maximum batch size.

3. The senders emit a packets every $\frac{1472\text{ B}}{3\text{ Gb}} \approx 3.9\ \mu s$ and have a 20000 μs budget. To send all 256 free buffers, the clients should take $3.9\ \mu s \times 256 \approx 1005\ \mu s$. This suggests that the smaller batch sizes are not due to budget exhaustion.

4. Using the GUI PDs page, we can find the number of pre-emptions and signals received for each PD. The clients send and receive a near identical number of signals, suggesting that the PDs are running out of free buffers as opposed to being pre-empted. Signal are received once every 286 μs of wake time for each





Queue ID	Producer	Consumer	Capacity	Type	Avg Fullness	Range (2%-98%)	Max Fullness	Wait time	Service time
Q0	EthDriver(ID:1)	NetVirtRX(ID:3)	256	RX	N/A	N/A			
Q1	SenderClient(ID:5)	NetVirtTX(ID:2)	256	TX	4.4%	0.0%-33.2%	100.0%	291519 cycles 242us	805 cycles 0us
Q2	NetVirtTX(ID:2)	EthDriver(ID:1)	512	TX	34.7%	24.6%-42.2%	50.0%	2502562 cycles 2085us	197 cycles 0us
Q3	SenderClient(ID:4)	NetVirtTX(ID:2)	256	TX	4.4%	0.0%-33.2%	100.0%	291519 cycles 242us	805 cycles 0us

Figure 6.4: Queue fullness analysis from pysddfsim for a system with two send-only PDs

PD, suggesting that each client has at most $\frac{256}{3.9} \approx 73$ free buffers every wake on average.

5. The sender clients are pre-empted roughly the same number of times as they send signals, further confirming our hypothesis that the PDs are not time limited or being interrupted.

On its face, this is a strange situation given what we know about the queuing network. Evidently, there is some constraint that is not from the senders or the virtualisers.

We can inspect this batching behaviour more closely by utilising the pysddfsim logging feature for quick and dirty tests. The same result may be achieved with the analysis engine directly, but in such cases the logging infrastructure is a simpler alternative and we can use this as an illustrative example for the power of the logging subsystem. We can add a print statement to the `SenderClient` class which states the number of free buffers available at the start of each send loop:

```python
    self.log("STARTING SEND")
    while not self.tx_queue.free_empty():
        self.log(f"Sending a packet. Free buffers left = {len(
  self.tx_queue.free)}")
```

Listing 6.2: Using the pysddfsim logging utility to amend the SenderClient

We can then save the output of a test run to a file and parse the output with `grep` and `awk`, standard commandline utilities on UNIX systems as shown in Listing 6.3.

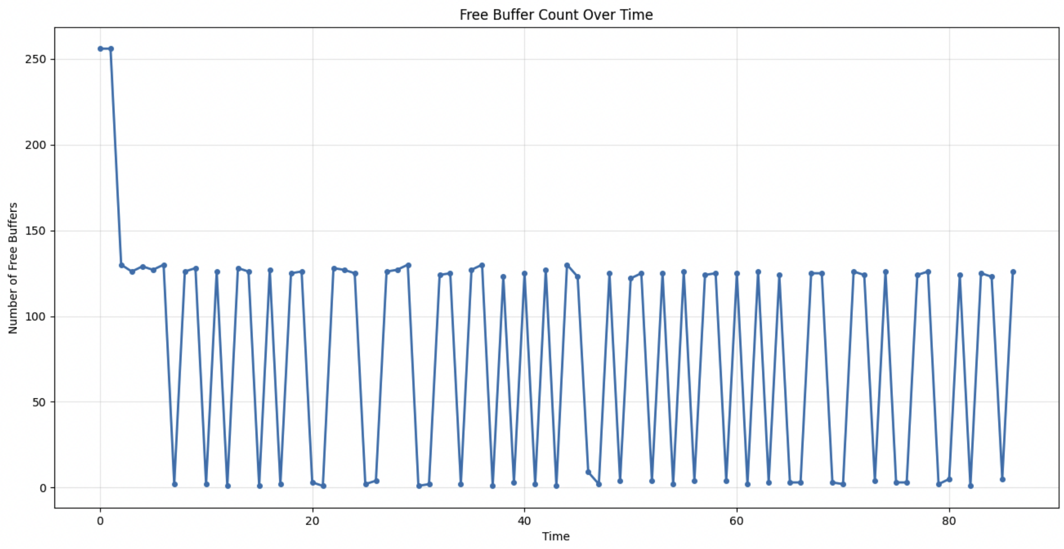


Figure 6.5: Number of buffers available to the sender client per wake up

```
 $ python echo_analysis.py 9000000000 results/
imx8mm_2tx_fast_1.sqlite3 imx8mm --clientbudget 20000 --
client_txonly 3g 3g --log > sender.log
 $ cat sender.log | rg "STARTING SEND" -A=1 | rg "Free
buffers left" | awk '{print $NF}' > sender_client_fullness.
txt
```

Listing 6.3: Running the test, saving the output to a file and creating a list of values

We can graph the data of Listing 6.3 with a short test run to create a graph of the initial number of free buffers at each PD wakeup, as depicted in Figure 6.5. This graph yields a surprising result: the clients alternate between having 128 buffers and single-digit quantities. As we can recall, the NIC has a 128 packet ring buffer and this is not a coincidence! The behaviour we observe is due to the finite size of the DMA ring of the device. Every time the driver is signalled by the virtualiser, it is only able to accept 128 out of the 512 buffers in the system.

Using the 2nd percentile fullness of the virtualiser-driver queue, we can conclude that there are approximately 127 packets trapped in the queue at all times. Since the driver only is able to process 128 packets at a time, it can return batches of at most 128 free buffers to the the virtualiser. The virtualiser then signals each client to return the free buffers. Given that the two senders take turns executing, we finally arrive at an explanation of the oscillatory behaviour we observe in Figure 6.5. Clients alternate between receiving a large batch of about 127 free buffers and a small batch of 1 buffer because this is the maximum batch that is released by the NIC and the consumption of packets by the driver is slightly out of phase with the execution of clients, leading to a small overlap in the packets consumed.

From these two examples, we have learned two important lessons:

1. The behaviour of sDDF systems can very significantly based on workload.
2. The behaviour of the driver and device cause dominating effects.

The remainder of this chapter will analyse the impact of compile-time parameters on the network device class, considering the impact of the clients, trusted PDs and device.

6.1.2 Scheduling parameters for virtualisers

As we have already established from Section 6.1.1, none of the trusted PDs typically run for any meaningful fraction of the default budget. The driver and copiers are the only PDs that are eligible to be meaningfully affected by budgets and periods as the driver has a possibly unbounded amount of work and the copiers are duplicated. Before we inspect the driver and copiers, let us quantify the relationship between virtualisers and the scheduling parameters.

Virtualisers typically have a budget and period of 20000 μs, but also typically have exclusive priorities. We can immediately observe that their scheduling parameters have no impact on a PD with a unique priority as there is no other equal-priority thread to switch to if the budget is consumed. Instead, the budget will immediately be refilled and they will act identically to a PD with infinite budget as there are no other PDs in the round robin queue!

In the event that the virtualisers did not have exclusive priority, the current budget settings are so large as to have no impact. We can perform some trivial arithmetic to prove this. Consider our network stack executing on a single core on a system with a 1.0GHz processor which receives and transmits packets at 1 GB/sec with 1472 byte packets.

A clock period is $\frac{1}{1.0\ GHz} = 1\ ns$ and packets arrive with a mean period of $T_{\text{net}} = \frac{1}{(1.0\text{Gb} \div 8) \div 1472} = 11.8\ \mu s$. Using the service time results from Figure 6.3, the virtualisers are able to process packets two orders of magnitude faster than the NIC produces them! Even a full queue of 512 packets can be processed in a small fraction of the budget.

Thus, we can conclude that the current scheduling parameters for virtualisers should have no impact under any circumstances. Furthermore, the network virtualisers are unlikely to ever need scheduling parameters applied, even in the event that they share time with another driver class. Service times are short and the virtualisers are always made runnable immediately upon being signalled and consequently should always respond to data immediately. We should note that the signalling behaviour that ensures signals make the virtualisers run is currently classified as a bug [Zupancic, Indan, 2022], so further thought may be required for virtualiser scheduling parameters in the future. That is to say, the current settings do not need changing despite their irrelevance.

We can validate this using pysddfsim with a simple experiment: we use an unbudgeted set of virtualiserss as our control case and vary budgets down from 20000μs.

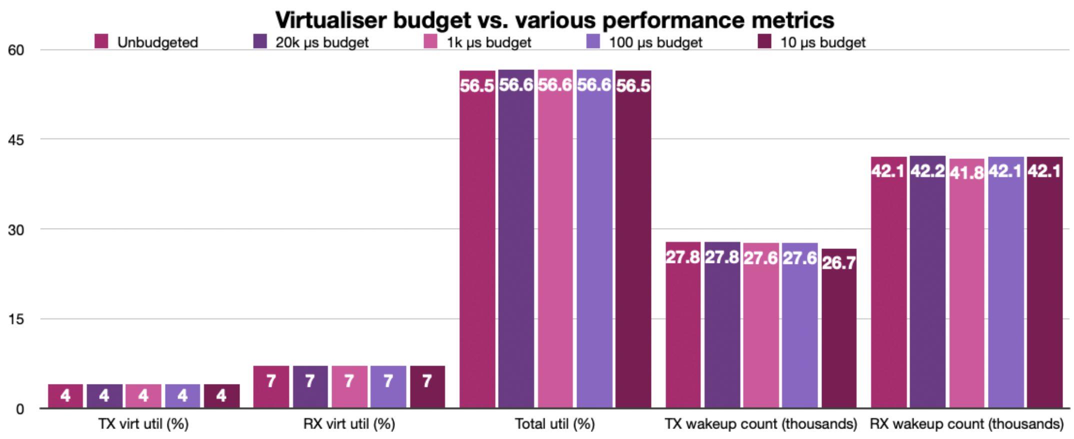


Figure 6.6: Comparison of various virtualiser budgets in microseconds

The results are shown in Figure 6.6, showing the CPU utilisation of the two virtualisers and the whole system, as well as wake up counts for the two PDs. The wakeup counts should indicate if the virtualisers ever get put to sleep by the scheduler. As predicted however, there is no impact to the budget setting as the PDs never run for long enough to encounter the budget.

6.1.3 Scheduling parameters for drivers

Unlike virtualisers, we cannot neglect scheduling parameters for the driver. As the highest-priority PD in the driver class, the driver is capable of pre-empting all other PDs as it receives interrupts from RX packets. If we allow PDs to be pre-empted too often, we reduce the maximum number of buffers they can process in a batch and reduce efficiency. In the upstream version of the sDDF at the time of writing, drivers have a period of 400 μs and a budget of 100 μs, bounding their execution time to one quarter of total CPU time.

The batch size of the driver itself can also be controlled directly using the scheduling parameters. Assuming a constant rate of arrival, the time the driver spends awaiting a budget refill allows packets to accumulate which may all be processed quickly once the budget is replenished. We should be careful when doing so however, as an excessively large budget can cause batching to only occur infrequently and thereby mitigate some of the performance gains.

To evaluate the impact of the scheduling parameters on the driver we can run a series of experiments in pysddfsim with different budgets and periods. We also require a control run with an unbudgeted driver, and both test invocations are shown in Listing 6.4. We reduce the packet size to 368 bytes and use a constant traffic distribution to emphasise the impact of the driver in the system and leave the configuration of the rest of the system as is standard for the echo server example.

```
$ python echo_analysis.py 9000000000 results/
    imx8mm_stdecho_368.sqlite3 imx8mm --protocolbudget 20000 --
    driverbudget 100 --driverperiod 400 --clientbudget 20000 --
    client_echo 2 --client_rx_dist 1 0 --packet_sz 368 --
    trafficgen constant
$ python echo_analysis.py 9000000000 results/
    imx8mm_stdecho_368budgetlessdriver.sqlite3 imx8mm --
    protocolbudget 20000 --driverbudget -1 --driverperiod -1 --
    clientbudget 20000 --client_echo 2 --client_rx_dist 1 0 --
    packet_sz 368 --trafficgen constant
```

Listing 6.4: Network test runner invocations for driver budget echo experiments

The results the test run in Figure 6.7 shows the performance overview of the unbudgeted system. System performance is suboptimal as the driver frequently interrupts the rest of the system. The driver CPU utilisation is very high and dominates the system CPU time, but the system still manages to echo all arriving packets.

Contrasting with the system with a 100/400 budget/period in Figure 6.8, we see a drastically different result. Despite the fact that we have bounded the time for the driver to execute, we see overall better performance. This is because packets continue to accumulate in the network interface as the driver is forced to sleep by its budget, allowing it to work through them all immediately upon wake. The cost of waking up the driver is far greater than the cost to process multiple packets, so this is a more efficient solution overall.

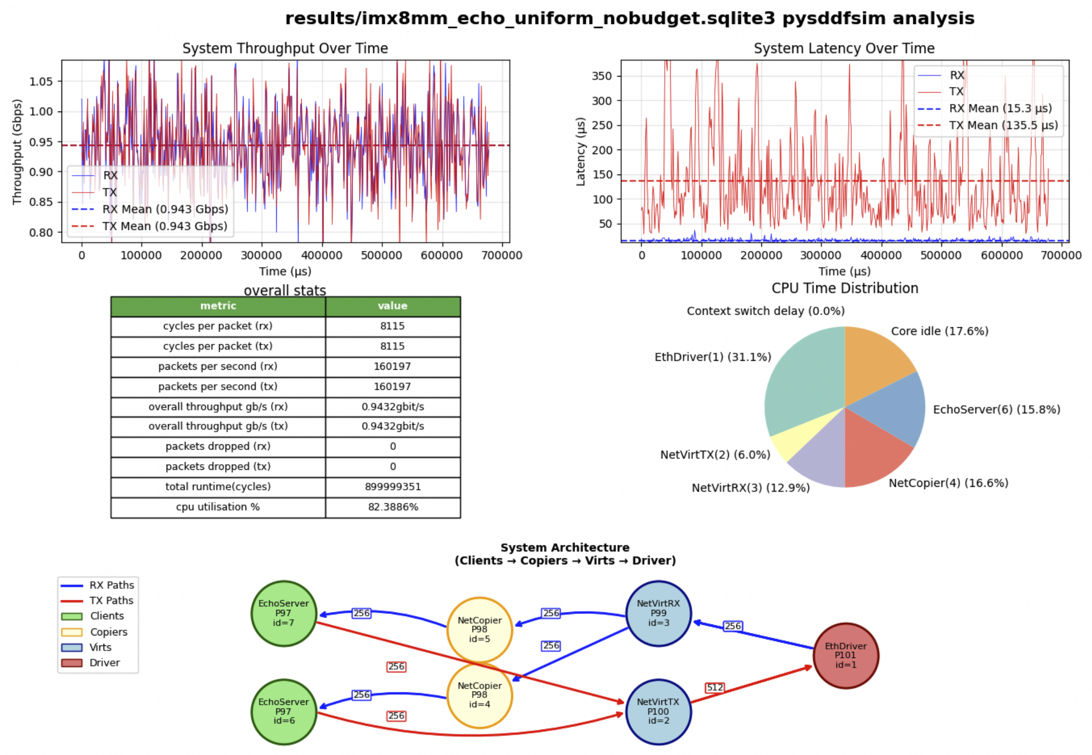
As we alter the budget and period, we can observe drastically different behaviours. Let's select a variety of values of smaller periods and budgets and use pysddfsim to inspect the results.

All systems manage to achieve the target throughput, so we should instead focus on the CPU utilisation as this gives us a direct measure of efficiency. An overview of results is presented in Figure 6.9. As we can see, an interesting trend emerges. Smaller period vs. budget fractions manage to allow us to drastically reduce CPU utilisation without a penalty to throughput! As the period vs. budget fraction decreases, the average system efficiency increases as more packets are processed in one batch.

Naively we would expect to see some clear point where throughput collapses as the driver no longer has enough time to do the work it needs to do. There is little use in randomly guessing values however, and we can analyse this algebraically instead.

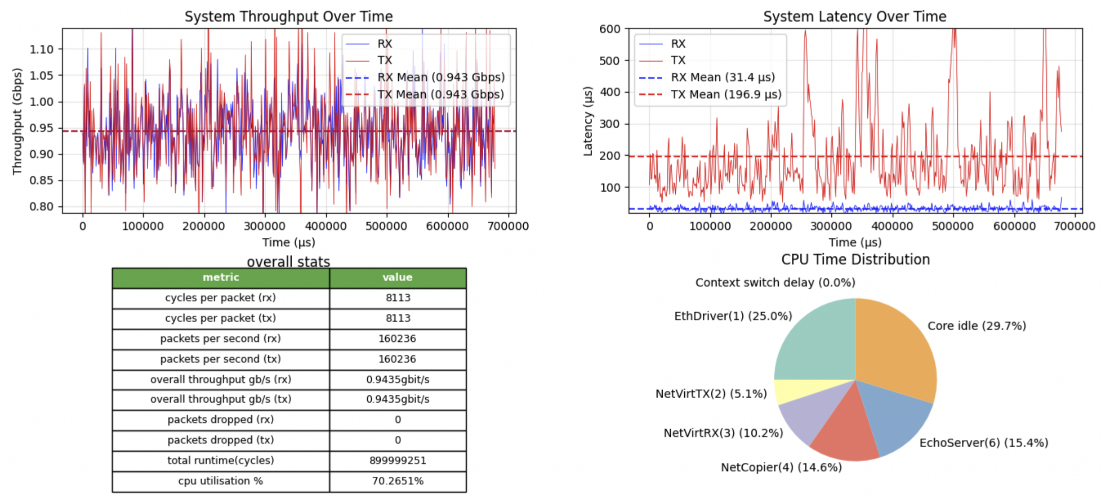
6.1.4 Tuning driver scheduling

As we see in the benchmarks in Section 6.1.3, the choice of scheduling parameters has a vast impact on the performance of the system. We do not have any clear strategy by which to select the parameters however, and we will address this problem in this section.



metric	value
cycles per packet (rx)	8115
cycles per packet (tx)	8115
packets per second (rx)	160197
packets per second (tx)	160197
overall throughput gb/s (rx)	0.9432gbit/s
overall throughput gb/s (tx)	0.9432gbit/s
packets dropped (rx)	0
packets dropped (tx)	0
total runtime(cycles)	899999351
cpu utilisation %	82.3886%

Figure 6.7: Overview of unbudgeted driver with uniform traffic and 736 byte packets



metric	value
cycles per packet (rx)	8113
cycles per packet (tx)	8113
packets per second (rx)	160236
packets per second (tx)	160236
overall throughput gb/s (rx)	0.9435gbit/s
overall throughput gb/s (tx)	0.9435gbit/s
packets dropped (rx)	0
packets dropped (tx)	0
total runtime(cycles)	899999251
cpu utilisation %	70.2651%

Figure 6.8: Overview of the driver with 100/400 budget/period

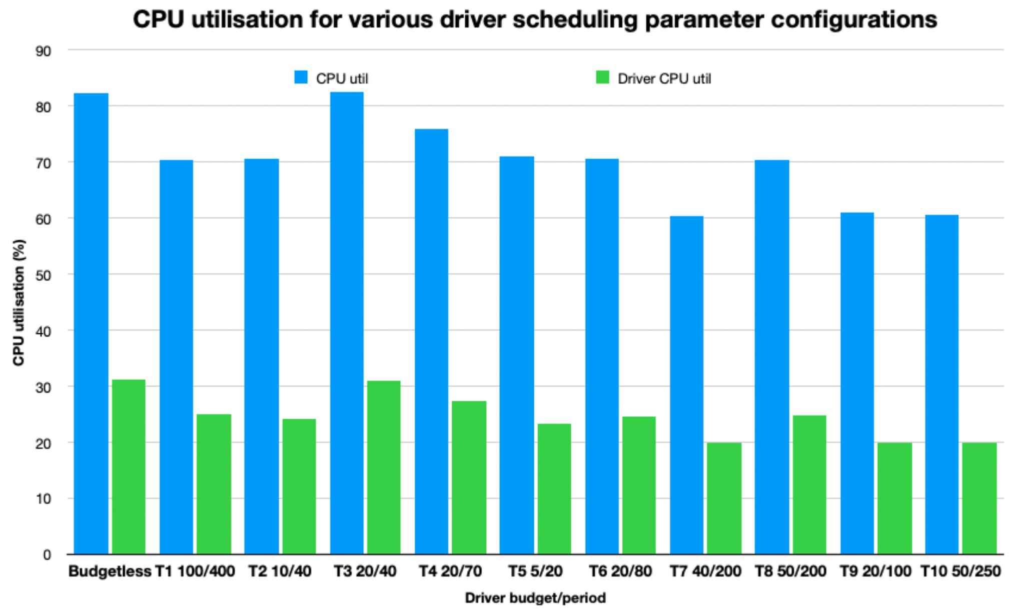


Figure 6.9: Results: CPU utilisations for different driver scheduling paramters with 368 byte packets

To start, let's consider the current sDDF scheduling parameters as a case study. Using the calculations in Section 6.1.2, we can calculate our expected batch sizes. Assuming a packet arrives every 11.8 μs and our system is configured with a period and budget of 400 μs and 100 μs respectively, we can expect $\frac{(400-100)}{11.8} \approx 26$ packets to accumulate in every sleep period.

Another $\frac{400}{11.8} \approx 8$ packets arrive over the 100 μs the driver is awake. We can approximate the time the driver takes to send a received packet to the virtualiser using the service time of the RX virtualiser as a lower bound using the results from Section 6.1.1. Thus, the driver should process the packets in approximately $26 + 8 \times 0.9\ \mu s \approx 30.6\ \mu s$.

The remaining 69.4 μs or $\frac{69.4}{400} = 17.7\%$ is time wasted. Every arriving packet will cause an interrupt and driver wake up, causing it to be sent to the virtualiser immediately. Other PDs will be excessively preempted by every packet in this window! We must carefully consider the average case when we set our period and budget.

We also must ensure that our fraction of time spent sleeping is not going to result in packets being dropped. For the i.MX8MM, the NIC has 128-packet ring buffers. This is not a problem with large packet sizes, but with smaller packets it can be problematic. With a packet size of 1472 we can idle for at most $128 * 11.8\ \mu s = 1510.4\ \mu s$, but with a packet size of 368 bytes we only have 377.6 μs of slack! There also should always be some leeway to cope with large bursts of packets.

With this simple math, we can construct a set of expressions to bound our period and budget. First, our sleep time must be bounded by the NIC RX buffer size such that there is always enough time to process a full buffer of packets:

$$\text{period} - \text{budget} < \frac{\text{size}(\text{packet}_{\text{max}})}{\text{throughput}_{\text{bytes/sec}}} \times \text{size}(\text{buffer}_{\text{rx}}) = T_{\text{packet}} \times \text{size}(\text{buffer}_{\text{rx}}), \quad (6.1)$$

where T_{packet} is the mean period of packet arrivals and $\text{size}(\text{buffer}_{\text{rx}})$ is the size of the RX ring buffer.

Next, we must bound our budget by the number of packets that arrive while sleeping. The number of packets that arrive over the sleeping period is given by:

$$\text{P}_{\text{sleep}} \approx \frac{(\text{period} - \text{budget})}{T_{\text{packet}}} < \text{size}(\text{buffer}_{\text{rx}}), \quad (6.2)$$

and our budget is thus bounded as:

$$\text{period} > \text{budget} > \text{P}_{\text{sleep}} \times S_{\text{driver}}, \quad (6.3)$$

where S_{driver} is the mean service time of the driver.

With these expressions, we now have a lower bound budget and upper bound for period. Let's now express the optimal budget for a given period.

For an optimal budget, we want to run for long enough to process just all packets that accumulated in the NIC as well as all packets that arrive while. We also need some degree of leeway to account for possible traffic spikes. The number of packets that arrive over the live period is given by:

$$\text{P}_{\text{awake}} \approx \frac{\text{budget}}{T_{\text{packet}}}. \quad (6.4)$$

Hence,

$$\text{budget} \approx (\text{P}_{\text{sleep}} + \text{P}_{\text{awake}}) \times S_{\text{driver}} + \text{T}_{\text{leeway}}, \quad (6.5)$$

but, P_{sleep} and P_{awake} can be expanded:

$$\text{budget} \approx (\frac{\text{budget} + \text{period} - \text{budget}}{\text{T}_{\text{packet}}}) \times S_{\text{driver}} + \text{T}_{\text{leeway}}, \quad (6.6)$$

Finally, simplifying:

$$\text{budget} \approx \frac{\text{period}}{T_{\text{packet}}} \times \text{S}_{\text{driver}} + \text{T}_{\text{leeway}}. \quad (6.7)$$

We cannot choose an optimal period without considering the workload. If we set our period such that batches will always fill the entire RX buffer, we will drastically increase latency! As a result, we should set our period based upon the maximum latency we wish to incur.

We can find the worst-case latency that our period will add to the processing of a packet easily: consider a packet that enters the NIC right as the driver goes to sleep. Assuming our budget is set appropriately such that all prior packets are processed while the driver runs, this packet will be transmitted after the driver wakes and has taken the time to move it into the virtualiser queue. Hence:

$$\mathrm{L(period)} = (\mathrm{period} - \mathrm{budget}) + \mathrm{S}_{\mathrm{driver}}. \tag{6.8}$$

A maximally sized period (equal to time to fill entire RX buffer) will ideally achieve the best efficiency, while smaller periods will have increasingly better latency.

Using this method, we can find an optimal budget for our i.MX8MM run with a period of 400 microseconds. We can arbitrarily set our leeway to be equal to $\mathrm{T}_{\mathrm{packet}}$. Instead of assuming the service time of the driver, we can find it precisely using the third interface to pysddfsim: the SQLite3 CLI. Any pysddfsim database dump can be opened using `sqlite3 dumpfile`. Reusing the echo example from Section 4.3.1, we can find the queue ID for the driver RX queue from the GUI easily and count the number of active events for this run as 24779.

```
  sqlite> select count(*) from QueuePutActiveEvents where
  queue_id=7;
  -----
  24779
```

We can then find the rate by dividing the driver runtime by the number of events, retrieving the runtime from the GUI PDs view. For this run, we find an execution time of 5.365×10^7 cycles and we know that the i.MX8MM is configured to execute at 1.2 GHz, hence the service time of the driver is given as:

$$(5.365 \times 10^7 \times \frac{1}{1.2\ \mathrm{GHz}}) \div 24779 \approx 1.8\ \mu s. \tag{6.9}$$

We can finally find our optimal budget for a period of 400 μs:

$$\mathrm{budget} \approx \frac{400\ \mu s}{11.8\ \mu s} \times 1.8\ \mu s + 2.23\ \mu s \approx 63\ \mu s. \tag{6.10}$$

To evaluate the efficacy of this calculation, let's test this configuration on a real system. As we can see from Figure 6.10, our predicted optimal budget beats the standard echo result by a significant margin as we achieve the best throughput measured thus far with this microbenchmark on the i.MX8MM.

6.1.5 sDDF and Microkit parameters for Copiers

Unlike the other trusted components, we have more room to tinker with the receive copiers as they are not in the critical path for multiple clients. Instead, there is one

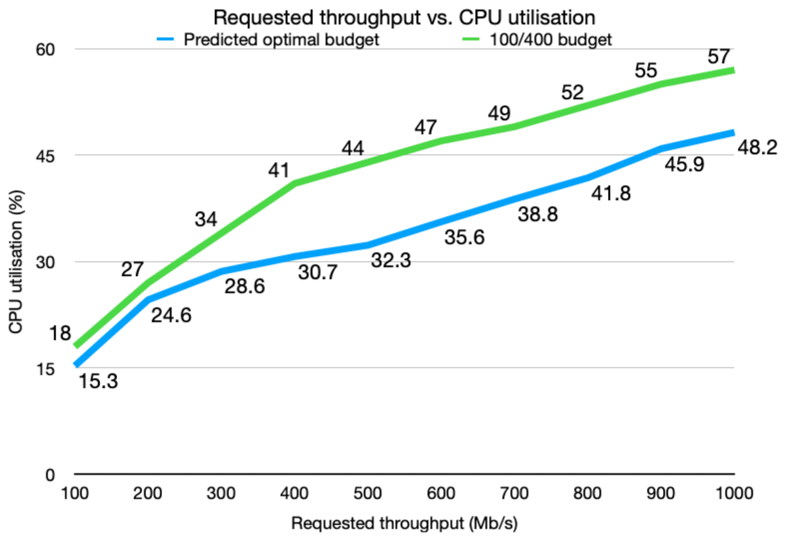


Figure 6.10: Comparison of ipbench results for the standard period on the i.MX8MM

copier per client. Every client which is able to receive packets from the network stack has a copier mediating the receive process. As packets arrive, the copier removes their data from the DMA buffer and places it into a copied (non-DMA) buffer for usage by the client.

Similarly to the virtualisers, copiers typically have a budget and period of 20000μs and a shared priority that is higher than all clients. Unlike the virtualisers however, the budget does matter as there are many of them and they are all of equal priority, meaning that it is possible for a copier to starve other copiers if incorrectly configured.

Copiers could be configured to have different priorities based upon their clients but there is little obvious reason to do so. In such a situation, RX buffers can become trapped in copier queues if a busy copier starves lower priority copiers of time. This can be managed by the RX virtualiser by ”rewinding” packets out of certain copiers, but there is no obvious advantage to such a strategy as compared to allowing equal-priority copiers to drop packets immediately as needed.

An important property of the seL4 scheduler is that any PD that is notified will be moved to the front of the round-robin queue. This has an important consequence: when we signal a PD which shares a priority level with other PDs, it will always be made runnable unless it is out of budget and has no refills available. When loads are balanced, this is not problematic, but it can possibly become an issue with asymmetric loads. As a higher-traffic copier can be signalled frequently enough that the lower-traffic copier may never be able to make progress. We explore this problem in Section 6.2 as this is a fundamental problem with the current design and exploit this problem in Section 6.3.1.

Even when ignoring the hazards of the signalling order, the 20000 μs budget is too large. Using the service times for the i.MX8MM in Figure 6.3, we can calculate that the copier takes $\frac{1}{1.2\ GHz} \times 1104\ \mu s = 0.92\ \mu s$ to move one packet to the client. Assuming a typical queue size of 512 entries, the copier is able to process the entire contents of the queue in 471 μ, or 2.3% of our budget! On systems with higher clock speeds, this issue will be more significant.

We can also consider altering the size of the copier-client queues because they do not contain DMA buffers and thus can be as large as necessary. The copier-client queue is effectively just a client-owned buffer. It stands to reason that the copier-client queue should be at least as large as the other RX queues, but its size may be increased to support clients who process packets slowly and may receive large bursts. Were we to make such a change however, there would be no impact on system performance other than reducing the number of dropped packets for a client with an eligible traffic pattern.

6.1.6 Virtualiser queue sizes

Virtualiser queues have a particular configuration as a part of the sDDF design specification:

- RX queues are sized identically to the client queues to ensure copiers can accept maximally-sized batches of packets, since client queue sizes match the number of RX buffers in the system.
- TX queues are sized to equal the sum of all client TX queues to prevent interference due to queue fullness as multiple clients transmit.

The queue sizes are both set to optimise transmission of DMA buffers. The RX queues cannot exceed the size of one client queue or it is likely that a single batch of RX buffers can overflow the copier and cause many packets to be dropped unnecessarily. All TX buffers are "owned" by clients and every client has a full queue worth of buffers. Each client has an isolated view of the transmit path.

Considering that there is no room to resize the virtualiser queues without a corresponding change to the client queues, we can consider them to be a dependent variable that is set by the clients and there are few remarks to make about queue sizings with respect to the virtualisers themselves.

6.1.7 Impact of system queue sizes

As we have established in the previous subsections of this section, queue sizes are an important variable but are largely not a factor that can be used to tune specific trusted PDs due to structural restrictions. This subsection will evaluate the impact of queue sizing on the entire system.

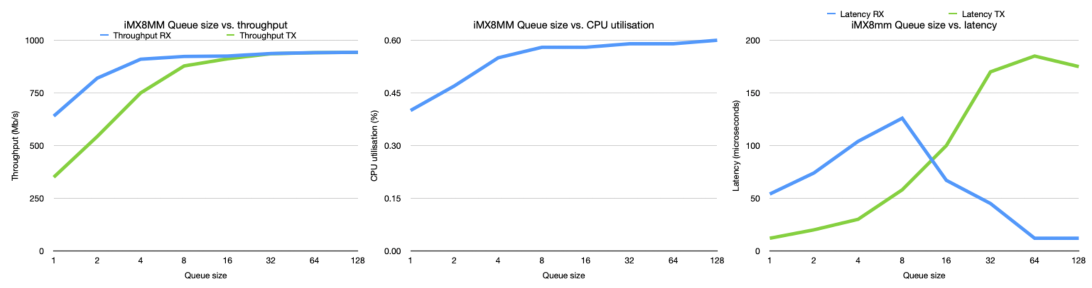


Figure 6.11: Echo server results while varying queue sizes

First, let us establish a reasonable lower bound for the queue size by evaluating the impacts of very small queues in systems that are not CPU bound. Using the i.MX8MM model once again, we can use an echo server client at the maximum throughput the i.MX8MM can achieve and 1472 byte packets. We keep the default period and budget settings of the echo server.

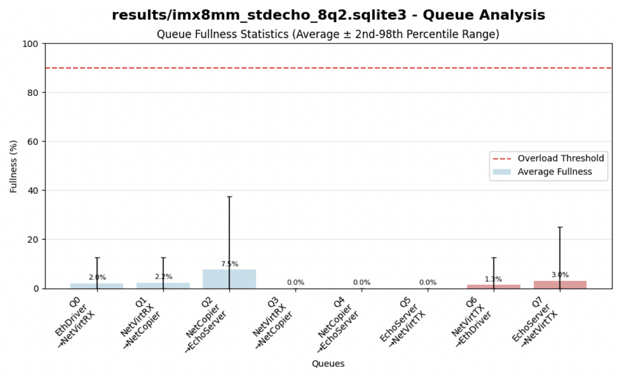
Figure 6.11 shows the results of the experiment. We can observe performance is limited by queue sizes that are smaller than 16 entries and quickly plateaus past this point. The latency measurements reveal an interesting result: initially, increasing the queue size increases our latency in both directions but there is a dramatic collapse in RX latency and a dramatic rise in TX latency once queues reach 16 entries.

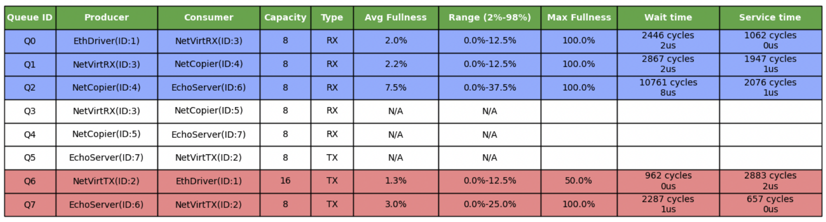
The symmetry in these results is no coincidence. Using the pysddfsim queue analysis view as listed in Figure 6.12, we can observe that the 8 entry queues face a bottleneck between the echo server and the RX copier, with the maximum queue fullness regularly hitting 100%. In the 16 entry queue system however, this queue never reaches maximum capacity.

Recalling our results from Section 4.3.1, we expect a maximal whole-RX batch size of 10 packets from the echo server under this configuration. Thus, it's unsurprising to find this limitation as the queues are no longer able to cope with the system batch size. Throughput on the RX path is very high with smaller queue sizes as every PD needs to wait for free buffers to be returned before more progress can be made.

Given the importance of batch sizes, we would notionally expect a far greater performance dropoff as a result of capping the batch size in the queues. As we can recall from our analysis of results from Section 6.1.4, the 1472 byte packet echo server with a period/budget of 100/400 is very inefficient for batching. Reducing the queue size here effectively only limits the performance of processing the packets that accumulate as the driver sleeps, while the rest of the packets arrive one at a time and are processed as normal.

We can conclude that the system queue sizes should be large enough to satisfy the largest anticipated batch size at a minimum. Any smaller than this will guarantee a performance degradation, and the degradation will be of a greater magnitude if the driver scheduling parameters are set to create optimally sized batches.





Queue ID	Producer	Consumer	Capacity	Type	Avg Fullness	Range (2%-98%)	Max Fullness	Wait time	Service time
Q0	EthDriver(ID:1)	NetVirtRX(ID:3)	8	RX	2.0%	0.0%-12.5%	100.0%	2446 cycles 2us	1062 cycles 0us
Q1	NetVirtRX(ID:3)	NetCopier(ID:4)	8	RX	2.2%	0.0%-12.5%	100.0%	2867 cycles 2us	1947 cycles 1us
Q2	NetCopier(ID:4)	EchoServer(ID:6)	8	RX	7.5%	0.0%-37.5%	100.0%	10761 cycles 8us	2076 cycles 1us
Q3	NetVirtRX(ID:3)	NetCopier(ID:5)	8	RX	N/A	N/A			
Q4	NetCopier(ID:5)	EchoServer(ID:7)	8	RX	N/A	N/A			
Q5	EchoServer(ID:7)	NetVirtTX(ID:2)	8	TX	N/A	N/A			
Q6	NetVirtTX(ID:2)	EthDriver(ID:1)	16	TX	1.3%	0.0%-12.5%	50.0%	962 cycles 0us	2883 cycles 2us
Q7	EchoServer(ID:6)	NetVirtTX(ID:2)	8	TX	3.0%	0.0%-25.0%	100.0%	2287 cycles 1us	657 cycles 0us

Figure 6.12: Queue fullness analysis for 8-packet queues

6.2 Impact of asymmetric RX and TX loads

When we consider asymmetric RX and TX loads, we must consider that there are two inherent asymmetries in the RX and TX paths:

1. When packets are received, consumers are likely to be signalled as soon as data arrives if they are able to consume data faster than packets arrive. We can recall from Section 4.4.6 that clients set a flag to indicate they require signalling when their queues are empty, and as a result fast consumers are frequently signalled. As we can recall, the scheduler will always make a PD run first among other equal-priority PDs if it is signalled and receiving PDs consequently will always get more execution time when compared against a transmitting PD of equal throughput.

2. When PDs transmit, their packets will never be sent until they signal the virtualiser. This can present a challenge because PDs have to choose between being able to fill their queues and yielding their timeslice by signalling the virtualiser.

This is to say that not only are receiving PDs advantaged due to scheduling behaviour, but transmitting PDs are implicitly disadvantaged by the signalling protocol. As a result, we are faced with significant challenges when deciding how to configure the sDDF for generality.

We can reveal the impact of this with a pysddfsim test run with a pair of clients, one receiver and one sender. We use 1472 byte packets and standard budgets and periods as we use in other examples. Similarly to the double sender example in Section 4.3.1, we configure the sender client to transmit at a high speed to prevent CPU monopolisation.

As a control case, let's first run a system with each client on its own. We can run the sender-only system using the invocation in Listing 6.6 and the receiver-only system in Listing 6.5.

```
 $ python echo_analysis.py 500000000 results/imx8mm_rxonly.
sqlite3\
 imx8mm --protocolbudget 20000 --driverbudget 100 \
 --driverperiod 400 --clientbudget 20000 --client_rxonly 1\
 --client_rx_dist 1 --rxthroughput 1g --interactive
```

Listing 6.5: Invoking the RX-only test system

```
 $ python echo_analysis.py 500000000 results/imx8mm_txonly.
sqlite3\
 imx8mm --protocolbudget 20000 --driverbudget 100\
 --driverperiod 400 --clientbudget 20000\
 --client_txonly 5g --interactive
```

Listing 6.6: Invoking the TX-only test system

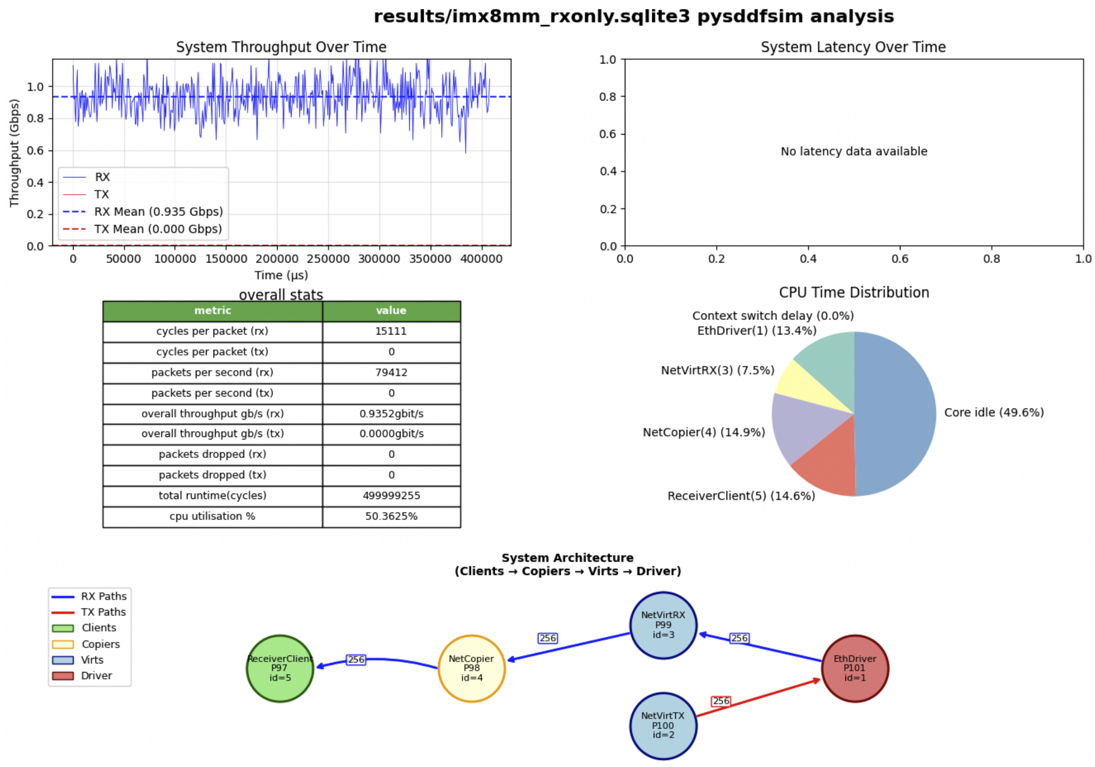
We can view the results of these systems in Figure 6.14 and Figure 6.13 for the TX and RX case respectively. These runs show little of note: the RX system uses substantially more CPU time as the suboptimal batching of the default driver scheduling parameters creates additional work. The RX system consistently manages a high receive throughput, bounded by the limits of the receive path. The TX system maintains a very low CPU utilisation as the lone client can reliably generate batches of up to 128 packets most times it is scheduled.

Inspecting the throughput graphs of each system, we can observe the TX-only system exhibiting aggressive oscillation between zero and 1.75 Gb/s of instantaneous throughput. The RX-only system exhibits random noise about the mean frequency of 0.935 Gb/s on the other hand, resembling the echo server examples we have previously observed. We can inspect the TX spectrum using the frequency domain tools to determine the source of the oscillations as listed in Figure 6.15. As we can see, the rate of the client being signalled is the largest component of the oscillation with the rate of the TX virt being signalled as a secondary component.

Now that we have inspected the individual results, we can run these clients concurrently to inspect the results. Ideally, these clients should interfere with each other minimally as they only share the driver in their critical paths and both clients execute with a large amount of free CPU time available. Invoking the test runner with the command in Listing 6.7, we see surprising results.

```
 $ python echo_analysis.py 500000000 results/imx8mm_rx_tx.
sqlite3\
 imx8mm --protocolbudget 20000 --driverbudget 100\
 --driverperiod 400 --clientbudget 20000\
 --client_rxonly 1 --client_txonly 3g\
 --client_rx_dist 1 --rxthroughput 1g --interactive
```

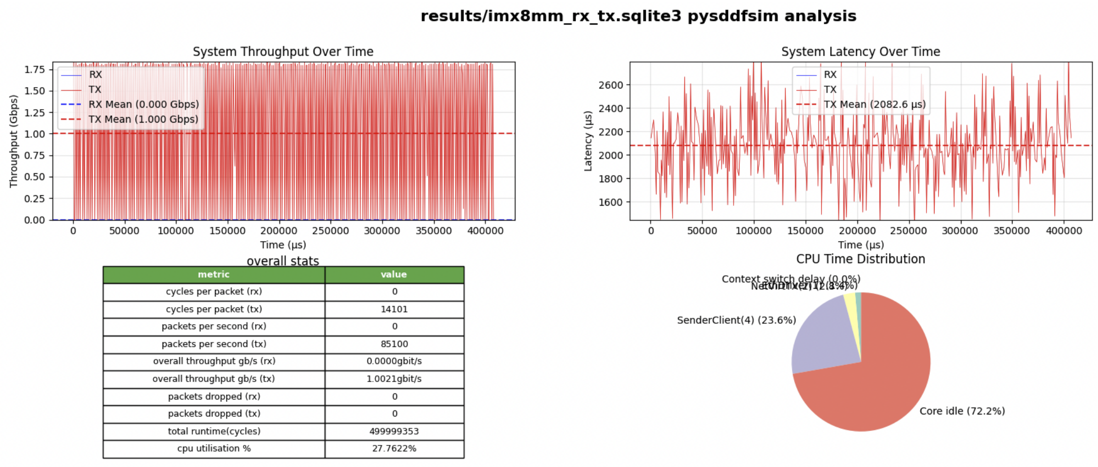
Listing 6.7: Invoking the network tester with the RX and TX clients



overall stats

metric	value
cycles per packet (rx)	15111
cycles per packet (tx)	0
packets per second (rx)	79412
packets per second (tx)	0
overall throughput gb/s (rx)	0.9352gbit/s
overall throughput gb/s (tx)	0.0000gbit/s
packets dropped (rx)	0
packets dropped (tx)	0
total runtime(cycles)	499999255
cpu utilisation %	50.3625%

Figure 6.13: Overview of the RX-only system



overall stats

metric	value
cycles per packet (rx)	0
cycles per packet (tx)	14101
packets per second (rx)	0
packets per second (tx)	85100
overall throughput gb/s (rx)	0.0000gbit/s
overall throughput gb/s (tx)	1.0021gbit/s
packets dropped (rx)	0
packets dropped (tx)	0
total runtime(cycles)	499999353
cpu utilisation %	27.7622%

Figure 6.14: Overview of the TX-only system

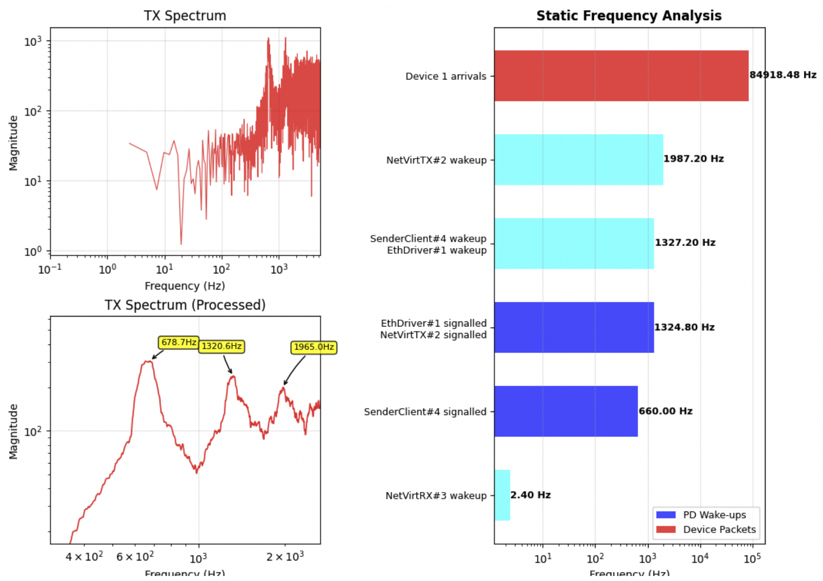


Figure 6.15: Frequency spectrum of the single transmitter control case exhibiting strong oscillations associated with signal rates

Looking at the results in Figure 6.16, we see that the two clients have affected each other substantially. The CPU utilisation of the sender client increases by 30%, while the utilisation of the receiver client decreases by about 4%. The persistent high-magnitude oscillation of the TX throughput changes drastically, becoming distantly spaces spikes with random noise between them instead of constant spikes.

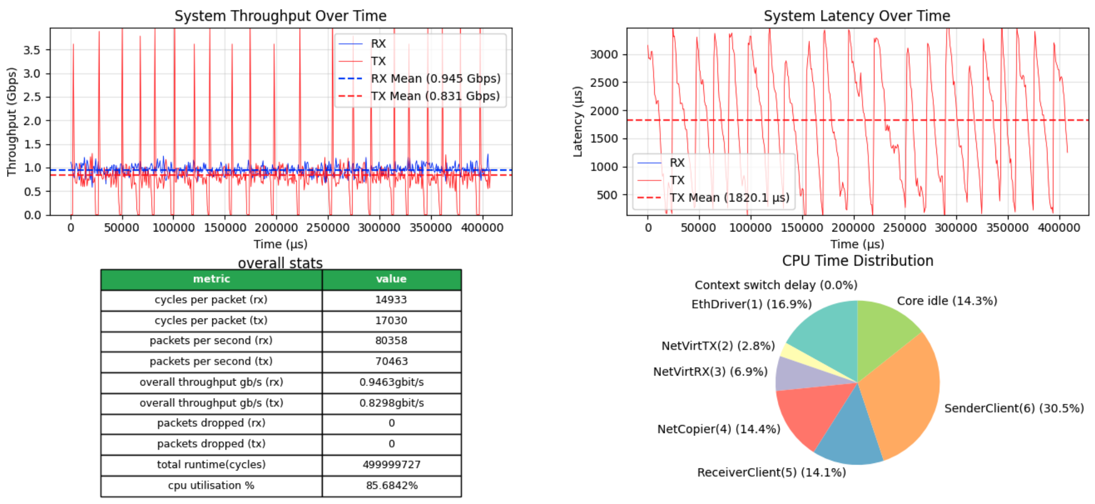
Zooming in, we can observe that the noise between the TX spikes appears to be exactly out of phase with the RX noise! This suggests that the RX client is impacting the TX client. If we investigate the frequency domain page in Figure 6.17, we see a surprising result. The TX spectrum spikes at the frequency where the RX spectrum is of low magnitude and rapidly decays as it increases.

It is clear that the RX path not only interferes with the TX path, but even benefits from doing so. The RX client receives larger batches as more packets arrive while the driver processes TX packets. The RX client is able to disrupt the TX client because of the scheduling impact of signals, confirming our hypothesis that RX clients have an inherent advantage in the sDDF.

In certain cases, this asymmetry may even deny service to transmit-dominated clients. We will demonstrate this in the form of a malicious client in Section 6.3.2.

6.3 Malicious clients

In this section we will explore the possible ways a malicious client can attempt to attack the driver stack. Fortunately, there are only a small number of ways this can occur due



metric	value
cycles per packet (rx)	14933
cycles per packet (tx)	17030
packets per second (rx)	80358
packets per second (tx)	70463
overall throughput gb/s (rx)	0.9463gbit/s
overall throughput gb/s (tx)	0.8298gbit/s
packets dropped (rx)	0
packets dropped (tx)	0
total runtime(cycles)	499999727
cpu utilisation %	85.6842%

Figure 6.16: Overview of combined RX-only and TX-only client run

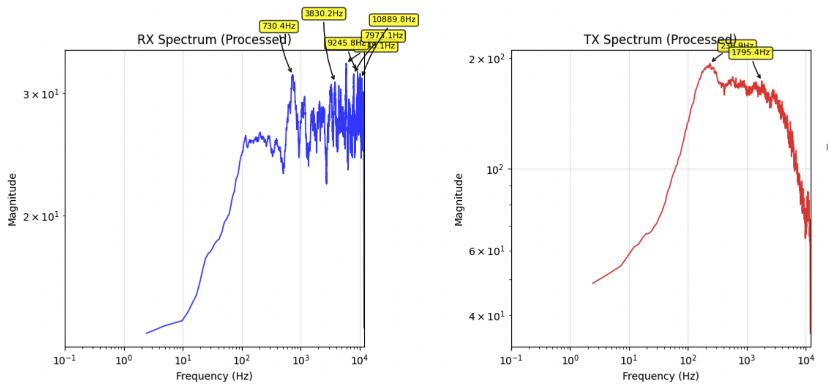


Figure 6.17: Spectra of the RX and TX clients. Note the frequency of the TX spectrum decaying as the TX spectrum increases in magnitude

to the restrictive interface between clients and the stack.

6.3.1 Malicious signalling

We have already explored throughout this chapter, the signals are a critical factor in scheduling and can incur large delays in themselves. Consequently, abuse of signals is an obvious vector for a malicious client to use to interfere with other clients by changing the scheduling order or wasting time.

Let's consider a simple case to start: a single client which will do nothing except send signals as fast as it can to the copier and the TX virt. Such a client will change the execution order of the system substantially as it causes frequent wakeups. We can refer to such a client as a "signal spammer".

We can model the signal spammer in pysddfsim by by creating a generic `FlexPD` which implements an infinite while loop in its notified function. There's no need to go to sleep for our malicious client as this will simply cost it time where it could be spamming.

The entire `pd_notified()` method can be simply written as follows in Listing 6.8.

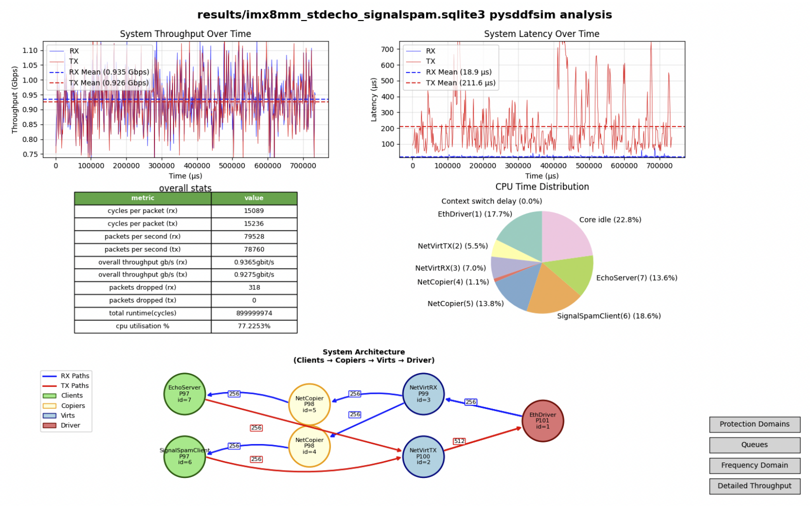
```
        def pd_notified(self, channel):
        while True:
            yield from self.rx_queue.signal_other_user(self)
            yield from self.tx_queue.signal_other_user(self)
            self.log("Mwuhahaha! Evil loop completed!")
```

Listing 6.8: Notified method for the signal spammer client

We can evaluate the impact of this client using our standard echo test on the i.MX8MM at 1 Gb/s with 1472 byte packets. So long as the echo server is still able to execute for the same amount of time and throughput is maintained, we can be reasonably sure that performance is not adversely affected. We execute our test run with the net tester command in Figure 6.18. Note that we divert a small portion of the RX traffic to the malicious client to simulate a regular networking client which has been taken over and ignores its traffic.

The results show that the echo server was still able to complete its work despite the presence of the signal spammer client. The echo server maintains approximately the same 13.7% CPU utilisation we expect. The virtualisers have slightly higher CPU utilisations than normal as the signal spammer successfully wakes them up on every signal, but this is ultimately negligible as the signal spammer is effectively yielding its timeslice to wake the virtualisers which do work for the other clients.

The copier attached to the signal client highlights the magnitude of the impact it has on the system. Since the signal spammer receives almost no traffic, the majority of the copier's 1.1% CPU utilisation is due to the signals received from the spammer. We



overall stats

metric	value
cycles per packet (rx)	15089
cycles per packet (tx)	15236
packets per second (rx)	79528
packets per second (tx)	78760
overall throughput gb/s (rx)	0.9365gbit/s
overall throughput gb/s (tx)	0.9275gbit/s
packets dropped (rx)	318
packets dropped (tx)	0
total runtime(cycles)	899999974
cpu utilisation %	77.2253%

Figure 6.18: pysddfsim overview of the signal spammer client test

can see a similar increase in the TX virt, rising to 5.5% from the 3.9% we see in an equivalent echo server without the spammer.

Spamming signals is simply not an effective technique for a malicious client. It principally will deny time from itself and not other clients.

6.3.2 Scheduler signalling exploitation

Although simply spamming signals is ineffective, we can take a more creative approach to abuse the signalling behaviour of the scheduler. As we can recall from prior sections, the scheduler will immediately make move a signalled PD to the front of its round robin queue. The result of this is that a PD can starve other equal-priority PDs if it is signalled frequently, possibly resulting in unfair service as we found in Section 6.2.

This begs the question: how can a malicious PD ensure it is signalled often?

We can answer this question by considering what causes a client PD to be signalled. There are only three sources:

1. The RX copier will signal the PD whenever it transfers a packet to the client.
2. the TX virtualiser will signal the PD every time there are free buffers to return.
3. The timer driver can signal the client whenever it desires.

A client is able to trigger any of these three signal sources on demand. The RX copier can signal the PD at most once per incoming packet, and the PD can cause a packet to arrive by sending a packet to itself. This is not particularly fast however and is difficult to exploit. Exploiting the TX virtualiser is far easier as the malicious PD can simply send one packet and immediately signal the TX virtualiser. The virtualiser will then send that one packet and immediately signal the driver. The timer driver is the most flexible option as as malicious client can repeatedly ask for an immediate signal.

Despite lacking the overt control of the timer as an exploit vector, abusing the transmit virtualiser is good enough to construct a competent attack. Additionally, the TX virtualiser is an excellent source of signals as it is of a higher priority than the RX virt and can thereby partially deny service to the entire RX path. Furthermore, this attack can succeed with a simple client without a timer driver which heightens the severity of the problem.

We can implement our transmit-based signal exploiter as shown in Listing 6.9.

```
    def pd_notified(self, channel):
        while True:
            while not self.tx_queue.free_empty():
                b = self.tx_queue.take_free()
                yield self.queue_interaction_delay()
                self.tx_queue.put_active(b)
                yield self.queue_interaction_delay()
            self.tx_queue.request_signal_active()
            self.tx_queue.request_signal_free()
            yield from self.tx_queue.signal_other_user(self)
            self.log("Mwuhahaha! Evil loop completed!")
```

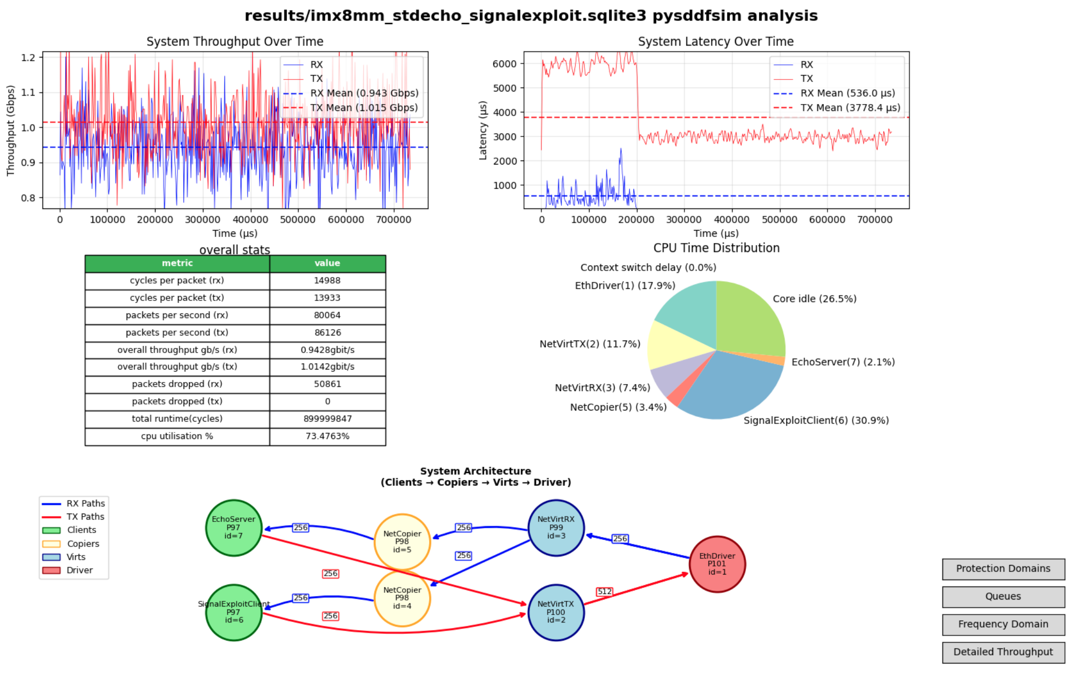
Listing 6.9: Business logic for TX virt signal exploiter

The behaviour of this client is very simple: we send packets to the TX virt and then immediately use the queue flag to indicate that we require signalling. The client then signals the TX virt, which will send the packet and immediately signal the client again. Running this test, we see the results in Figure 6.19.

At a glance, the results seem fine as we achieve a high TX throughput albeit with an unusually high CPU utilisation. Inspecting the PDs view reveals a drastically different story however, shown in Figure 6.20.

The malicious client was able to almost completely deny service to the echo server, despite the fact that the echo server constantly receives signals from the copier to awaken it! This is a serious vulnerability in the signalling protocol. This problem could be mitigated in two ways:

1. Clients can have their ability to request to be signalled removed. Ideally, clients are meant to set this flag once they are out of data to indicate that they are ready for more. The PDs on both sides of the queue can read the length of the free and active portion of the netqueue, so producer PDs can be modified to simply check capacity instead of the flag. This scenario may still be vulnerable to similar exploits however.



overall stats

metric	value
cycles per packet (rx)	14988
cycles per packet (tx)	13933
packets per second (rx)	80064
packets per second (tx)	86126
overall throughput gb/s (rx)	0.9428gbit/s
overall throughput gb/s (tx)	1.0142gbit/s
packets dropped (rx)	50861
packets dropped (tx)	0
total runtime(cycles)	899999847
cpu utilisation %	73.4763%

Figure 6.19: pysddfsim overview page showing the TX signal exploit example with an echo server

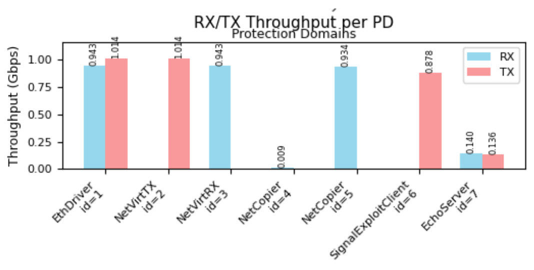


Figure 6.20: pysddfsim PDs view, showing that the echo server was unable to receive almost any packets

2. The seL4 kernel can be patched to alter the signalling behaviour such that a signalled PD remains in the same place in the round robin queue.

6.3.3 Buffer starvation

Another obvious method for a malicious client to attack the network stack is to try and hoard buffers. This threat was considered when designing the sDDF however, and RX copiers are designed to make such a scenario impossible.

Using pysddfsim, we can easily evaluate the efficacy of the copiers as well as the impact of such a client with the copier removed. We can implement our ”buffer hog” client easily as follows in Listing 6.3.3.

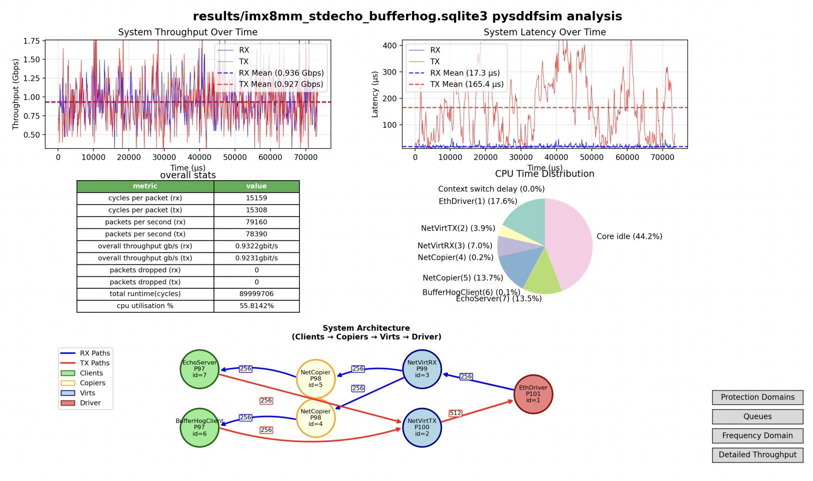
```
    def pd_notified(self, channel):
        # We just eat active buffers and never return the free ones!
        while len(self.rx_queue.active) > 0:
            buf = self.rx_queue.take_active()
            yield self.queue_interaction_delay()
            self.analytics.log_pd_consume(self.db_id, hash(buf.get_contents))
            yield 1000 # packet eat delay
            self.log("oink oink yum")
            yield get_overhead_profile().dma_interaction_delay(self.packet_sz)
            self.stomach.append(buf) # eat it for ever.

        self.rx_queue.request_signal_active()
```

This client is very simple: it simply takes all buffers that arrive and puts them in a list. We can repeat the same echo server test to evaluate non-interference for this client. Using the network tester tool, we can launch a test and add the buffer hog client as shown in Listing 6.10.

```
$ python echo_analysis.py 9000000000 \
results/imx8mm_stdecho_bufferhog.sqlite3\
imx8mm --protocolbudget 20000\
--driverbudget 100 --driverperiod 400\
--clientbudget 20000 --client_echo 1\
--badclient_bufferhog 1 --client_rx_dist 0.9 0.1
```

Listing 6.10: Invoking a test with the bufferhog client

The results in Figure 6.21 show that the buffer hog client has no impact on the echo server whatsoever. It consumes all the packets it is simply never awoken again. We can evaluate the impact of the buffer hog client without a copier by using the network tester `--no-copier` flag to disable copiers. This test yields a drastically different result: the system throughput drops to zero as the buffer hog consumues all RX buffers gradually.



overall stats

metric	value
cycles per packet (rx)	15159
cycles per packet (tx)	15308
packets per second (rx)	79160
packets per second (tx)	78390
overall throughput gb/s (rx)	0.9322gbit/s
overall throughput gb/s (tx)	0.9231gbit/s
packets dropped (rx)	0
packets dropped (tx)	0
total runtime(cycles)	89999706
cpu utilisation %	55.8142%

Figure 6.21: Overview page analysing a run with the buffer hog client.

Thus, we can conclude that clients are not able to reasonably exploit buffers to attack the system in the general case. Copiers are an important component for isolation and play an important role in decoupling the client perspective of the receive path from the driver stack.

6.3.4 CPU time abuse

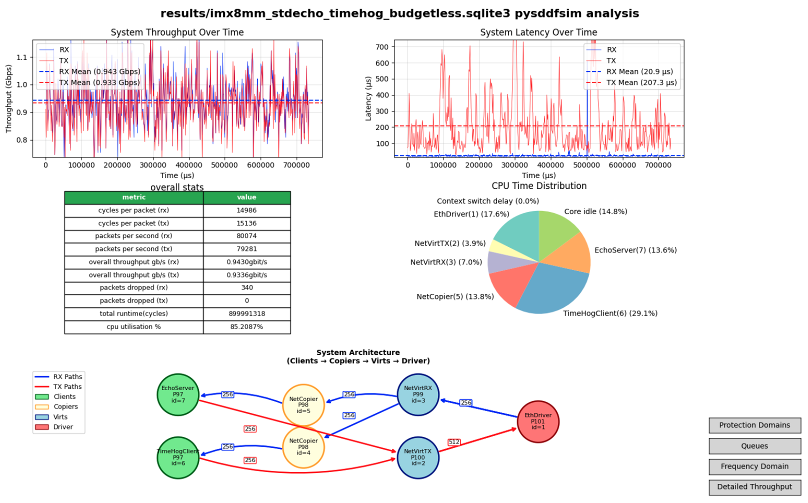
Equal-priority clients face another possible attack vector: a malicious client can seek to deny execution time to other clients by using all CPU time available to it. As we have established, all clients notionally have a budget of $20000\mu s$, and if a malicious client was able to delay other clients by a whole budget it may have a substantial impact on system performance.

We can implement a client to test this case very easily in pysddfsim. We implement an infinite loop in `pd_notified()` which yields large delays every iteration as detailed in Listing 6.11.

```
        def pd_notified(self, channel):
        while True:
            # All we do is scream into the void and never sleep.
            # We have queues, but we totally ignore them.
            self.log("Preparing to oink...")
            yield 100000
            self.log("Oink")
```

Listing 6.11: Business logic for the time hog client

We can test this malicious client using an i.MX8MM echo server with 1472 byte packets similarly to the other malicious client cases. The results in Figure 6.22 reveal that the



overall stats

metric	value
cycles per packet (rx)	14986
cycles per packet (tx)	15136
packets per second (rx)	80074
packets per second (tx)	79281
overall throughput gb/s (rx)	0.9430gbit/s
overall throughput gb/s (tx)	0.9336gbit/s
packets dropped (rx)	340
packets dropped (tx)	0
total runtime(cycles)	899991318
cpu utilisation %	85.2087%

Figure 6.22: pysddfsim overview of the echo server test with the time hog client

time hog had no impact echo server as it successfully manages to return all packets received, excluding the small fraction of throughput ignroed by the time hog client.

The time hog client successfully absorbs all free time on the system, but it simply has no effect due to the scheduler round robin reordering behaviour on signals. This raises a tricky question for the scheduling behaviour: is it worth removing it to guard against exploits such as Section 6.3.2 when it will also open weaken clients against trivial attacks?

Chapter 7

Discussion

This thesis has evaluated the sDDF from a variety of directions that have been largely unexplored thus far. Pysddfsim is a powerful tool which greatly expands the power of introspection we have over sDDF systems and will enable more sophisticated design and analysis going into the future. Although we have highlighted several serious problems in the sDDF, we have also broadly seen that the framework is robust under most conditions. Further, almost all of the troubles encountered are caused by a single seL4 bug which can be easily removed.

To conclude, we will remark on several important findings and discuss future work.

7.1 Factors governing general system performance

The single most important factor to system performance for the sDDF network stack is the number of packets serviced per wakeup for all PDs. We observed the significance of this in Section 6.1.3, where an appropriately selected budget and period to efficiently batch was able to achieve the best network benchmark measured thus far for the sDDF.

Fundamentally, the cost of switching between PDs is simply too large to ever justify wake up for small numbers of packets unless it is unavoidable for latency reasons. At a minimum, the cost to send a signal and `seL4_Recv()` it is on the order 1000-4000 cycles depending on the platform, and this is before considering the work PDs need to do to process packets themselves.

An insight from this thesis is the significance of apparently minor details of the driver and the network interface in the network device class. Transmit and receive performance is bounded by the size of the network interface ring buffers as they dictate the maximum batch size the system can achieve.

7.2 The peril of scheduling parameters

Given what we know about the performance constraints of the sDDF and the magnitude of impact that budgets and periods have on systems, we must question whether it makes sense to use the seL4 scheduling parameters as a performance tuning tool. The results we have seen thus far strongly imply that it does not make sense!

If we tune period and budget of a driver to be optimal for a specific throughput and packet size, we are always conceding to suboptimal performance in other cases. This is fine for systems with predictable traffic patterns, but it is an issue for general-purpose operating systems such as Djawula or systems with inconsistent traffic.

If we want to maximise batch sizes at any given load, there is simply no way we can set budgets and periods appropriately. The methodology to tune the driver scheduling paramters in Section 6.1.3 depends on anticipating both the maximum (ideally average) throughput as well as an expected service time, but it can be reformulated in the future for a different batching mechanism that abstractly controls the fraction of driver CPU time.

Budgets and periods are largely irrelevant for other trusted PDs in the network device class as all such PDs have tightly bounded execution time and very fast service time. Similarly, the current scheduling behaviour also makes budgets and periods mostly irrelevant to clients as the most recently signalled PD always runs first.

7.3 The seL4 scheduler notification bug and its implications

The seL4 scheduler bug which causes notifications to promote PDs to the front of their round-robin queues has been a frequent topic of conversation throughput this thesis. Presently it is listed as an issue to be fixed for the seL4 kernel, but there is ongoing discussion about the impact of changing it. Removing this bug will mitigate the RX/TX asymmetry of Section 6.2 and fix the hazardous signal exploit of Section 6.3.2, but will promote several non-issues into hazards such as the time hogging malicious clients in Section 6.3.4.

Without this behaviour, all PDs in all driver classes will need to be considered more carefully for any given system. Dataflow will no longer guarantee consumers react to data, and most PDs likely will need a period applied.

Presently, the scheduling parameters of clients and copiers has little to no impact on performance as PDs respond to signals immediately as data arrives. Without this bug, any misconfiguration of the scheduling parameters is likely to compromise system performance as clients will need to wait to be scheduled if any other equal-priority clients are running.

These problems can be handled with relative ease however. A principled method of assigning periods and budgets to clients as well as privileged PDs will be needed, and additionally some structural changes may be required. For example, the receive copier-client queues will likely need to become much larger as clients may need to wait a substantial amount of time before they can begin to process packets.

Once this bug has been mitigated, pysddfsim will remain an ideal tool to check for possible problems and design optimal solutions.

7.4 Future work

There is a gamut of future work that may follow this thesis, including some work that has immediate applications for other research projects involving the sDDF. Some of the future work described in this section is already implemented in pysddfsim but could not be added to this thesis due to time constraints.

7.4.1 x86 modelling and analysis

High-throughput x86 systems are particularly interesting as a target to analyse with pysddfsim as we can detect resonant behaviours at very high throughputs. The sDDF x86 network class supports 10 Gb/s network interfaces and previous benchmarking has yielded some evidence of resonant behaviour as is.

Pysddfsim already has an x86 device class and frequency analysis tools, but there was insufficient time to tune the `OverheadModel` and perform tests for this thesis.

7.4.2 Extended overhead models

As discussed in Section 5.2, the current pysddfsim overhead model is not sufficiently accurate to predict performance in all cases. Pysddfsim currently fails to include any notion of instructions and hence is unable to simulate the impact of the I-cache and I-TLB in real systems, compromising accuracy at lower throughputs.

In the future this can be mitigated by altering how the FlexPD class understands time. Currently, FlexPDs step through business logic and yield delays in cycles to indicate the amount of time that should pass. Overheads are calculated by the FlexPD and yielded explicitly, calling the OverheadModel where appropriate. A better model could be to yield a number of instructions and provide some unique identifier for every yield block such that an instruction cache simulation can track temporal locality of the yields.

Chapter 8

Conclusions

This thesis has presented pysddfsim, a discrete-event simulator and comprehensive analysis suite for the seL4 Device Driver Framework. Through extensive modelling and experimentation, we have demonstrated that the sDDF maintains robust performance under most extreme conditions, validating its core design principles while identifying a family of issues with a common root cause.

Our analysis reveals that batch size processed per PD wakeup emerges as the dominant factor governing system performance. The empirical methodology developed for optimising driver scheduling parameters achieved the highest throughput thus recorded on the i.MX8MM, demonstrating the practical value of simulation-guided parameter tuning.

The analysis uncovered critical insights regarding system dynamics. Queue sizes must accommodate maximum anticipated batch sizes to prevent performance degradation, while scheduling parameters for most non-driver PDs prove largely irrelevant due to their rapid service times and current scheduling behaviour. Relevantly to scheduling parameters and most significantly, we identified that a scheduler bug causing signalled PDs to jump round-robin queues causes unfair service and enables a denial of service exploit.

Pysddfsim itself is a useful contribution for analysing seL4 Microkit systems in general. By abstracting protection domains as event-emitting iterators and modelling cores as discrete processes, we achieved simulation speeds orders of magnitude faster than cycle-accurate approaches while maintaining sufficient fidelity for meaningful analysis. Domain-specific modelling ensures accessibility to end-users without simulation expertise, while the comprehensive analysis engine enables finer-grained investigation of system behaviour than is otherwise possible.

This work establishes that while the current sDDF design successfully remains performant and supplies isolation under typical workloads, its reliance on flawed scheduler behaviours creates vulnerabilities that must be addressed. Removing this scheduler behaviour will require a re-evaluation of performance of the sDDF with complex systems.

Looking forward, pysddfsim provides the foundation for principled optimisation of sDDF systems, enabling developers to evaluate parameter choices and architectural decisions without running slow benchmarks.

Bibliography

Gianfranco Balbo. Introduction to Stochastic Petri Nets. In Ed Brinksma, Holger Hermanns, and Joost-Pieter Katoen, editors, *Lectures on Formal Methods and PerformanceAnalysis: First EEF/Euro Summer School on Trends in Computer Science Bergen Dal, The Netherlands, July 3–7, 2000 Revised Lectures*, pages 84–155. Springer, Berlin, Heidelberg, 2001. ISBN 978-3-540-44667-5. doi: 10.1007/3-540-44667-2_3.

U. Narayan Bhat. Imbedded Markov Chain Models. In U. Narayan Bhat, editor, *An Introduction to Queueing Theory: Modeling and Analysis in Applications*, pages 85–125. Birkhäuser, Boston, MA, 2015a. ISBN 978-0-8176-8421-1. doi: 10.1007/978-0-8176-8421-1_5.

U. Narayan Bhat. Queueing Networks. In U. Narayan Bhat, editor, *An Introduction to Queueing Theory: Modeling and Analysis in Applications*, pages 159–176. Birkhäuser, Boston, MA, 2015b. ISBN 978-0-8176-8421-1. doi: 10.1007/978-0-8176-8421-1_7.

U. Narayan Bhat. Simple Markovian Queueing Systems. In U. Narayan Bhat, editor, *An Introduction to Queueing Theory: Modeling and Analysis in Applications*, pages 37–83. Birkhäuser, Boston, MA, 2015c. ISBN 978-0-8176-8421-1. doi: 10.1007/978-0-8176-8421-1_4.

Laura Carnevali, Marco Paolieri, and Enrico Vicario. The ORIS tool: app, library, and toolkit for quantitative evaluation of non-Markovian systems. *SIGMETRICS Perform. Eval. Rev.*, 49(4):81–86, June 2022. ISSN 0163-5999. doi: 10.1145/3543146.3543164. URL `https://dl.acm.org/doi/10.1145/3543146.3543164`.

Andrew Daw, Brian Fralix, and Jamol Pender. Non-Stationary Queues with Batch Arrivals, June 2022. URL `http://arxiv.org/abs/2008.00625`. arXiv:2008.00625 [math].

Jack B Dennis and Earl C Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9(3), 1966.

Free Software Foundation. https://savannah.nongnu.org/projects/lwip/, October 2002. URL `https://savannah.nongnu.org/projects/lwip/`.

Yunhai Fu, Lin Ma, and Yubin Xu. A Novel Component Carrier Configuration and Switching Scheme for Real-Time Traffic in a Cognitive-Radio-Based Spectrum Aggregation System. *Sensors (Basel, Switzerland)*, 15:23706–23726, September 2015. doi: 10.3390/s150923706.

Vijay Gehlot and Carmen Nigro. An introduction to systems modeling and simulation with colored petri nets. In *Proceedings of the Winter Simulation Conference*, WSC '10, pages 104–118, Baltimore, Maryland, December 2010. Winter Simulation Conference. ISBN 978-1-4244-9864-2.

Peter W. Glynn and Peter J. Haas. On simulation of non-Markovian stochastic Petri nets with heavy-tailed firing times. In *Proceedings of the Winter Simulation Conference*, WSC '12, pages 1–12, Berlin, Germany, December 2012. Winter Simulation Conference.

Geoffrey Gordon. A general purpose systems simulation program. In *Proceedings of the December 12-14, 1961, eastern joint computer conference: computers - key to total systems control on - AFIPS '61 (Eastern)*, pages 87–104, Washington, D.C., 1961. ACM Press. doi: 10.1145/1460764.1460768. URL `http://portal.acm.org/citation.cfm?doid=1460764.1460768`.

Gernot Heiser, Lucy Parker, Peter Chubb, Ivan Velickovic, and Ben Leslie. Can We Put the "S" Into IoT? In *2022 IEEE 8th World Forum on Internet of Things (WF-IoT)*, pages 1–6, Yokohama, Japan, October 2022. IEEE. ISBN 978-1-66549-153-2. doi: 10.1109/WF-IoT54382.2022.10152198. URL `https://ieeexplore.ieee.org/document/10152198/`.

Gernot Heiser, Peter Chubb, Alex Brown, Courtney Darville, and Lucy Parker. sDDF design: design, implementation and evaluation of the seL4 device driver framework. 2024. URL `https://trustworthy.systems/publications/papers/Heiser_CBDP_24.abstract,/publications/papers/Heiser_CBDP_24.abstract`.

Heiser, Gernot. The seL4 Microkernel – An Introduction. White paper., January 2025. URL `https://sel4.systems/About/seL4-whitepaper.pdf`.

Tetsuji Hirayama. Analysis of multiclass feedback queues and its application to a packet scheduling problem. In *Proceedings of the 4th International Conference on Queueing Theory and Network Applications*, QTNA '09, pages 1–8, New York, NY, USA, July 2009. Association for Computing Machinery. ISBN 978-1-60558-562-8. doi: 10.1145/1626553.1626568. URL `https://dl.acm.org/doi/10.1145/1626553.1626568`.

Kurt Jensen, Lars Michael Kristensen, and Lisa Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer*, 9(3):213–254, June 2007. ISSN 1433-2787. doi: 10.1007/s10009-007-0038-x.

David G. Kendall. Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain. *The Annals of Mathematical Statistics*, 24(3):338–354, September 1953. ISSN 0003-4851, 2168-8990. doi: 10.1214/aoms/1177728975. URL https://projecteuclid.org/journals/annals-of-mathematical-statistics/volume-24/issue-3/Stochastic-Processes-Occurring-in-the-Theory-of-Queues-and-their/10.1214/aoms/1177728975.full. Publisher: Institute of Mathematical Statistics.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, February 2014. ISSN 0734-2071. doi: 10.1145/2560537.

Kurtis Konrad and Yunan Liu. Real-Time Estimations for the Waiting-Time Distribution in Time-Varying Queues. In *Proceedings of the Winter Simulation Conference*, WSC '23, pages 315–326, San Antonio, Texas, USA, February 2024. IEEE Press. ISBN 9798350369663.

Timo Lask. Stochastic time petri nets: time processes modelling in Modelica and application in hospital and healthcare. In *Proceedings of the 8th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools*, EOOLT '17, pages 37–44, New York, NY, USA, December 2017. Association for Computing Machinery. ISBN 978-1-4503-6373-0. doi: 10.1145/3158191.3158196. URL https://dl.acm.org/doi/10.1145/3158191.3158196.

G. Latouche and V. Ramaswami. *Introduction to Matrix Analytic Methods in Stochastic Modeling.* ASA-SIAM Series on Statistics and Applied Mathematics. Society for Industrial and Applied Mathematics, January 1999. ISBN 978-0-89871-425-8. doi: 10.1137/1.9780898719734. URL https://epubs.siam.org/doi/book/10.1137/1.9780898719734.

Linux Kernel Development Community. Networking — The Linux Kernel documentation. URL https://linux-kernel-labs.github.io/refs/heads/master/labs/networking.html.

Linux Kernel Development Community. Linux Device Model — The Linux Kernel 5.10.14 documentation, 2024. URL https://linux-kernel-labs.github.io/refs/heads/master/labs/device_model.html.

John A. Miller, Jun Han, and Maria Hybinette. Using domain specific language for modeling and simulation: scalation as a case study. In *Proceedings of the Winter Simulation Conference*, WSC '10, pages 741–752, Baltimore, Maryland, December 2010. Winter Simulation Conference. ISBN 978-1-4244-9864-2.

Rupert G. Miller, Jr. A Contribution to the Theory of Bulk Queues. *Journal of the Royal Statistical Society: Series B (Methodological)*, 21(2):320–337, July 1959. ISSN 0035-9246. doi: 10.1111/j.2517-6161.1959.tb00340.x.

Lucy Parker. *High-performance Networking on seL4.* Honours Thesis, The University of New South Wales, January 2024.

Tomasz Rolski. Queues with nonstationary inputs. *Queueing Systems*, 5(1):113–129, November 1989. ISSN 1572-9443. doi: 10.1007/BF01149189.

seL4 Foundation. MCS | seL4 docs, 2024a. URL `https://docs.sel4.systems/Tutorials/mcs.html#scheduler`.

seL4 Foundation. seL4 Microkit Manual - Github · seL4/microkit, 2024b. URL `https://github.com/seL4/microkit/blob/main/docs/manual.md`.

seL4 Foundation. seL4/sel4bench, August 2025. URL `https://github.com/seL4/sel4bench`. original-date: 2016-06-02T01:38:23Z.

C.E. Shannon. Communication in the Presence of Noise. *Proceedings of the IRE*, 37(1): 10–21, January 1949. ISSN 0096-8390. doi: 10.1109/JRPROC.1949.232969. URL `http://ieeexplore.ieee.org/document/1697831/`.

Taejoon Song and Youngjin Kim. Comparative Study on Fuchsia and Linux Device Driver Architecture. In *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*, SAC '23, pages 1305–1308, New York, NY, USA, June 2023. Association for Computing Machinery. ISBN 978-1-4503-9517-5. doi: 10.1145/3555776.3577828. URL `https://dl.acm.org/doi/10.1145/3555776.3577828`.

SQLite project. SQLite, April 2025.

Team Simpy. Simpy Documentation. URL `https://simpy.readthedocs.io/en/latest/`.

Yentl Van Tendeloo, Randy Paredis, and Hans Vangheluwe. An Introduction to Discrete-Event Modeling and Simulation with Devs. In *Proceedings of the Winter Simulation Conference*, WSC '23, pages 1531–1545, San Antonio, Texas, USA, February 2024. IEEE Press. ISBN 9798350369663.

Gerd Wagner. An abstract state machine semantics for discrete event simulation. In *Proceedings of the 2017 Winter Simulation Conference*, WSC '17, pages 1–12, Las Vegas, Nevada, December 2017. IEEE Press. ISBN 978-1-5386-3427-1.

Gerd Wagner. Towards a non-proprietary modeling language for processing network simulation. In *Proceedings of the 2019 Summer Simulation Conference*, SummerSim '19, pages 1–12, San Diego, CA, USA, July 2019. Society for Computer Simulation International.

Weisstein, Eric W. Hanning Function, September 2025. URL `https://mathworld.wolfram.com/HanningFunction.html`.

Wienand, Ian and Luke Macpherson. ipbench - A Framework for Distributed Network Benchmarking. pages 163–170, Melbourne, Australia, 2004. URL `https://github.com/au-ts/ipbench`.

Cathy Wu. Markov Chains -1.041 L9 Beyond basic facility dynamics - MIT Open Courseware, March 2024a. URL `https://web.mit.edu/1.041/www/lectures/L9-markov-chains-2024sp.pdf`.

Cathy Wu. Stochastic simulation - 1.041 L10 - MIT Open Courseware, April 2024b. URL `https://web.mit.edu/1.041/www/lectures/L10-stochastic-simulation-2024sp.pdf`.

Hee Chang Yoon, Seung Heon Oh, Jong Hun Woo, Jung-Hoon Chung, Hyuk Lee, and Sun-Ah Jung. Discrete Event Simulation of Aircraft Sortie Generation on an Aircraft Carrier. In *2023 Winter Simulation Conference (WSC)*, pages 2439–2449, San Antonio, TX, USA, December 2023. IEEE. ISBN 9798350369663. doi: 10.1109/WSC60868.2023.10407756. URL `https://ieeexplore.ieee.org/document/10407756/`.

Xiao Yu, Shi Han, Dongmei Zhang, and Tao Xie. Comprehending performance from real-world execution traces: a device-driver case. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, ASPLOS '14, pages 193–206, New York, NY, USA, February 2014. Association for Computing Machinery. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541968. URL `https://dl.acm.org/doi/10.1145/2541940.2541968`.

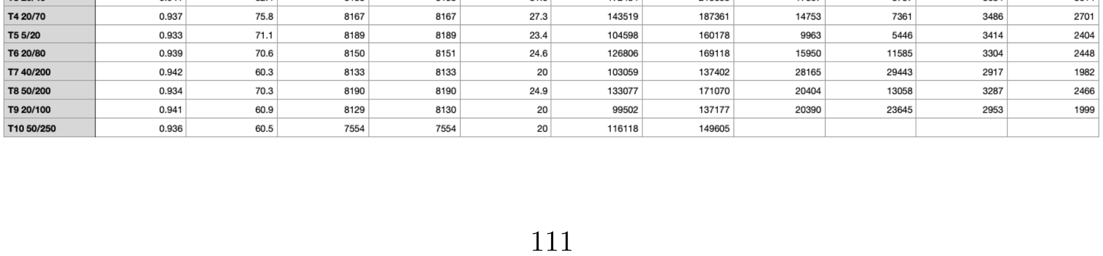
Zupancic, Indan. Append instead of enqueue tasks woken up by notifications · Issue #902 · seL4/seL4, August 2022. URL `https://github.com/seL4/seL4/issues/902`.

Supplementary Materials

A.1 Various results spreadsheets

Submitted in artefacts due to an issue with the UNSW thesis template in OverLeaf.

A.2 Benchmark results: driver periods and budgets



RUN 1: echo	Throughput (TX)	CPU util	Cycles per packet RX	Cycles per packet TX	Driver CPU util	Driver sleeps	Driver signals sent	Total RX wait time	Total TX wait time	Total RX service time	Total TX service time
Budgetless	0.943	82.3	8161	8162	31.1	172357	212502	17149	3701	3674	3011
T1 100/400	0.943	70.3	8113	8113	25	134656	171599	25382	14433	3258	2440
T2 10/40	0.94	70.6	8139	8139	24.1	117693	166047	12822	8711	3324	2423
T3 20/40	0.941	82.4	8133	8133	31.0	172484	213055	17367	3737	3684	3014
T4 20/70	0.937	75.8	8167	8167	27.3	143519	187361	14753	7361	3486	2701
T5 5/20	0.933	71.1	8189	8189	23.4	104598	160178	9963	5446	3414	2404
T6 20/80	0.939	70.6	8150	8151	24.6	126806	169118	15950	11585	3304	2448
T7 40/200	0.942	60.3	8133	8133	20	103059	137402	28165	29443	2917	1982
T8 50/200	0.934	70.3	8190	8190	24.9	133077	171070	20404	13058	3287	2466
T9 20/100	0.941	60.9	8129	8130	20	99502	137177	20390	23645	2953	1999
T10 50/250	0.936	60.5	7554	7554	20	116118	149605				