THE UNIVERSITY OF
NEW SOUTH WALES

**SCHOOL OF ELECTRICAL ENGINEERING
AND TELECOMMUNICATION**

Usermode OS Components On seL4 With Rump Kernels

by

Kent McLeod

Thesis submitted as a requirement for the degree
Bachelor of Engineering (Electrical Engineering)

Submitted: October 27, 2016
Supervisor: Kevin Elphinstone

Student ID: z3377644
Course Code: ELEC4121

Abstract

seL4 is a formally-verified high-assurance microkernel that provides isolation to properly designed applications that it executes. Real-world cyber-physical systems can use seL4 for increased security. Many applications rely on the operating system to provide system services, such as device drivers, file systems and networking capabilities, however seL4 only provides these in a limited capacity which limits its deployment. Adding support to this wide array of systems that can benefit from the additional security seL4 provides would require reimplementing the millions of lines of operating system code that these systems require. This is infeasible without an approach that reuses existing components. Current methods either require providing services by running a paravirtualised version of the Linux kernel which provides only coarse isolation or by developing specific services on an as-needed basis which does not scale for many devices. Rump kernels are a NetBSD project for running NetBSD system services in different environments such as in user-mode on a microkernel.

This thesis evaluates rump kernels as an approach to provide driver-like operating system components in user-mode on the seL4 microkernel. This will be achieved by adding a seL4 platform to the Rumprun unikernel, an existing project that uses rump kernels. We evaluate our implementation to compare its performance with other software systems and to investigate the level of overhead our implementation adds. We also show that the effort required to use rump kernels is low and by using NetBSD system services we increase the amount of devices that can be used with seL4. This thesis contains background information and related work, details on our design and implementation and our evaluation, future work and conclusion.

Contents

List of Figures

List of Tables

Acronyms

DDE Device Driver Environment

DMA Direct Memory Access

IPC Inter-process communication

ISA Instruction Set Architecture

L4RE L4 Runtime Environment

lwp light weight process

MMIO Memory Mapped I/O

NDIS Network Driver Interface Specification

OS operating system

SWI Software Interrupt

UDI Uniform Driver Interface

1 Introduction

This thesis evaluates an approach for driver-like operating system component reuse on the seL4 microkernel. This is achieved by porting the Rumprun unikernel, a project that uses rump kernels to reuse NetBSD operating system components, to run in user-mode on top of seL4. We:

1. perform a comparative performance evaluation of the NetBSD networking stack running inside a rump kernel, and
2. evaluate the effort required to use rump kernels to provide unmodified NetBSD driver-like operating system (OS) components that are runnable on seL4.

seL4 is a microkernel with highly critical applications. Its comprehensive formal verification makes it the world's most secure OS kernel. Real-world cyber-physical systems can use seL4 for increased security. However, to run seL4 on a wide range of heterogeneous devices, driver software is required, which is millions of lines of code, therefore limiting seL4's deployment. Allowing driver reuse will increase the number of drivers that work with seL4 so as to ultimately increase the range of real world devices that can be made more secure.

There are existing solutions for reusing this driver software, but they require either a large amount of effort per driver, or significant runtime overhead. Therefore, we are seeking a method of reusing device drivers and other driver-like OS components on seL4 in user-mode that is low effort per component and also has a low runtime overhead.

Rump kernels were designed to allow driver-like OS components to run in other environments with minimal effort while still providing the same performance, together with low runtime overhead. The Rumprun unikernel is an existing project that uses rump kernels designed for running directly on hardware, either on bare metal or on top of a virtual machine monitor such as Xen. By adding support to the Rumprun unikernel to run on top of seL4 we intend to run unmodified NetBSD driver-like OS component with low effort and low runtime overhead.

Our contributions are porting the Rumprun unikernel to run in user-mode on top of seL4, performing a performance evaluation of the NetBSD networking stack running in the unikernel on seL4 to compare our performance with the unikernel running without seL4. We also compare performance with other operating systems Linux and NetBSD so as to evaluate overall rump kernel performance. We evaluate the effort required to use rump kernels to provide unmodified NetBSD driver-like OS components that are runnable on seL4.

Our future work would be to perform a further exploration of rump kernel features such as running rump kernels on other architectures such as ARM, extending our solution to use the rump kernel with multiple seL4 threads and evaluate the rump kernels syscall proxy features to enable using the rump kernel across multiple address spaces allowing us to take advantage of seL4's ability to provide fault isolation.

This thesis presents relevant background information in Section 2. In Section 3 we set out the existing approaches for reusing device drivers as already described in literature. In Sections 4 and 5, we provide background on the Rumprun unikernel, rump kernels and seL4. Section 6 introduces our approach for porting the Rumprun unikernel and evaluation while the following section provides our detailed design. Section 8 contains our results and evaluations. Our final sections set out future work and our conclusion.

2 Background Information

We first describe what an operating system (OS) is and two features one should provide: application portability and fault isolation. We describe what a kernel is and look at two popular kernel designs: A monolithic kernel design and the NetBSD OS as well as a microkernel design and the seL4 microkernel. We also look at common system components that are required to provide application portability and how they can be implemented differently with different OS designs. Finally we describe the low level system primitives that an OS uses to implement these components.

2.1 Operating Systems and kernel design

An OS is responsible for two things: allowing applications to run on top of underlying hardware by presenting this hardware as a selection of system components with defined interfaces, and managing the use of this hardware, allowing it to be used efficiently and often shared among many concurrent applications [Tanenbaum, 2009, chap.1]. These abstractions can enable applications to run on a diverse range of hardware platforms. Additionally, applications should be protected from incorrect or adversarial behavior of other applications running on the same system and system resources should be used efficiently but also fairly.

Mechanisms are built into the hardware to enable the OS to implement and enforce its management policies. The kernel is the core part of the OS that that takes advantage of these hardware mechanisms. As different applications have different requirements, it is hard to design and build an OS that can achieve all these requirements effectively, some OS designs focus more on some requirements than others. As a consequence there are many different OS and kernel designs.

2.1.1 Application portability

An application can be considered portable if the effort to move it to a new environment is much less than the effort required to rewrite it from scratch [Johnson and Ritchie, 1978]. Portability is important because it increases the amount of devices that an application can run on without as much additional effort. Originally, when an application had to be coded in assembly language, moving an application to a different machine, even one with the same architecture, was no small task [Johnson and Ritchie, 1978]. Today however, a developer is able to build an application and have it run on multiple devices with ease. For example, there are over 24,000 distinct device models which run the Android OS [Open Signal, 2015].[1] A developer prefers developing an application once that can run on multiple environments than port and maintain their application several times as the cost of development increases with each additional version of their application they have to maintain.

An operating system provides application portability by presenting hardware as a selection of system components with defined interfaces that applications can use. This saves the application developer from having to develop these components themselves. Two ways that an application can be made portable is by developing an application that can run on multiple OSes, or developing an application to run on one OS but developing an OS that can run on multiple environments. The POSIX standard provides portability at the OS level [POSIX]. If an application uses system services through a POSIX interface,

[1] That information was collected by those devices all running the same application.

then the application will be able to be compiled and run on any POSIX compliant OS. Many OSes are mostly POSIX compliant such as GNU/Linux, OSX and the BSD variants.

Alternatively, an OS that can run on many different machines has been widely used for a long time [Johnson and Ritchie, 1978]. This method is important if an application relies on specific features of its OS that are not within a common standard such as POSIX. OSes like GNU/Linux can be run on over 25 different architectures and many more different devices [Torvalds, 2016].

When developing a new OS, if it wants to be used for a wide range of applications, it is necessary that it supports many different environments in order for developers to commit to developing applications for it.

2.1.2 Processor modes and fault isolation

An OS needs to be able to enforce the policies that it implements to manage resources and applications. Processors provide privileged instructions in an Instruction Set Architecture (ISA) that can only be executed when the processor is in a privileged mode. This enables OSes to enforce their policies. On an ARMv7 processor, if an application in user mode tries to perform an illegal instruction, such as read a memory address only valid in supervisor mode, then the processor will trigger an exception and the OS will be invoked to handle the application's illegal behavior [ARM, 2008]. Thus, an application is unable to perform illegal actions.

The kernel is the part of the OS that executes in the privileged processor mode that enables these special instructions. When an application wants to perform a protected operation, it has to ask the kernel to perform it on the application's behalf. In the case of ARMv7, the application can trigger a Software Interrupt (SWI) which changes the processor into supervisor mode and calls the kernel. The kernel can then choose to perform the operation or not. This allows the OS to selectively prevent applications from performing unauthorised actions.

An exception is also known as a fault. Fault isolation means that if one part of the system has a fault, the rest of the system is unaffected and the fault can be contained by the OS. Faults can occur because an application is trying to act maliciously, or because of a software bug such as an array out-of-bounds access. Without the OS present to handle faults, the system can crash. Applications inevitably have bugs, even in high quality projects [Synopsys, 2015], however because the OS can handle the faults, the system does not crash and other applications can continue operating.

However, if the kernel performs an action that is incorrect, either the action will be performed, putting the system into an undefined state, or a fault will occur and the system can crash as the kernel cannot handle a fault it did not expect. When this happens, all applications running on the system will crash and the OS has failed to correctly perform its required tasks. It is very important that this never happens, though this is very hard to guarantee. Different kernel designs can help reduce the chances of this outcome.

2.1.3 Monolithic Kernel: NetBSD

A monolithic kernel is an OS design where a majority of the system components reside in the kernel and are executed in privileged mode. User applications run in user-mode and use OS components through

system calls (syscalls) into the kernel. Many modern OSes such as Microsoft Windows NT, Linux, OSX, and the BSD variants are all based on monolithic kernel designs.

Monolithic kernel designs were initially used because the earliest computer architectures did not have memory protection and there was no advantage from separating other OS services from the rest of the kernel [Ritchie and Thompson, 1978]. However, as the amount of services that the OS provides and the number of hardware platforms that the OS supports has increased, the amount of code in the kernel has increased significantly. In 2014 the Linux kernel had over 9 million lines of source code [Synopsys, 2015]. This large amount of code for components is good for application portability, but is practically impossible to guarantee that a monolithic kernel of that size will never fault. The Linux kernel has a few thousand known defects [Synopsys, 2015]. When there is a fault in the kernel, there is no way for the OS to recover. This makes it hard to build systems with good fault isolation using a monolithic kernel design.

NetBSD is a monolithic kernel that is one of the open source descendants of the BSD monolithic operating system. NetBSD focuses on portability across a large number of platforms [NetBSD]. It has 8 officially maintained architectures, and 49 unofficially supported architectures. NetBSD has a more structured kernel design than Linux which makes it suitable for reusing OS components in other environments. Rump kernels are part of the NetBSD project and use unmodified NetBSD OS components. We will discuss them in detail in Section 4.

2.1.4 Microkernel: seL4

Microkernels, as a contrast to monolithic kernels, try to minimise amount of functionality in the kernel. Any extra OS functionality is implemented outside the kernel whenever possible [Liedtke, 1995]. This additional functionality is implemented by using user-mode processes commonly referred to as servers. Communication is achieved through Inter-process communication (IPC) message passing. Original microkernels became known for slow IPC [Bershad, 1992] and many popular microkernels started including more OS services in the kernel as performance optimisations [Singh, 2006, page.50]. However Liedtke demonstrated that IPC can be fast [Liedtke, 1993] by illustrating OS IPC rates orders of magnitude faster than previous implementations. This led to a resurgence of microkernel OS development.

By minimising the amount of functionality implemented in the kernel, a microkernel based OS can considerably reduce the chance of a fault occurring and crashing the system. If a fault occurs in an OS service, such as a device driver, the fault occurs in user-mode and the kernel is invoked to handle the fault and the remainder of the system keeps functioning correctly. The system services that still need to be implemented by the kernel are usually basic resource management, access control, scheduling, and low level IPC.

seL4 is an example of a microkernel. It is used to build secure high-assurance systems. Its design specifies that the kernel will never fault provided it was initialised correctly. There is a comprehensive formal verification that under the right circumstances, the implementation of seL4 correctly implements its design down to the binary level [Klein et al., 2014]. This formal verification was only possible because the implementation of seL4 is very small compared with a monolithic kernel. seL4 supports 2 architectures and only runs on a handful of devices. We provide an overview of seL4 followed by some details relevant to this thesis in section 5.

Throughout this thesis we also refer to unikernels and rump kernels. A unikernel is a purpose built kernel for a single application [Madhavapeddy et al., 2013]. When the application is compiled, only the OS components it needs are included in the final binary. Everything executes in the same address space and

usually in a privileged processor mode. A rump kernel is a name used by the rump kernel project and does not refer to a particular design[2]. Rump kernels are usually used to run OS components in a non privileged mode. As a quick summary, monolithic kernels, microkernels and unikernels are kernel designs, but rump kernels refers to a project name.

2.2 Operating System components

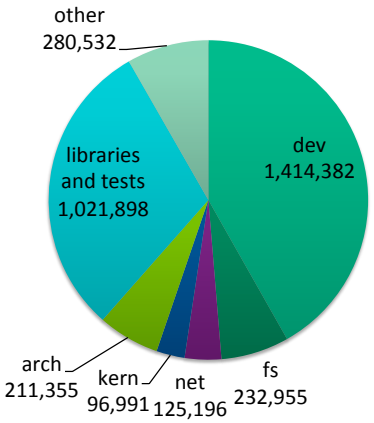


Figure 1: Lines of code in NetBSD kernel

We include a brief description of the various components that make up an OS. Figure 1 shows a breakdown of the lines of code in version 7.99.29 of the NetBSD kernel[3].

2.2.1 Device drivers

Device drivers are the device specific code within an OS that have the capability to interact with underlying hardware. Functionality specific to an individual I/O device is encapsulated in its specific driver. This can include how the device is addressed, how interrupts are handled and how to translate application requests to a format that the underlying device can understand.

As can be seen in Figure 1, device drivers usually make up a significant portion of a modern OS. This is not because drivers are overly complicated, but because there are a very diverse range of different devices. Within one platform, there are devices for timing, storage, networking, entropy, audio, video, user input among others. Every manufacturer has proprietary hardware and thus requires a different driver. In order to support a wide range of platforms, an OS needs to provide a large amount of drivers.

As drivers rely on performing tasks such as interrupt handling and interacting with physical resources, they often require to be run in a privileged mode and hence are often located in the kernel. Drivers are the largest sources of kernel faults [Chou et al., 2001] and implementing them in user-mode is often considered to be inefficient. However it has been shown that it is possible to implement drivers in user-land and still have acceptable performance [Leslie et al., 2005].

[2]The design is an Anykernel
[3]Generated using David A. Wheeler's 'SLOCCount'.

2.2.2 File systems

Filesystems provide an interface for applications to store and retrieve information. They take an underlying storage mechanism and present a file based interface for the application to store, retrieve and manage data. As can be seen in Figure 1, filesystem code also accounts for a large amount of code in an OS. This is to account for the different methods of storing information, such as file based storage, record based storage, or raw data. There are many different storage mechanisms such as disk, flash, tape, or no local storage at all such as network based storage. These different storage mechanisms require different strategies for reading and writing to the underlying data.

Filesystems require use of an underlying driver and are also often located in the kernel. They are also a source of kernel faults such as faulting when loading incorrectly formatted data [Yang et al., 2006]. There are a few ways of implementing file systems in user mode that rely on opening a block device through a /dev/ interface or by using socket IPC [Mazieres, 2001].

2.2.3 Networking

Networking can be considered an extension of IPC where the process is on a different machine in a different geographic location. Network communication can be done over different mediums (Ethernet, WiFi, GSM are examples) and by using different protocols, (TCP, UDP). Today, traffic is almost exclusively routed using IP, but there but there are other protocols and many OSes provide support. The networking subsystem in an OS provides support for this and exposes a simple unified interface (in POSIX this is the socket API) for applications to communicate.

Networking is located in the kernel for performance reasons, most networking is done by sending small packets of information at sufficiently high frequencies that constantly changing modes would be inefficient. However it has been demonstrated that efficient networking stacks can be implemented in user-mode [Jeong et al., 2014].

2.2.4 Core routines

The core kernel routines implement core kernel functionality such as process management, scheduling, memory management and IPC. They cannot easily be migrated to user-land for reasons such as a dependence on privileged operations.

2.2.5 Architecture specific

An OS can be split into two sections, the part of the implementation that is architecture independent, which can be the same for all architectures, and the part that is architecture specific. This is almost entirely low level code where abstractions break down. It includes trap and exception handling code, hardware execution contexts (register layout), low level interrupt handling, thread local storage, memory mapped I/O registers and emulated hardware instructions.

2.3 Low level operating system primitives

While the above components are higher level components that the OS makes available to applications, the following low level primitives are required to implement them: the execution model, access control, memory management, I/O, IPC, synchronisation and timing. There are different designs for providing these primitives and if we want to take the OS services from one OS and reuse them in another, we need to consider different designs of managing them.

2.3.1 Execution model, scheduling and interrupts

The execution model is how the OS manages access to the system's processors. In a typical environment there are many competing tasks requiring access to the system's processors. The OS has to manage access in a way that is fair and efficient for the many different types of activities. The execution model can be broken up into two components, how the OS schedules work, and how the OS handles interrupts. Scheduling is associated with picking which tasks to run, deciding how long they can run for, and deciding whether to stop a task from running so that another task can run. Interrupts are asynchronous events that can be externally triggered and normally need to be processed quickly. Unless the OS has blocked external interrupts, when an interrupt occurs the processor will jump to a special location for the OS to process the interrupt and then return. There are different execution models that incorporate different approaches to handling these two things. We briefly describe the various models: threads, events, coroutines and continuations.

Threads are a way of expressing tasks as units that can be taken and run on a processor, removed from one processor and run on another one, or suspended. A thread has some local state such as the current value of all of its registers, the position in the program it is running, and a stack where it has stored local variables. An OS can have a large number of threads, and use a scheduling algorithm that decides what thread should be run when a processor is free.

There are two ways of scheduling threads, pre-emptive scheduling and cooperative scheduling. If an OS has pre-emptive scheduling, a thread can be stopped at any point. This allows the OS to handle interrupts whenever they occur or quickly change to a different thread if it has a higher priority. The disadvantage of this is that it's dangerous to operate on shared data without using synchronisation strategies which can be complicated to do correctly. The other approach, cooperative scheduling solves this problem by allowing a thread to continue executing until it has finished. However with this design, if a thread has an error, or is selfish, it can stop other threads from getting a fair turn by never finishing. It is also possible to miss interrupts if the OS doesn't process interrupts while the thread is executing. Most OSes use a pre-emptive scheduling model.

Coroutines are an approach of solving this problem, where long running tasks can explicitly yield and remove themselves from the processor. When it is their turn to run again, they get scheduled back on the processor and a different entry point is called and they resume executing the task.

Events are a model where the tasks are represented by a series of events in an event queue. The OS takes an event from the queue, processes it and then repeats. Processing events can cause other events to be added to the queue. If events are quick to process, then the entire event can be processed before checking for interrupts. This also means that shared data is more safe to operate on. However if some events require a long running operation, then it may be infeasible to process the entire event immediately.

Continuations are one approach for solving this problem, a continuation is where a task can be broken up

into checkpoints, and when a program reaches a checkpoint, it saves the state that it requires somewhere and suspends to be continued again later. With an event model, continuations allow events with long-running operations to have those operations broken up into smaller pieces that can be treated as if they are separate events.

2.3.2 Access control

Access control is how the OS controls which applications have access to certain computing resources. The OS needs to keep track of who (subjects) can access what (objects). This can be expressed abstractly as an access control matrix where at any point in time there is a record for every enumeration of subjects and objects that specifies what access the subject has to the object (i.e. read or write access).

Access control lists are one approach of implementing access control where for each object the OS has a list of subjects have what access [Sandhu and Samarati, 1994]. In most OSes this method is used, such as the ownership structure of files and folders in most UNIX OSes with owner, group and everyone permissions. Another mechanism of access control is capabilities. A capability design is where each subject has a list of capabilities to the objects that it has access to [Sandhu and Samarati, 1994]. Capabilities allow for easier propagation and restriction of access as well as revocation.

2.3.3 Memory Management

Memory management is how the OS controls access to the volatile memory of the system. All OS services need memory for tracking state or interacting with applications. Memory can be allocated to services statically or dynamically. Static allocations cannot be changed and if a service runs out of memory, it has to either find some of its own memory to reuse, or crash. Dynamic memory allows services to request more memory, but is more complicated to implement as now the system can run out of memory if services ask for too much. The OS can implement mechanisms for reclaiming memory it believes is unused such as swapping it out and saving it to disk. This can cause problems in cases such as device drivers, where the driver needs memory readily available for the device to directly read from which can cause problems if the OS decides to use that memory region for something else.

2.3.4 I/O

Drivers use I/O to interact with hardware devices. There are different ways that the OS can interact with underlying hardware. Sometimes the processor provides special privileged instructions to read and write data to devices using special hardware registers (PMIO). Another approach is using Memory Mapped I/O (MMIO) where I/O devices are mapped into addresses in the address space. To access a device, the device's memory is accessed and device controllers can monitor the memory bus for addresses assigned to that device. MMIO is more efficient than PMIO and modern architectures favor it.

Direct Memory Access (DMA) is a method that allows devices to directly read and write to memory without involving the processor. This can enable higher bandwidth I/O as the processor is not needed to process I/O data, just coordinate the memory DMA addresses.

2.3.5 Inter process communication

Inter-process communication is how processes communicate with each other. IPC can be implemented by using shared files, shared memory, network, or dedicated OS mechanisms. Historically, IPC was considered too slow to use in core OS functionality. Microkernels need to have fast IPC to allow system services to operate efficiently.

2.3.6 Synchronisation

Synchronisation primitives are required when system resources can be accessed a limited number of times at once and also for communication between different tasks. Common synchronisation primitives are mutexes, semaphores, reader-writer locks, spinlocks and condition variables. A mutex enables a task to try and acquire a resource, but if that resource is held by another task, then the OS suspends the requesting task until the other task releases it. When a task acquires a mutex, it knows that it has exclusive access to the resource until it releases it again. Semaphores are similar to mutexes but they support multiple access slots and once all available resource slots are used, additional tasks will be blocked until a slot is free. Spinlocks are similar to mutexes where a task will be blocked until the resource is free, only the task will remain executing on the processor. They are used to synchronise across multiple processors. Reader-writer locks are locks that allow multiple readers at once, or only one exclusive writer. Condition variables are used for tasks to sleep until events happen where either one arbitrary task is resumed, or they can all be resumed. Synchronisation is hard do correctly, and if done incorrectly it can cause the system to deadlock, livelock, corrupt shared data, cause inconsistent execution and not make meaningful progress.

2.3.7 Timing

An external timing source usually provided by dedicated hardware is used to perform many low level important tasks, such as causing an interrupt every fixed amount of time to trigger the scheduler, or allowing device drivers to coordinate with their devices, performing networking, allowing tasks to run at certain times of the day, or tasks to sleep for a certain amount of time.

2.4 Summary

Operating systems are useful as they provide application portability and fault isolation. Application portability is important is it reduces the engineering effort required to produce an application while also increasing the range of devices and applications that the application can be used for. Fault isolation is important because it makes the system more resilient to failures. Many operating systems do not provide a high level of fault isolation because of the large amount of code that executes in privileged mode in the kernel. Device drivers, and driver-like components such as filesystems and networking stacks, which are needed for portability, contribute millions of lines of code to a monolithic OS kernel, but do not need to be executed in privileged mode. seL4 can provide better fault isolation by moving these components out of the kernel to execute in a non privileged mode. Yet reimplementing this code is infeasible as it would require rewriting millions of lines of code that have been developed over many years already. Instead, it

is possible to reuse the code and there exist different strategies to achieve this that will be discussed in the next section. In order to move this code correctly, it is important that the low level primitives that it relies on: the execution model, access control, memory management, I/O, IPC, synchronisation and timing. In this thesis we use Rump kernels to allow us to reuse components from the NetBSD kernel on seL4 with minimal modification. Background for rump kernels and seL4 can be found in sections 4 and 5, after we introduce rump kernels as related work in section 3.

3 Related Work

There are several methods for reusing OS system services from one OS in another.

- Full virtualisation and paravirtualisation where the entire source OS is run by the target OS,
- Reverse engineering and porting is where the target OS ports services from the source OS,
- replicating runtime environments, which is similar to virtualisation but can be done at smaller scale than the entire source OS,
- utilising standardised interfaces, and
- using driver synthesis from hardware specification.

This section looks at the various methods including using rump kernels which is the chosen method for this thesis.

3.1 Virtualisation and Paravirtualisation

When an unmodified OS is run on top of another OS it is said to be virtualised. If the virtualised OS is modified such that it knows it is not running on bare hardware then it is said to be paravirtualised. In the context of this thesis, using virtualisation techniques allows the use of system services that the guest OS provides. This method is currently used by some seL4 projects as an approach of using device drivers.

3.1.1 Virtualisation

Virtualisation is method of running one OS on top of another by emulating the same hardware environment. A virtualised OS cannot easily tell if it is virtualised. Fault isolation is provided as if the guest OS faults and crashes, the host (base) OS remains running. Some difficulties associated with full virtualisation is that it requires the underlying architecture to be virtualisable. Virtualisation typically works by running the guest OS in user-mode and whenever the guest kernel tries to perform a privileged instruction the processor faults and the host OS emulates the behavior of the privileged instruction. This can pose difficulties for two main reasons. Firstly, it takes many cycles to process these faults and if a kernel assumes it is in a privileged mode and performs these instructions frequently, it can significantly reduce the performance of the system [Barham et al., 2003]. Additionally, it may not be possible to emulate the underlying hardware. Originally x86 was not virtualisable [Robin and Irvine, 2000] as certain privileged instructions would silently fail if performed in user-mode. When this happens the host OS cannot emulate the behavior.

3.1.2 Paravirtualisation

As a response to the above problems with virtualisation, paravirtualisation techniques were developed which modified the guest OS to explicitly call the host OS to perform more privileged instructions instead of the host OS trapping and emulating. This approach enabled virtualised OSes to run on a host with minimal overhead [Barham et al., 2003].

Eventually better hardware support was released that made it easier to do full virtualisation, such as the x86 and ARM virtualisation extensions. Generally most modern OSes can be virtualised with less than 10% overhead compared with native execution [Dall and Nieh, 2014].

Virtualisation techniques are effective for providing a large range of driver-like OS components as it provides the same level of application portability as the virtualised OS through using the same components. However, running an entire monolithic kernel has a large runtime memory footprint as well as the overhead required by the host OS to support itself. It is unsuitable to run an entire monolithic kernel just to get access to one component.

3.2 Reverse engineering and porting

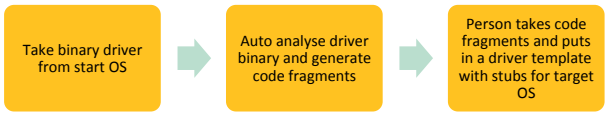


Figure 2: RevNIC: Reverse engineering of binary device drivers

Reverse engineering and porting approaches look at taking existing OS functionality and re-implementing it for the target OS. RevNIC is a tool that tries to automate this process [Chipounov and Candea, 2010]. Looking at Figure 2, the process takes an existing binary driver, analyses it by 'wiretapping' its I/O operations while running it and then uses these operations to generate code fragments of the behavior. An experienced developer then takes the code fragments and fills in an existing driver template for the target system.

This approach is designed to make it easy to re-implement drivers in situations where a driver vendor releases binary only drivers for a few OSes. The approach is intensive per driver and needs to be repeated if the source driver is updated due to improvements or bug fixes. Additionally the developer needs to be experienced at driver implementation for both the source and target OS. This method is not appropriate for what we want to do in this thesis because it is high effort and can only be applied one driver at a time which makes it an infeasible way to reuse a large amount of drivers.

3.3 Interfaces

Uniform Driver Interface (UDI) was an Intel project to provide a unified interface between device drivers and OS kernels [Barned and Richards, 2002]. The goal was to be able to have one driver for a certain device and run it on any OS that implemented the interface. The project is now defunct as it never gained enough momentum. It would have required existing device drivers to be rewritten and also allowed hardware vendors to release more binary only drivers which make it hard to guarantee the absence of bugs if the source code cannot be analysed.

3.4 Driver synthesis

Driver synthesis is the process of generating a device driver from an existing formal specification of the device and the target OS by using a tool, Termite [Ryzhyk et al., 2009]. In order to generate a new driver, Termite takes a driver specification and OS specification and generates a driver implementation that can translate OS commands into a device commands. What makes this approach novel is that the generated driver is bug free according to the provided specifications. However it does not guarantee that the driver is efficient and a developer is required to guide the synthesis in order to generate efficient implementations. Additionally, device specifications are not released by vendors and must be manually created from reading the hardware manual which is prone to errors. In the future this could be a better method for generating a wide range of drivers, however currently the project is not mature enough to easily use for our purposes.

3.5 Replicating runtime environments

Replicating runtime environments refers to taking functionality, such as a driver, from a source OS and then running it unmodified in another OS by emulating the same environment. This is similar to a virtualisation approach however it can be used to run a smaller amount of functionality than the entire OS. There are different approaches of implementing this that work at different abstraction levels from implementing higher level internal OS interfaces, to providing the same low level OS primitives.

3.5.1 Driver wrapper

Driver wrappers replicate the runtime environment of a single driver. NDISWrapper was a project that enabled the use of Windows XP network device drivers on Linux [Kiszka et al.]. This was achieved by implementing the Windows kernel Network Driver Interface Specification (NDIS) APIs in Linux. This made it possible to run an unmodified network driver from Windows on a Linux system. This approach minimises runtime overhead to perform the emulation as the target OS only needs to emulate functionality that is needed on a per driver basis. However it is not low effort as the target OS still needs to implement the required APIs and this only provides one class of driver. Also, very few wrapper APIs are available to do this.

3.5.2 L4RE and DDE

The L4 Runtime Environment (L4RE) and Device Driver Environment (DDE) was a project for the original L4 microkernel and the Fiasco microkernel [DROPS]. It uses OS specific glue code to provide an environment for device drivers to run unmodified in a microkernel environment . Work was done to reuse unmodified Linux drivers [Helmuth, 2001]. There is no support for this with seL4, however if there was it would meet our requirements for reusing unmodified drivers from other OSes. However, the work required to add support in seL4 would be greater than the scope of this thesis.

3.5.3 rump kernels

The rump kernel is a NetBSD project that exposes NetBSD kernel modules to be used in other environments [Kantee et al., 2012]. Projects have used this to provide a library OS for single applications [Kantee, 2016]. Support for running rump kernel modules requires the host to provide a series of low level OS primitives through defined host interfaces. By implementing these interfaces once, it is possible to use the rump kernel modules which include many OS services unmodified from the NetBSD kernel. It could also be possible to run these modules in isolation from each other in different rump kernels on the same system. The next section will go into rump kernels in more detail as it is what we will use for this thesis.

3.6 Summary

When reusing driver-like OS components in other systems, we want our approach to be low effort to reuse a large range of components and we want the runtime overhead introduced to be low. We looked at Virtualisation and Paravirtualisation approaches which have low effort but introduce high runtime overhead. Reverse engineering and porting approaches potentially have low overhead but is high effort to reuse each component. Interfaces and driver synthesis both can be good approaches but are not available for a large range of components. Replicating runtime environments is a method that has been used before with success on L4 based systems, but requires an upfront investment which has not been performed for seL4 yet. However, rump kernels seem like an approach where this upfront investment is low and allow us access to a wide range of components with low effort and low runtime overhead. More detail on rump kernels is provided in the next section.

4 Rump kernels

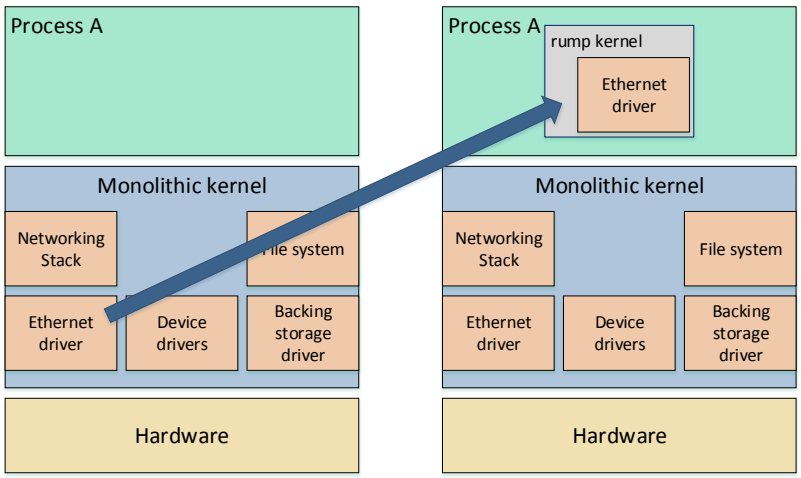


Figure 3: Moving drivers to user-mode using a rump kernel

The rump kernels project within NetBSD is a project to make NetBSD OS components more portable to allow reuse in other systems. Device drivers, filesystems and networking modules can be run inside a rump kernel in an unmodified way. The rump kernel can then be easily ported to run in different configurations such as in a different OS, on top of a POSIX application interface or as its own system. This is made possible by a set of common software interfaces that are designed to provide low level OS primitives to the rump kernel. The rump kernel then uses these primitives to support the NetBSD OS components. This section presents a summary of the rump kernel design, the host interfaces that systems can implement to support rump kernels and also describes the Rumprun unikernel which is a project that uses a rump kernel to provide OS components to single address space applications without needing an actual OS.

4.1 Motivations for rump kernels

The original motivation for rump kernels was to make it possible to run unmodified NetBSD modules in user-mode to increase the speed of development and testing them. Without rump kernels, a developer would have to update the kernel with the modified driver in order to test new changes. This was non-ideal as it increased the time taken to get feedback on a change as well as limiting the amount of debugging tools available. A solution to this problem was to provide a way for these modules to be run in user-mode on the same system that they were being developed for. This would make it faster to test changes as well as providing access to better debugging tools. These motivations were what drove the initial rump kernel design requirements:

Modules should be unmodified: It needs to be possible to take a NetBSD kernel component and be able to run it in a rump kernel without significant modification. Having to significantly modify modules in order to run them in user-mode would limit the quality of testing feedback and not increase the speed of development. Additionally if a module has previously been run inside a rump kernel, any additional changes to the module should not reduce its ability to be run in a rump kernel.

Minimal amount of kernel code: The amount of additional code required to support the module

should be minimal. In order to provide a benefit over requiring the full NetBSD kernel as support to run the single driver, running it in the rump kernel should require less overall code. This will also enable the rump kernel to be more resource efficient and faster to initialise.

Easy to add new components: It shoudl be easy to add components to the rump kernel. Again, in order to make module reuse low effort.

Modules should still act the same way: There should be a low performance impact and the modules should still function the same way.

An additional requirement was later added when it became apparent that rump kernels could be used to make NetBSD modules generally reusable in a much wider range of applications than for testing and development for the NetBSD kernel.

Easy to reuse rump kernels: It should be easy to run rump kernels in many different systems. Thus the underlying APIs that the rump kernel uses should be generalised and not tied to running in user-mode of NetBSD systems.

Many of these motivations are aligned with our own goals for driver-like OS component reuse on seL4. In the next subsection we summarise the design of rump kernels and also describe some potential issues involved in choosing a good design that achieves the above requirements.

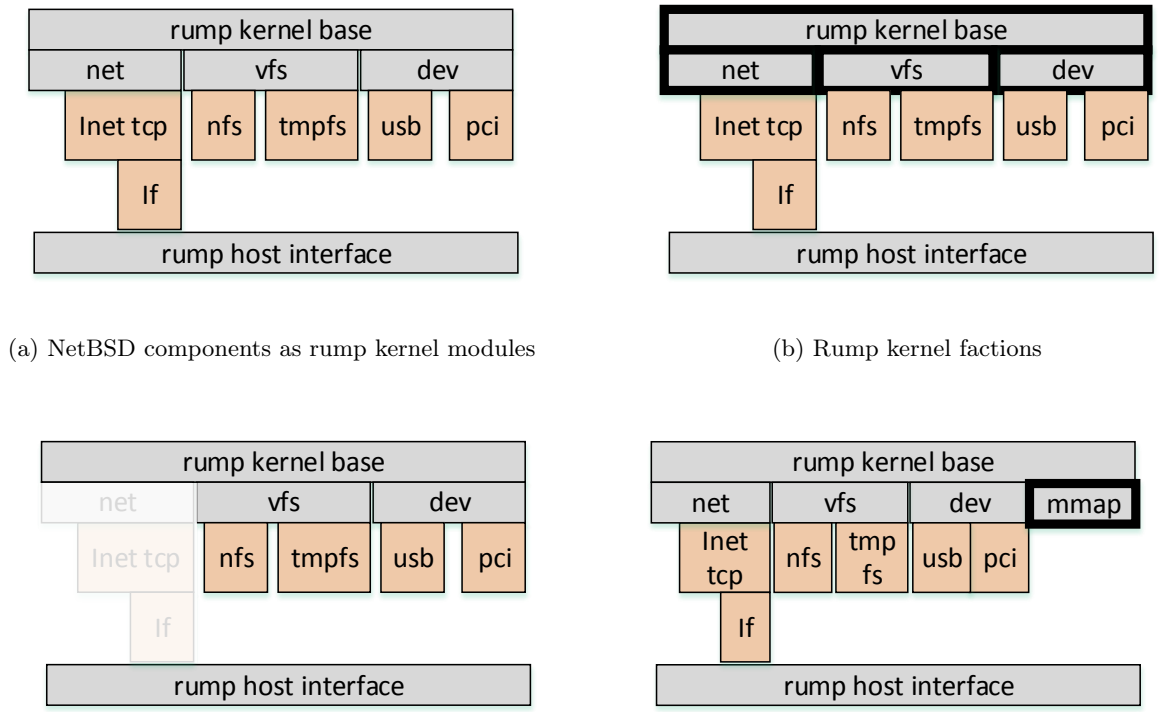
4.2 The rump kernel design

The rump kernel design is based on taking NetBSD components and providing them as modules that can be loaded into the rump kernel. Common kernel functionality is provided by the core rump module and three extra modules known as the three factions: net, dev and vfs which contain common functionality for networking, device and filesystem components. It is possible to add additional modules fairly easily. This subsection describes some details about rump kernels relevant to this thesis.

4.2.1 Modules

A typical rump kernel structure is shown in Figure 4a. In a rump kernel each component is provided in a module. For example, a component for an ethernet card driver has its own module. Different components are provided in different modules. If the ethernet card is connected to a PCI bus, the driver for the PCI bus is in a different module from the ethernet card driver module. Modules for components that are not needed in the rump kernel do not need to be provided. This reduces the amount of code required to run rump kernels compared to running the NetBSD kernel.

Modules need to depend on other modules, and a final application needs to interact with these modules through the rump kernel. The kernel provides a system call (syscall) table for applications to communicate with it. When the kernel is initialised, many of the syscalls are unimplemented. Modules are able to register syscall handlers with the syscall table so that when an application makes a rump syscall, the request can be handled by the module. Additionally, if the virtual file system module is present, modules can register device nodes in the /dev directory with the vfs module. This allows an application to interact with modules using the standard unix file interface.



(a) NetBSD components as rump kernel modules

(b) Rump kernel factions

(c) Only use modules that are needed

(d) Providing additional modules

Figure 4: Rump kernel figures

Modules interact with each other in different ways depending on their dependency relationship. A module can have strong or weak dependencies on other modules. A strong dependency means that the module cannot run if the other module is not present and a weak dependency means that a module can take advantage of other modules if they are present, but is not required for the module to run.

The main way that strong dependencies are handled is by function interfaces that the compiler linker links together when the application is being built. The PCI ethernet card we mentioned before, depends on an interface that the PCI bus driver provides. When the application is built, if the PCI bus driver module is present then the linker will link the ethernet card driver pci function calls with the bus driver. If the bus driver is not present, then the application will fail to build.

Weak dependencies are handled with other mechanisms. VFS device nodes can be used which allow modules to interact with each other, but not cause the application to fail to build if one module is not provided. Additionally, link-sets are used. Link-sets are when the compiler linker makes a list of links(pointers to data or functions) with a special section name in the binary file. This is used when a module wants to do something to an unknown number of other modules. The main example where this is used is when a module is initialising other modules. A module responsible for initialising other modules does not know at compile-time if the other modules are going to be available in the kernel. The link-set acts as a list where if other modules were included, the linker added them to this list, and now the module can initialise them by iterating through this list.

With this design we end up with a module hierarchy where modules can depend on other modules, but only the modules that are needed by the application need to be included in the rump kernel. This reduces the size of the final kernel due to the lack of unnecessary lines of code.

4.2.2 Modularization with factions

In order to achieve the goal of having a minimal amount of NetBSD kernel code needed to support rump modules, the components were separated into three categories.

Core shared-functionality is included in the base rump module. This is required in every rump kernel and includes initialisation code, syscall dispatching and other common interfaces that modules commonly need such as memory allocators and scheduling. It has no strong dependencies on any other modules and can be initialised by itself.

The remainder shared functionality is split into three categories: devices, networking and virtual filesystems. These then become three modules: dev, net and vfs. While they have strong dependencies on the base rump module, each of the three modules cannot have strong dependencies on each other. Each module provides common interfaces required for the driver-like NetBSD components to be included unmodified in the rump kernel. These core modules are highlighted in Figure 4b.

The final category is the actual NetBSD kernel components. These are able to be turned into rump modules with minimal effort and modification. This structure should allow the minimisation of extra kernel code required in rump kernels as only the modules needed to support the desired NetBSD kernel components are required. The remainder of the modules can be excluded as shown in Figure 4c. Additionally, the existence of the three dev, fs and vfs modules should enable the components to be run unmodified.

4.2.3 Bring your own modules

Adding a module is fairly easy. All the components exist in the same address space so additional modules can be compiled and linked into the binary in the same way. Figure 4d shows an additional mmap module that the Rumprun unikernel adds to provide memory mapping functionality to the rump kernel. A new module can depend on other modules using all of the mechanisms discussed above: direct linking, providing or using device vfs nodes and using link-sets. One of the link-sets allows a module to register any syscalls that it can provide that can then be called by the application.

If the module is added to the dev, vfs or net initialisation link-sets then the module will be initialised by these modules upon initialisation. At this point the new module is the same as any other rump kernel module.

New modules can be existing NetBSD kernel components, or additional components from third-parties. Additionally, an existing rump kernel module can be replaced with a customised one if necessary. The ability to easily add additional modules that can be based on NetBSD components or custom modules makes it easy to extend the functionality of the rump kernel.

4.3 Kernel configurations

Rump kernels can be used in different configurations. There are two configurations relevant to seL4. In one configuration, the rump kernel runs in the same process and address space as the single application that is using it. In Figure 5 this is the local configuration. In this configuration the rump kernel is used as a library OS for a single application running in that environment. Underlying software or hardware is used to

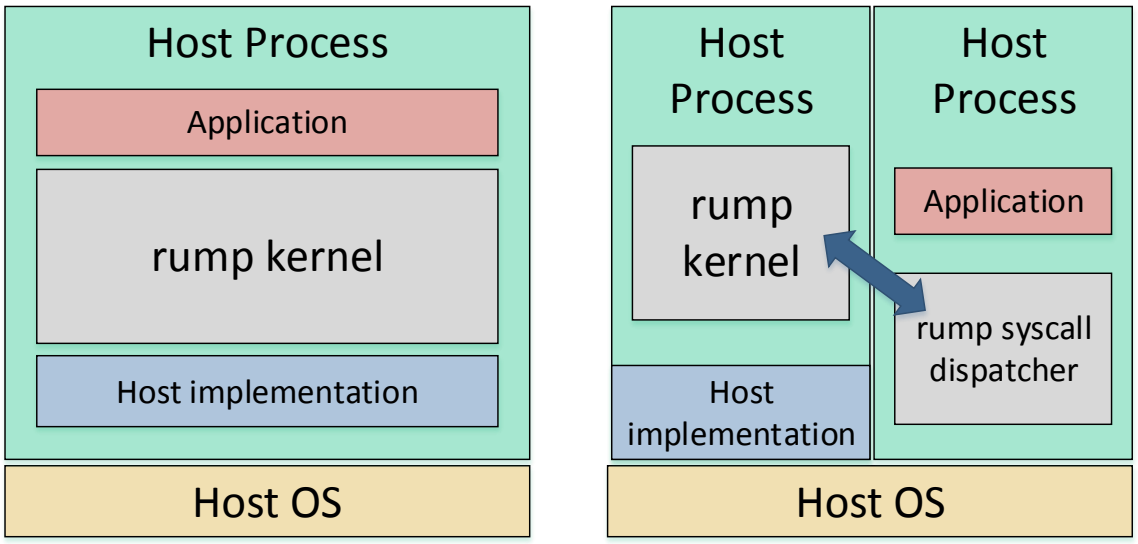


Figure 5: Rump kernel local and remote configurations

isolate different applications, each with their own rump kernel, from each other. This is the configuration that the Rumprun unikernel uses and is also the configuration that we use for this thesis.

The other configuration is a remote configuration where the rump kernel is used to implement a server that provides certain OS system services. The rump kernel provides a syscall proxy module that can redirect syscalls from an application in a different address space to a rump kernel existing elsewhere. This could potentially be used to provide fault isolation between NetBSD OS components running in a rump kernel and the application using it. This is discussed more in the future work section.

4.4 Software interfaces

In order for NetBSD kernel components to function correctly and efficiently as rump modules, a mechanism is needed to access the same low level OS primitives that are originally used. This is achieved with the rump host interfaces. These interfaces allow rump kernels to be provided low level OS primitives that are otherwise controlled by the host system the rump kernel is running on. This subsection looks at what is provided by these interfaces. The actual interfaces can be found in Appendix B.

4.4.1 Thread and execution model

The rump kernel's execution model is cooperative multi-threading on virtual CPUs. A virtual CPU is an abstraction of how the host system runs rump kernel tasks. When the kernel initialises it is given a value of how many virtual CPUs are available. This informs it how many rump kernel threads that the host system will run concurrently. Concurrently can either mean in parallel on separate physical CPUs, or time-sliced with preemption on a single physical processor. A thread inside the rump kernel is known as a light weight process (lwp). Rump kernel threads require host system support and can be created and destroyed by the rump kernel through calling into the host interface. The rump kernel performs its

own scheduling of lwps on to virtual CPUs internally, but the underlying hardware can schedule which virtual CPUs are actually being executed. The host to inform the rump kernel of its scheduling behavior for optimisation purposes.

It is also possible for the host to remove a rump kernel thread from its virtual CPU. This usually happens when the rump thread becomes blocked on a synchronisation primitive (also provided in the host interface) and the virtual CPU can be used to process other tasks. The host interface provides call backs that allow the host to call functions that schedule and unschedule rump kernel threads on virtual CPUs.

There are often situations where host threads call rump kernel functions. When this happens, the rump kernel may not be aware of threads created by the host but not by rump. Upon entry to the rump kernel, any non-rump-created thread temporarily becomes a lwp and needs to be scheduled on a vCPU before it can run.

Interrupts are delivered into the rump kernel by scheduling a host thread on a free rump virtual CPU and then calling into the kernel. The thread then registers a soft interrupt and returns. The rump kernel can then processes the soft interrupt using one of its own threads.

4.4.2 Device I/O

There are three main ways of performing I/O in rump kernels. For networking there is a POSIX socket interface that the host can choose to implement (Appendix B.2). This method allows the host to provide socket I/O and its own networking stack underneath. The second method is a PCI interface for using PCI device drivers (Appendix B.3). The rump PCI ethernet driver would use this interface. Finally, a file based I/O interface is included in the main host interface (Appendix B.1). This method allows the rump kernel to call open on the host with a special device name and then use the device as a character or block device. This is used to allow some rump drivers to run on top of lower level host drivers. An example of this is a low level USB host controller provided by the host that the rump kernel's USB mass storage driver can use.

4.4.3 Virtual Memory

For virtual memory, the rump kernel calls malloc in the host interface but also specifies an alignment value that the returned memory must be aligned to. There is also an interface for a memory map function to allocate larger amounts of memory. The rump kernel can perform its own paging internally and usually acquires all memory it requires at start-up. There is no explicit way for the host to reclaim memory given to a rump kernel.

4.4.4 Timing

The rump kernel requires two clock sources, a wall clock and a monotonically increasing count since boot. Additionally, the host needs to provide functionality for a sleep call that will return after a fixed amount of time.

4.4.5 Synchronisation

The rump kernel requires three synchronisation primitives: mutexes, reader writer locks and condition variables. They are expected to have standard behavior. Synchronisation is delegated to the host in order to handle low level synchronisation hardware instructions and rescheduling rump kernel threads as they become blocked and unblocked. There is a configuration option when the rump is operating in an environment with only one processor. In this case many rump kernel synchronisaton calls are disabled.

4.4.6 Other

The remainder of the host interfaces are for miscellaneous functionality such as configuration, dynamic linking, debugging, signal handling, and remote system call dispatching.

4.5 Rumprun unikernel

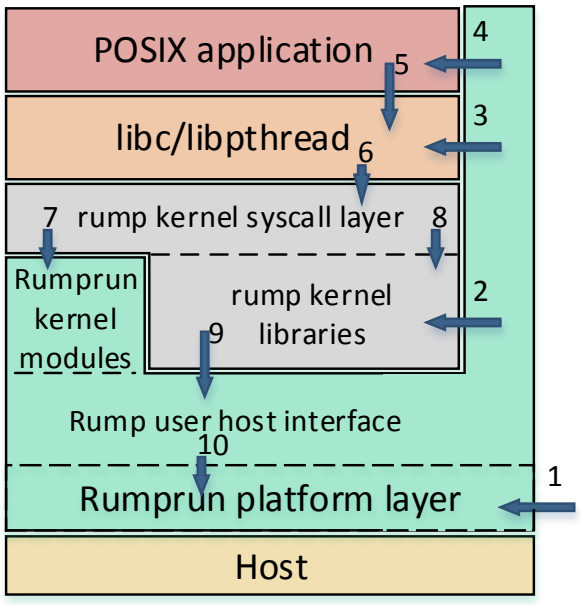


Figure 6: Rumprun system architecture

Rumprun is a unikernel project that uses a rump kernel to provide OS functionality to singular POSIX applications. This design is commonly used when an application does not require the support of a full time-sharing operating system.

The system architecture of the Rumprun unikernel is shown in Figure 6. The entirety of the unikernel runs in kernel mode and every component shares the same address space. Therefore it would be possible for the posix application to bypass the rump kernel and interact with low level hardware directly. This is allowed because the Rumprun unikernel assumes that it is running on a hypervisor and the hypervisor is

providing fault isolation and protection between different applications, where each application runs inside its own Rumprun unikernel.

The following steps describe how the system uses the rump kernel to support applications. Each step is also shown in Figure 6:

1. The system is booted. Rumprun's low level subsystems are initialised.
2. Rumprun initialises and configures the rump kernel. There is one rump kernel and it is configured to run on only one virtual CPU. In this step, any configuration settings such as networking or disk drives are initialised in the rump kernel by the Rumprun system.
3. Rumprun initialises the libc and libpthread runtimes.
4. Rumprun enteres the POSIX application's main thread.
5. POSIX application operates as normal. When it wants to interact with system resources, it does this by using the C standard library.
6. The libc and libpthread libraries perform syscalls if necessary. These are handled by the rump kernel syscall layer.
7. Some syscalls will be redirected to special Rumprun rump kernel modules. This method is used to provide mmap functionality that rump kernels do not natively provide.
8. The remainder of the syscalls are handled by the rump kernel.
9. When required, the rump kernel will interact with the low level system through the rump kernel host interface. This is implemented by the Rumprun unikernel.
10. The host interface implementation interacts with the low level Rumprun platform layer to perform actions on the system. At this stage Rumprun supports bare metal hardware or Xen as low level platform layers.

The Rumprun unikernel is included here as an example of using rump kernels in a practical setting. We plan on using the Rumprun unikernel to aid us in running rump kernels in user-mode on seL4.

4.6 Summary

Rump kernels take driver-like NetBSD kernel components and provide them as reusable modules that can run inside a rump kernel. Rump kernels can then be easily run in different systems, thereby allowing the NetBSD kernel components to be easily run in different systems. Rump kernels have the goals that modules should be minimally modified from original source, a minimal amount of kernel code is required to support them in the different environment, easy to add new components, components should still act the same way with similar performance and behavior and that rump kernels are easy to reuse in different systems. These goals align with our goals of: low effort to reuse a large range of components and low runtime overhead introduced on seL4. The Rumprun unikernel is an existing project that uses rump kernels to provide application portability on low-level hardware without the overhead of a monolithic OS. The Rumprun unikernel is designed to run on different low level application interfaces such as KVM, Xen or bare metal. This should make it possible to add support to Rumprun for it to run on seL4, making it possible to reuse OS components with low effort and low runtime overhead while also running POSIX applications allowing application portability.

5 seL4

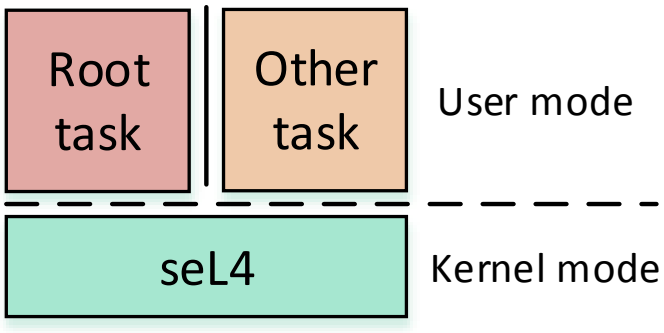


Figure 7: seL4 architecture

This section provides background about seL4 that will be relevant in later sections. In particular, the general system architecture of seL4 and how user applications interact with the kernel. The capability model that seL4 uses to manage resources and objects. What resources and objects that seL4 provides and controls. What IPC mechanisms that seL4 provides and how they can be used. How hardware interrupts are handled by seL4 and delivered to user-mode processes. Finally, we finish this section with a brief description of a typical seL4 application structure that is used.

5.1 System architecture

Referring to Figure 7, the seL4 kernel is the only part of the system that operates in kernel mode. It has access to all system resources and can perform all privileged hardware instructions. All applications operate in user-mode. In order to use system resources and perform privileged instructions, applications must communicate with seL4 to gain access to the resources or have seL4 perform the instructions on the application's behalf. Interactions between applications and the kernel are done with syscalls. seL4's syscalls stem from three basic syscalls: send, receive and yield. Send sends a message, receive receives a message, and yield yields the remainder execution time of the thread to the kernel. There are additional debug syscalls that are available when the kernel is built with different debug settings. These are used for debug printing and measuring certain benchmarking information for performance evaluations.

5.2 Capabilities

seL4 uses capabilities to manage access to system's resources and objects. We previously described capabilities as a way to record a subject's (user) access to an object. In seL4 every resource or object that the kernel manages has associated capabilities that record what rights user applications have. Each process has a list of capabilities that are stored in the process' capability space (cspace). If a process needs the kernel to perform an action on an object or resource, the process performs a send syscall with the capability ID provided in the message contents with additional parameters included to inform seL4

how to handle the request. This is known as an invocation, as the process has invoked the capability. If the process has the correct rights, seL4 will perform the action. The first process started by seL4 is called the root task. It receives capabilities that give it access to all system resources and kernel objects when it starts. The root task can use these capabilities to interact and manage the system.

5.3 seL4 objects

We describe some seL4 objects. Some objects are omitted from this list as they are not relevant to the thesis. Further information about seL4 can be found in the user manual [Trustworthy Systems Team, 2016].

Memory Frames (ram): seL4 memory frames are regions of the system RAM that is available to processes. This memory can be accessed if a process maps the frame into its virtual memory address space. A process can only access memory that is mapped into its address space. Memory can be mapped in by calling a map invocation on a frame object.

Device memory: This is similar to RAM memory but it refers to memory mapped device regions. A region of device memory can also be mapped into a process' address space in order to be used.

Threads: seL4 thread objects represent a thread that can be scheduled to run on the processor. In seL4 threads can be preempted, which means that they can be stopped and temporarily replaced with a different thread on the processor. To control scheduling behavior each thread can be assigned a priority. seL4 guarantees that the runnable thread with the highest priority will always be scheduled. A thread's priority can be modified by an invocation on the thread object. A thread also has a cspace and can have its own virtual memory address space.

Page directory objects: A page directory object contains a collection of mappings of memory frames (device or RAM). Each thread has a reference to a page directory object which specifies what memory is mapped in its address space. Threads with the same page directories have the same address spaces and can access the same memory. Threads with different page directories have different address spaces and may not be able to access the same memory.

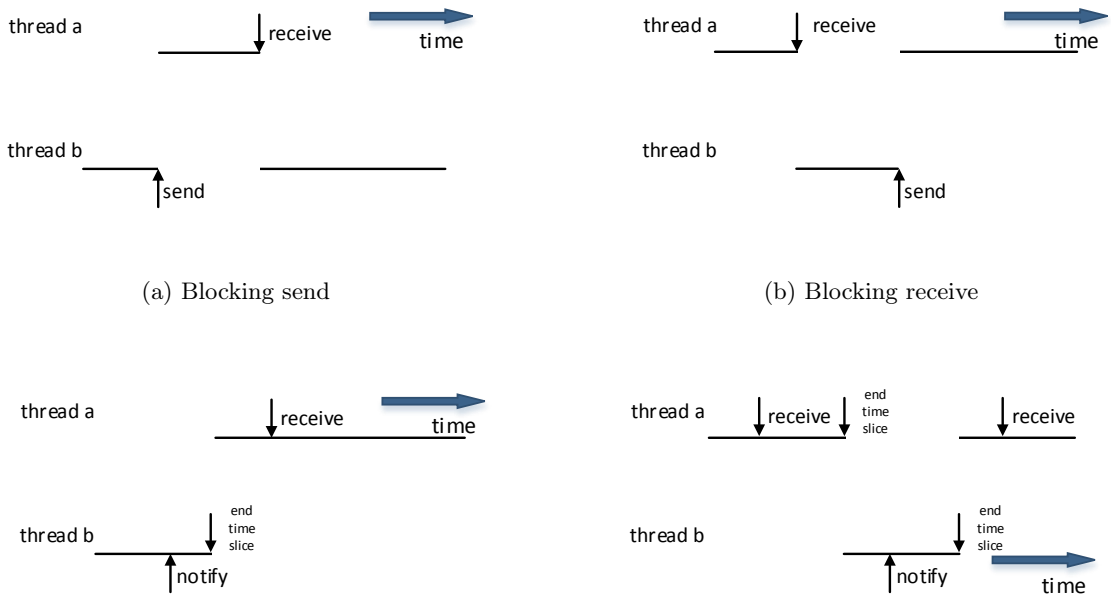
Synchronous endpoints: Synchronous endpoints are objects used for communication between different threads. A endpoint is used to send and receive messages. If a thread calls send on an endpoint, it will be blocked until another thread calls receive on the endpoint. seL4 can queue multiple threads in both waiting or sending states on the endpoint. It is a synchronous endpoint because a message will not be transmitted unless both the sender and receiver rendezvous.

Notification objects: Notification objects allow for asynchronous communication between threads and the kernel. A notification object can be signaled by a send syscall. This will change the object into a triggered state. It will remain in this state until a thread calls receive on the object. This means communication can happen asynchronously.

Interrupt handlers: Interrupt handler objects are used to manage the delivery of interrupts. An interrupt handler can be used to tell seL4 to deliver interrupts to a specific notification object. Threads can then receive interrupts on these notification objects. seL4 will mask further interrupts from the same source until the original interrupt is acknowledged. This is achieved by performing an acknowledge invocation on the interrupt handler capability.

IO Port caps: On x86 there are privileged IOPort processor instructions. They are used to read and write data to hardware devices. If a thread wants to perform these instructions, then it calls the read or write invocations on the IOPort capability. The kernel will then perform the processor instructions. A thread's permission to perform these operations is implied by it having the capability in its cspace.

5.4 Interprocess communication



(a) Blocking send

(b) Blocking receive

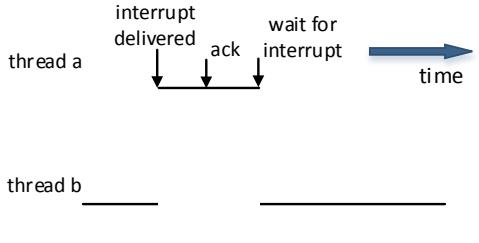
(c) Non-blocking send

(d) Non-blocking receive

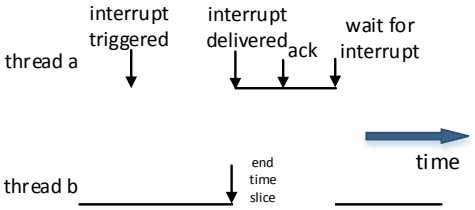
Figure 8: seL4 IPC mechanisms

seL4's interprocess communication mechanisms are used by threads to communicate with the kernel and with each other. IPC can be used to pass small amounts of data directly from one thread to another via a thread's IPC buffer. The IPC buffer is a small region of memory that is copied from one thread to another when IPC is performed. In addition to communicating with the kernel through invocations, IPC can be used by different threads to communicate or synchronise with each other. We briefly show how this is achieved. Figure 8 shows IPC using synchronous endpoints and notification objects. When using a synchronous endpoint, IPC must be synchronous. As shown in Figure 8a, when thread b sends a message, it will be blocked until thread a is available to receive the message. The same happens when a thread wanting to receive has to wait for a corresponding sender in Figure 8b. By using notification objects, sends and receives can be non blocking as shown in Figures 8c and 8d. This means that a sender can send without waiting for a receiver, and a receiver is able to check multiple times if a message has been sent. For a notification object the message length is less than 32 bits long and the messages for multiple sends get bitwise orred together. It is also possible for a receiver to block on a notification object. This mechanism can be used for construction of semaphores and other synchronisation primitives for use between seL4 threads.

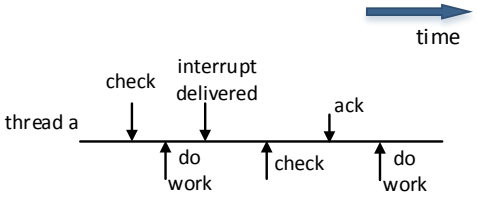
5.5 Interrupt delivery



(a) Thread with higher priority receiving interrupt



(b) Thread with same priority receiving interrupt



(c) Single thread handling interrupts

Figure 9: seL4 Interrupt delivery

Figure 9 illustrates different ways user-mode threads can receive hardware interrupts on seL4. When a hardware interrupt occurs, the processor will run the interrupt handler which involves entering the seL4 kernel. The kernel then checks to see if there is a notification object registered for that interrupt number. If no interrupt is registered then the interrupt is discarded and the source masked from sending further interrupts. If there is a notification object, then seL4 will deliver the interrupt by triggering the notification object. If there is a thread waiting on the object with a higher priority than the current running thread, seL4 will preempt the current thread and schedule the blocked thread to run. This is the example we see in Figure 9a. If the thread waiting is the same priority of the current thread, then it will not be woken up until seL4 schedules it normally. This is what is seen in Figure 9b. If no thread is waiting on the notification object, seL4 will still deliver the interrupt message. The message will be kept until a thread calls receive on the object. This is shown in Figure 9c where a single thread periodically checks for interrupts in between doing other work. When a thread receives an interrupt it can then process the relevant interrupt handler if there is one. seL4 will not deliver further interrupts from the same source until the interrupt has been acknowledged by acking the interrupt handler object.

5.6 Typical application structure

We finish this section with a brief description of a common seL4 application pattern. A typical application structure is one where the root task acts as a 'driver' for loading and configuring other applications. The root task receives capabilities to all system resources when it is started by seL4. The root task then can configure parts of the system as required through using the capabilities it is given. The secondary application is loaded by the root task using an in memory filesystem that contains its binary. The root task creates the new seL4 thread, cspace and address space objects and creates the initial RAM memory

frame mappings. It loads the applications binary into memory, sets the program counter of the new thread to the entry point of the new application, and sets the thread to runnable. seL4 can schedule the new thread to run, and a new application has been started. seL4 provides mechanisms not policy. This means that the root task can do anything with the initial resources that seL4 gives it, including 'shooting itself in the foot' through incorrect use. There are many seL4 libraries available to make managing seL4 objects easier and more user friendly.

5.7 Summary

seL4 controls access to underlying resources using capabilities. If a process has a capability to a resource, it is authorised to use that resource. The first task started by the seL4 kernel is the root task and it is given capabilities to all system resources. Usually the root task uses these resources to create and initialise a secondary process which it then gives a subset of the capabilities to. In addition to hardware resources such as RAM, device memory and IOPorts, seL4 can also create objects such as threads, messaging and interrupt delivery objects. Messaging objects can be used for synchronisation between different threads, while interrupt objects can control the delivery of hardware interrupts. By design, these resources and objects provide mechanisms instead of policy and can be used to provide low level OS primitives to driver-like OS components running in user-mode on seL4.

6 Approach

In this section we outline our approach for providing driver-like OS components on seL4 by using NetBSD rump kernels and also performing an evaluation of the performance of the resulting driver-like components and relative effort required to use rump kernels to achieve this goal. We reiterate our motivation for wanting a way to easily reuse OS components, then we outline our design requirements and how rump kernels can help us achieve them. We discuss the potential issues that may be faced and finally introduce our experimental setup that will be required to perform the performance evaluation and measure the relative effort required to use rump kernels. Section 7 will provide more low level information about our implementation.

6.1 Motivation

seL4 is a microkernel with highly critical applications. Its comprehensive formal verification makes it the world's most secure OS kernel. Real-world cyber-physical systems can use seL4 for increased security. However, to run seL4 on a wide array of heterogeneous devices, driver software is required, which is millions of lines of code, limiting seL4's deployment. Allowing driver reuse will increase the amount of drivers that work with seL4 which will ultimately increase the rage of real world devices that can be made more secure. There are existing solutions for reusing this driver software, but they either require a large amount of effort per driver, or the runtime overhead of using them is large. Therefore, we want to find a way of reusing device drivers and other driver-like OS components on seL4 in user-mode that is low effort per component and also has a low runtime overhead.

6.2 Design requirements

Our design requirements are as follows:

Low effort: We want it to be easy to take a driver-like OS component and run it in user-mode on seL4. Diverse devices and hardware platforms mean that there is a large amount of drivers available. If the effort required to run each component on seL4 is too high then it our solution is infeasible.

Low runtime overhead: We want to minimise the additional amount of hardware resources required to run the driver in user-mode compared with it running natively. We want to provide OS components with lower overhead than existing solutions such as virtualising the Linux kernel in user-mode on seL4.

Similar performance: We want to minimise the amount of performance degradation compared with the OS components running natively. If similar performance cannot be achieved, then the applications relying on the OS components will be negatively impacted.

6.3 Using rump kernels and the Rumprun unikernel

Comparing our requirements with the original requirements of rump kernels we see that they are similar. Rump kernels were designed to allow driver-like OS components to run in other environments with minimal

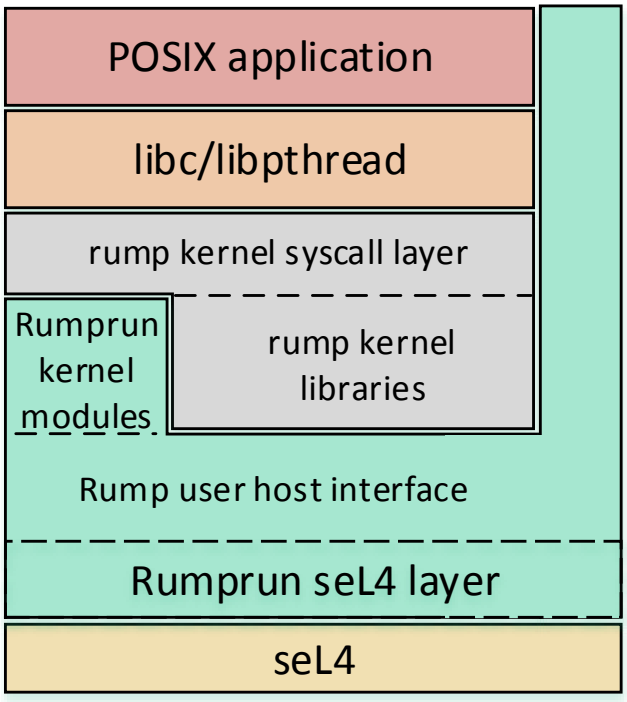


Figure 10: Rumprun system architecture

effort while still providing the same performance and with minimal runtime overhead. Additionally, the Rumprun unikernel is an existing project that uses rump kernels to provide a library OS to applications running in a single address space designed for running directly on hardware, either bare metal or on top of a virtual machine monitor such as Xen. By adding support to the Rumprun unikernel to run on top of seL4 we will be able to evaluate the performance of rump kernels and compare it with other systems. We will also be able to evaluate the effort required to use different rump kernel modules as well as add new ones.

The purpose of using the Rumprun unikernel is that it already has support for running unmodified POSIX applications. These applications are linked against the NetBSD C library which has syscalls handled by the rump kernel. Once the Rumprun unikernel can run on seL4 it will be possible use POSIX applications to evaluate the performance of the system, allowing us to directly compare our driver performance with other POSIX systems such as Linux and the NetBSD kernel that the drivers and other components originated. This will allow us to perform an effective evaluation of rump kernels and how they perform on top of seL4. Figure 10 is a diagram of the Rumprun unikernel with seL4 platform support added.

6.4 Issues

We discuss some of the issues we will encounter when trying to add support to the Rumprun unikernel for running on seL4 and also when performing an evaluation.

Different semantics between primitives: There is a difference between the semantics of the low level

system resources that seL4 provides, and the resources that the rump kernel host interface expects. The main difference is the execution model. Rump kernel threads are non preemptive while seL4 threads are. A way of providing seL4 threads in a way that the rump kernel can effectively use them is needed.

Interrupt disabling: The Rumprun unikernel expects to be able to disable interrupts and reenable them to provide mutual exclusion for some data. We need to be able to replicate this behavior using seL4 mechanisms.

Thread local storage: The Rumprun unikernel and applications that run on it rely on thread local storage. On x86 thread local storage uses a special segment register that seL4 uses for providing IPC mechanisms. We will need to find a way for seL4 to provide TLS as well as still effectively handle IPC.

Lack of documentation for interfaces: Some interfaces that the rump kernel provides for the Rumprun unikernel to implement are undocumented. The main one is a PCI driver interface that the rump kernel PCI bus module uses to configure PCI devices and map in PCI device memory. Additionally there is an undocumented API for handling DMA. We will need to ensure that our port successfully implements these APIs otherwise undesired device behavior could occur.

Lack of instrumentation in rump kernels: There is no existing mechanisms for measuring CPU utilisation in either the unikernel or rump kernels. A mechanism will need to be added in order to effectively compare performance to other systems.

Build system integration: Both seL4 and the rump kernel projects have very complex build systems that are responsible for producing a final binary image. In order to end up with Rumprun application binaries that can run on seL4 we will need to integrate these build systems.

We will present our low level design that attempts to solve these issues in the next section.

6.5 Design

Our high level design is to support the Rumprun unikernel as a single process on top of seL4. The seL4 root task will be responsible for initialising the system, loading the unikernel image, providing its initial resources and executing it. We will add seL4 as a new supported platform in the unikernel project so that when it is executed, it can interact with seL4's APIs and use existing seL4 user-mode libraries to initialise the rump kernel and start the loaded user applications. It will therefore be possible to take a POSIX application, compile it to run on the Rumprun unikernel, build the unikernel with the seL4 configuration. The resultant image can be loaded into a seL4 binary and the POSIX application can run on seL4. Eventually our design can be extended to run multiple Rumprun unikernels on top of seL4 in different applications all started by the initial process, however this is future work and is out of scope for this thesis. The implementation section of this thesis presents this design at a more detailed level.

6.6 Experimental setup

In order to evaluate the performance of rump kernel drivers we need to conduct experiments. Our benchmarks will be designed in order to find out the following:

- How the performance of rump kernel drivers on SeL4 compares with different systems.
- What overhead is introduced by SeL4.
- An understanding of how the drivers perform in general.

We will need to design our experiments to enable fair comparison across different systems. Supporting the Rumprun unikernel on seL4 will make it easier to run comparable benchmarking programs, and additionally, our results will be directly comparable with the unikernel running on bare metal. This should allow us to directly measure the overhead introduced by seL4.

It will be necessary to measure the CPU utilisation of the systems we run the benchmarks on. These measurements will be calculated from measuring how much cpu time is spent in the idle thread and subtracting it from the total CPU time spent on the benchmark. The remainder of the cpu time must have been spent on our benchmarks. This assumption requires that the system was not spending time doing any other work while the benchmark was running.

We will need to instrument some of the systems in order to measure thread utilisation information. We need to ensure that this instrumentation accurately measures the CPU idle time and also does not distort the actual utilisation measurements we are trying to take.

Similar instrumentation will need to be done to measure seL4 kernel entries. This is necessary to measure what the kernel spent time on in order to understand the source of overheads that will be introduced. There is existing support in seL4 for this, however we will need to quantify the overhead introduced by the additional instrumentation.

6.7 Summary

In this section we came up with our design requirements for adding support to the Rumprun unikernel for running on seL4. We presented some issues that our design will need to solve and presented a generic design. We also introduce some requirements of our experimental setup in order to accurately perform a performance evaluation of our constructed system.

7 Implementation

In this section we introduce our implementation to add support to the Rumprun unikernel for running on seL4. We focus our initial implementation on the x86 32-bit architecture. Expanding our support over multiple architectures is future work. We break our design up into three parts: Changes made to the seL4 kernel, our root task, and adding seL4 as a platform to the Unikernel.

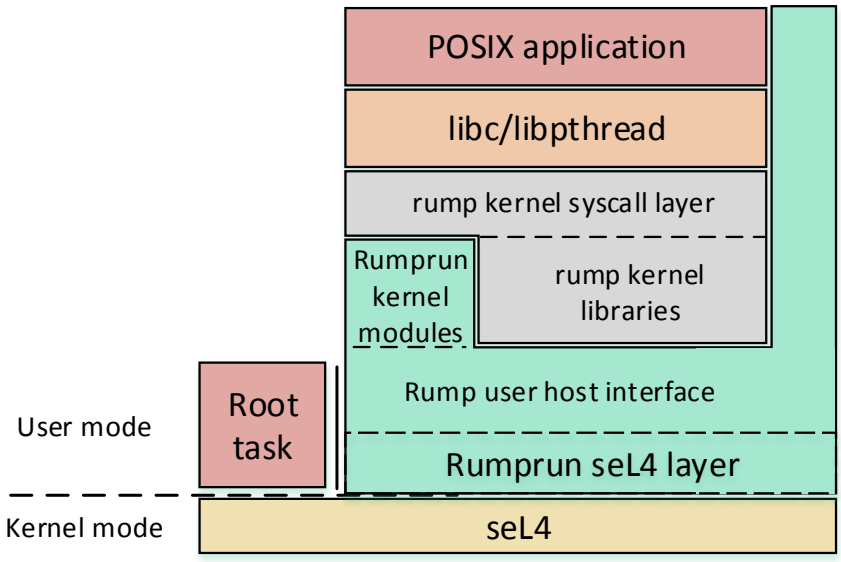


Figure 11: System architecture design

Figure 11 shows the block diagram of our design. To add support to Rumprun to run on seL4 we plan to run it as a single process in user-mode on seL4. To achieve this we need to:

- Implement the Rumprun seL4 layer which adds seL4 as compatible platform that Rumprun can run on. This layer manages seL4 objects and resources that are provided with the unikernel is initialised. We need to implement our design in a way that minimises overhead so that our performance is still comparable with the unikernel running on bare metal. To achieve this we need to make sure that we manage seL4's resource effectively and we minimise entries into the seL4 kernel as this significantly contributes to our overhead.
- Implement a seL4 root task that can create the unikernel's process, load its binary image and configure and provide its initial resources.
- Make any required modifications to the seL4 kernel in order to support features that Rumprun relies on efficiently.
- Additional modifications may potentially be required to Rumprun and the rumpkernel in order to better function on seL4.

The following subsections describes our intended design in more detail.

7.1 Adding seL4 support to Rumprun

Adding seL4 support to the Rumprun unikernel requires implementing the unimplemented sections of the rump kernel host interface, as Rumprun provides many of the implementations for us, implementing support for the additional interfaces for PCI and DMA functionality, as well as performing initialisation of the Rumprun system using seL4 resources and objects.

7.1.1 Host interface

The Rumprun unikernel implements some functionality of the rump kernel host interface. The remainder of the functionality we implement:

Threads: As previously mentioned, seL4's threading model is priority based preemption. If a lower priority thread is running and a higher priority thread becomes runnable, seL4 will preempt the currently running thread and schedule the higher priority thread.

Rump kernel threads are non preemptive. Rump kernels have a single runnable thread queue and when rump threads become available to run they are added to this queue. Threads can only be run when the currently running thread yields the processor.

A consequence of this non-preemption is that when a thread needs to run, it has to wait for a currently running thread. The rump kernel provides a way around this problem by having an assumption that there are enough CPUs available for the current set of runnable threads. When a thread becomes runnable, there will be an unused CPU for it to run on. The Rumprun unikernel configures the rump kernel with only a single CPU however. There is an assumption that the typical workload that the unikernel receives does not involve contention of the single virtual CPU due to only running a single application: when interrupts arrive, the application is already waiting for the interrupts to occur and as such there are no threads running allowing the interrupt to be processed. The unikernel uses interrupt handlers to handle interrupts immediately, however these handlers are used to mark a rump kernel interrupt handler thread as runnable, and save the interrupt number. The rump kernel then schedules this thread and the thread processes the interrupt.

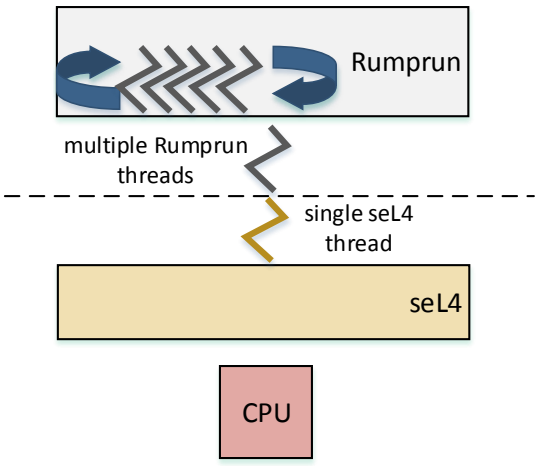


Figure 12: Thread model

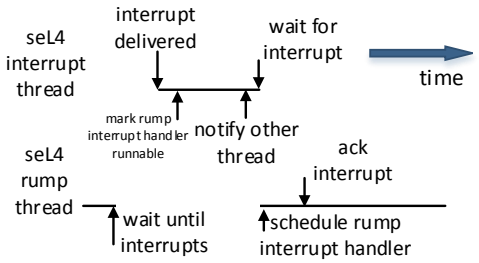
In our design, we map seL4 threads to rump kernel CPUs. This means that there will be a single seL4 thread that runs all rump kernel threads. This is a green-threading model and is shown in Figure 12. Every rump kernel thread runs in the same address space. Thus when a thread is switched, only certain hardware registers need to be updated. This means that we can have fast thread switching that does not need to significantly involve the seL4 kernel. There is an additional requirement that a thread local storage base address needs to be updated but we discuss this in the next subsection. We also have two additional seL4 threads to handle timer and device interrupts.

Synchronisation: We make use of standard notification objects and seL4's user level synchronisation libraries to provide synchronisation between the different seL4 threads. One synchronisation requirement is replicating masking of interrupts. The unikernel has the ability to disable interrupts. We replicate this in our system by using a binary semaphore. The binary semaphore is constructed out of a notification object and a globally shared memory variable. When an interrupt is delivered the interrupt thread reads the global variable and if the semaphore is free, marks it to held and continues executing. When the interrupt thread is finished, it returns the semaphore to the free state by changing the global variable. However if the other seL4 thread needs to disable interrupts, it marks the semaphore held. When this happens, if an interrupt is received, it will notice that the semaphore is held, and wait for it to become free by blocking on the seL4 notification object. When the rump kernel seL4 thread is ready to enable interrupts, it will mark the semaphore as free and then message the notification object to wake up the seL4 interrupt thread that will then handle the interrupt.

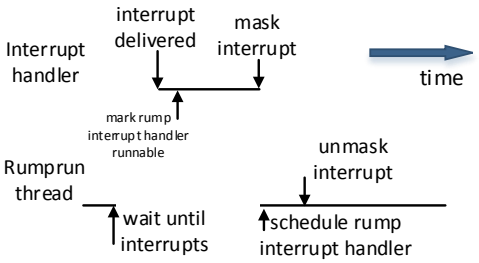
The other mechanism is simply a single notification object that is used to wake up the main seL4 rump kernel thread if it is blocked waiting for a timer or hardware interrupt. When the interrupt is received this notification object is messaged which causes the main thread to resume running after the interrupt thread finishes.

Interrupt delivery: Hardware interrupts can be triggered by timer events or other devices. When an interrupt arrives we need to be able to process it in the same way that the unikernel would without seL4. Figure 13 shows the typical ways that an interrupt is delivered on the Rumprun unikernel. There are 3 different modes. Mode 1 is when the system is waiting for an interrupt and not doing anything else. When this happens, the interrupt is triggered, the interrupt handler runs, and then the unikernel immediately schedules the rump kernel interrupt handler thread and processes the interrupt. Afterwards, interrupt is unmasked. The second mode is when an interrupt is triggered while the unikernel is already performing work. In this case the interrupt handler will run, mark the rump kernel interrupt thread as runnable, and when the currently executing work finishes, the interrupt thread will be scheduled, the interrupt processed, and finally unmasked. The final mode is when interrupts are masked in order to provide some mutual exclusion. When this occurs no interrupts will be delivered until interrupts are enabled, in which case any pending interrupts will be handled and then mark the rump kernel interrupt thread as runnable.

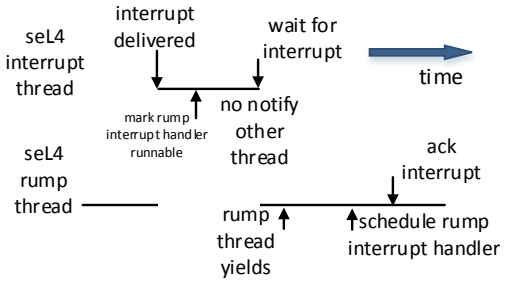
To replicate this behavior using seL4, we use a secondary seL4 thread with a higher priority than the seL4 thread running the rest of the unikernel. In mode 1, when seL4 receives the interrupt, it will send it to the notification object making our seL4 interrupt thread runnable. seL4 will immediately schedule the thread and it will start executing. Our thread marks the rump kernel interrupt thread as runnable and then signals the seL4 semaphore which will wake up the other seL4 thread that is blocked waiting for interrupts. In mode 2, the same happens, only the semaphore won't message the other seL4 thread as it is currently executing. In mode 3, we use the seL4 binary semaphore to replicate masking of interrupts. When the interrupt is received, the seL4 interrupt thread will block



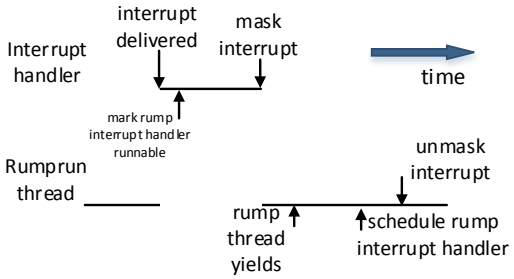
(a) Mode 1: seL4



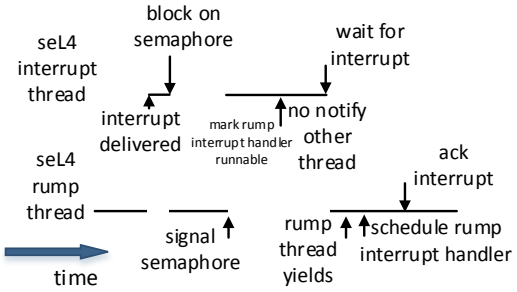
(b) Mode 1: Rumprun



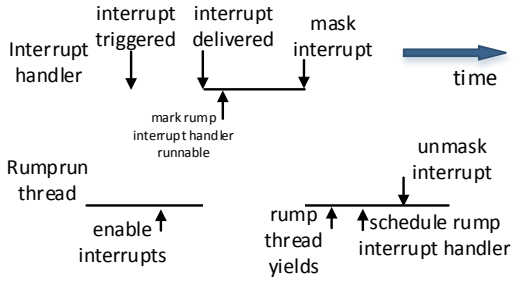
(c) Mode 2: seL4



(d) Mode 2: Rumprun



(e) Mode 3: seL4



(f) Mode 3: Rumprun

Figure 13: Interrupt delivery

until interrupts are unmasked and then resume executing.

We therefore replicate the same behavior as a native Rumprun unikernel using seL4 threads and synchronisation primitives.

Timestamps and timers: There are two time components required by the unikernel. The first is a monotonically increasing time counter which returns the number of nanoseconds since boot. The second time component is a programmable interrupt timer which causes a hardware interrupt after an amount of time. For the time stamp counter we use the built-in x86 RDTSC instructions as the x86 architecture provides a cycle counter that is updated on every instruction. By using the CPU clock speed we can convert this to number of nanoseconds since boot. There were better methods to achieve this that used different clock hardware, however the base version of Rumprun uses this method, therefore we use it too in order to remain comparable. For the programmable interrupt timer, IOPort instructions are required to program the device. Therefore we need seL4 capabilities to these port instructions as well as a seL4 interrupt handler object for the timer interrupts. We configure our root task to provide these to us upon initialisation. We use a secondary seL4 thread to receive the timer interrupts. When the timer is used, the typical behavior is that the seL4 thread

running the rump kernel has no work and it is set to idle until the next thread becomes runnable. We program the timer and then wait on a seL4 synchronisation semaphore until the timer is received. This will eventually wake the seL4 thread up when either the timer interrupt or another interrupt is received using our interrupt delivery design.

Memory: The Rumprun unikernel has a memory submodule that is initialised with a start address and end address. We use the seL4 user-mode memory allocator libraries to map in a contiguous range of memory from the initial resources that we pass through from the root task. This memory range is then passed on to the unikernel. The unikernel manages the rest of the memory.

7.1.2 DMA and PCI interfaces

See Section B.3 for the full interface. The interface can be split up into two categories: providing DMA functionality, and configuring PCI devices.

For PCI functionality:

port_out/port_in: These were functions that we had to add to the interface. The rump_pci driver was using the x86 IOPort operations directly. We replaced those operations with calls to these functions. These functions call the seL4 invocations for performing the actual x86 IOPort operations in kernel mode.

confread/confwrite: We used the same implementation as the original, with the IOPort operations substituted for our port_out and port_in functions.

irq_map: Registers a PCI device IRQ number.

irq_establish: Provides an interrupt handler to the previously registered IRQ number. This handler needs to be called when an interrupt is received. We create the seL4 interrupt object and configure it to set seL4 to deliver received interrupts to our seL4 interrupt thread. The interrupt handler is added to a list of interrupt handlers that the rump kernel interrupt thread processes once the seL4 interrupt thread marks it as runnable and it is scheduled to run.

map: Map PCI device memory into the address space. This is for memory mapped IO. We use the device frames that the root task gave us capabilities for and use the seL4 user-mode libraries for handling mapping them into our virtual address space.

unmap: Un-map PCI device memory from the address space. We added this function to the interface as the current unikernel did not support it. It is currently used when some PCI devices are mapped in in order to be configured, but are then unmapped until they are used, such as SATA disks.

For DMA functionaility:

dmalloc: Memory allocation of memory for sharing with PCI devices. It takes in a size and alignment, and returns a virtual address and the equivalent physical address of the underlying memory. To implement this we use a DMA allocator provided by the seL4 user-mode libraries. We save the virtual and physical addresses in a local data structure.

dmamem_map: This is a secondary function for mapping in the DMA memory into the virtual address. We already did this when dmalloc was called, as such we don't need to do anything for this function, other than return the virtual address for that mapping.

dmafree: Frees the DMA region that was previously allocated.

rumpcomp_pci_virt_to_mach: This function is used to translate a virtual address to the underlying physical address. The original Rumprun unikernel has a direct mapping where each physical address is the virtual address. As our design has different mappings, we need to provide an implementation of the translation. Reverse engineering the behavior of function, it appears that it is called for both DMA addresses and PCI mappings. Because of this we need to check for a mapping in two places. We initially search in our collection of existing DMA allocations. If we find the address in here, then we return the physical address. If not, then we search the device mappings from the map function we discussed above. We use the seL4 user-mode libraries to perform the lookup.

7.2 Modifying the Kernel

One issue mentioned in the previous section was the lack of required support for thread local storage. Thread local storage is a mechanism that allows thread-local variables in an address space that is otherwise shared amongst several threads. The main reason why this is used in the Rumprun unikernel is for a rump thread to figure out what its thread ID is and access a thread control block. TLS is provided by the compiler and the OS that the application runs on. TLS works by annotating a variable with '__thread' in the source code. This tells the compiler that the variable is to be stored in the TLS region in the address space. The compiler gives these variables a special section in the binary file. The OS that loads this binary is then required to instantiate a new copy of the variables in memory for each thread that is started. When a thread wants to access these variables, it looks at a global location for its TLS region base address. It then performs a constant offset from this address to get the address for the particular variable. When a thread is switched by the system, the global TLS base address is updated to reflect the memory location of the next thread's TLS region.

On x86 this global base address is stored in the global descriptor table, a table of base addresses referred to by the x86 segment registers (CS, DS, SS, ES, FS, GS), and is referred to by the GS segment register. There are two issues here with regards to seL4. Firstly, updating entries in the global descriptor table requires the processor to be in kernel mode and seL4 currently does not provide a way to update these entries. Secondly, seL4 already uses the GS segment register as the base address for a thread's IPC buffer. In order to support TLS on seL4 we need to provide a way for a user application to update the global descriptor table with a new base address, and also find a way to use the GS segment register without breaking IPC on seL4.

We achieve this by switching the segment register that seL4 uses for the IPC buffer to the FS segment register. This register is not currently used and can be used in the same way as the GS segment register can. Switching this register requires modifying seL4 to use this different register. We also add an invocation to allow a process to update the base address for its TLS region. By invoking the thread capability a process can get the seL4 kernel to update the new base address in the global descriptor table. Now whenever a thread is changed in user-mode, the invocation can be used to update the TLS base address. This should enable TLS functionality for the unikernel and rump kernels.

Additional changes were made to the seL4 kernel for the purposes of performance evaluation and they will be discussed in the evaluation section.

7.3 The Root task

The root task is the first user-mode application started by the seL4 kernel once it has initialised. seL4 will load its binary file into memory and configure it with the initial resources required to run. seL4 also provides this application with the capabilities to the remainder of system resources. These capabilities represent the authority to control these system resources. The root task is then usually responsible for initialising the rest of the system.

Our root task implementation follows a common seL4 design pattern where the root task acts as an "application driver" for loading multiple other processes. The root task is responsible for the following:

- Creates a new process and loads the Rumprun unikernel binary
- Adds a fault handler for handling any user or virtual memory faults that the unikernel causes
- Copies/moves the relevant capabilities to system resources onto the unikernel process
- Initialises any instrumentation of the seL4 kernel for use in performance evaluation

We reuse this design rather than loading the unikernel directly because:

- The root task is special. It has the most rights that we don't necessarily want to give to the unikernel immediately. By using the root task to load the unikernel task, we can more finely control what resources that the unikernel can use.
- Allows us to extend the root task to handle multiple unikernels without having to modify the unikernel. This would allow us to run multiple driver-like components that are fault isolated from each other in the future.
- If the unikernel crashes we can restart it from the root task. In some cases recovery may be possible, but if not, then the root task can simply reload the unikernel process and restart it. The alternative is the entire system crashing and having to be restarted.
- Easier to add intermediate processes such as a debugging process that boots the unikernel process in a different environment.
- No performance costs associated with having an extra process. When the Rumprun unikernel is running, the root task threads are blocked and unused.

7.4 Summary

In this section we presented our design for adding seL4 support to the Rumprun unikernel. Our design is divided into three parts: changes made to the seL4 kernel, our root task, and adding seL4 as a platform to the unikernel. We presented solutions to the challenges identified in the previous section such as effectively handling the different execution models, delivering interrupts, adding thread local storage support and how we plan to implement the DMA and PCI interfaces which do not have a large amount of documentation. In our next section we describe our experimental setup and how we add instrumentation to our system as well as present the results of our evaluation.

8 Evaluation

Our primary goal is to evaluate rump kernels as an approach for driver-like operating system component reuse on the seL4 microkernel. We have added support to the Rumprun unikernel, a project that uses rump kernels, to run in user-mode on seL4. We want the following:

- Low effort to reuse components
- Low runtime overhead
- Similar performance to components performing natively

To evaluate how rump kernels achieve our goals, we measure the following:

Rump kernel module sizes: We measure both the number of lines of source code and final binary size of rump kernel modules required to perform different tasks.

Networking stack driver performance: We measure the performance of the NetBSD networking stack running in Rumprun on seL4 and compare this to Rumprun running natively, NetBSD running natively and an equivalent Linux networking stack running on Linux.

Effort required to reuse components: We measure the amount of effort that was required to use rump kernels on seL4 with a quantitative method of number of lines of source code additions and modifications required as well as a qualitative review of our experiences.

We initially describe our experimental setup and then provide results for our measurements in the following subsections. Discussion is presented alongside the results and we summarise our overall findings at the end of this section.

8.1 Experimental setup

This section provides implementation details of the experimental setup. It includes information about the hardware used, operating systems used and how they were configured, details about the benchmark tools used and what methods were used to collect results.

8.1.1 Hardware

During initial development of our system the hardware emulator QEMU was used to test our implementation. The benefit of using QEMU was that it provided a way to attach a debugger to the running system that greatly aided in debugging. Additionally, it provided an assurance that any bad driver code could not damage any actual hardware.

For conducting experiments, actual hardware was used. To ensure comparability the same physical machine was used to run benchmarks on all systems. The machines details are provided in Table 1.

As seL4 and the Rumprun unikernel do not currently offer full support for multiple cores, all but one core was disabled in the BIOS settings. Additionally, hyper-threading support was also disabled.

This machine was connected to the host machine using a direct ethernet connection between two gigabit ethernet cards, and a serial connection. The details of the host machine are included in Table 2.

Detail	Value
num processors	4
vendor id	GenuineIntel
model name	Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz
CPU MHz	1600.011
cache size	6144 KB
ethernet cart	Intel Corporation 82579LM Gigabit Network Connection (rev 04)
vendor id	8086
device id	1502

Table 1: Hardware details of test machine.

Detail	Value
num processors	4
vendor id	GenuineIntel
model name	Intel(R) Core(TM) i7 CPU 870 @ 2.93GHz
CPU MHz	2934.000
cache size	8192 KB
ethernet card	Intel Corporation 82574L Gigabit Network Connection
vendor id	8086
device id	10d3

Table 2: Hardware details of host machine.

The host machine was used to update and install software binaries and to act as a client during benchmarks.

8.1.2 Systems tested

Throughout our experiments we run benchmarks across four different operating systems/software stacks. These systems and their general configurations are described here. Any specific benchmark changes will be mentioned in the relevant section.

Rumprun on seL4: Our system that we have built. It consists of the Rumprun unikernel on mainline seL4. No serious modification were made other than the additional features added, instrumentation for measuring CPU utilisation and kernel entry points. It has been built in release mode where kernel and user debugging features have been disabled. Printing has been disabled.

Rumprun on BMK (Bare-metal kernel): A standard instance of the Rumprun unikernel built for bare-metal x86 32-bit architecture. The rump kernel sources were built using the same configuration as for the seL4 version. Some instrumentation changes were added in order to measure CPU utilisation, however as we will argue in the next section, this should have no effect on the total utilisation. For the purpose of benchmarking printing has also been disabled.

Native NetBSD: A NetBSD kernel built from the same sources as the unikernel. It was built for release. When running benchmarks we reverted back to single user mode in order to reduce background processes and attributed CPU utilisation. It was also configured to boot with one CPU.

Linux: A Linux kernel running an up to date Debian distribution. Configured to boot in single user mode in order to reduce amount of background processes. Configured to use only 1 CPU to reduce synchronisation overhead.

8.1.3 Benchmarks

Initially our goals were to run multiple benchmarks to test network performance, disk I/O performance and then a benchmark that tested both networking and disk I/O at the same time. However, due to time constraints it was decided to instead pick one benchmark to focus on to be able to go into a reasonable amount of depth in collecting and analysing the results.

Iperf3 is a networking benchmark, designed for measuring network statistics such as bandwidth and packet loss percentages. We use it to measure networking performance of the rump kernel networking stack. An Iperf3 server is started on the target system, and a client is used on our host machine to run the benchmark. When the client is started, it opens a TCP control connection to the server, specifies what test is being run and then either the server or the client starts the test. We use this benchmark to measure throughput under applied load for each of our systems as well as CPU utilisation. We also run a test to compare the different UDP packet loss rates for each of the systems.

8.1.4 Measuring CPU Utilisation

We measure CPU utilisation because all four systems reach maximum networking card throughput before the maximum CPU utilisation. In order to compare performance we compare CPU utilisation at the same network throughput. In all four systems, CPU utilisation was calculated by measuring idle system time and subtracting it from the total. For NetBSD and Linux this was achieved by reading the /proc/uptime file which contains total system time and total idle time. By reading these values at the start of our measurement period and then reading them at the end we can calculate system time and idle time differences over the period. This gives us idle time to a 0.01s which is the resolution of the timestamps in the file.

For seL4 and BMK, idle-time was calculated by modifying the idle loops of each system to sample the in-built timestamp counter. Each sample is compared to the previous sample. If the difference is small enough, we can assume that the processor has not handled any interrupts or that another thread was scheduled. By sampling this counter in the idle loop, the total idle time is able to be summed up. The benefit of this approach is that the instrumentation only runs when the processor would otherwise be idle meaning that our instrumentation adds no overhead. This approach has been used before in other seL4 projects.

8.1.5 Additional instrumentation

We take advantage of additional instrumentation available in seL4 to track kernel entries and exits. This allows us to measure how much CPU time was spent inside the kernel and also track what code paths were used. As this is an existing feature, a configuration setting is needed to enable it. Each sample contains a cycle count of when the kernel was entered, the number of cycles spent in the kernel, and also what the kernel was doing, this can either be processing a syscall, interrupt, user level fault, or virtual memory fault. A syscall fault is split up into the syscall number and if the entry is due to a seL4_call syscall to invoke a kernel object there is additional information specifying what kernel object (capability) was called, and what invocation number on the kernel object.

A memory buffer must be added to store the entry and exit information. This buffer is limited to 1MiB in

size, which can hold about 55000 entries. This quickly fills up which means that samples can only be taken for periods less than 3 seconds. When using this method during a benchmark, start sampling after the warmup period and sample for 3 seconds.

8.1.6 Test harnesses

In order to automate the benchmarking process, we use a test harness to handle uploading the correct binary images, restarting and configuring the different systems, starting and stopping the benchmarks and collecting the relevant measurements.

All systems have the initial console configured to read and write to the serial connection between the machines. This allows the host machine to read and write from the shell for both Linux and NetBSD when started in single user mode. For seL4 and BMK as there is no shell; the desired benchmark program is compiled into the initial binary that is loaded. It is then executed automatically when the system initialises.

If possible, the benchmark is configured by running a client program on the host machine which handles connection and management. One benchmark program, Iperf3 achieves this by opening a TCP socket to a server running on the system to configure, start and stop the benchmark and to also collect results.

Even though Iperf3 has capability to calculate CPU utilisation statistics, the mechanisms that it relies on are not present on the seL4 or BMK systems. In order to collect our measurements, we add a serial interrupt handler to each system that parses serial input commands. These can start, reset or stop CPU utilisation counters and dump any other required system information to the serial console which the host machine can then parse. Sometimes, the serial drops characters which the host can then detect that the response is malformed. When this happens the test is rerun automatically in case the delay due to resampling affects the results. For the Linux and NetBSD systems, a UDP server is started that responds to any messages with the contents of /proc/uptime. The host machine then messages this server each time it wants to take a CPU utilisation measurement.

We now present our results.

8.2 Rumpkernel library sizes

In this section we measure the binary sizes and source lines of code of rump modules. There are three reasons for measuring these:

- To understand the amount of code that we are able to reuse unmodified from NetBSD
- The binary sizes are a weak indicator of runtime size overhead, we can also compare them to a Linux binary which is our other approach for reusing os components
- To measure the amount of extra code used to support rump components that beyond the components themselves.

We initially present a summary of all rump modules, followed up by the rump modules required for running different software stacks: networking, USB, SATA disk drivers and a Virtio driver stack. A full list of rump kernel modules with the number of files, source lines of code and binary sizes can be found

in Appendix C. Additionally, the tables contain the module names, module desciptions can be found in Appendix A

Library	**Num files**	**SLOC**	**Filesize**
net	207	108154	1.8MiB
kern	265	75244	1.6MiB
dev	247	135291	2.5MiB
vfs	238	135191	2.7MiB
Total	**957**	**453880**	**8.7MiB**

Table 3: Summary of total NetBSD code provided by rump kernels.

Table 3 shows a summary of all rump kernel modules that can be run. The subtotals net, kern, dev and vfs are based on the category of OS component. The sizes of each individual module can be found in Appendix C. Although it is possible to use rump kernels that have all of these modules included, we can cut down the modules used on an as needed basis. This allows us to customise the size of the rump kernel based on what is required.

8.2.1 Networking stack

Table 4 shows the rump kernel modules required to run a NetBSD networking stack. The required categories of modules are:

- the core rump kernel module
- temp filesystem required for the Rumprun unikernel
- network card modules that provide device drivers for the PCI Intel e1000e gigabit network card
- network stack modules that provide functionality for TCP, IP, UDP network protocols and the unix socket interface
- other Rumprun miscellaneous modules

Groupings	**Library**	**Num files**	**SLOC**	**Filesize**
Core	rump	198	53681	1.2MiB
Temp filesystem	rumpvfs	49	24884	485.8KiB
	rumpfs_tmpfs	7	2705	61.1KiB
Network stack	rumpnet	21	8092	156.9KiB
	rumpnet_config	10	2452	54.1KiB
	rumpnet_netinet	1	49	2.7KiB
	rumpnet_net	71	56846	934.3KiB
	rumpnet_netinet6	1	22	1.8KiB
	rumpnet_local	3	1388	23.6KiB
Network card	rumpdev	5	3052	67.8KiB
	rumpdev_bpf	3	2033	32.6KiB
	rumpdev_pci	13	5596	158.4KiB
	rumpdev_pci_if_wm	2	8649	120.0KiB
	rumpdev_miiphy	5	857	197.3KiB
Rumprun misc	rumpkern_mman	2	195	5.1KiB
	rumpkern_bmktc	2	43	2.9KiB
Total		**393**	**170544**	**3.4MiB**

Table 4: Breakdown of rump kernel code required to use the NetBSD networking stack.

8.2.2 USB Stack

Table 5 shows the rump kernel modules required to run a NetBSD USB stack. The required categories of modules are:

- the core rump kernel module
- temp filesystem required for the Rumprun unikernel
- USB modules that provide the USB stack
- filesystem modules that provide functionality for mounting filesystems on top of provided block devices
- other Rumprun miscellaneous modules

Groupings	**Library**	**Num files**	**SLOC**	**Filesize**
Core	rump	198	53681	1.2MiB
USB	rumpdev_pci_usbhc	8	10237	131.9KiB
	rumpdev_usb	10	5310	133.6KiB
	rumpdev_umass	5	2578	34.0KiB
	rumpdev_scsipi	11	8622	159.2KiB
	rumpdev_pci	13	5596	158.4KiB
	rumpdev	5	3052	67.8KiB
Filesystems	rumpvfs	49	24884	485.8KiB
	rumpfs_ffs	24	15736	319.0KiB
	rumpfs_cd9660	7	2540	51.7KiB
	rumpfs_ext2fs	11	4627	97.9KiB
	rumpdev_disk	6	2852	57.5KiB
	rumpfs_tmpfs	7	2705	61.1KiB
Rumprun misc	rumpkern_bmktc	2	43	2.9KiB
	rumpkern_mman	2	195	5.1KiB
Total		**358**	**142658**	**2.9MiB**

Table 5: Breakdown of rump kernel code required to use the NetBSD USB stack.

8.2.3 Sata disk stack

Table 6 shows the rump kernel modules required to run a SATA disk drivers and filesystem support. The required categories of modules are:

- the core rump kernel module
- SATA modules that provide SATA disk functionality
- filesystem modules that provide functionality for mounting filesystems on top of provided block devices
- other Rumprun miscellaneous modules

8.2.4 Virtio stack

Table 7 shows the rump kernel modules required to run a Virtio stack. Virtio is a standard for accessing virtual networking and block devices when the system is aware it is running in a virtual environment.

Groupings	Library	Num files	SLOC	Filesize
Core	rump	198	53681	1.2MiB
Sata	rumpdev	5	3052	67.8KiB
	rumpdev_pci_ahcisata	7	3421	68.9KiB
	rumpdev_ata	2	1628	30.2KiB
	rumpdev_pci	13	5596	158.4KiB
Filesystems	rumpvfs	49	24884	485.8KiB
	rumpfs_ffs	24	15736	319.0KiB
	rumpfs_cd9660	7	2540	51.7KiB
	rumpfs_ext2fs	11	4627	97.9KiB
	rumpdev_disk	6	2852	57.5KiB
	rumpfs_tmpfs	7	2705	61.1KiB
Rumprun misc	rumpkern_bmktc	2	43	2.9KiB
	rumpkern_mman	2	195	5.1KiB
Total		**333**	**120960**	**2.5MiB**

Table 6: Breakdown of rump kernel code required to use the NetBSD SATA stack.

The required categories of modules are:

- the core rump kernel module
- Virtio modules that provide device drivers and networking interface drivers
- filesystem support
- network stack modules that provide functionality for TCP, IP, UDP network protocols and the unix socket interface
- other Rumprun miscellaneous modules

Groupings	Library	Num files	SLOC	Filesize
Core	rump	198	53681	1.2MiB
Virtio	rumpdev	5	3052	67.8KiB
	rumpdev_virtio_if_vioif	2	1195	23.2KiB
	rumpdev_virtio_ld	3	939	22.4KiB
	rumpdev_virtio_viornd	2	196	7.4KiB
	rumpdev_pci_virtio	2	976	21.1KiB
	rumpdev_pci	13	5596	158.4KiB
Filesystems	rumpvfs	49	24884	485.8KiB
	rumpfs_ffs	24	15736	319.0KiB
	rumpfs_cd9660	7	2540	51.7KiB
	rumpfs_ext2fs	11	4627	97.9KiB
	rumpdev_disk	6	2852	57.5KiB
	rumpfs_tmpfs	7	2705	61.1KiB
Networking	rumpnet_config	10	2452	54.1KiB
	rumpnet	21	8092	156.9KiB
	rumpdev_bpf	3	2033	32.6KiB
	rumpnet_netinet	1	49	2.7KiB
	rumpnet_net	71	56846	934.3KiB
	rumpnet_netinet6	1	22	1.8KiB
	rumpnet_local	3	1388	23.6KiB
Rumprun misc	rumpkern_bmktc	2	43	2.9KiB
	rumpkern_mman	2	195	5.1KiB
Total		**443**	**190099**	**3.7MiB**

Table 7: Breakdown of rump kernel code required to use the NetBSD Virtio stack.

To summarise this section, rump kernels do provide a large amount of NetBSD driver-like OS components based on numbers of lines of source code. By using rump kernels we are able to reuse up to around 450 thousand lines of NetBSD code, some of which represents over 30 years of kernel development. Modularisation of the kernel modules allow us to require less lines when we want to use less drivers. Using the four example software stacks of networking, USB, SATA disk drivers with filesystems and a Virtio stack we only require between 23% and 42% of the total rump kernel lines of code. The amount of overhead required to use these modules, the base rump and vfs modules contribute a total of 75 thousand lines of code required each time which make up 16% points on each of those measurements. Compared to current alternative approaches for OS component reuse on seL4 the overhead required for running different component stacks is smaller.

8.3 Performance evaluations

Our goal is to evaluate the seL4 Rumprun port to get an understanding of the performance characteristics of drivers provided by rump kernels. We also want to identify what overhead is introduced by seL4 to the new system. To evaluate the performance of the seL4 Rumprun port, we perform some benchmarks and compare seL4 against three other systems: BMK Rumprun unikernel, native NetBSD and Linux. See the previous experimental setup section for a description of our experimental setup.

8.3.1 Networking benchmarks

We use Iperf3 which is a tool designed to test bandwidth and other networking statistics. The network topology was two machines, each with a 1Gb/s network card directly connected to each other. Measurements were taken in steps of 90Mb/s from 90Mb/s up to 900Mb/s. Each test was run for 20 seconds where the first 10 seconds were ignored to allow for the TCP sockets to reach a stable state. This initial warmup period was based on measurements taken that will be discussed. Each sample is run 7 times, and error bars indicate standard deviation. Where error bars are not shown it is because the standard deviation is too low. The socket size is set to 87380 (85KiB). For receiving and sending and the TCP congestion algorithm is set to cubic and a 128KiB buffer is used for each read and write request.

For our first test, we send packets from the host machine to the target machine. The amount of packets sent is controlled in order to send a controlled load. We can then measure the amount of data received at the target as the Iperf server running on it can report back what it received. We expect that the amount of packets received will be the same as the amount of packets sent. However if the amount of packets sent is at a rate higher than the target machine can process, then packets will be dropped, causing the TCP socket to retransmit them. This should result in a slower amount of packets reported received. If this occurs it should be reflected in a 100% CPU utilisation measurement of the target machine reflecting that it cannot process any additional packets.

Figure 14 shows the results of this initial test. We ran the same test on all four systems. The single line on our graph is because all four results appear on top of each other. It can be seen that the target machine can keep up with the source machine at all levels of applied load. This is not too unexpected as we are using 1 Gbps cards as both source and target. Thus the rate we can send saturates before the targets CPU utilisation becomes 100%. To be able to compare performance we need to measure CPU utilisation.

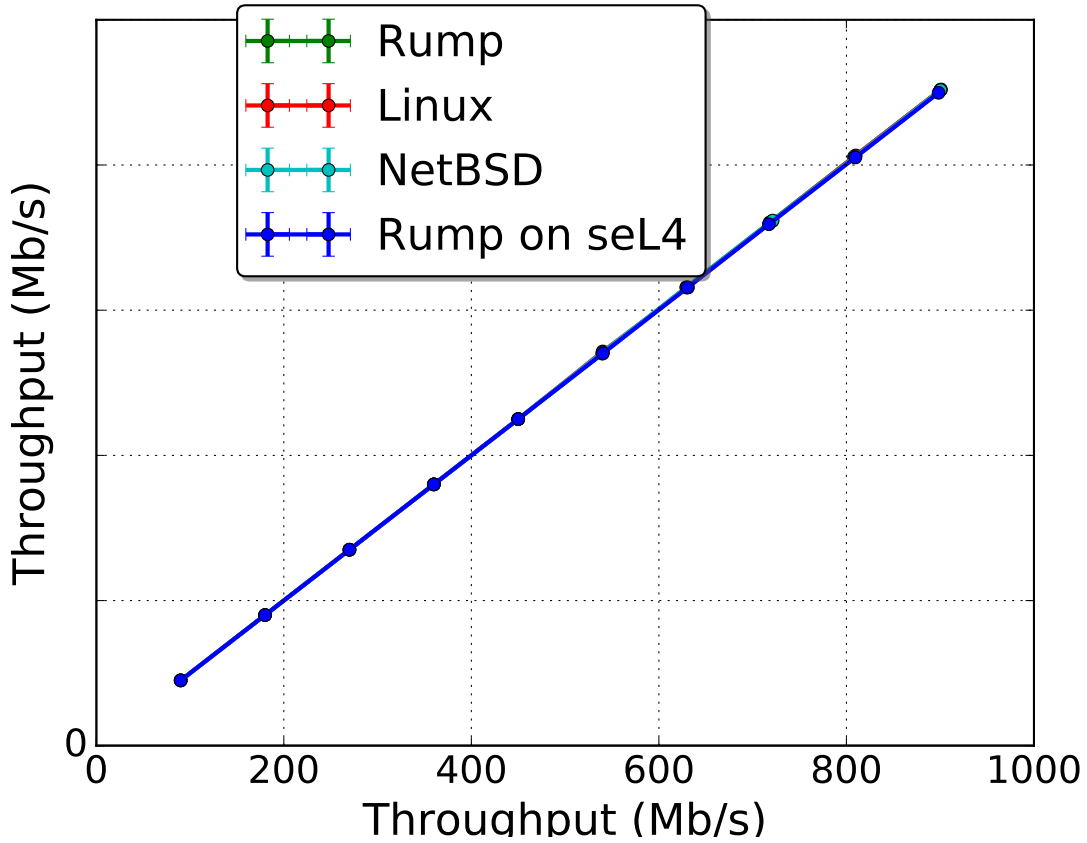


Figure 14: Throughput received over applied TCP load

We also plot the measured CPU utilisation of each system for each test in Figure 15. As can be seen, the CPU utilisation remains under 25% for all systems. Thus there is still remaining CPU available to process additional packets which is why we see no drop-off in throughput at different loads.

We can still compare these systems based on their CPU utilisation under different applied loads. All four systems have a similar utilisation at 90Mb/s of between 1% and 2.5%. As the applied load increases the utilisation increases at different rates. BMK having a lower utilisation than NetBSD is expected as BMK is essentially a stripped down version of the NetBSD kernel running no background tasks and with no preemption. They both run the same driver. It is also expected that seL4 will introduce some amount of overhead compared with BMK. This overhead is between 15% and 20%. However it is still faster than the same driver running in NetBSD. The Linux system is running on the same hardware, but uses a Linux e1000e driver rather than the NetBSD one. This could explain Linux's better performance. Other things that were considered include that Linux does dynamic interrupt throttling while NetBSD has a static throttle rate. This could explain the fifth Linux data point which has a high variance in samples due to the switching between different interrupt throttle rates. NetBSD's variance is hard to explain. The amount of variance grows as the throughput increases meaning that it is not caused by unknown background tasks. Low NetBSD variance is not important for us to effectively collect the measurements we require.

The 10 second warmup period is used as it takes that long for the system utilisation to reach a steady state. For the first 10 seconds, the TCP socket is establishing itself and there is a variable utilisation while this happens. To indicate this we run the same benchmark for one minute at each throughput level and measure the utilisation in one second intervals. The results of this is shown in Figure 16. As can be seen

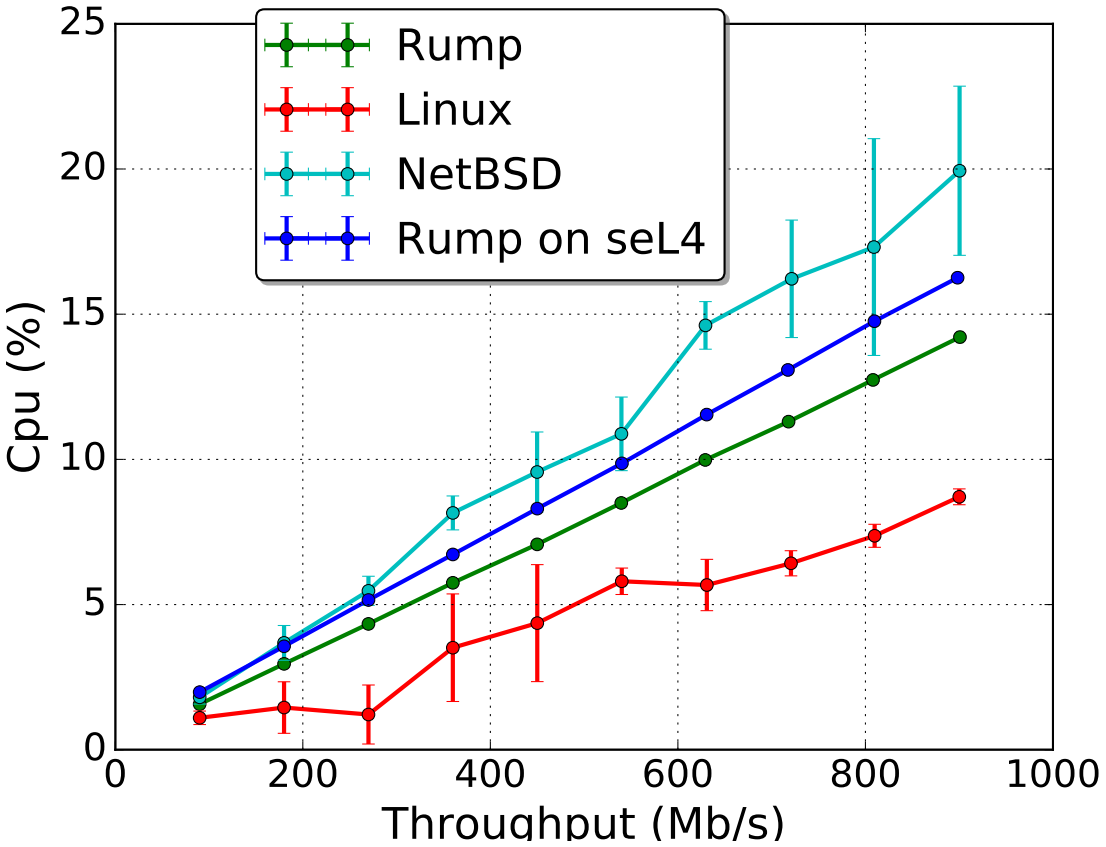


Figure 15: Thread utilisation under applied TCP load

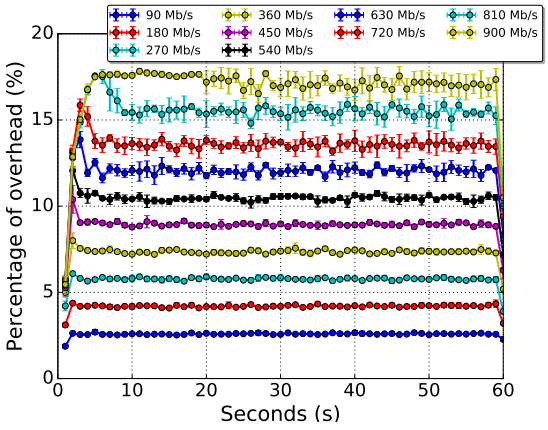
once 10 seconds has passed, the utilisation for the seL4 and BMK systems mostly stabilises. The excessive variance in utilisation for Linux and NetBSD is partly due to the accuracy of our utilisation measurements. Our utilisation measurements are accurate to 0.01s. When we measure in 1 second increments, this becomes a 1% point error. When our utilisation percentage is under 20% points this becomes a higher percentage overhead. Normally this is not an issue as we sample over a 10 second period. Our second source of variance is a higher amount of execution required to support running the Linux and NetBSD systems that is not present on the seL4 or BMK systems.

A quick summary:

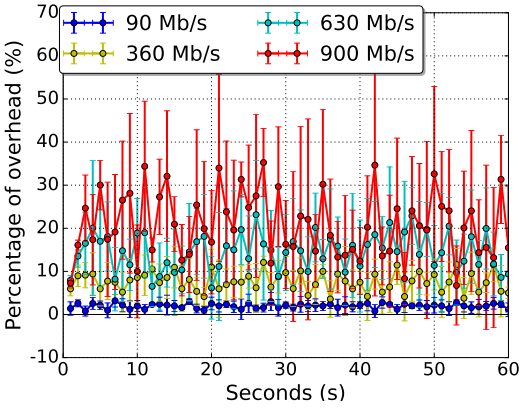
- We see that seL4 has some overhead compared to the BMK unikernel.
- Both seL4 and the BMK have less cpu utilisation than the same driver running on native NetBSD.
- A separate linux e1000e networking driver has the lowest cpu utilisation.

We next breakdown the source of seL4's overhead. In order to do this we turn on kernel entry tracking in order to track how much time was spent in the kernel and understand what that time is spent on. Turning on the kernel tracking provides a small additional overhead. We rerun our TCP benchmark for seL4 with the new configuration. Figure 17 shows that while some overhead is present, it is not a substantial amount to negatively affect the results.

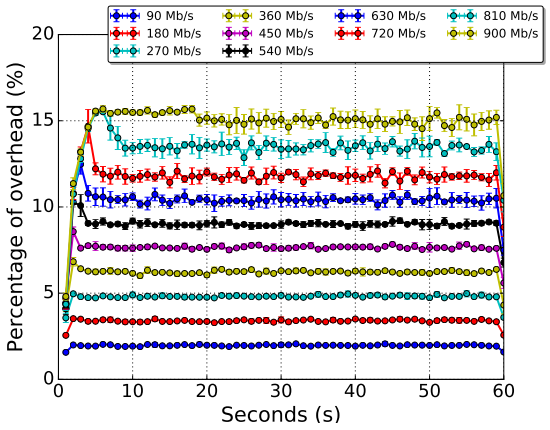
One issue we face however, is that the kernel log buffer to track entries fills up after roughly 3 seconds worth of sampling at the rate of entries we receive while running the benchmark. Our networking benchmark currently takes longer than this. Thus we sample the overhead for 3 sec in the middle of the benchmark



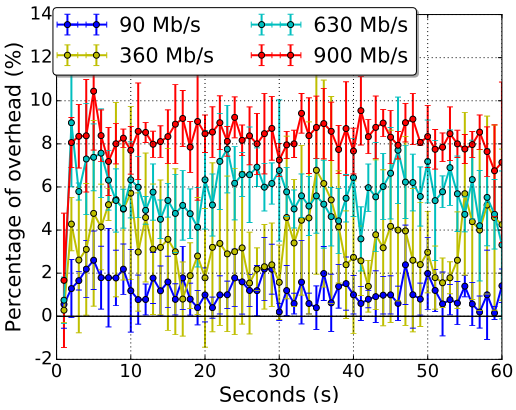
(a) seL4



(b) NetBSD



(c) BMK



(d) Linux

Figure 16: Utilisation measured over 60s in 1s intervals

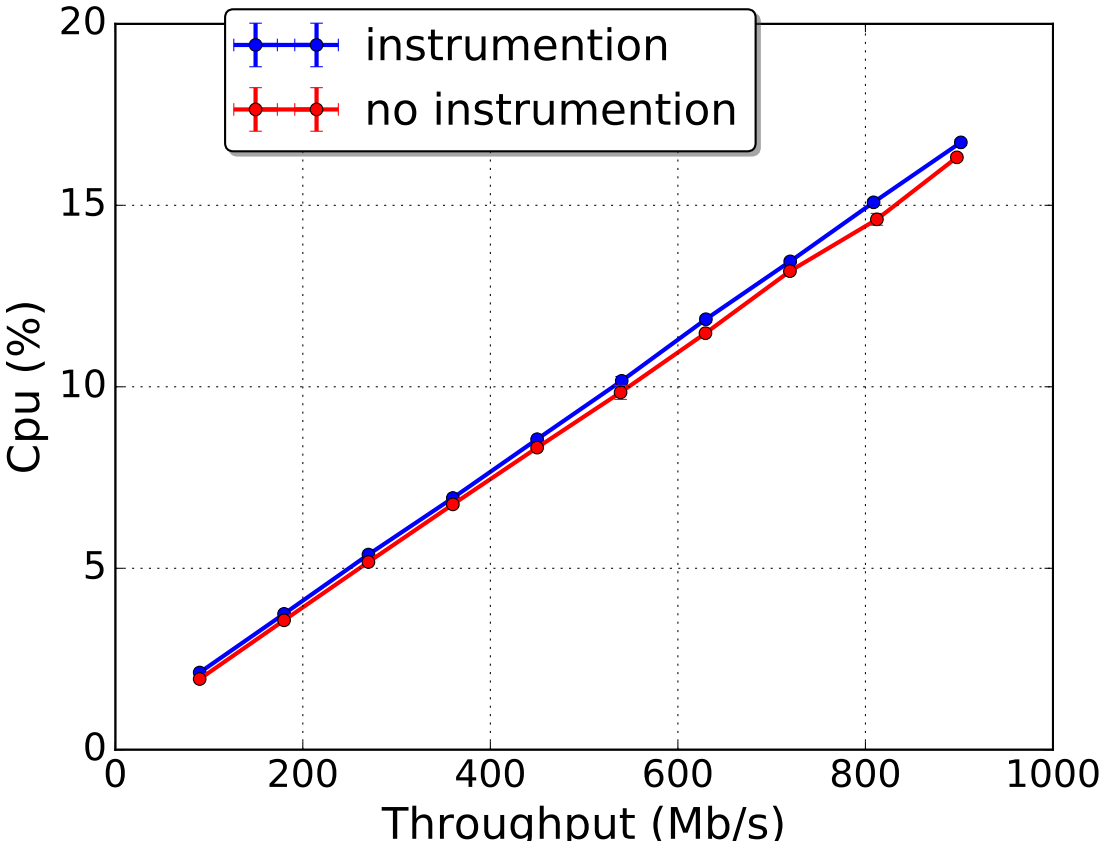


Figure 17: TCP Load benchmark with kernel instrumentation comparison

after the warmup period. We are confident that the overhead is stable at this point as we saw in Figure 16a. This should give us an accurate result of where kernel time is spent.

Figure 18 shows the results of the throughput benchmark running for only 3 seconds. Now we can break the seL4 utilisation results down into time spent in kernel-mode and time spent in user-mode. From the figure we can see that most of our overhead is attributed to time spent in kernel. The overhead attributed to user-mode time is less than kernel time and appears to increase at the same rate.

We ran a few experiments to try and explain the sources of user-mode overhead, such as roughly measuring the time spent in extra user-mode functions we added to our implementation however we were unable to find any significant contribution to our overhead. Another source of overhead is the mode switching time and interrupt time that occurs before we start measuring time in the kernel. However, initial experiments that we ran showed these to be less than 5% extra of the kernel overhead and so we did not invest more investigation time. It is possible that it is attributed to time spent managing our records of seL4 kernel objects as well as different compiler settings that the BMK was compiled with compared with seL4. Additionally, including the seL4 would increase our total amount of code which could reduce the effects of the instruction cache, however we would need to measure cache miss events to be sure of this assumption. A more comprehensive analysis is required to understand the source of user-mode overhead that we have introduced and is left for future work.

However, most of our overhead is attributed to time spent in the kernel. The overhead starts small, for lower throughputs, but then grows linearly until the end of the benchmark. It is possible to further break down this overhead into what the kernel was actually doing with this time. We do this in the next figure.

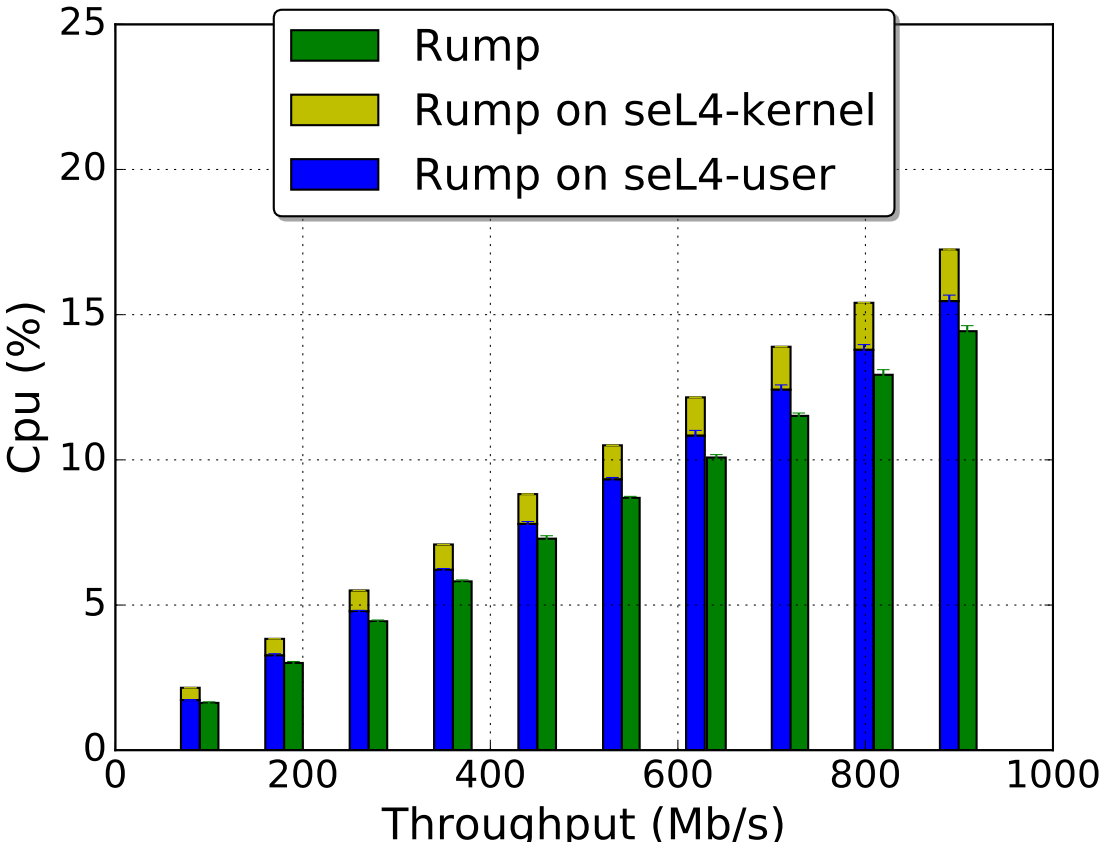


Figure 18: TCP Load benchmark split into sources of overhead

Figure 19 shows the kernel overhead broken down into what task the kernel was spending its time doing. The task that takes the longest time is handling IOPort operations. x86 has IOPort instructions that are used to read and write to devices. These are privileged instructions and thus the kernel has to be entered each time. In this case we use the IO ports to program an interrupt timer. Modern devices use memory mapped IO and the IOPort ops can be considered legacy instructions. It is possible to remove this source of overhead by using a different timer. However the BMK system uses the same programmable interrupt timer, so to keep our results comparable we didn't switch to the different timer.

The next source of overhead is attributed to time spent receiving interrupts and delivering them to the user process, and then handling the acknowledgement of the interrupt which is a separate syscall. The combination of these two tasks make up about 50% of our overhead. The remainder of the overhead is attributed to send and receive syscalls which are used to provide synchronisation primitives between our seL4 user-mode threads, and the invocation that we added to update the thread local storage pointer. As the throughput of the benchmark increases, there appears to be a slight increase in the amount of synchronisation related syscalls.

We finish up our benchmarking of the rump kernel networking stack with some UDP benchmarks. UDP doesn't provide the same delivery guarantees that TCP provides, and we can analyse the drivers based on the percentage of packets that they drop under different loads. We standardise our UDP socket buffer sizes to 87380 (85KiB) and we do not omit the first 10 seconds as we do not require time for TCP congestion control algorithms. In Figure 20 we see that utilisation measurements are roughly the same as for TCP,

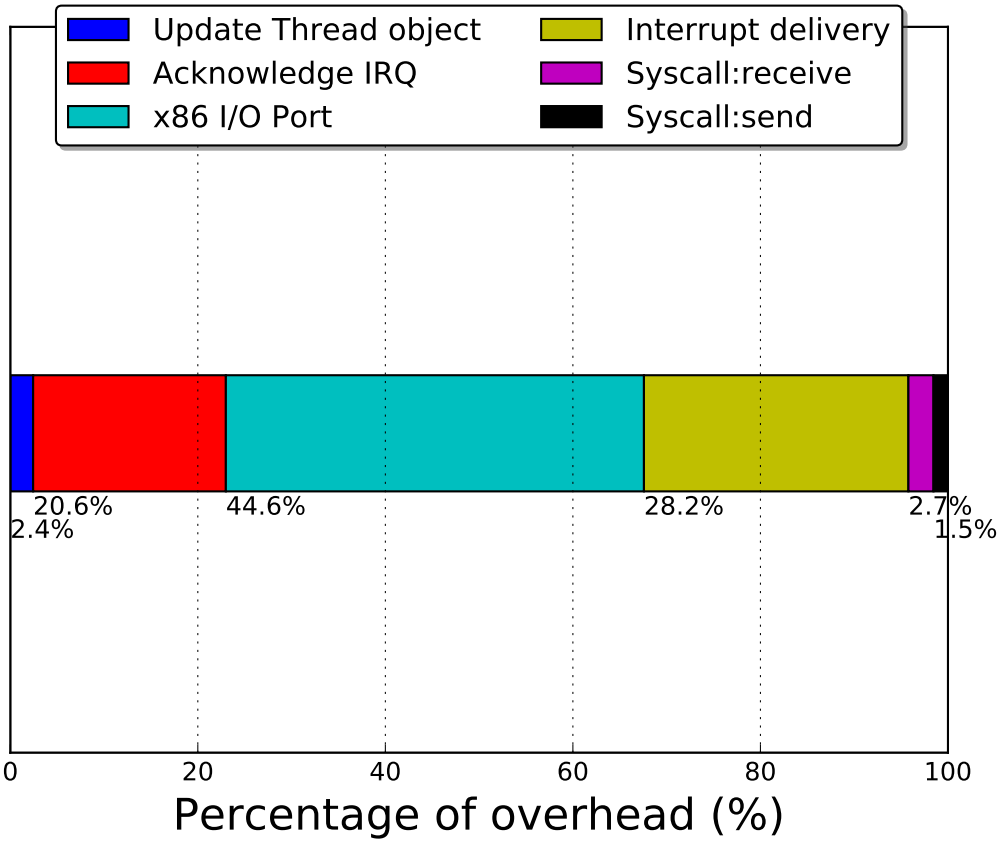


Figure 19: Kernel overhead from previous figure split into code paths

and in Figure 21 we see that our packet loss percentages are 0% for Linux and are around 20% for all systems that use NetBSD drivers. If we increase our socket buffer size we can reduce the amount of packet loss, but our Linux system performs better.

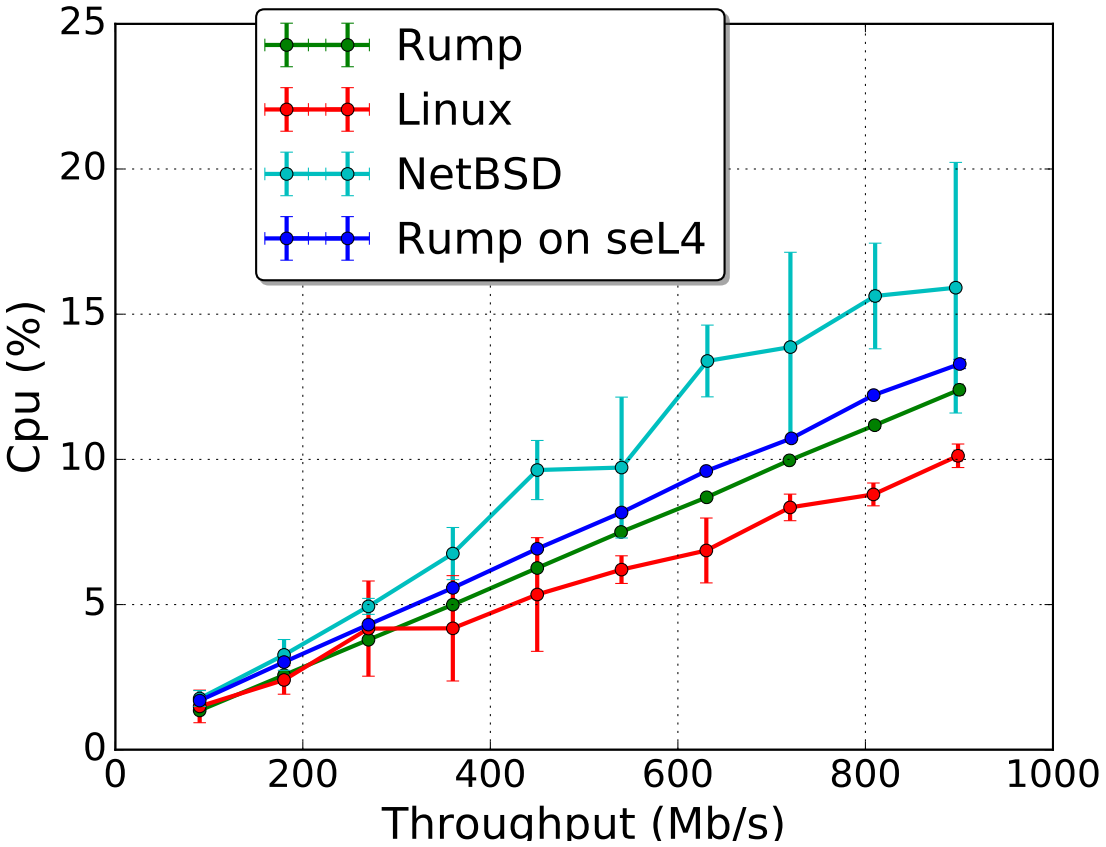


Figure 20: UDP Throughput benchmark

In conclusion, the OS components provided by rump kernels perform well compared to running them in their native NetBSD environment. The total utilisation is low, yet still higher than the comparable Linux system. Our port of the Rumprun unikernel onto seL4 has limited overhead and appears to be a good candidate for reusing device drivers and other kernel modules from other systems to reuse on seL4. Future work is required to provide a more comprehensive analysis. Areas where future work will benefit is in analysing other drivers as well as performing more macro based benchmarks that test the system in different ways as well as finding an explanation for the source of user-mode utilisation overhead. We also did not compare our networking stack against existing stacks that run on seL4.

8.4 Port evaluation

One of our goals was to find an approach that required low-effort to access a wide range of drivers. We review what was required to use the rump kernel drivers on seL4. Our approach consisted of the following:

- Add support to Rumprun unikernel to run as a process on seL4
- Write a seL4 root task that can create and initialise the resources for the unikernel process
- Modify the seL4 kernel to add support for required features
- Modify the rump kernel implementation slightly to be compatible with seL4

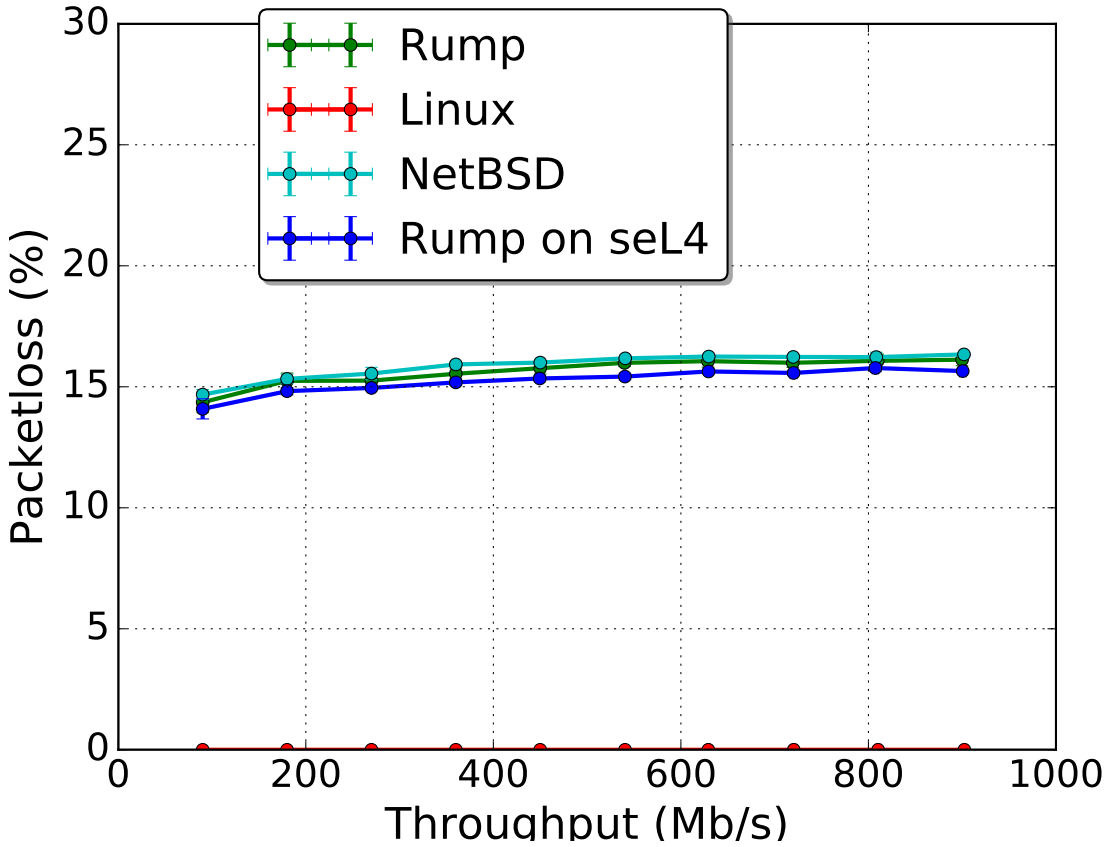


Figure 21: UDP Benchmark packet loss

- Configure and modify the build infrastructure to produce runnable binaries
- Tune the finished system to reduce overheads
- Create new rump kernel modules from existing NetBSD components

In this section we provide a quantitative and qualitative review of what effort was required. Quantitative analysis is presented as a summary of files edited as tracked by a source control management tool, git. Where relevant we also provide a qualitative review on amount of effort required.

8.4.1 Implementation effort

Our implementation is split across two seL4 components. The seL4 root task is used to receive the initial system resources from seL4 and use them to create, load and initialise the Rumprun unikernel process. In Table 8 this is the root task component. A more granular breakdown by file can be found in Appendix D.1. The implementation effort here is similar to a standard seL4 root task that is used to load and initialise another process.

The second component was the Rumprun unikernel. Again in Table 8, the Rumprun unikernel on seL4 component represents the amount of lines required to add support for seL4, further breakdown found in Appendix D.1. This implementation includes our initialisation code that takes the seL4 resources and uses them to initialise and configure the rest of the unikernel, our DMA and PCI interface implementations,

timer and timestamp implementations, and our functionality to handle interrupts delivered by seL4. It also includes our build scripts which add support for linking seL4 user-mode libraries into low level Rumprun code.

These lines of code represent several iterations of our design that was necessary in order to improve our performance.

Component	**Lines added**	**Lines removed**
Rumprun Unikernel on seL4	2660	0
seL4 Root task	838	0
Total	**3498**	**0**

Table 8: Breakdown of user-mode changes required to add Rumprun support.

8.4.2 seL4 kernel changes

This section contains a breakdown of the changes that were made to the seL4 kernel. Changes were made to achieve the following as per our design goals:

- adding the TLS invocation
- swapping x86 segment registers
- changes required for performance evaluation

Table 9 contains a summary of files modified and lines of code changed. Again a further breakdown can be found in Appendix D.2. As can be seen from the magnitude of lines added and removed, our TLS invocation and segment register modification were both small changes. Modifications for benchmarking purposes were larger but do not need to be kept in a production version. Changes required to be made to the seL4 kernel to support the Rumprun unikernel were small.

Feature	**Lines added**	**Lines removed**
Segment register changes	16	17
TLS Invocation	33	2
Benchmark changes	337	33
Total	**386**	**52**

Table 9: Kernel changes required to support Rumprun unikernel.

8.4.3 Rump kernel changes

We had to make minor modifications to some of the rump kernel modules in order for them to function correctly on seL4. One of the changes required replacing the x86 port instructions with seL4 invocations to execute them in a privileged mode. We also had to add a PCI un-map function to the rump kernel PCI interface. This was because some PCI devices try and un-map themselves after configuration. The other change required some small bug fixes to the USB stacks to enable a USB hub to be connected to another USB hub which had been disabled due to a requirement that was valid when running rump kernels on a POSIX api, but not valid when running on seL4. The amount of code changed to achieve this is shown in Table 10. Appendix D.2.1 has a more fine grained breakdown.

Change	Lines added	Lines removed
rump kernel libpci changes	29	10
rump kernel libusb changes	7	6
Total	**36**	**16**

Table 10: Rump kernel changes required to run on seL4.

8.4.4 Reducing overheads

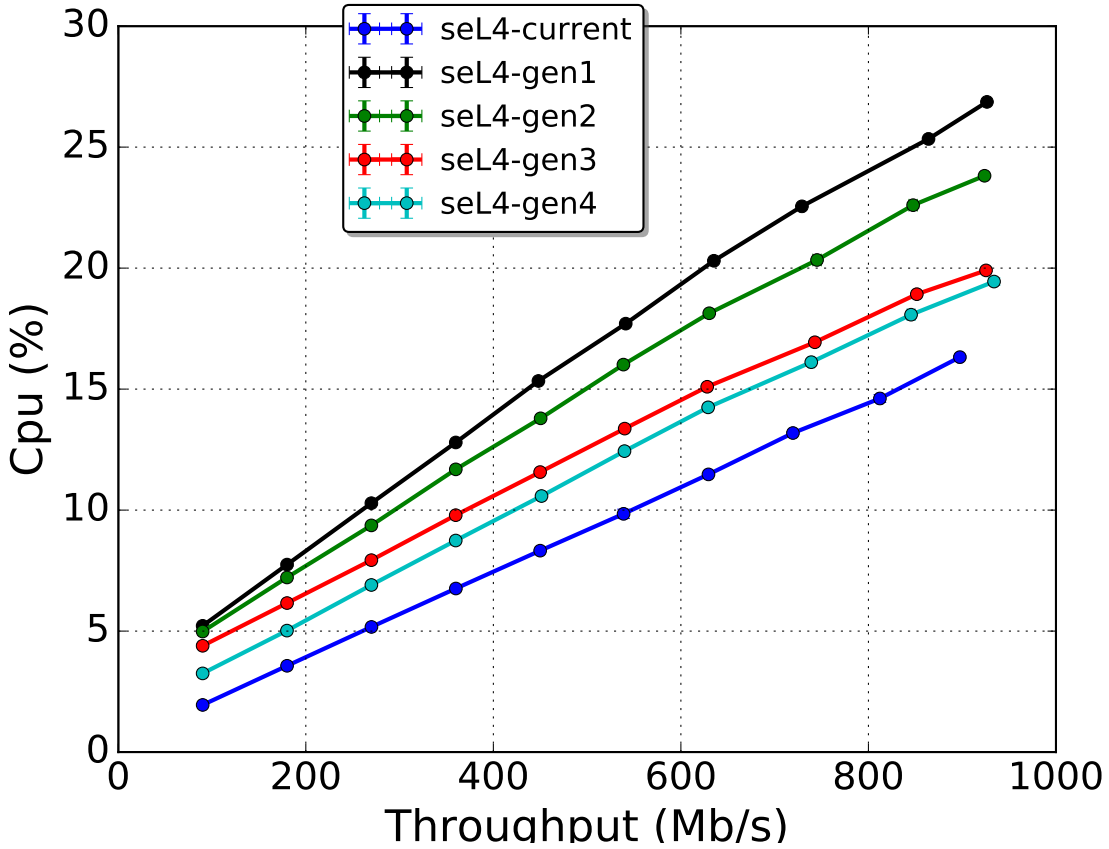


Figure 22: Iterative improvement

Figure 22 shows our Rumprun on seL4 implementation through several design iterations. Several of the changes came as a result of our kernel overhead measurements identifying the sources of overhead that we could then eliminate. Note that between our gen4 and current design we changed the TCP congestion control algorithm from newreno to cubic which is what likely shifted the 900Mb/s sample to the left on the graph. seL4-gen1 was our first design iteration and as we iterated, our generation number increased until our current design seL4-current. We list what improvements contributed to the increases in performance efficiency:

- One of the functions in the rump kernel PCI interface converts a virtual address to a physical address. In our initial implementation we were using a seL4 invocation to perform this mapping. At high network throughput, this function was getting called millions of times per second. Changing the implementation to instead rely on initial physical address information that is known when the root task is given the system resources by seL4 we can eliminate the entry to the kernel. This reduced our utilisation by about 11%.

- When we first started measuring kernel overhead (Figure 19), a significant amount of the overhead was caused by the IOPort instructions. In our gen2 design, our time stamp implementation would use the programmable interrupt timer (PIT) to fire an interrupt at 100Hz which would increase a global counter in 10ms increments. When the time stamp function was called, we would query the hardware register of the PIT for its current value and add this to the counter to provide nanosecond precision. However, to perform this read, 3 IOPort instructions were required. Reading the time stamp is supposed to be a cheap operation it was being called very frequently. By removing hardware register query on each time stamp call we saw the reduction on the figure between gen2 and gen3 of around 17%.
- Once we realised that our time stamp and interrupt timer functions were a source of overhead, we redesigned the system to use the inbuilt x86 time stamp counter for a global time stamp and using the programmable interrupt timer which is now only used to program interrupts when the system enters the idle state. This is the same as the bare metal rumprun kernel implementation. This gave us a slight reduction between gen3 and gen4 with a higher effect at lower CPU utilisations as we were no longer programming the PIT every 10ms.
- Our final reduction in overhead between gen4 and current was because we realised that the binaries had all been optimised for debugging and rebuilding all of the libraries and applications for release mode saw this final 20% utilisation drop.

8.4.5 Porting new netbsd drivers: AHCI/SATA

Library	**Lines added**	**Lines removed**
rump kernel ATA changes	74	1
rump kernel AHCI chagnes	73	1
Total	**147**	**2**

Table 11: Changes required to port NetBSD components to rump kernel modules.

Rump kernels did not currently support running NetBSD AHCI/SATA components. As a way of evaluating how easy it is to add additional NetBSD components as rump modules we added rump kernel support for these drivers. The total amount of changes we made can be seen in Table 11. Line changes per file can be found in Appendix D.2.2. The entirety of these line changes were associated with rump kernel wrapper additions. No changes were required to the actual NetBSD components themselves. 68 lines overall were added to add the new modules to the correct rump kernel link sets and add the devices as device nodes in the VFS subsystem. The remainder of the lines were required to make the build system aware of the new modules and to also add support for the device auto-configuration the the dev faction provides for auto-configuring devices on hardware busses. The drivers were tested by copying files on both the bare-metal and seL4 Rumprun implementations.

8.4.6 Running posix applications

We also include a list of the POSIX applications that we successfully ran on our Rumprun on seL4 implementation. These were applications that were run in the course of our development but we did not try to measure the ease at which POSIX applications can be run. From the experience we did have however, it appears to be easy to add POSIX applications provided that they support running without the

Application	Description
hello world	A simple hello world c program
nginx	A http webserver
iperf3	Networking benchmarking tool
redis	Scalable keyvalue storre
netperf	Networking benchmarking tool
lmbench(dd)	lmbench file copy

Table 12: POSIX applications we ran on Rumprun on seL4.

fork syscall, and support being built as a static binary (rather than having libraries loaded dyamically by a system runtime). There is an existing list of Rumprun supported applications that can be found here[4]. The list contains over 15 differnet large projects such as Python and OpenJDK8 that support POSIX systems which have been configured to run on Rumprun unikernels. We expect that these should also now be able to run on our Rumprun on seL4 system. Two of the items in our table redis and nginx come from this list. Other applications in our table we built using the rump kernel compiler toolchain and GNU's autotools build system framework that makes it easy to cross-compile POSIX compliant applications do different systems.

Based on our port evaluation, it has been low-effort to reuse driver-like NetBSD OS components on seL4 using the Rumprun unikernel and rump kernels. Our implementation was a reasonable size and modifications required to seL4 and the rump kernel were small. Adding new rump kernel modules appears to be low-effort and building POSIX applications for the unikernel also. Our iterative improvements of our implementation resulted in a final low level of overhead compared with Rumprun running on bare-metal.

8.5 Summary

In summary, we measured the amount of source lines of code provided by rump kernels and how rump kernel modularisation allows us to use significantly less when we want to use less drivers. Comparing the performance of the networking stack with other systems we found that we get good performance and minimal overhead was introduced when porting the Rumprun unikernel to run on seL4. When we evaluated the effort required to use rump kernels and the Rumprun unikernel we found that the effort required is low. Rump kernels appear to be an acceptable way of achieving reuse of driver-like OS components on top of seL4. Future work could involve performing an analysis of the user-mode overhead that our implementation introduces, performing a further analysis of different components by using different benchmarks, and also eliminating more of our kernel overhead by using different timer devices that do not require x86 IOPort operations. More of our future work is discussed in the following section.

[4] `https://github.com/rumpkernel/rumprun-packages`

9 Future work

There are many opportunities for future work related to using rump kernels on seL4. We briefly list and describe them here. Opportunities can be broken into categories of further analysis, further exploration of rump kernel features, other designs and potential uses in other seL4 projects.

9.1 Further analysis

As mentioned in the evaluation section, opportunities for further analysis involve:

Analysing user-mode overhead: We do not have a good argument for what contributes towards the user-mode overhead of our Rumprun port. This would require running more comprehensive profiling tools to try and reconstruct where time is spent in user-mode.

Benchmarks of other components: We only focussed on running networking benchmarks. Further benchmarks can be run to evaluate different OS components provided by rump kernels. This can also include benchmarks that utilise a wider range of rump kernel modules to see if there are any performance bottlenecks associated with the non-preemptive scheduling model.

Investigation on how to remove kernel entry: In addition to receiving interrupts, seL4 is entered to use the PortIO instructions, perform the new TLS invocation and to perform synchronisation between threads. We know that the PortIO instructions can be eliminated, but it may be possible to remove the TLS invocation and further reduce rate of synchronisation related syscalls.

9.2 Further exploration of Rump kernel features

Other architectures: We currently support only 32-bit x86 architectures. The Rumprun unikernel supposedly supports ARM platforms but we did not test this directly. NetBSD supports a range of hardware architectures so it would be worth trying to extend seL4 rump kernel use to these architectures also.

Multicore aware rump kernels: Rumprun uses rump kernels with multicore support disabled. This means that there is no way to preempt rump kernel threads once they are scheduled. It is possible to schedule multiple rump kernel threads at the same time at the expense of more synchronisation primitive use.

Syscall proxy for multiple address spaces: It is possible to dispatch rump kernel syscalls from one process to another using parts of the rump kernel host interface. The Rumprun unikernel also supports this. It can be configured to dispatch syscalls over a socket connection to another process where they are processed. This mechanism can be used to run applications in a different address space from the rump kernel which provides a level of fault isolation.

Fork capability: Many POSIX applications require the fork syscall functionality. Once applications are executing in different address spaces from the rump kernel it may be possible to implement the fork related syscalls.

9.3 Other non-unikernel designs

One option for future work is using rump kernels without relying on the Rumprun unikernel. Other designs involve using multiple rump kernels in different seL4 processes to provide OS components that are fault isolated from each other. It should also be possible to use rump kernel modules at the same time as native drivers written to seL4. An example of this would be using rump kernel networking modules on top of native seL4 ethernet card drivers.

9.4 Use in other seL4 projects

Rump kernels can be used in other seL4 projects where engineering effort is currently required to implement OS components that are not currently available on seL4 and it may be too expensive to run a virtualised Linux instance. However, some of the seL4 driver development is for trusted drivers and rump kernel modules would not be able to replace these. Yet they could still be used to provide comparative performance analysis and for faster prototyping of system designs before a high assurance driver is then developed.

In summary, there appear to be further opportunities to use rump kernels with seL4 such as opportunity for further analysis, further exploration of rump kernel features, other designs and potential uses in other seL4 projects.

10 Conclusion

We evaluated an approach for reusing NetBSD driver-like operating system components on the seL4 microkernel. This was achieved through porting the Rumprun unikernel, a project that uses rump kernels to reuse NetBSD operating system components. We performed a comparative performance evaluation of the NetBSD networking stack running inside a rump kernel, and evaluated the effort required to use rump kernels to provide unmodified NetBSD driver-like OS components that are runnable on seL4.

We measured the amount of source lines of code provided by rump kernels and how rump kernel modularisation allows us to only use components that are needed. Comparing the performance of the networking stack with other systems we found that we get good performance and minimal overhead was introduced when porting the Rumprun unikernel to run on seL4. When we evaluated the effort required to use rump kernels and the Rumprun unikernel we found that the effort required is low. Rump kernels appear to be an acceptable way of achieving reuse of driver-like OS components on top of seL4.

There is a range of opportunities available for future work including other areas for performance analysis, opportunities to further reduce our overhead, other rump kernel features, other designs and potential uses in other seL4 projects.

In this thesis we have presented the background information and related work regarding reusing operating system services. We also presented further background on rump kernels, the Rumprun unikernel and seL4. We provided our approach for porting the Rumprun unikernel and performing an evaluation and also provided our detailed design. We presented our evaluation and discussion of results and set out opportunities for future work.

References

A. Tanenbaum, *Modern operating systems.* Pearson Education, Inc.„ 2009.

S. C. Johnson and D. M. Ritchie, "UNIX time-sharing system: Portability of C programs and the UNIX system," *Bell System Technical Journal*, vol. 57, no. 6, pp. 2021–2048, 1978.

Open Signal, "Android fragmentation visualized," *Retrieved from opensignal.com: http://opensignal.com/assets/pdf/reports/2015_08_fragmentation_report.pdf*, 2015.

Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language], 1990, iEEE Std 1003.1-1990, ISO/IEC 9945-1:1990.

L. Torvalds, "Linux linux/arch," https://github.com/torvalds/linux/tree/master/arch, 2016.

ARM Architecture Reference Manual, ARM v7-A and ARM v7-R, ARM Ltd., Apr. 2008, aRM DDI 0406B.

Synopsys, "Coverity scan, open source report 2014," *Retrieved from scan.coverity.com: http://go.coverity.com/register-for-scan-report-2014.html*, 2015.

O. Ritchie and K. Thompson, "The unix time-sharing system," *Bell System Technical Journal, The*, vol. 57, no. 6, pp. 1905–1929, 1978.

NetBSD, "Netbsd," http://www.netbsd.org/.

J. Liedtke, "On μ-kernel construction," in *ACM Symposium on Operating Systems Principles*, Copper Mountain, CO, USA, Dec. 1995, pp. 237–250.

B. N. Bershad, "The increasing irrelevance of IPC performance for microkernel-based operating systems," in *Proceedings of the USENIX Workshop on Microkernels and other Kernel Architectures*, Seattle, WA, US, Apr. 1992, pp. 205–211.

A. Singh, *Mac OS X internals: a systems approach.* Addison-Wesley Professional, 2006.

J. Liedtke, "Improving IPC by kernel design," in *ACM Symposium on Operating Systems Principles*, Asheville, NC, USA, Dec. 1993, pp. 175–188.

G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," in *ACM Transactions on Computer Systems*, vol. 32, no. 1, Feb. 2014, pp. 2:1–2:70.

A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 461–472. [Online]. Available: http://doi.acm.org/10.1145/2451116.2451167

A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *ACM Symposium on Operating Systems Principles*, Lake Louise, Alta, CA, Oct. 2001, pp. 73–88.

B. Leslie, P. Chubb, N. FitzRoy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. R. Shen, K. Elphinstone, and G. Heiser, "User-level device drivers: Achieved performance," *Journal of Computer Science and Technology*, vol. 20, no. 5, pp. 654–664, Sep. 2005.

J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler, "Automatically generating malicious disks using symbolic execution," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 15–pp.

D. Mazieres, "A toolkit for user-level file systems." in *USENIX Annual Technical Conference, General Track*, 2001, pp. 261–274.

E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mtcp: a highly scalable user-level tcp stack for multicore systems," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 489–502.

R. S. Sandhu and P. Samarati, "Access control: principle and practice," *Communications Magazine, IEEE*, vol. 32, no. 9, pp. 40–48, 1994.

P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM Symposium on Operating Systems Principles*, Bolton Landing, NY, US, Oct. 2003, pp. 164–177.

J. S. Robin and C. E. Irvine, "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor," in *Proceedings of the 9th USENIX Security Symposium*, Denver, CO, Aug. 2000. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec2000/robin.html

C. Dall and J. Nieh, "KVM/ARM: The design and implementation of the Linux ARM hypervisor," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, Salt Lake City, UT, US, Mar. 2014, pp. 333–347.

V. Chipounov and G. Candea, "Reverse engineering of binary device drivers with revnic," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 167–180.

R. M. Barned and R. J. Richards, "Uniform driver interface (udi) reference implementation and determinism," in *Real-Time and Embedded Technology and Applications Symposium, 2002. Proceedings. Eighth IEEE*. IEEE, 2002, pp. 301–310.

L. Ryzhyk, P. Chubb, I. Kuz, E. Le Sueur, and G. Heiser, "Automatic device driver synthesis with Termite," in *ACM Symposium on Operating Systems Principles*, Big Sky, MT, US, Oct. 2009, pp. 73–86.

J. Kiszka, G. Pemmasani, and P. Fuchs, "Ndiswrapper," http://ndiswrapper.sourceforge.net/.

DROPS, "Fiasco.oc," http://os.inf.tu-dresden.de/fiasco/.

C. Helmuth, "Generic porting of linux device drivers to the drops architecture," Master's thesis, 2001.

A. Kantee *et al.*, "Flexible operating system internals: the design and implementation of the anykernel and rump kernels," Ph.D. dissertation, 2012.

A. Kantee, "Rumprun," https://github.com/rumpkernel/rumprun, 2016.

Trustworthy Systems Team, *seL4 Reference Manual, API Version 3.2.0*, NICTA, University of NSW, Sydney 2052, Australia, 2016, available from http://sel4.systems/Info/Docs/seL4-manual-latest.pdf.

Appendicies

Appendix A Rump components

```
rump                    Rump kernel base
rumpdev                 Rump kernel device faction
rumpnet                 Rump kernel networking faction
rumpvfs                 Rump kernel file system faction
rumpdev_audio           Audio support (incl. /dev/audio and /dev/mixer)
rumpdev_bpf             Berkeley Packet Filter
rumpdev_cgd             Cryptographic disk driver (block device crypto layer)
rumpdev_disk            Disk-like device support (used e.g. by file systems)
rumpdev_dm              Device-mapper driver (for LVM)
rumpdev_drvctl          /dev/drvctl driver
rumpdev_fss             File system snapshot device
rumpdev_md              Memory disk device driver
rumpdev_netsmb          SMB protocol communicator (required by SMB/CIFS)
rumpdev_pad             Pseudo Audio Device
rumpdev_pud             Userspace character and block driver framework
rumpdev_putter          User/kernel protocol transporter (for puffs and pud)
rumpdev_raidframe       RAIDframe (software RAID)
rumpdev_rnd             /dev/{,u}random
rumpdev_scsipi          SCSI & ATAPI mid-layer
rumpdev_sysmon          System monitoring and power management
rumpdev_vnd             Present a regular file as a block device (/dev/vnd)
rumpdev_wscons          Workstation console support
rumpdev_opencrypto      OpenCrypto, incl. /dev/crypto
rumpdev_ubt             USB BlueTooth driver
rumpdev_ucom            USB serial driver
rumpdev_ugenhc          USB host controller using /dev/ugen
rumpdev_ulpt            USB printer driver
rumpdev_umass           USB mass storage driver
rumpdev_usb             USB support
rumpdev_pci             PCI bus support
rumpdev_pci_if_iwn      Intel wireless device driver
rumpdev_pci_if_pcn      PCnet Ethernet device driver
rumpdev_pci_if_wm       Intel GigE device driver
rumpdev_pci_usbhc       PCI USB host controller drivers
rumpdev_pci_virtio      VirtIO bus support
rumpdev_virtio_if_vioif VirtIO network interface driver
rumpdev_virtio_ld       VirtIO block device driver
rumpdev_virtio_viornd   VirtIO entropy driver
rumpdev_virtio_vioscsi  VirtIO SCSI driver
rumpdev_audio_ac97      AC97 audio driver
rumpdev_pci_auich       AC97 Intel Audio driver
rumpdev_pci_eap         Ensoniq AudioPCI driver
rumpdev_pci_hdaudio     HDaudio PCI attachment
rumpdev_hdaudio_hdafg   High Definition Audio (hdaudio) driver
rumpdev_miiphy          MII and PHY drivers (for networking)
rumpfs_cd9660           ISO9660
rumpfs_efs              SGI EFS
rumpfs_ext2fs           Linux Ext2
rumpfs_fdesc            /dev/fd pseudo file system
rumpfs_ffs              Berkeley Fast File System
rumpfs_hfs              Apple HFS+
rumpfs_kernfs           /kern fictional file system
rumpfs_lfs              Log-structured File System
```

```
rumpfs_mfs              Memory File System (in-memory FFS)
rumpfs_msdos            FAT
rumpfs_nfs              NFS client
rumpfs_nilfs            NILFS
rumpfs_ntfs             NTFS
rumpfs_null             Loopback file system
rumpfs_ptyfs            /dev/pts pseudo file system
rumpfs_smbfs            SMB/CIFS
rumpfs_syspuffs         puffs in-kernel driver
rumpfs_sysvbfs          System V boot file system
rumpfs_tmpfs            tmpfs (efficient in-memory file system)
rumpfs_udf              UDF
rumpfs_umap             uid/gid mapping layer
rumpfs_union            union file system (fan-out layer)
rumpfs_v7fs             Unix 7th edition file system
rumpfs_zfs              ZFS
rumpfs_nfsserver        NFS server
rumpvfs_fifofs          File system FIFO support
rumpvfs_layerfs         Layer file system support (used by other drivers)
rumpvfs_aio             POSIX asynchronous I/O system calls
rumpkern_crypto         Cryptographic routines
rumpkern_sysproxy       Remote system call support (rump kernel as a server)
rumpkern_tty            TTY/PTY support
rumpkern_z              Data compression
rumpkern_sys_cygwin     Cygwin system call translation
rumpkern_sys_linux      Linux system call translation
rumpkern_sys_sunos      SunOS/Solarisa system call translation
rumpkern_sljit          Stackless JIT compiler
rumpkern_solaris        Solaris compatibility layer (for ZFS)
rumpnet_agr             Link aggregation pseudo interface (L2 trunking)
rumpnet_bridge          Bridging for IEEE 802
rumpnet_net             Network interface and routing support
rumpnet_net80211        IEEE 802.11 (wireless LAN) support
rumpnet_netbt           BlueTooth (PF_BLUETOOTH)
rumpnet_netinet         IPv4 incl. TCP and UDP (PF_INET)
rumpnet_netinet6        IPv6 incl. TCP and UDP (PF_INET6)
rumpnet_gif             generic tunnel interface
rumpnet_netmpls         Multiprotocol Label Switching (PF_MPLS)
rumpnet_npf             NPF packet filter
rumpnet_local           Local domain sockets (PF_LOCAL/PF_UNIX)
rumpnet_shmif           Shared memory bus network interface
rumpnet_tap             /dev/tap virtual Ethernet interface
rumpnet_bpfjit          JIT compiler for Berkeley Packet Filter
rumpnet_virtif          Network interface which uses hypercalls for I/O
rumpnet_sockin          PF_INET/PF_INET6 via hypercalls
```

artifacts/rumpcomponents.txt

Appendix B Rump host interfaces

B.1 rumpuser.h

```
/* init */
/* hypervisor upcall routines */
struct rumpuser_hyperup {
  void (*hyp_schedule)(void);
  void (*hyp_unschedule)(void);
  void (*hyp_backend_unschedule)(int, int *, void *);
  void (*hyp_backend_schedule)(int, void *);
  void (*hyp_lwproc_switch)(struct lwp *);
  void (*hyp_lwproc_release)(void);
  int (*hyp_lwproc_rfork)(void *, int, const char *);
  int (*hyp_lwproc_newlwp)(pid_t);
  struct lwp * (*hyp_lwproc_curlwp)(void);
  int (*hyp_syscall)(int, void *, long *);
  void (*hyp_lwpexit)(void);
  void (*hyp_execnotify)(const char *);
  pid_t (*hyp_getpid)(void);
  void *hyp__extra[8];
};
int rumpuser_init(int, const struct rumpuser_hyperup *);

/* memory allocation */
int rumpuser_malloc(size_t, int, void **);
void rumpuser_free(void *, size_t);
int rumpuser_anonmmap(void *, size_t, int, int, void **);
void  rumpuser_unmap(void *, size_t);

/* files and I/O */
int rumpuser_open(const char *, int, int *);
int rumpuser_close(int);
int rumpuser_getfileinfo(const char *, uint64_t *, int *);
typedef void (*rump_biodone_fn)(void *, size_t, int);
void rumpuser_bio(int, int, void *, size_t, int64_t, rump_biodone_fn, void *);
struct rumpuser_iovec {
  void *iov_base;
  size_t iov_len;
};
int rumpuser_iovread(int, struct rumpuser_iovec *, size_t, int64_t, size_t *);
int rumpuser_iovwrite(int, const struct rumpuser_iovec *, size_t, int64_t, size_t *);
int rumpuser_syncfd(int, int, uint64_t, uint64_t);

/* clock and zzz */
int rumpuser_clock_gettime(int, int64_t *, long *);
int rumpuser_clock_sleep(int, int64_t, long);

/* host information retrieval */
int rumpuser_getparam(const char *, void *, size_t);

/* system call emulation, set errno is TLS */
void rumpuser_seterrno(int);

/* termination */
int rumpuser_kill(int64_t, int);
void rumpuser_exit(int) __dead;

```

```
/* console output */
void rumpuser_putchar(int);
void rumpuser_dprintf(const char *, ...) __printflike(1, 2);

/* access to host random pool */
int rumpuser_getrandom(void *, size_t, int, size_t *);

/* threads, scheduling (host) and synchronization */
int  rumpuser_thread_create(void *(*f)(void *), void *, const char *, int, int, int, void
     **);
void rumpuser_thread_exit(void) __dead;
int  rumpuser_thread_join(void *);
enum rumplwpop {
  RUMPUSER_LWP_CREATE, RUMPUSER_LWP_DESTROY,
  RUMPUSER_LWP_SET, RUMPUSER_LWP_CLEAR
};
void rumpuser_curlwpop(int, struct lwp *);
struct lwp *rumpuser_curlwp(void);

void rumpuser_mutex_init(struct rumpuser_mtx **, int);
void rumpuser_mutex_enter(struct rumpuser_mtx *);
void rumpuser_mutex_enter_nowrap(struct rumpuser_mtx *);
int  rumpuser_mutex_tryenter(struct rumpuser_mtx *);
void rumpuser_mutex_exit(struct rumpuser_mtx *);
void rumpuser_mutex_destroy(struct rumpuser_mtx *);
void rumpuser_mutex_owner(struct rumpuser_mtx *, struct lwp **);

void rumpuser_rw_init(struct rumpuser_rw **);
void rumpuser_rw_enter(int, struct rumpuser_rw *);
int  rumpuser_rw_tryenter(int, struct rumpuser_rw *);
int  rumpuser_rw_tryupgrade(struct rumpuser_rw *);
void rumpuser_rw_downgrade(struct rumpuser_rw *);
void rumpuser_rw_exit(struct rumpuser_rw *);
void rumpuser_rw_destroy(struct rumpuser_rw *);
void rumpuser_rw_held(int, struct rumpuser_rw *, int *);

void rumpuser_cv_init(struct rumpuser_cv **);
void rumpuser_cv_destroy(struct rumpuser_cv *);
void rumpuser_cv_wait(struct rumpuser_cv *, struct rumpuser_mtx *);
void rumpuser_cv_wait_nowrap(struct rumpuser_cv *, struct rumpuser_mtx *);
int  rumpuser_cv_timedwait(struct rumpuser_cv *, struct rumpuser_mtx *, int64_t, int64_t)
    ;
void rumpuser_cv_signal(struct rumpuser_cv *);
void rumpuser_cv_broadcast(struct rumpuser_cv *);
void rumpuser_cv_has_waiters(struct rumpuser_cv *, int *);

/* dynloader */
void rumpuser_dl_bootstrap(rump_modinit_fn, rump_symload_fn, rump_compload_fn);

/* misc management */
int rumpuser_daemonize_begin(void);
int rumpuser_daemonize_done(int);

/* syscall proxy */
int rumpuser_sp_init(const char *, const char *, const char *, const char *);
int rumpuser_sp_copyin(void *, const void *, void *, size_t);
int rumpuser_sp_copyinstr(void *, const void *, void *, size_t *);
int rumpuser_sp_copyout(void *, const void *, void *, size_t);
int rumpuser_sp_copyoutstr(void *, const void *, void *, size_t *);
```

```
int rumpuser_sp_anonmmap(void *, size_t, void **);
int rumpuser_sp_raise(void *, int);
void  rumpuser_sp_fini(void *);
```

artifacts/rumpuser.h

B.2 sockin_user.h

```
int  rumpcomp_sockin_socket(int, int, int, int *);
int  rumpcomp_sockin_sendmsg(int, const struct msghdr *, int, size_t *);
int  rumpcomp_sockin_recvmsg(int, struct msghdr *, int, size_t *);
int  rumpcomp_sockin_connect(int, const struct sockaddr *, int);
int  rumpcomp_sockin_bind(int, const struct sockaddr *, int);
int  rumpcomp_sockin_accept(int, struct sockaddr *, int *, int *);
int  rumpcomp_sockin_listen(int, int);
int  rumpcomp_sockin_getname(int, struct sockaddr *, int *, enum
    rumpcomp_sockin_getnametype);
int  rumpcomp_sockin_setsockopt(int, int, int, const void *, int);
int  rumpcomp_sockin_poll(struct pollfd *, int, int, int *);
```

artifacts/sockin_user.h

B.3 pci_user.h

```
void *rumpcomp_pci_map(unsigned long, unsigned long);
void rumpcomp_pci_unmap(void *);

int rumpcomp_pci_confread(unsigned, unsigned, unsigned, int, unsigned int *);
int rumpcomp_pci_confwrite(unsigned, unsigned, unsigned, int, unsigned int);

int rumpcomp_pci_irq_map(unsigned, unsigned, unsigned, int, unsigned);
void *rumpcomp_pci_irq_establish(unsigned, int (*)(void *), void *);

int rumpcomp_pci_port_out(uint32_t port, int io_size, uint32_t val);
int rumpcomp_pci_port_in(uint32_t port, int io_size, uint32_t *result);

int rumpcomp_pci_dmalloc(size_t, size_t, unsigned long *, unsigned long *);
void rumpcomp_pci_dmafree(unsigned long, size_t);

struct rumpcomp_pci_dmaseg {
  unsigned long ds_pa;
  unsigned long ds_len;
  unsigned long ds_vacookie;
};
int rumpcomp_pci_dmamem_map(struct rumpcomp_pci_dmaseg *, size_t, size_t, void **);

unsigned long rumpcomp_pci_virt_to_mach(void *);

#ifdef RUMPCOMP_USERFEATURE_PCI_IOSPACE
int rumpcomp_pci_iospace_init(void);
#endif
```

artifacts/pci_user.h

Appendix C Rump kernel binary sizes

Library	Num files	SLOC	Filesize
rumpkern_bmktc	2	43	2.9KiB
rump	198	53681	1.2MiB
rumpkern_crypto	15	2517	96.3KiB
rumpkern_mman	2	195	5.1KiB
rumpkern_sljit	2	1557	39.1KiB
rumpkern_sys_linux	20	5662	124.3KiB
rumpkern_sysproxy	1	121	4.4KiB
rumpkern_tty	9	4318	87.2KiB
rumpkern_z	1	3877	44.3KiB
rumprun_base	15	3273	75.5KiB
Total	**265**	**75244**	**1.6MiB**

Table 13: Rump kernel modules.

Library	Num files	SLOC	Filesize
rumpdev_ubt	3	1499	25.1KiB
rumpdev_ucom	4	2557	42.0KiB
rumpdev_ugenhc	2	732	13.7KiB
rumpdev	5	3052	67.8KiB
rumpdev_ulpt	2	673	13.0KiB
rumpdev_ata	2	1628	30.2KiB
rumpdev_umass	5	2578	34.0KiB
rumpdev_audio	6	5444	90.7KiB
rumpdev_usb	10	5310	133.6KiB
rumpdev_audio_ac97	1	1837	30.9KiB
rumpdev_virtio_if_vioif	2	1195	23.2KiB
rumpdev_bpf	3	2033	32.6KiB
rumpdev_virtio_ld	3	939	22.4KiB
rumpdev_cgd	3	1069	20.2KiB
rumpdev_virtio_viornd	2	196	7.4KiB
rumpdev_disk	6	2852	57.5KiB
rumpdev_virtio_vioscsi	2	422	9.8KiB
rumpdev_dm	9	2093	60.7KiB
rumpdev_vnd	2	1507	22.4KiB
rumpdev_drvctl	2	530	12.3KiB
rumpdev_wscons	5	2762	38.5KiB
rumpdev_fss	2	1021	23.3KiB
rumpdev_hdaudio_hdafg	2	3923	45.9KiB
rumpdev_md	2	534	11.8KiB
rumpdev_miiphy	5	857	197.3KiB
rumpdev_netsmb	13	4958	96.2KiB
rumpdev_opencrypto	10	4859	99.3KiB

rumpdev_pad	2	626	13.0KiB
rumpdev_pci	13	5596	158.4KiB
rumpdev_pci_ahcisata	7	3421	68.9KiB
rumpdev_pci_auich	2	1338	23.8KiB
rumpdev_pci_eap	2	1586	22.8KiB
rumpdev_pci_hdaudio	2	1467	29.2KiB
rumpdev_pci_if_iwn	2	5020	78.5KiB
rumpdev_pci_if_pcn	2	1341	27.4KiB
rumpdev_pci_if_wm	2	8649	120.0KiB
rumpdev_pci_usbhc	8	10237	131.9KiB
rumpdev_pci_virtio	2	976	21.1KiB
rumpdev_pud	3	529	11.9KiB
rumpdev_putter	2	476	10.3KiB
rumpdev_raidframe	61	23658	346.2KiB
rumpdev_rnd	2	397	9.6KiB
rumpdev_scsipi	11	8622	159.2KiB
rumpdev_sysmon	11	4292	88.5KiB
Total	**247**	**135291**	**2.5MiB**

Table 14: Rump device modules.

Library	**Num files**	**SLOC**	**Filesize**
rumpnet	21	8092	156.9KiB
rumpnet_agr	18	2767	60.6KiB
rumpnet_bpfjit	1	1660	14.4KiB
rumpnet_bridge	3	2742	34.2KiB
rumpnet_config	10	2452	54.1KiB
rumpnet_gif	5	1455	23.6KiB
rumpnet_local	3	1388	23.6KiB
rumpnet_net80211	15	12611	210.1KiB
rumpnet_net	71	56846	934.3KiB
rumpnet_netbt	20	7298	132.5KiB
rumpnet_netinet6	1	22	1.8KiB
rumpnet_netinet	1	49	2.7KiB
rumpnet_netmpls	3	729	13.1KiB
rumpnet_npf	26	6589	143.9KiB
rumpnet_pppoe	2	1324	20.1KiB
rumpnet_shmif	3	666	14.4KiB
rumpnet_sockin	2	572	12.2KiB
rumpnet_tap	2	892	20.0KiB
Total	**207**	**108154**	**1.8MiB**

Table 15: Rump networking modules.

Library	**Num files**	**SLOC**	**Filesize**

rumpfs_v7fs	14	3840	76.9KiB
rumpfs_cd9660	7	2540	51.7KiB
rumpfs_efs	4	1540	33.7KiB
rumpfs_ext2fs	11	4627	97.9KiB
rumpfs_fdesc	2	905	18.7KiB
rumpfs_ffs	24	15736	319.0KiB
rumpfs_hfs	5	3469	69.5KiB
rumpfs_kernfs	2	1142	26.5KiB
rumpfs_lfs	28	17527	394.0KiB
rumpfs_mfs	2	507	15.8KiB
rumpfs_msdos	7	5606	85.4KiB
rumpfs_nfs	14	10553	251.2KiB
rumpfs_nfsserver	6	5363	152.0KiB
rumpfs_nilfs	3	2452	44.2KiB
rumpfs_ntfs	6	2876	56.2KiB
rumpfs_null	2	182	7.8KiB
rumpfs_ptyfs	3	1207	25.4KiB
rumpfs_smbfs	7	3799	75.3KiB
rumpfs_syspuffs	8	4724	101.5KiB
rumpvfs	49	24884	485.8KiB
rumpfs_sysvbfs	5	1771	43.1KiB
rumpvfs_aio	1	804	14.0KiB
rumpfs_tmpfs	7	2705	61.1KiB
rumpvfs_fifofs	1	474	10.9KiB
rumpfs_udf	11	12229	180.8KiB
rumpvfs_layerfs	3	693	13.9KiB
rumpfs_umap	3	626	18.0KiB
rumpfs_union	3	2410	46.9KiB
Total	**238**	**135191**	**2.7MiB**

Table 16: Rump VFS modules.

Appendix D Source code changes

D.1 Our implementation

File	Lines added	Lines removed
platform/sel4/Makefile	103	0
platform/sel4/arch/i386/Makefile.inc	13	0
platform/sel4/arch/i386/kern.ldscript	84	0
platform/sel4/arch/i386/machdep.c	51	0
platform/sel4/arch/x86/arch.c	83	0
platform/sel4/arch/x86/clock.c	211	0
platform/sel4/arch/x86/cons.c	63	0

platform/sel4/arch/x86/cpu_subr.c	80	0
platform/sel4/entry.c	411	0
platform/sel4/include/arch/i386/md.h	27	0
platform/sel4/include/arch/i386/pcpu.h	7	0
platform/sel4/include/arch/x86/inline.h	71	0
platform/sel4/include/arch/x86/reg.h	44	0
platform/sel4/include/arch/x86/var.h	29	0
platform/sel4/include/sel4/clock_subr.h	56	0
platform/sel4/include/sel4/helpers.h	165	0
platform/sel4/include/sel4/kernel.h	37	0
platform/sel4/include/sel4/multiboot.h	182	0
platform/sel4/include/sel4/test.h	108	0
platform/sel4/include/sel4/types2.h	11	0
platform/sel4/intr.c	204	0
platform/sel4/kernel.c	90	0
platform/sel4/pci/Makefile	21	0
platform/sel4/pci/Makefile.pcihyperdefs	18	0
platform/sel4/pci/rumpcomp_userfeatures_pci.h	2	0
platform/sel4/pci/rumpdma.c	205	0
platform/sel4/pci/rumppci.c	237	0
platform/sel4/platform.conf	1	0
platform/sel4/undefs.c	46	0
Total	**2660**	**0**

Table 17: Additions to rumprun unikernel.

File	**Lines added**	**Lines removed**
thesisrr/Kbuild	30	0
thesisrr/Kconfig	15	0
thesisrr/Makefile	85	0
thesisrr/src/arch/x86/arch.c	29	0
thesisrr/src/main.c	548	0
thesisrr/src/test.h	131	0
Total	**838**	**0**

Table 18: Additions for seL4 root task.

D.2 Kernel changes

File	**Lines added**	**Lines removed**
include/arch/x86/arch/32/mode/fastpath/fastpath.h	2	3
include/arch/x86/arch/32/mode/machine.h	6	6
libsel4/sel4_arch_include/ia32/sel4/sel4_arch/functions.h	4	4
src/arch/x86/32/kernel/thread.c	2	2
src/arch/x86/32/kernel/vspace.c	2	2

Total	**16**	**17**

Table 19: Changes for switching segment registers.

File	Lines added	Lines removed
libsel4/arch_include/x86/interfaces/sel4arch.xml	6	0
src/object/tcb.c	27	2
Total	**33**	**2**

Table 20: Changes for adding thread local storage invoation.

File	Lines added	Lines removed
Kconfig	8	3
include/arch/x86/arch/benchmark.h	2	0
include/arch/x86/arch/kernel/cmdline.h	2	2
include/arch/x86/arch/kernel/vspace.h	7	2
include/arch/x86/arch/model/statedata.h	2	2
include/benchmark_track.h	5	1
include/machine/io.h	2	2
include/object/structures.h	1	0
libsel4/include/sel4/benchmark_track_types.h	1	1
libsel4/sel4_arch_include/ia32/sel4/sel4_arch/mapping.h	1	1
libsel4/sel4_arch_include/ia32/sel4/sel4_arch/syscalls.h	17	0
src/api/syscall.c	2	1
src/arch/x86/32/kernel/thread.c	31	1
src/arch/x86/kernel/boot.c	4	1
src/arch/x86/kernel/cmdline.c	2	2
src/arch/x86/kernel/vspace.c	2	2
src/arch/x86/model/statedata.c	2	2
src/benchmark_track.c	13	2
src/machine/io.c	5	0
src/object/interrupt.c	7	1
src/object/tcb.c	2	2
src/plat/pc99/machine/io.c	219	5
Total	**337**	**33**

Table 21: Changes required for benchmarking.

D.2.1 Rumpkernel changes

File	Lines added	Lines removed
sys/rump/dev/lib/libpci/pci_user.h	5	1
sys/rump/dev/lib/libpci/rumpdev_bus_space.c	24	9
Total	**29**	**10**

Table 22: Modifications to PCI modules.

File	Lines added	Lines removed
sys/dev/usb/umass.c	1	1
sys/dev/usb/usbdi.c	5	5
sys/rump/dev/lib/libusb/USB.ioconf	1	0
Total	**7**	**6**

Table 23: Modifications made to usb modules.

D.2.2 Porting new netbsd drivers: AHCI/SATA

File	Lines added	Lines removed
sys/rump/dev/Makefile.rumpdevcomp	3	1
sys/rump/dev/lib/libpci_ahcisata/Makefile	24	0
sys/rump/dev/lib/libpci_ahcisata/PCI_AHCISATA.ioconf	14	0
sys/rump/dev/lib/libpci_ahcisata/pci_ahcisata_component.c	32	0
Total	**73**	**1**

Table 24: Modifications required for PCI AHCI component.

File	Lines added	Lines removed
sys/rump/dev/Makefile.rumpdevcomp	3	1
sys/rump/dev/lib/libata/ATA.ioconf	12	0
sys/rump/dev/lib/libata/Makefile	23	0
sys/rump/dev/lib/libata/ata_component.c	36	0
sys/rump/include/opt/ataraid.h	0	0
sys/rump/include/opt/opt_ata.h	0	0
sys/rump/include/opt/sata_pmp.h	0	0
Total	**74**	**1**

Table 25: Modifications required for ATA component.