Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 250b

Systems Group, Department of Computer Science, ETH Zurich

Using NetBSD Kernel Components on Barrelfish Through Rump Kernels

by

Leo Horne

Supervised by

Reto Achermann, Roni Haecki, Lukas Humbel,
Daniel Schwyn, David Cock, Timothy Roscoe

March 2019–September 2019

Informatik
Computer Science

Abstract

A major task in the implementation of new operating systems is writing code for filesystems, the network stack, and various device drivers, which allow the operating system to run on a large range of hardware. Using the NetBSD rump kernel, a tool that allows running parts of the NetBSD kernel in userspace, we show how to reuse code from the NetBSD operating system in Barrelfish with the goal of avoiding having to manually port a large amount of code from other operating systems. We implement a basic hypervisor for the rump kernel which enables using OS components such as filesystems, the network stack, and PCI device drivers. We additionally show how to integrate portions of the NetBSD kernel into the Barrelfish ecosystem by wrapping them inside Barrelfish applications and using RPCs to communicate with them. Finally, we evaluate the performance of our implementation and show that overheads are typically kept within reasonable limits.

Contents

Chapter 1

Introduction

Operating systems are arguably the most important piece of software running on any computer. They provide applications with a simple, standardized view of the platform they are running on and abstract away all the machine and hardware-specific details while providing support for a wide range of hardware and software protocols.

Often, the core functionality of an OS (e.g. memory management, interrupt handling, scheduling, etc.) constitutes only a small part of its total codebase, as can be seen in Figure 1.1. One of the main reasons why implementing an OS from scratch is non-trivial is because it must support a wide range of hardware and because software expects the OS to provide a wide range of services, including support for different filesystems and network protocols.

One of the biggest challenges in writing new operating systems is therefore providing OS-specific implementations for supporting myriad hardware devices and OS services. In his work *The Design and Implementation of the Anykernel and Rump Kernels* [12], Antti Kantee notes that in almost all cases, these OS components follow a rigid protocol, and implementing them involves simply mapping the functionality of that protocol to the OS in question. For example, a device driver adheres to the specification of its device and translates device commands into an OS-understandable format so that the OS can export the services of the device to other applications. An implementation of a filesystem or network protocol takes the specification of that particular filesystem or network protocol and implements it using the framework offered by the OS.

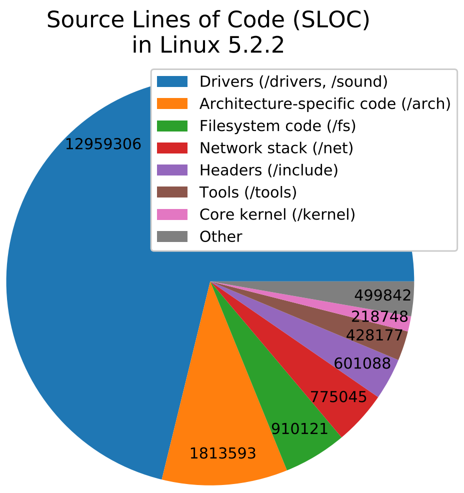

Figure 1.1: SLOC in Linux Kernel version 5.2.2. Generated using David A. Wheeler's 'SLOCCount'.

This mapping between protocols and OS frameworks constitutes an extremely large amount of the operating system's codebase. As can be seen in Figure 1.1, the vast majority of the codebase of the Linux kernel is dedicated to device drivers, filesystems, and network stack code. Implementing a new OS would therefore be much simpler if one could reuse these components from a different, more mature operating system instead of having to port or reimplement existing frameworks for each new operating system, which is often a very monotonous and tedious task due to it simply consisting of translating various protocols.

In this thesis, we will examine an approach for using components from the NetBSD operating system in the Barrelfish operating system using a NetBSD architecture called a rump kernel. More specifically, we seek to answer the following questions:

- Can the NetBSD rump kernel be easily ported to the multikernel architecture of Barrelfish? What difficulties are involved?
- How can we give the rump kernel the access to the low-level hardware resources it requires in order to run device drivers?
- How can rump kernels be integrated into the existing Barrelfish frameworks (e.g. for device drivers) in such a way that they are treated as first-class citizens of the Barrelfish ecosystem? Which API can regular Barrelfish applications use to access the OS components provided by a rump kernel?
- What is the performance of NetBSD OS components running on Barrelfish? If there are sources of substantial overhead, what are they?

To answer these questions, we will proceed in the following manner. Chapter 2 will present background information to understand the technologies and concepts related to Barrelfish and the NetBSD rump kernel with the goal of providing sufficient high-level knowledge to understand the implementation. Chapter 3 will then survey related work that provides similar functionality to rump kernels as well as examine implementations of the rump kernel on other platforms. After that, Chapter 4 will discuss the implementation on Barrelfish and Chapter 5 will present a brief performance evaluation of certain components. Chapter 6 will discuss the limitations of the current implementation and list some features that would be interesting to implement in future work. Chapter 7 will then conclude our discussion.

A Note on Manual Pages and the NetBSD Source Tree

This document references several NetBSD (not Linux!) manual pages. As usual, they appear in the format *tool/library*(*section*), e.g. printf(3). The relevant manual page for printf(3) can be found by typing `man 3 printf` on a NetBSD system or by using the online manpage tool at `https://netbsd.gw.com/cgi-bin/man-cgi?`.

We also make several references to the NetBSD source tree. The latest version of the NetBSD source tree can be found under `http://cvsweb.netbsd.org/` or checked out via CVS as explained in `https://www.netbsd.org/docs/guide/en/chap-fetch.html#chap-fetch-cvs-netbsd-release`. The version of the NetBSD kernel used for this document is 7.99.34, and its sources can most easily be obtained by running Hake on the Barrelfish source tree and then executing `make checkout-netbsd`. The sources can then be found in `lib/rump/buildrump.sh/src`.

Chapter 2

Background Information

2.1 The Barrelfish Operating System

Barrelfish is a research operating system created from scratch by the Systems Group at ETH Zürich, Switzerland. Its distinguishing feature is that it treats the machine it runs on as a distributed system of cores. The motivation behind this is that modern computers are already so complex they can be viewed as such [8]. The following subsections aim to introduce the parts of Barrelfish that are relevant to running the NetBSD rump kernel on Barrelfish.

2.1.1 The multikernel architecture

The Barrelfish kernel is structured as a so-called *multikernel*. In a conventional operating systems, the kernel runs on all available cores using mutual exclusion and shared memory. A multikernel treats the machine as a distributed system of cores by running independent kernels on each core, and any communication between cores is done via message passing in userspace [7]. Each core runs a small, non-preemptible kernel called a *CPU driver* that does not share any memory with other cores. Instead, each CPU driver possesses a local copy of the system state, and updates to the system state are propagated using message passing.

The functions of a CPU driver are very limited compared to the services offered by monolithic kernels such as Linux. On a per-core basis, CPU drivers are responsible for receiving hardware interrupts and faults and routing them to user-space handlers, providing access to hardware resources and certain kernel data structures through capability invocations (see section 2.1.2), responding to system calls, scheduling user-space tasks (see section 2.1.3), and providing a form of core-local message passing (see section 2.1.4) [4].

2.1.2 Access control using capabilities

Barrelfish uses *capabilities* [15] to control access to almost all resources. A capability can be thought of as an unforgeable token owned by a process that gives it the right to access the resource referenced by the capability. Examples include capabilities to regions of raw RAM or capabilities to page tables or I/O ports (on x86). There is a specific type capability for each type of resource, and certain types of capabilities may be retyped to other types following an inheritance tree. Each type of capability supports various *invocations* that allow the holder of the capability to use the underlying resource in various ways [31]. For example, a capability to a mappable chunk of physical memory (a `Frame`) may have an invocation to map it into the holder's virtual address space, and a capability to an I/O port on x86 may have an invocation to send a 32-bit data word to that I/O port.

Capabilities allow many of the functions traditionally performed inside the kernel to be moved to trusted userspace daemons in a spirit similar to that of an exokernel [9]. For example, Barrelfish applications are self-paging: instead of relying on the kernel to manage their virtual memory address space, they can use capabilities to get the kernel to install the desired page tables and manage their own page faults (see section 2.1.5).

Each applications's capabilities are stored in a separate part of its address space called a capability space or CSpace. The CSpace is organized as a two-level tree of CNodes. Each application has a pointer to a level 1 CNode, which typically contains capabilities to Level 2 CNodes. Each level 2 CNode contains capabilities to various types of resources.

2.1.3 Domains, dispatchers, and scheduling

A Barrelfish application (or *domain* in Barrelfish parlance) can run on several different cores and make use of shared memory and user-level threads by using userspace libraries. Each domain has one or more *dispatchers* (one per core) that constitute the core-local component of that domain. Dispatchers are also the unit of scheduling in Barrelfish that the CPU driver preemptively switches between.

In addition to preemptive scheduling of dispatchers, Barrelfish employs a scheduling technique known as *scheduler activations* [2]: once the CPU driver decides to schedule a dispatcher, instead of simply resuming the application code from where it was when the dispatcher was unscheduled, it upcalls a `run()` function of that dispatcher. This function executes user-space thread scheduling and is also responsible for handling events from the kernel such as page faults or incoming messages from other domains [4].

For example, assume two threads, thread A and thread B, are running on a single core (and therefore also a single dispatcher), and that thread A is currently executing. Once the dispatcher running the two threads gets unscheduled and rescheduled onto the processor core, it may decide to switch to thread B instead of continuing the execution of thread A.

2.1.4 Message passing

Since Barrelfish treats the machine as a distributed system of cores, message passing is an integral part of how the OS performs basic functions. Barrelfish implements several different backends for message passing (also known as inter-dispatcher communication or IDC). LMP, inspired by L4 IPC [16], is used for messages between applications on the same core and is supported by the local CPU driver. UMP is used for messages between applications running on different cores and multihop is used for applications running on different machines connected through a network.

Barrelfish also has a domain-specific language (DSL) called Flounder that, given a specification for a set of messages, can generate helper methods to route these messages using any of the aforementioned backends [6].

2.1.5 Virtual memory

As mentioned earlier, Barrelfish domains are self-paging. This means that they can manage their virtual address space (VSpace) themselves by obtaining a capability to raw RAM and retyping it to a mappable `Frame` or a capability for a page table. They can then perform invocations on these capabilities, which allow them to install their own page tables and mappings. Furthermore, domains can register a page fault handler that the CPU driver upcalls in case of a page fault. However since managing paging is not necessary for many applications, the library libbarrelfish provides a default paging implementation.

2.1.6 Kaluga, the SKB, and device drivers

To manage hardware resources, Barrelfish has a database (the *System Knowledge Base* or SKB) that contains information about the hardware in the system, such as which devices are connected and which drivers correspond to which devices. The SKB is queried using Prolog.

Barrelfish also has a device manager, Kaluga, that is responsible for managing the drivers attached to devices. At system startup, it queries the SKB for attached devices, finds the appropriate drivers, and starts them. It also hands drivers the necessary capabilities for interacting with their devices from userspace.

Device drivers in Barrelfish are composed of two parts: the *driver domain* and one or more *driver modules*. The driver domain typically simply dispatches events it receives on a special queue called a *waitset*, which allows it to receive messages from Kaluga as well as interrupts from the hardware it is controlling. It is usually statically linked with a library called `driverkit` which is responsible for handling messages from Kaluga.

Kaluga may send a *create* message to a driver domain, which will cause it to start a *driver instance* from one of the driver modules embedded into the driver binary. The driver instance is responsible for actually talking to the device. This way, Kaluga can dynamically create driver instances depending on the hardware configuration of the system (or destroy them via the *destroy* message) [5].

2.1.7 Build system: Hake

The build system for Barrelfish is called Hake. It is written in Haskell and defines a Haskell-based syntax that allows the user to declaratively (rather than imperatively, as with Make) specify how to build a library or application for Barrelfish using a special file called a Hakefile. Hake scans these Hakefiles and generates one large Makefile from them that is has recipes to build all the components of Barrelfish.

It does this by scanning the source tree for Hakefiles and then assembling all of them into one large Haskell expression containing a set of rules. These rules are parsed and translated into Make recipes [29].

2.2 The NetBSD Operating System

The rump kernel we are interested in running on Barrelfish is based on NetBSD. Therefore, the relevant features of NetBSD are introduced in this section.

NetBSD is a Unix-like operating system directly descended from the original Unix through the Berkeley Software Distribution (BSD). It was forked from 386BSD (itself directly descended from the original BSD) in 1993 [1].

2.2.1 Design principles of NetBSD

The NetBSD project emphasizes clean, high-quality code and the main goal of the NetBSD OS is to be able to run on as many different architectures as possible. This is why the slogan of the NetBSD project is "of course it runs NetBSD!" [1].

NetBSD achieves this by strictly dividing the OS into *machine-dependent* (MD) and *machine-independent* (MI) parts. The typical paradigm is that a MI interface is defined for use by various parts of the kernel, and this interface is implemented by separate MD implementations for each architecture NetBSD supports. For example, for reading from/writing to device registers, NetBSD defines the bus_space(9) interface, which allows drivers to be MI and not have to be rewritten for each architecture NetBSD runs on.

2.2.2 NetBSD kernel structure

In contrast to Barrelfish, NetBSD is based on a *monolithic* kernel: this means that almost all functions typically fulfilled by an OS (i.e. features that aim to provide abstractions for hardware and protecting processes from interfering with each other) are implemented inside the kernel. Therefore, the NetBSD kernel is very large and contains code not only for managing virtual memory and exceptions, but also for filesystems, the network stack, device drivers, and more.

2.2.3 Device drivers in NetBSD

As previously mentioned, device drivers in NetBSD run in the kernel. The kernel provides a large set of MI interfaces for drivers to use to talk to their devices. Besides bus_space(9), there is also bus_dma(9) for direct memory access (DMA), pci(9) for PCI devices, pci_msi(9) for message-signaled interrupts (MSI/MSI-X), and various other interfaces for different kinds of buses.

2.3 The NetBSD Rump Kernel

The focus of this project is to port the NetBSD rump kernel to Barrelfish and use it to run NetBSD OS components on Barrelfish. Because understanding the NetBSD rump kernel is key to understanding how it is run on Barrelfish, this section is devoted to explaining the fundamental concepts of the NetBSD rump kernel, based on Antti Kantee's *The Design and Implementation of the Anykernel and Rump Kernels* [12].

2.3.1 What is a rump kernel?

The term *rump kernel* is closely related to the term *anykernel*. As such, let us first define what an anykernel is.

Antti Kantee, who coined these two terms and designed the NetBSD rump kernel, defines an anykernel as "a term describing a kernel-type codebase from which drivers [...] can be extracted and integrated to *any* operating system model – or at least near any – without porting and maintenance work" [12, p. 27]. Here, drivers do not refer merely to device drivers, but a broader range of driver-like components of an OS, such as filesystems or the network stack.

A rump kernel is "a time-sharing style kernel from which portions have been removed. What remains are drivers and the basic support routines required for the drivers to function – synchronization, memory allocators, and so forth. What is gone are the policies of thread scheduling, virtual memory, application processes, and so forth. Rump kernels have a well-defined (and small!) portability layer, so they are straightforward to integrate into various environments." [12, p. 27].

The relationship between an anykernel and a rump kernel is that a rump kernel is typically built by selecting and extracting components from an anykernel, and then run on a certain platform (the *host*) by using its portability layer (e.g. a set of hypercalls).

2.3.2 Motivations behind rump kernels

The first motivation for rump kernels was to allow easy kernel driver development in the comfort of userspace, and they are still used by NetBSD for this purpose [10]. However, rump kernels also have the advantage that they can be fine-tuned to specific use cases by selecting the required components from an anykernel. As we will see in the next chapters, this dynamism makes rump kernels ideal for our purposes of running arbitrary NetBSD OS components in Barrelfish userspace.

2.3.3 Rump kernel structure: factions and components

As mentioned previously, a rump kernel is constructed by selecting different components from an anykernel. In this case, the anykernel is NetBSD, which was transformed into an anykernel during the development of the NetBSD rump kernel. Selecting a component from NetBSD consists of writing a set of files that define which files of the NetBSD source tree the component is based on and how to build and initialize the component. Based on these definitions, a specialized build script creates C libraries containing the specified behavior. A rump kernel is then constructed by (statically or dynamically) linking together all the desired components. Each of the components falls into three classes or *factions*, which we will discuss next.

Factions In order to facilitate extracting components from NetBSD, the NetBSD source tree was divided into a base subsystem and three factions: the device faction, the virtual filesystem faction, and the network faction. The base subsystem provides basic functionality required for all components to run and provides functions such as bootstrapping the rump kernel. The device (`dev`), network (`net`), and virtual filesystem (`vfs`) factions are orthogonal libraries (i.e., they do not depend on one another) that each rely on the base subsystem, and provide basic functionality for device drivers, network stack components, and filesystems respectively. This allows components to be defined without redundantly specifying the same base behavior in each component.

Components A *component* forms the basic building block of a rump kernel and contains behavior from a specific part of NetBSD. Each component is based upon one or more of the three factions. For example, a component containing the implementation of the TCP/IP protocols would be based upon the `net` faction, a component containing a driver for a network card would be based upon the `dev` faction and the `net` faction, and a component containing code for using the NTFS filesystem would be based upon the `vfs` faction.

2.3.4 Hypercalls

As mentioned in Subsection 2.3.1, kernels are not designed to run independently and need some form of support from their host, be it a full-fledged operating system or raw hardware, in the form of a portability layer. In the NetBSD rump kernel, this portability layer is implemented in the form of several *hypercall interfaces*. For example, if the rump kernel needs to allocate memory, it simply performs a hypercall instead of manually providing different implementations for different platforms. Hence, porting rump kernels to a new platform involves simply implementing these hypercalls and turning them into a library that can be linked with rump kernel components.

2.3.5 Bootstrap

"Booting" the rump kernel consists of initializing all of the components. The base faction is initialized by calling `rump_init()`, which initializes all the basic functionality needed by all components, then proceeds to initialize all the other components included in the rump kernel. The order of this initialization is specified by the use of different macros in the files defining that component. If the component is provided as a NetBSD kernel module, initialization simply consists of calling the module's initialization routine. For other components, the initialization routine is defined in the file specifying how to extract the component from the NetBSD source tree.

2.3.6 Virtual memory

The rump kernel does not manage its own virtual memory, since this would create a large amount of overhead with little benefit. Instead, it relies on the host to provide an address space and uses hypercalls to allocate memory. However, the NetBSD kernel relies on its virtual memory subsystem to perform basic tasks such as reading files. Therefore, the virtual memory subsystem is reimplemented by the rump kernel, but mostly consists of empty stubs or identity mappings and functions for allocating memory using hypercalls.

2.3.7 Kernel threads & scheduling

Like any modern operating system, the rump kernel operates using several threads (or lightweight processes (LWPs) in NetBSD parlance) in order to perform multiple tasks concurrently. However, the rump kernel does not include NetBSD's full preemptive scheduler and instead uses the host's thread and scheduling facilities. Since rump kernel LWPs are effectively mapped to host threads, the host can decide when to schedule and when to preempt the rump kernel.

To avoid data structure corruption due to LWPs being preempted when they would normally disable preemption while running on full NetBSD, the rump kernel introduces the notion of *virtual CPUs*. Each virtual CPU can only be running one LWP at a time, and each LWP must be running on exactly one virtual CPU. In other words, virtual CPUs can be regarded as a resource that an LWP must hold in order to execute code in the rump kernel. Each LWP runs to completion before it gives up the virtual CPU, or it can also yield the virtual CPU voluntarily. An internal scheduler then takes care of scheduling the next idle LWP onto a free virtual CPU. This internal scheduling can be viewed as a form of cooperative multitasking.

As an example, consider two LWPs, `lwp0` and `lwp1`. Assume `lwp0` is currently holding the (single) virtual CPU. Even if the host unschedules the thread running `lwp0` and wants to schedule the thread running `lwp1`, `lwp1` will not be able to execute in the rump kernel since `lwp0` is still holding the virtual CPU.

2.3.8 Avoiding namespace clashes

The rump kernel contains many symbols that are often already defined on the host such as `printf`. To avoid clashes with the host symbols, the rump kernel renames all of its symbols to be in a unique, separate namespace by prefixing them with `rumpns_`. Interfaces exposed by the rump kernel, such as system calls, are also renamed, e.g. the open(2) system call is renamed to `rump_sys_open`.

2.3.9 The rump kernel build system

The build system for the rump kernel is very similar to cross-compiling NetBSD. Regular NetBSD uses a shell script called `build.sh` which first compiles some tools for building NetBSD, and then uses a system of Makefiles to build the OS. The rump kernel uses a shell script called `buildrump.sh` which does something very similar. It first builds a toolchain for building NetBSD and performs some checks of the compiler specified by the `CC` environment variable, and then calls `build.sh` with the necessary parameters for building the rump kernel instead of the full kernel.

2.4 PCI

Since a substantial part of this project is devoted to allowing NetBSD PCI drivers to run on Barrelfish, we will give a brief description of the relevant features of PCI based on PCI-SIG's PCI Local Bus Specification (Revision 3.0) [24] and how support for it is implemented in Barrelfish, NetBSD, and the NetBSD rump kernel.

2.4.1 Basics of PCI

PCI (Peripheral Component Interconnect) is a widely used I/O bus for connecting hardware devices. It allows assigning each device a specific range of addresses in the processor's physical address or I/O space such that no device's address range overlaps with that of another. Each PCI device on the bus is assigned a specific device number, and each PCI device may implement multiple *functions*. Each function appears as a separate logical device, although they are physically implemented on the same hardware chip. A PCI device can therefore uniquely be identified by its bus:device:function numbers. Each physical PCI device is also assigned a unique ID consisting of two main parts: a unique vendor ID and a unique device ID. Together, these two IDs can be used to determine the correct device driver for a specific hardware device. PCI also supports *bridging*, where a special device to which further PCI devices can be connected can be attached to the main PCI bus.

A distinguishing feature of PCI compared to other buses is that each device contains 256 kB of *configuration space*, which is used by the BIOS, firmware, or OS to automatically configure the devices (e.g. by assigning certain physical address ranges to be used by each device). Once all the devices have been configured, device drivers can find out relevant information for their devices (e.g. which address ranges to read from/write to and which interrupts to service) by reading the configuration space. The configuration space layout for non-bridge devices is given in Table 2.1. For our purposes, the device ID, vendor ID, base address registers, interrupt pin, and interrupt line are most relevant; the others can be ignored.

To start the appropriate device drivers for each PCI device, the PCI bus must be enumerated. This entails querying the configuration space of each possible bus:device: function to check if a device is there. Trying to read from the configuration space of a non-existent device returns `0xFFFFFFFF`. If a device is present at a given bus:device: function, some basic information about it (such as the number of BARs it implements and sizes of the memory ranges they point to) is recorded. Later, the corresponding device driver for each device is determined by looking at its vendor and device IDs and matching it with information provided by the device driver.

Table 2.1: PCI configuration space layout for non-bridge devices

Offset	Bits 31:24	23:16	15:8	7:0
`0x00`	Device ID		Vendor ID	
`0x04`	Status		Command	
`0x08`	Class code	Subclass	Prog IF	Revision ID
`0x0C`	BIST	`0x00`[1]	Latency timer	Cacheline size
`0x10`	Base address register 0 (BAR0)			
`0x14`	Base address register 1 (BAR1)			
`0x18`	Base address register 2 (BAR2)			
`0x1C`	Base address register 3 (BAR3)			
`0x20`	Base address register 4 (BAR4)			
`0x24`	Base address register 5 (BAR5)			
`0x28`	CardBus CIS pointer			
`0x2C`	Subsystem ID		Subsystem vendor ID	
`0x30`	Expansion ROM address			
`0x34`	*Reserved*			Capabilities pointer
`0x38`	*Reserved*			
`0x3C`	Max latency	Min grant	Interrupt pin	Interrupt line

2.4.2 Memory-mapped and port-mapped I/O for PCI devices

As seen in Table 2.1, the configuration space for each device contains six 32-bit BARs, or Base Address Registers. There are two types of BARs: I/O BARs, which contain the start of a range in I/O port space (for x86-based processors) and memory BARs, which contain the start of a physical address range. These ranges of I/O space or memory are used by device drivers to talk to their devices. The two types of BARs are distinguished between by looking at the least-significant bit of the BAR. Tables 2.2 and 2.3 show the layout of each type of BAR.

Table 2.2: Layout of memory BAR

Bits 31:4	3	2:1	0
16-byte aligned base address	Prefetchable	Type	`0`

Table 2.3: Layout of I/O BAR

Bits 31:2	1	0
4-byte aligned base address	Reserved	`1`

The size of the address or I/O port range needed by the device can be determined as follows: first, one saves the original value of the BAR. Then, one writes all ones (`0xFFFFFFFF`) to the BAR and reads back the BAR. Since only the bits of the BAR that are beyond the range of memory needed by the device can be modified, one can then determine the amount of memory needed by performing a bitwise NOT and adding 1. Finally, one restores the original value of the BAR.

[1]Different for bridge devices

2.4.3 PCI capabilities

PCI capabilities refer not to a form of access control, but to additional abilities a PCI device may possess. The configuration header (Table 2.1) contains a capabilities pointer, which points to a linked list in configuration space. Each item of the linked list contains an 8-bit capability ID to indicate the type of capability, a pointer to the next capability in the linked list, and some further data specific to the capability.

2.4.4 Interrupts on PCI devices

PCI devices support three different kinds of interrupts: legacy interrupts, MSI interrupts, and MSI-X interrupts. In the following, we will briefly describe each kind of interrupt.

Legacy interrupts Legacy interrupts are akin to traditional interrupts on other, non-PCI devices. Each PCI device possesses an interrupt pin which is asserted if the device wants to raise an interrupt. The interrupt is then routed via the IOAPIC and LAPIC to the CPU, which can then execute the appropriate interrupt handler. One issue with this approach is that several different devices may share interrupt lines, which increases interrupt handling overhead, since every time an interrupt on that line occurs, all the handlers associated with that interrupt line must be executed. A further issue is that race conditions between DMA memory writes and interrupts can occur: for example, if a device performs DMA and then signals via interrupt that it is done, the interrupt may arrive before the DMA is actually completed, since the DMA write may be buffered or otherwise postponed by the memory controller.

MSI interrupts MSI interrupts (**M**essage **S**ignaled **I**nterrupts) aim to solve the above issues by a mechanism by which the device simply issues a write to a specific memory address in order to raise an interrupt. This solves the race conditions, interrupt line sharing issues, and also easily enables a device to generate multiple different interrupts by writing different data to the MSI interrupt memory address. Since Barrelfish does not support MSI, we will not provided a detailed explanation of MSI. However, a basic explanation follows. MSI interrupts are entirely configured in the configuration space of the device. A device indicates that it supports MSI by providing a capability with ID `0x05`. The MSI capability contains the address to which the device should write to generate interrupts (typically set by the BIOS, firmware, or OS) as well as the data that should be written to that address. It also contains a field that specifies how many different interrupts should be supported. The device generates each of these different interrupts by changing the least significant bits of the data. For example, if the device supports 4 interrupts, it will modify the 2 least significant bits of the data to generate different interrupts.

MSI-X interrupts MSI-X interrupts are an extended version of MSI interrupts. Instead of being configured solely in the configuration space of the device, they are configured partially by a PCI capability and partially by a table pointed to by one of the BARs. The layout of the capability register is presented in Table 2.4, and the layout of the table is presented in Table 2.5. The capability ID for MSI-X is `0x11`.

Table 2.4: Structure of the MSI-X capability

Offset	**Bits 31:16**	**15:8**	**7:3**	**2:0**
`0x00`	Message Control	Next cap. ptr	Capability ID	
`0x04`	Message upper address			
`0x08`	Table offset			BIR

Table 2.5: Structure of the MSI-X table

Offset	**Table contents**	
`0x00`	Message address (32 bits)	Message data (32 bits)
`0x08`	Message address (32 bits)	Message data (32 bits)
...	...	
`0x08`$\times(n-1)$	Message address (32 bits)	Message data (32 bits)

The message upper address in the capability structure contains the upper 32 bits of the address that is written to to generate interrupts on 64-bit systems. The BIR corresponds to the index of the BAR which contains the base address to access the MSI-X table. Adding this base address to the table offset gives the base address of the MSI-X table.

Each MSI-X table entry corresponds to a separate interrupt and contains an address that must be written to in order to generate that interrupt as well as some message data to write to that address.

2.4.5 PCI device drivers on Barrelfish

PCI on Barrelfish is handled by a PCI domain that performs basic initialization such as bus enumeration and populates the SKB with facts about each discovered device. It initially possesses a capability to the entire PCI device space, which it breaks down into more fine-grained capabilities for each detected device. PCI device drivers are then started by Kaluga just like any other driver.

In order to automatically spawn the correct driver domain for each device, the SKB contains a database of PCI vendor/device IDs and their corresponding driver domains and modules. Upon startup, Kaluga iterates over all connected PCI devices and queries the SKB for their corresponding driver. It obtains capabilities for each PCI device's BARs and interrupts through the PCI domain, which it stores in a CNode that is handed to the driver. The driver then uses these capabilities to map the BARs and set up interrupt handling.

2.4.6 PCI device drivers on NetBSD

NetBSD uses a process called autoconfiguration (see autoconf(9)) which matches devices to the appropriate driver. Each supported PCI device is listed in a PCI vendor/device ID database. The driver is then added to a file containing a list of PCI device drivers, and the PCI driver is added to the kernel configuration using a special configuration language (see config(5)).

Each PCI driver contains a function called `match()` which takes a description of a PCI device (including its vendor/device IDs) as input and returns `true` if the driver supports that device. Finding the driver for a certain device therefore entails calling the `match()` function of all registered PCI drivers until one is found that returns `true`.

The actual programming of the device in NetBSD is done using four MI frameworks that we have mentioned before: bus_space(9), bus_dma(9), pci(9), and pci_msi(9).

2.4.7 PCI device drivers on the NetBSD rump kernel

The rump kernel has a component that provides PCI bus support. If this component is linked into the rump kernel, it performs a basic bus enumeration upon initialization and starts PCI drivers (which are provided in the form of further components) in the same way as the full NetBSD kernel.

The way the rump kernel provides support for running PCI drivers is by providing hypercall-based (or *emulated*) versions of the three frameworks bus_space(9), bus_dma(9), and pci(9). Support for pci_msi(9) was provided as part of this project; see Chapter 4 for more details. In Chapter 4, we will also discuss the implementation of the hypercalls for the three included frameworks.

Chapter 3

Related Work

In this chapter, we will examine related work that also allows running OS components from one OS, the *source OS*, on another OS, the *target OS*. We will proceed by first examining various alternative approaches to achieving this and then by going over implementations of the NetBSD rump kernel on OSes other than Barrelfish.

3.1 Alternative Approaches

In this section, we will examine the various approaches designed with the goal of running part or all of the source OS on a target OS.

3.1.1 Virtualization and paravirtualization

A conceptually simple way to run components from the source OS on the target OS is simply to run the entire OS in userspace, providing the illusion of raw access to an entire machine in software. This approach is called *virtualization*. It has two components: an unmodified OS, called a guest OS (in this case, this is the source OS), and a virtualization layer called a *hypervisor* that typically emulates hardware and traps privileged instructions from the guest OS and simulates their execution in software. It has the benefit guest kernel is running in the environment it expects to run in, i.e. on raw hardware. This means that guest OS components can be run unmodified and without having to perform any configuration beyond what would be necessary on real hardware.

There are two types of hypervisors: type 1 hypervisors (such as Xen [3]), which run directly on raw hardware, and type 2 hypervisors, which run in software on top of an existing platform such as a general-purpose OS, such as VirtualBox [23].

A similar concept called *paravirtualization* differs from virtualization in that the guest OS knows it is not running on real hardware and thus provides special drivers for "devices" that actually rely on host OS components. These "devices" usually appear as pseudo-devices with similar interfaces to their physical counterparts. An example of this is the networking interface of Xen.

One drawback to this approach is that implementing such a hypervisor on a new platform is typically a non-trivial task. Furthermore, without implementing additional compatibility layers, the guest OS and host are completely isolated from each other and thus only applications designed for the guest kernel can use the guest's OS components, and data in the guest OS is completely invisible to the host and vice versa. The virtualized guest OS also often runs considerably more slowly than on real hardware.

3.1.2 Porting

A more technically involved method of running OS components from one OS in another is by *porting*. Porting an OS component involves reading the source code of the source OS component and rewriting it to conform to the software libraries and conventions of the target OS. The effort involved in doing this varies according to how closely related the two OSes are. For example, porting OS components between the various BSDs (FreeBSD, OpenBSD, and NetBSD) is often not very difficult due to their common BSD core and the fact that they often collaborate on various software interfaces. However, porting an OS component from NetBSD to Barrelfish is substantially more involved, since the architectures of these two OSes varies considerably. In fact, because of Barrelfish's approach to device drivers, porting OS components from other OSes to Barrelfish often requires completely restructuring the code.

3.1.3 Library operating systems

Not all OS components are necessarily run in the kernel. Even in the case of monolithic kernels, parts of what can be considered the operating system run in userspace. In order to use such components on the target OS, a crucial observation can be made: the only way a regular userspace application can access the kernel is through system calls. Providing an implementation for these system calls therefore theoretically enables running userspace OS components from the source OS. This is precisely what *library operating systems* aim to do.

An example of such a library OS is Graphene [32], a library OS that allows running unmodified Linux binaries. The high-level idea of Graphene is quite similar to that of the NetBSD rump kernel: it provides a set of libraries that emulate Linux system calls, relying on the host for machine-dependent tasks through a hypercall interface called the Platform Adaptation Layer (PAL). The difference is that Graphene does not actually provide an implementation for these system calls by using Linux kernel code but by providing the machine-independent functionality through a library called `libLinux`. Similarly, the Drawbridge library OS [25] transforms Windows 7 into a library OS. This is achieved by emulating the API exposed to Windows applications and relying on the host OS to perform tasks whenever possible. This shows that while library OSes are suitable for running userspace components or entire applications, their goal is typically not to provide complete versions of OS components running in kernel, which limits their usefulness for our purposes, since the most common desktop OSes are monolithic and run almost all of their components in kernel mode.

3.1.4 Library emulation

Specific types of OS components often rely on a well-defined subset of the software interfaces provided by that OS. Reimplementing these software interfaces on a different OS can therefore provide a means to run such OS components.

One such software interface is the Network Driver Interface Specification (NDIS) by Microsoft [20]. NDIS is a library that "specifies a standard interface between kernel-mode network drivers and the operating system [and] a standard interface between layered network drivers, thereby abstracting lower-level drivers that manage hardware from upper-level drivers, such as network transports" [20]. NDIS is implemented in Microsoft's Windows OS, but has also been ported to various other operating systems such as Linux (through ndiswrapper [22]) and FreeBSD (through NDISulator [21]).

Additionally, some work has been done on running Linux USB drivers on FreeBSD [28] by reimplementing several Linux driver interface routines in FreeBSD.

Such approaches allow running unmodified components from other OSes without considerable overhead, but require writing emulated versions of each possible call an OS component makes. For arbitrary components, this could be any function in the kernel (for components of a monolithic kernel) or even any function available in any userspace library (for OS components implemented in userspace). This makes a generic solution that allows running a wide range of OS components very difficult to implement, and therefore library emulation typically only works well for a very restricted subset of OS components (such as network card drivers in the case of ndiswrapper/NDISulator).

Another option that goes in the direction of library emulation is the UDI project [26]. Instead of emulating different device driver interfaces from various OSes, device drivers are written to conform to the UDI specification, which specifies an interface for performing operations such as memory-mapped I/O, DMA, and handling interrupts. The UDI libraries can then be ported to any OS, which allows UDI drivers to easily run across several different OSes. This solution avoids having to emulate hundreds of kernel routines, but does require drivers to be written to conform to the UDI specification. However, the main problem with this approach is that there are very few device drivers of practical value written using UDI.

3.1.5 Formal specifications

If the interface for an OS component, along with the interfaces it depends on, can be formally defined using some sort of specification language, then it is possible to automatically generate the code for such a component for various OSes. A concrete example of this is Termite [30]. As input, Termite takes three kinds of specifications: a device specification describing the device registers and the effects of reading from or writing to them, a device class specification specifying common behavior exhibited by various device classes (e.g. a device class specification for an Ethernet controller might specify that it can transmit a packet), and an OS specification that models the ways in which an OS interacts with its drivers. It then outputs an automatically-generated driver that is guaranteed to adhere to the constraints defined in the specifications and complete its tasks in finite time.

Reusing components from the source OS therefore becomes easy if they are generated by a tool like Termite, since writing an OS specification for the target OS suffices to automatically generate OS components for the target OS.

However, while very promising in theory, such an approach would require that a large number of device manufacturers and OS developers develop specifications for their devices or OSes respectively. Since this can be very time-consuming and many devices already have hand-written drivers for the most popular OSes, there are very few device drivers available that are automatically generated, which severely impairs its usefulness for running general OS components from the source OS on the target OS.

3.1.6 Running other kernels in userspace

The idea of running OS kernels in userspace is not new and has been done before. For example, the Linux Kernel Library (LKL) [27] provides the full, monolithic Linux kernel as a library that applications can call into via the Linux system call interface. It is quite similar to the NetBSD rump kernel in that it relies on a hypercall layer to do things like managing semaphores and thread creation/deletion. It has two main differences compared to the NetBSD rump kernel: it is not modular, i.e. the entire Linux kernel is turned into one library, and it uses the Linux scheduler for scheduling its own kernel threads, resulting in two layers of scheduling (the host scheduler and the Linux scheduler).

The benefit of such a library is that the Linux kernel supports a wide array of filesystems and devices, which makes using unmodified Linux components very desirable. However, the non-modularity of the LKL means that it is not as flexible as the NetBSD rump kernel, e.g. starting multiple instances of the LKL would consume much more resources compared to a very slim rump kernel. Also, the two layers of scheduling are an additional source of overhead.

Another approach which aims to run the full Linux kernel in userspace is the Windows Subsystem for Linux (WSL) [18]. Although its first version simply consisted of a compatibility layer to translate Linux calls to Windows calls, its second version runs a full Linux kernel using a highly optimized version of the Hyper-V hypervisor [19]. This allows Linux applications to be run on Windows with minimal performance overhead and also allows these Linux applications to make use of Linux kernel components. However, because this approach is essentially a lightweight form of virtualization, it comes with many of the same drawbacks: native Windows applications cannot use the resources provided by WSL in the same way as they would use Windows components, and device drivers do not control physical devices, but only the virtual devices provided by the hypervisor.

3.2 Rump Kernels on Other Platforms

In the previous subsections, we have examined other approaches to running components from the source OS on the target OS. We conclude see that rump kernels provide one of the easiest and most flexible approaches to this problem, since they allow easily running unmodified NetBSD kernel code by simply implementing some hypercalls, rely on the NetBSD kernel, for which filesystems and drivers are widely available, and provide a large degree of flexibility thanks to their modularity. In this section, we will therefore examine implementations of the rump kernel on other platforms, namely POSIX-compliant platforms that use ELF binaries and the seL4 microkernel.

3.2.1 POSIX userspace rump kernels

Rump kernels were originally developed to run on NetBSD, so the `rumpuser` library could very easily be adapted to run on virtually all POSIX platforms. In fact, the POSIX implementation of the `rumpuser` library is available by default in the NetBSD source tree under `lib/librumpuser`. *Building* the NetBSD rump kernel for non-NetBSD platforms is a different story, however. For several years after the advent of the NetBSD rump kernel, it could not easily be built for platforms other than NetBSD. It was only after a comprehensive build script called `buildrump.sh` (inspired by NetBSD's cross-platform build script `build.sh`) was introduced that building the rump kernel for non-NetBSD became easily feasible. As we will see later, this build script is also easily usable for building the NetBSD rump kernel for Barrelfish.

The POSIX implementation for the NetBSD rump kernel also provides several facilities for making the services of a rump kernel available to other applications. The basis for this is a server component implemented by the rump kernel that exports a URL. There is also a client component that can use this URL to communicate with the rump kernel at that URL using BSD sockets. By linking against a rump kernel client, applications can issue remote system calls to the rump kernel using the server URL. Dynamically linked applications can be "forced" to use a rump kernel by means of *system call hijacking*, whereby the dynamic linking process of the application is hijacked to link system calls to the rump kernel instead of to the host.

3.2.2 Rumprun on seL4

In 2016, the NetBSD rump kernel was also ported to the seL4 OS [17] with the goal of being able to reuse NetBSD kernel components. seL4 [14] is a formally verified microkernel-based OS that shares quite a few features with Barrelfish, including the use of capabilities to manage access to resources. The seL4 rump kernel is fully functional and provides access to NetBSD filesystems, the network stack, and also allows the use of basic PCI device drivers by providing the rump kernel with the proper capabilities to access the necessary low-level hardware resources. These capabilities are provided to the rump kernel manually directly from seL4's root (init) task, which is a trusted userspace task that initially possesses all capabilities.

One main difference between the seL4 implementation and the implementation presented in this document is that the seL4 implementation focuses on porting Rumprun [13], which is a wrapper around the rump kernel that allows running unmodified POSIX applications on top of the rump kernel by modifying NetBSD's libc to call into the rump kernel instead of making system calls. This means that it does not provide an interface to export the services provided by the rump kernel.

The seL4 approach has two main limitations from the point of view of the goals of this thesis: firstly, since the the rump kernel must be manually started from the root task, the OS cannot dynamically react to the hardware present in the machine by automatically providing capabilities to various devices. Secondly, since the services provided by the rump kernel are not exported for use by other seL4 applications, only the POSIX application running on top of the rump kernel can use these services.

Chapter 4

Implementation

In Subsection 2.3.1, we saw that rump kernels can easily be run on different platforms by implementing their portability layer. In this case, this consists of two sets of hypercalls: one set (called `rumpuser`) for running rump kernel components that do not require direct access to hardware, and one set of PCI hypercalls for accessing PCI devices. In this chapter, we will examine the implementation of each of these sets of hypercalls and describe an extension to the rump kernel that allows it to run drivers that use MSI-X interrupts.

Moreover, as stated in the introduction, one of the goals of this thesis was to integrate NetBSD OS components into Barrelfish as first-class members of the ecosystem. In this chapter, we also discuss mechanisms to allow regular Barrelfish applications to use NetBSD OS components such as the network stack and PCI device drivers in a way very similar to existing Barrelfish services.

4.1 General Architecture

Let us begin by describing the general architecture of the implementation before explaining each part in more detail. Figure 4.1 illustrates an example of a rump kernel running on Barrelfish. As mentioned in Subsection 2.3.3, a rump kernel is run in the virtual address space of an application, which we call the *wrapper application*, by linking together the hypercall layer, the base, zero or more factions, and zero or more components. Currently, wrapper applications run over a single dispatcher, which means that rump kernels cannot take advantage of the true parallelism offered by multiple cores.

Each layer in Figure 4.1 runs independently of the layers above it, except components, which may cross-reference each other (for example, the component for the Intel e1000 network card driver references the PCI driver component and the low-level networking component). Porting the rump kernel therefore requires implementing the two hypercall libraries `rumpuser` and `pci_hyper` using the libraries provided by Barrelfish, which are the only interfaces to the host the rump kernel uses. Section 4.2 will discuss the former and Section 4.3 will discuss the latter. Finally, in order make the components running inside a rump kernel hosted inside a wrapper application available to other Barrelfish applications, we discuss an RPC-based system call approach in Section 4.4.

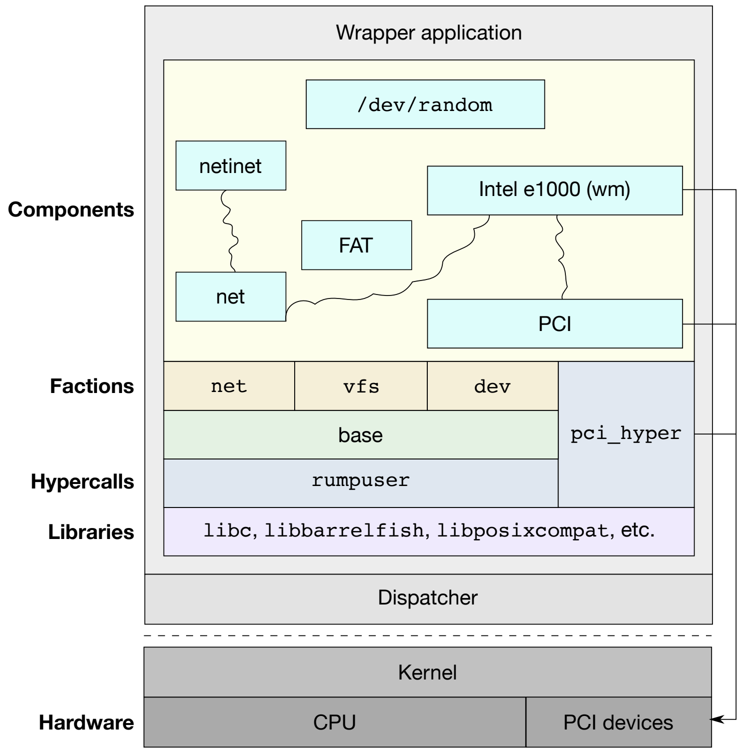

Figure 4.1: General architecture of the implementation

4.2 Base, Factions, Filesystems, and Network Stack

We will first describe running the rump kernel base and the three factions as well as components for filesystems and the network stack. These components are easy to implement because they rely solely on the `rumpuser` hypercall library and do not require access to hardware resources. The `rumpuser` library can be divided into eight categories: initialization/termination, memory allocation, files and I/O, clocks, parameter retrieval, randomness, threads, and mutexes.

Almost all of the `rumpuser` implementation could be adapted from the existing code for POSIX platforms by using Barrelfish's `posixcompat` library and by reusing code from a previous attempt to port the rump kernel to Barrelfish by Pravin Shinde. Using the `posixcompat` library makes sense, since a lot of the hypercalls roughly correspond to POSIX calls such as `open()` and `close()`, and writing those hypercalls from scratch would have resulted in implementations closely mirroring those of the `posixcompat` library anyway. The remaining hypercalls are also closely related to the POSIX versions but using various (near-)equivalent native Barrelfish APIs. This design choice was made for two reasons: it simplifies maintenance in case of new versions of the hypercall interface, and it also provides some insurance that the implementations are correct. The issue is that the hypercall interface is not thoroughly documented besides the rather informal documentation provided by the rumpuser(3) man page, so mirroring the POSIX implementation helps guarantee that the hypercalls provide an interface compatible with the rump kernel.

We will now proceed to describe the various hypercalls. Their implementation can be found in the `lib/rump/rumpuser` directory of the Barrelfish source tree.

4.2.1 Initialization and termination

As discussed in Subsection 2.3.5, to boot the rump kernel, an application calls a function provided by the rump kernel called `rump_init()`. This function initializes all the components and internally performs a hypercall to a function called `rumpuser_init()`. This is used in order to give the hypervisor a pointer to a struct of function pointers that allow the hypervisor to manage scheduling on the rump kernel, and also allows the hypervisor to perform any necessary internal initialization. The reason for the former is that the hypervisor sometimes needs to manage scheduling; for example, when a rump kernel LWP performs a blocking hypercall, the hypervisor temporarily unschedules it from its virtual CPU while the hypercall is blocked in order to allow other LWPs to proceed. As for the latter, the only extra initialization performed is to initialize threading, which we will discuss more closely in Subsection 4.2.7, and to initialize the Barrelfish VFS layer.

Explicit termination of the rump kernel is usually not necessary, as all resources it occupies are freed when the application hosting it in its address space exits. However, the rump kernel may need to exit in case of a fatal error or if the application wants to explicitly shut down the rump kernel. For this purpose, it internally calls the `rumpuser_exit()` function, which is essentially just a wrapper around the libc `exit()` function.

4.2.2 Memory allocation

All memory allocation in the rump kernel is performed by means of hypercalls. There are four memory allocation-related hypercalls, as seen in Listing 4.1.

```
int rumpuser_malloc(size_t howmuch, int requested_alignment, void **memp);
void rumpuser_free(void *ptr, size_t size);
int rumpuser_anonmmap(void *prefaddr, size_t size, int alignbit,
                      int exec, void **memp);
void rumpuser_unmap(void *addr, size_t len);
```

Listing 4.1: Hypercalls for memory allocation

The function `rumpuser_malloc()` mainly differs from the libc `malloc()` in that it returns the pointer as an out parameter and also takes a parameter for the alignment. In the POSIX implementation, this is taken into account by a call to `posix_memalign()`. However, since this function is not available on Barrelfish, memory alignment must be performed manually. We use a trick originally from Pravin Shinde's version of `rumpuser`, which involves `malloc`'ing a region R large enough so that (1) R is guaranteed to contain an address A with the desired alignment; (2) there is enough space between A and the end of R to satisfy the `howmuch` parameter; and (3) there is enough space before A to store a pointer P to the start of R. We will then return A for use by the rump kernel, and internally use P in order to properly free the entire allocated region of memory

R, since `rumpuser_free()` will be passed *A* by the rump kernel for the `ptr` parameter (and not *P*, i.e. the start of the allocated region). Allocating a region of size `howmuch` + `requested_alignment` + `sizeof(void *)` satisfies this request.

The `rumpuser` library must also provide a primitive for anonymous memory mapping (but not file-based memory mapping). This is easy to implement in Barrelfish, since libbarrelfish provides the `vspace_map_anon_aligned()` and `vspace_map_anon_fixed()` functions which perform essentially the same thing. The only optimization is that in order to ensure that even large chunks of virtual memory can be mapped, we try to allocate physical frames in sizes of decreasing powers of two until the allocation request is fulfilled. We pagefault on each frame so it can be added to the memobj returned by one of the `vspace_map_anon_*()` functions. `rumpuser_unmap()` is then just a wrapper around `vregion_destroy()`.

4.2.3 Files and I/O

Listing 4.2 shows the hypercalls related to files and I/O. Notice that many of the hypercalls are quite similar to POSIX calls. In fact, in the POSIX implementation, all of these hypercalls are implemented using standard POSIX functions such as `open()`, `close()`, `read()`, `write()`, `fsync()`, and `stat()` that are also available in the `posixcompat` library, meaning that these functions could all be copied verbatim from the POSIX implementation. The only difficulty was that the `posixcompat` did not implement the POSIX `readv()` and `writev()` calls (which are required for the `rumpuser_iovread()` and `rumpuser_iovwrite()` hypercalls), so they had to be implemented to allow this approach to work. Because the approach is the same as the POSIX version, we will not provide an extensive discussion here but advise the reader to consult the POSIX implementation in the NetBSD source tree under `lib/librumpuser/rumpuser_file.c` and to read the rumpuser(3) manual page.

```c
void rumpuser_putchar(int c);
void rumpuser_dprintf(const char *format, ...);  // debug printf
int rumpuser_getfileinfo(const char *path, uint64_t *sizep, int *ftp);
int rumpuser_open(const char *path, int ruflags, int *fdp);
int rumpuser_close(int fd);
int rumpuser_iovread(int fd, struct rumpuser_iovec *ruiov, size_t iovlen,
                     int64_t roff, size_t *retp);
int rumpuser_iovwrite(int fd, const struct rumpuser_iovec *ruiov,
                      size_t iovlen, int64_t roff, size_t *retp);
int rumpuser_syncfd(int fd, int flags, uint64_t start, uint64_t len);
```

Listing 4.2: File and I/O hypercalls

4.2.4 Clocks

The hypervisor must provide two clocks to the rump kernel: one which provides wall time, and one monotonically increasing clock (relative to some reference). According to the rumpuser(3) man page, if this is not possible, "the hypervisor must make a reasonable effort to maintain semantics". Listing 4.3 lists the relevant hypercalls.

```c
int rumpuser_clock_gettime(int enum_rumpclock, int64_t *sec, long *nsec);
int rumpuser_clock_sleep(int enum_rumpclock, int64_t sec, long nsec);
```

Listing 4.3: Clock-related hypercalls

The two different types of clock are distinguished by the `enum_rumpclock` parameter. In our implementation, for simplicity, we do not make a distinction between the two types This is a valid design choice because (1) for the `rumpuser_clock_gettime()` hypercall, the POSIX implementation also does not make a distinction between the two on some platforms and (2) for the `rumpuser_clock_sleep()` hypercall, what matters is that the calling thread sleeps for a given amount of time, no matter the type of the clock.

4.2.5 Parameter retrieval

The rump kernel can be configured by a number of parameters which it can get using the hypercall `int rumpuser_getparam(const char *name, void *buf, size_t blen)`. In the Barrelfish implementation, we support two parameters: `RUMPUSER_PARAM_NCPU`, the number of virtual CPUs available to the rump kernel, and `RUMPUSER_PARAM_HOSTNAME`, which returns a descriptive name for the rump kernel. For now, we always only use one virtual CPU (since we are only running on one dispatcher, we cannot use multiple physical cores anyway), and the hostname is set to `rump-<dispatcher name>-<core id>-<thread id>`. There are further parameters (e.g. a parameter that regulates the verbosity of the rump kernel's output), but as these are not necessary for supporting the core functionality of the rump kernel, we did not implement them.

4.2.6 Randomness

The rump kernel needs a way to obtain random bytes of data. The function
`int rumpuser_getrandom(void *buf, size_t buflen, int flags, size_t *retp)`
is responsible for this. It basically calls the libc function `random()` repeatedly until enough randomness is generated.

4.2.7 Threads

As mentioned previously, the rump kernel manages its LWPs (kernel threads) through hypercalls. The hypervisor must therefore provide primitives for (1) creating and destroying threads, (2) joining threads, and (3) for managing thread-local storage (TLS). The threading implementation is done using `pthreads` on Barrelfish, since this allows the POSIX implementation to be closely followed (thereby reducing the chance of implementation bugs), and `pthreads` also provide an easy way to manage TLS, as we will see.

Creating, destroying, and joining threads is done through an interface very similar to `pthreads`, as can be seen in Listing 4.4. The call for creating a thread takes some additional parameters that cannot be directly passed to `pthread_create()`, such as setting a thread's priority and whether it is joinable or not. In our implementation, setting the priority is not supported, since there does not appear to be a simple interface for doing so using `pthreads` on Barrelfish. The `joinable` bit simply specifies whether to return the ID of the created thread in `ptcookie` and is easy to support. The `cpuidx` bit is also ignored, since in our implementation, the rump kernel is only running over one dispatcher and cannot support true multicore execution. `rumpuser_thread_exit()` and `rumpuser_thread_join()` are just wrappers around `pthread_exit()` and `pthread_join()` respectively.

```c
int rumpuser_thread_create(void *(*f)(void *), void *arg,
                           const char *thrname, int joinable, int priority,
                           int cpuidx, void **ptcookie);
__dead void rumpuser_thread_exit(void);
int rumpuser_thread_join(void *ptcookie);
```

Listing 4.4: Rump kernel thread creation, deletion, and joining

The rump kernel requires two hypercalls to manage its TLS, as seen in Listing 4.5. Rump kernel threads (i.e. LWPs) store their TLS in the form of a `struct lwp`. The mechanism we use for managing it is through `pthread` keys, which allow each `pthread` to have their own chunk of memory to use for TLS. Upon initializing the hypervisor using `rumpuser_init()` (see Subsection 4.2.1), we issue a call to `pthread_key_create()`, which sets up a key that each thread can use to access its TLS. The use of the library functions `pthread_getspecific()` and `pthread_setspecific()` then allows easily setting/getting the TLS, which is what is required by the calls in Listing 4.5.

```c
void rumpuser_curlwpop(int enum_rumplwpop, struct lwp *l);
struct lwp *rumpuser_curlwp(void);
```

Listing 4.5: Rump kernel TLS management hypercalls

4.2.8 Mutexes

The rump kernel requires the hypervisor to provide three kinds of mutexes: (1) standard "lock/unlock" type mutexes, (2) reader-writer locks, and (3) condition variables. We will describe the implementation of each of these three kinds in the following. In all cases, the implementation again closely mirrors the POSIX implementation. This design choice is particularly suitable for this part of the implementation, as bugs in this area of the code are very difficult to find and debug and following the POSIX implementation provides some reassurance that the implementation is correct. Moreover, there is no precise documentation on the multitude of hypercalls necessary to properly support the various mutexes, making a self-designed implementation difficult to get right without thoroughly reading the POSIX implementation and understanding all the corner cases brought about by the use of various POSIX library functions. Pursuing an idea set forth by Pravin Shinde in the previous implementation, we also replace `pthread` mutex types such as pthread_mutex_t with ones from the Barrelfish threading library wherever possible, since the Barrelfish pthread implementation internally relies on the Barrelfish threading library anyway.

```c
struct rumpuser_mtx {
    struct thread_mutex bf_mutex;
    struct lwp *owner;
    int flags;
};
```

Listing 4.6: The Barrelfish rumpuser_mtx

Standard mutexes Standard "lock/unlock" mutexes are implemented in the rump kernel using an opaque (i.e., hypervisor-specified) struct rumpuser_mtx. On Barrelfish, this struct contains three fields, as seen in Listing 4.6. This allows easily implementing the required features, namely initializing, acquiring/releasing the mutex, and finding out who owns the mutex. The implementation is mostly identical to the POSIX implementation, with the exception that any calls related to pthread_mutexes are replaced by equivalent calls using thread_mutexes. For example, by using the thread_mutex_trylock() function instead of the pthread_mutex_trylock() function, we are able to support blocking on mutex acquisition by unscheduling the calling LWP from the virtual CPU if the mutex is currently held by someone else.

Reader-writer locks Similarly to standard mutexes, reader-writer locks are implemented using an opaque struct rumpuser_rw. On Barrelfish, this struct is defined identically to the POSIX version, since we use have to use pthread_rwlock_ts because Barrelfish does not natively provide reader-writer locks in its threading library. The code is therefore essentially identical to the POSIX implementation, just with some POSIX calls (e.g. nanosleep()) replaced by (near-)equivalent Barrelfish ones (e.g. barrelfish_usleep()) and some system-specific #ifdefs removed. Because of the extreme similarity of the two implementations, we will not provide a detailed description of the code here but instead advise the reader to refer to the POSIX implementation, which is well-commented and can be found in the NetBSD source tree under lib/rumpuser/rumpuser_pth.c.

Condition variables Once again, condition variables are implemented using an opaque struct rumpuser_cv which contains two members: a host condition variable and a count of the number of threads waiting on the condition variable. For this part, we can use native Barrelfish thread_conds instead of pthread_cond_ts as a backend for the implementation. We again follow the pattern of replacing the pthread_cond_t-related calls in the POSIX implementation with equivalent Barrelfish thread_cond-related ones and replacing the few POSIX calls with equivalent Barrelfish ones, and advise the reader to refer to the POSIX implementation for a detailed description.

4.3 NetBSD PCI Drivers and the PCI Hypercall Library

While the above implementation of the rumpuser library is sufficient for running a broad range of NetBSD OS components, running PCI drivers cannot be done using only the rumpuser library, since PCI device drivers need access to their hardware devices. A considerable portion of time therefore had to be spent implementing the additional support necessary to accomplish this, which is why this section is specifically devoted to PCI-related support routines. The task was made easier by the face that the rump kernel supports PCI devices by using an undocumented, unnamed PCI hypercall library which we have named `pci_hyper` for our implementation. This hypercall library contains the hypercalls used by the rump kernel reimplementations of bus_space(9), bus_dma(9), and pci(9) referred to in Subsection 2.4.7. In its default form, this hypercall library provides support for reading from/writing to PCI configuration space, performing port and memory-mapped I/O, handling (legacy) interrupts, and doing non-scatter-gather DMA. In addition, the PCI hypercall library was extended with hypercalls for initialization and supporting MSI-X interrupts, thereby allowing drivers to use pci_msi(9).

4.3.1 Initialization and architecture

Currently, the `pci_hyper` library is limited to controlling one device at a time, which means that a running rump kernel can only be running a device driver for one PCI device at a time. This is because doing so makes obtaining capabilities and device information easy by using Kaluga to start a driver wrapper, which performs initialization and starts the rump kernel (see Subsection 4.4.3), and Kaluga is only able to provide capabilities and device information for one device at a time. However, further PCI devices can be supported by starting multiple rump kernels, although using multiple instances of the same driver module is not supported because the rump kernel internally uses many `static` variables and because there can only be one rump kernel per address space.

The `pci_hyper` library expects to be initialized with a `struct bfdriver_instance` (typically obtained through Kaluga; Subsection 4.4.3 will describe the typical process for doing this). This `bfdriver_instance` contains all the necessary device memory and interrupt capabilities as well as information about the PCI address of the device and which types of interrupts it supports. It also contains a capability to connect to the Barrelfish PCI server, which the PCI hypercall library makes use of upon initialization to perform the connection to the PCI server.

Upon initialization, we also pass a flags bitmask (which currently only contains an option to use polling instead of interrupts) and a waitset on which interrupts are dispatched.

4.3.2 PCI configuration space

The PCI hypercall library needs to support accessing PCI configuration space for several reasons: (1) to perform the basic bus enumeration mentioned in Subsection 2.4.7, (2) to provide support for the NetBSD PCI libraries which need to access the configuration space for the implementations of various library functions (for example, the library function for mapping device memory needs to read the BARs), and (3) to allow NetBSD device drivers to access the configuration space of their devices, which may be required for them to find out certain device-specific information.

The basic mechanism for reading from/writing to the configuration space is by using RPCs to the PCI server. However, there are two issues which must be mitigated before making the RPC. First, NetBSD indexes the configuration space by byte offset, whereas Barrelfish indexes it by 32-bit register offset, meaning we must convert the NetBSD offset to a Barrelfish offset by dividing by 4 (the NetBSD offset is always 4-byte-aligned, so this does not cause any compatibility issues). Secondly, the rump kernel does not initially know which device it must start drivers for, which is why it does a bus enumeration, which involves trying to read from the configuration space of all possible PCI devices, but the specific device whose configuration space the PCI server reads from is "hardwired" in the capability used while connecting to it. This means we must check the PCI address whose configuration space is being accessed and only perform the RPC if this address matches the address in the `bfdriver_instance` received upon initialization. Otherwise, upon reads, we return `0xFFFFFFFF` to tell the rump kernel that no device that it should control is connected at that address.

4.3.3 Memory-mapped I/O

Memory-mapped I/O is one of the most important features the PCI hypercall library must support, since this is the mechanism used by device drivers to configure and issue commands to their devices. The PCI hypercall library implements a hypercall to map device memory into the address space used by the rump kernel. This is easy because the `bfdriver_instance` received upon initialization contains capabilities to map all of the device's BARs. The only complication is that the NetBSD driver does not know about this and instead directly passes the address which should be mapped. We therefore need to iterate over all the BAR capabilities to find the one which contains the right address and then map it read/write/non-cacheable using Barrelfish's VSpace mapping functions.

4.3.4 Direct memory access

In order to support DMA, the PCI hypercall library must support four primitives: (1) allocating physically contiguous memory for DMA use, (2) freeing this memory, (3) mapping this memory into the rump kernel's virtual address space, and (4) converting virtual addresses to physical addresses.

Primitive (1) is easy to implement in Barrelfish. It is simply a matter of making use of `frame_alloc()` to allocate a physically contiguous chunk of memory. We also follow the model of the Linux implementation of this library by already mapping this frame into the rump kernel's virtual address space, since in almost all cases, the driver will do this immediately afterwards anyway.

Primitive (2) is therefore also easy to implement. It can be accomplished by reversing all the steps taken while allocating and mapping the memory, i.e. destroying the mapping and frame.

Since we already map the memory while allocating it, the only thing that must be implemented in primitive (3) is returning the mapping.

Primitive (4) is conceptually easy to implement, since Barrelfish theoretically provides mechanisms to find the physical address of a given virtual address by using `frame_identify()` to find the base address of the frame the virtual address maps to and the `lookup()` function of the pmap to find the offset within that frame. However, there appears to be a bug in the implementation of this `lookup()` function (at least on x86_64) that causes it to almost always return 0. We work around this bug by using the following property. Assume v to be the virtual address for which we wish to find the corresponding physical address p. Let p_0 be the base address of the frame v is mapped to (obtained through `frame_identify()`). Assume v_0 is the virtual address closest to v that maps to the physical address p_0. Then, assuming there is no address between v_0 and v that also maps to p_0, $v - v_0 = p - p_0$, i.e. the virtual and physical offsets within the frame are the same.

We therefore use the algorithm presented in Listing 4.7 to discover v_0 and recover p by calculating $p = p_0 + (v - v_0)$. This algorithm, starting at v, iterates backwards in page-sized increments until it finds a virtual address that maps to a different frame or until it finds an unmapped virtual address. This means that the previous iterate was the virtual base address of the frame, i.e. v_0. As an optimization, instead of executing this entire algorithm every time we need to find the virtual base address of a frame, we cache the already-found base virtual addresses in a linked list. This saves time because executing `frame_identify()` several times means executing several system calls, which is inefficient. Instead of performing the iteration, if the base virtual address for a frame is already cached, we can just directly use it. In practice, the cache typically only contains a few items (in the order of 10), so iterating over the linked list does not create too much overhead.

```
algorithm virt_to_phys($v$):
    $c \leftarrow$ capability for frame $v$ is in;
    $p_0 \leftarrow$ frame_identify($c$);
    $v_0 \leftarrow$ undefined;
    for ($v_i \leftarrow v$; $v_0 =$ undefined; $v_i \leftarrow v_i -$ pagesize):
        $c_i \leftarrow$ capability for frame $v_i$ is in;
        $p_i \leftarrow$ frame_identify($c_i$);
        if $p_i \neq p_0$ or error occurred:
            $v_0 \leftarrow v_i +$ pagesize;
    return $p_0 + (v - v_0)$;
```

Listing 4.7: Virtual address translation workaround

4.3.5 Legacy interrupts

Many devices signal that events have occurred by issuing an interrupt. If interrupts are enabled, they preempt whatever the processor is doing and the processor starts executing code for one or more handlers (or interrupt service routines) associated with that interrupt. This preemption poses a challenge for our purposes because the rump kernel cannot be preempted. In the following, we describe a method to handle interrupts by scheduling the interrupt handlers onto a rump virtual CPU like any other rump LWP.

The PCI hypercall library keeps a linked list of `struct irqs` (see Listing 4.8) to keep track of the interrupts it has to handle. If a NetBSD driver wants to register for an interrupt and receive interrupts, the execution flow is as follows (see Figure 4.2 for a graphical depiction):

1. The NetBSD driver wants to register an interrupt and calls `intr_alloc()` with a parameter designating the desired type of interrupt.
2. The rump kernel's emulated version of the NetBSD PCI library internally makes a hypercall to `rumpcomp_pci_irq_map()`, passing a unique identifier (cookie), which creates a `struct irq` for the new interrupt and inserts it into the list of interrupts. The emulated PCI library returns a handle to the interrupt (i.e. the cookie).
3. After a while, the NetBSD driver is ready to receive and handle interrupts, so it calls `intr_establish()` with the previously returned handle and provides a handler and pointer to a handler argument.
4. The emulated PCI library makes a hypercall to `rumpcomp_pci_irq_establish()`. This function finds the `struct irq` for the specified interrupt and associates the handler and argument with it.
5. In the case of legacy interrupts, `rumpcomp_pci_establish()` creates a new thread with the `struct irq` as its argument which will create a rump kernel thread context for itself and then block on the condition variable inside the `struct irq`. The original thread also registers an interrupt with the Barrelfish interrupt routing service, and sets the handler for this interrupt to a function called `intrhandler()` with the `struct irq` as its argument.
6. If an interrupt occurs, `intrhandler()` will lock the mutex inside the `struct irq`, signal the condition variable to wake up the interrupt handling thread, and release the mutex.
7. The interrupt handling thread unblocks, schedules itself onto a free rump virtual CPU, executes the actual handler, unschedules itself, and blocks again.

```
struct irq {
    int type;                     // Legacy, MSI, or MSI-X
    unsigned cookie;              // Unique identifier
    int (*handler)(void *arg);    // Interrupt handler
    void *arg;                    // Argument to interrupt handler
    struct thread_cond cond;      // To signal if an interrupt arrives
    struct thread_mutex mutex;    // Mutex associated with cond
    int msix_index;               // Index in MSI-X table (only for MSI-X)
    int msix_bir;                 // Index of BAR where MSI-X table resides
    int intr_occurred;            // 1 if interrupt occurred
    struct irq *next;             // Next irq in list
};

```

Listing 4.8: IRQ struct

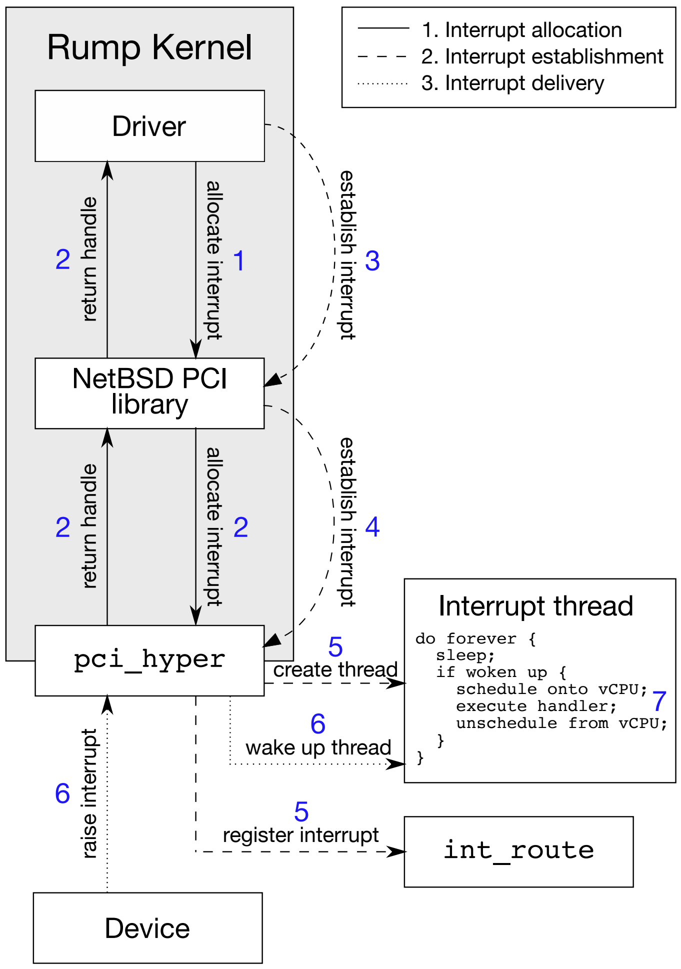

Figure 4.2: Diagram of interrupt allocation, establishment, and delivery. The blue numbers correspond to the steps in the execution flow.

4.3.6 MSI-X interrupts

As many PCI devices nowadays can take advantage of the benefits of MSI-X (as discussed in Subsection 2.4.4), it makes sense to support them. However, by default, the rump kernel does not emulate the necessary calls to support MSI-X (i.e. the pci_msi(9) interface). Therefore, we implemented extensions to the rump kernel PCI library and extensions to the PCI hypercall interface to allow using MSI-X interrupts.

Extensions to the rump kernel PCI library

The rump kernel PCI library needed to be extended to support the pci_msi(9) interface. The relevant functions are implemented as part of the emulated PCI library, but in a new file. Most of the functions cannot be implemented directly without support from the hypervisor, so they are essentially wrappers around new hypercalls, which we will introduce shortly. There are certain convenience functions that can be directly implemented without hypercalls, such as `intr_alloc()`, which takes an array expressing a list of types of interrupts (MSI-X, MSI, or legacy) in order of decreasing preference from the driver and tries to satisfy the driver's request to the best extent by calling various other functions from the pci_msi(9) interface. Since the pci_msi(9) interface contains functions related both to MSI and MSI-X functions, we also use macros for conditional compilation in case the host supports one type of interrupt but not the other, as is the case in Barrelfish, which supports MSI-X but not MSI. By not specifying any of these macros, the new interface is also fully backwards-compatible with the non-extended version of the emulated PCI library.

Extensions to the PCI hypercall interface

A summary of the new hypercalls is presented in Listing 4.9. These are very similar to the equivalent calls in the pci_msi(9) interface. In addition to these, the hypercall `rumpcomp_pci_intr_establish()` had to be extended to not only support establishing legacy, but (in our case) MSI-X interrupts. To illustrate how a driver can use MSI-X interrupts, we present an execution flow, referring back to the execution flow presented in the case of legacy interrupts in Subsection 4.3.5 in case of identical behavior. The execution flow is, in fact, very similar to that of legacy interrupts:

1. See step 1 of previous execution flow.
2. The emulated PCI library tries to satisfy the driver's request by calling the function for requesting the preferred type of interrupt. Assume that the driver prefers MSI-X interrupts.
3. The emulated PCI library makes a hypercall to `rumpcomp_pci_msix_alloc()`, passing how many interrupts are to be allocated and unique handles (cookies) for each of them along with the index of the BAR containing the address of the MSI-X table (found in the MSI-X capability).
4. `rumpcomp_pci_msix_alloc()` allocates a `struct irq` for each of the interrupts and inserts it into the list of interrupts. The emulated PCI library returns an array of handles (cookies) to the allocated interrupts.
5. See steps 3 and 4 of the previous execution flow.
6. For MSI-X interrupts, `rumpcomp_pci_establish()` follows a similar process to legacy interrupts. It uses Barrelfish's MSI-X service to establish an MSI-X interrupt with the same `intrhandler()` function as the handler and the `struct irq` as the handler argument, which requires mapping the MSI-X table in the virtual address space. It then creates a new thread for handling interrupts that does exactly the same thing as the thread for legacy interrupts.
7. If an interrupt occurs, the same things happen as in steps 6 and 7 of the previous execution flow.

Since supporting other types of interrupts is so straightforward, we also provide the possibility to not use interrupts at all and instead use polling to execute the interrupt handler at regular intervals. This mode can be activated by passing a flag to the PCI hypercall library upon initialization and does almost the same thing as in the case of legacy and MSI-X interrupts, except instead of the interrupt thread being unblocked upon interrupts, it periodically schedules itself onto a rump virtual CPU, executes the interrupt handler, and then unschedules itself and sleeps for a while. This mode can be used in case of unexpected behavior when using regular interrupts and for debugging.

Step 3 of this execution flow specifies that the emulated PCI library passes the hypercall library the index of the BAR containing the address of the MSI-X table. As we saw in Subsection 2.4.4, this is stored in the MSI-X capability as the BIR. However, there is currently a bug on Barrelfish where the capabilities to the BARs passed to PCI drivers from Kaluga do not correspond to the actual BAR numbers if some of the BARs are BARs to I/O space, which means that the BIR in the MSI-X capability may correspond to the right BAR number in hardware, but not in the list of capabilities passed by Kaluga. A concrete of example of this is the Intel 82574 (e1000e) network card, which has four BARs [11, p. 264]: BAR0 contains the base memory address for performing memory-mapped I/O, BAR1 is a flash BAR (a memory BAR), BAR2 contains the base I/O port address for performing port-mapped I/O, and BAR3 contains the base memory address of the MSI-X table. However, a PCI driver will receive the following capabilities from Kaluga: BAR0 and BAR1 will correspond to the hardware versions and BAR2 will be the BAR for the MSI-X table. Notice that the hardware BAR2 (an I/O BAR) is skipped, which messes up the BAR numbering so that the MSI-X table, from a driver's point of view, is at BAR2 instead of BAR3 as indicated by the BIR. Therefore, for the Barrelfish implementation of the PCI hypercall interface, we cannot trust the BIR. We work around this by providing a way to manually communicate the proper BAR number to the PCI hypercall interface when it is initialized by means of a flag.

```c
// Given a handle (cookie), determine which type (MSI-X/MSI/legacy)
// of interrupt is referred to by that handle
unsigned rumpcomp_pci_irq_type(unsigned cookie);

// Request one or more MSI interrupts
int rumpcomp_pci_msi_alloc(unsigned bus, unsigned dev, unsigned fun,
                           int *cookies, int count, int *retcount);

// Request one or more MSI-X interrupts
int rumpcomp_pci_msix_alloc(unsigned bus, unsigned dev, unsigned fun,
                            int bir, int *cookies, int count, int *retc);

// Cancel a request for an interrupt (typically used in case not
// enough MSI-X or MSI interupts could be allocated)
int rumpcomp_pci_irq_release(unsigned bus, unsigned dev, unsigned fun,
                             unsigned cookie);
```

Listing 4.9: New pci_msi(9)-related hypercalls

4.4 Integrating the Rump Kernel with Barrelfish

Now that we have seen how the rump kernel is implemented on Barrelfish, we need a way to use the rump kernel seamlessly within Barrelfish. This entails the following objectives:

Source tree and build system integration. The rump kernel uses a radically different build system than Barrelfish. However, to make the rump kernel easier to work with in Barrelfish, it is desirable to somehow integrate the two different build systems such that the Barrelfish build system (Hake) can be "tricked" into executing the rump kernel build system (`buildrump.sh`). In essence, the goal is to be able to provide the rump kernel as a regular Barrelfish library that can be used as any other Barrelfish library, i.e. compiled into the `lib` directory of the build tree and referenced in Hakefiles of other applications. We accomplish this by means of shell scripts and custom Hake rules.

Barrelfish application integration. As stated in the introduction, one of the goals of this thesis is to provide regular Barrelfish applications with a means to access the services of NetBSD OS components. Since the rump kernel runs in the address space of one (and only one) Barrelfish application, we need a way to share the services of a rump kernel with other applications. Naively, one could just let each application desiring to use NetBSD services start their own rump kernel, but this does not work. Consider the case of device drivers: having multiple device drivers controlling a single device is a recipe for disaster, and there is no easy way to provide a rump kernel with the capabilities to arbitrary devices from within a regular Barrelfish application.

In this section, we will analyze how to achieve these two objectives to the highest possible degree.

4.4.1 Source tree and build system integration

Before starting our discussion for this subsection, let us first present the general workflow when manually (i.e. without using Hake) compiling the rump kernel.

1. Clone the `buildrump.sh` repository, which contains the script needed to compile the rump kernel.
2. Run the `buildrump.sh` script, which does the following:
    (a) Clones a subset of the NetBSD source tree stripped of unnecessary userspace utilities.
    (b) Builds the tools to compile the rump kernel. This entails running some test on the compiler provided in the `CC` environment variable.
    (c) Invokes `build.sh` to build the rump kernel libraries.
3. If needed, copy the generated libraries to a suitable location.

For step 1, we provide the `buildrump.sh` repository as a git submodule in the Barrelfish source tree. This simplifies maintenance due to the fact that a git submodule is linked to a specific commit of the `buildrump.sh` repository, and each commit of the `buildrump.sh` repository is linked to a specific version of the NetBSD sources. This means that future commits to the `buildrump.sh` repository do not break the functionality of the implementation discussed in this report, and future commits can be gradually pulled in and tested for compatibility.

The general idea to automate step 2 is to write shell scripts to execute each of the subitems 2a, 2b, and 2c by invoking `buildrump.sh` with the appropriate arguments. We then write Hake rules which generate phony Make recipes that execute the shell scripts. For the sake of convenience, each of the shell scripts checks to see if the subitems that precede it have been executed and executes them if this is not the case.

However, there are a few issues which must be resolved before building the rump kernel libraries, namely that a default run of `buildrump.sh` does not build any PCI-related libraries and that we have made quite a few modifications to the rump kernel sources (e.g. extending the emulated PCI library). The resolution to the former is to edit the Makefile of the rump kernel's PCI library, and the latter requires adding code to various existing files and adding new files. We can solve both of these issues by using git patches which are applied to the rump kernel sources immediately before building the rump kernel libraries.

Step 3 is implemented by passing a flag to `buildrump.sh` to tell it where to put the generated libraries. In this case, we pass the path to the `lib` directory of the build tree.

This approach is sufficient to satisfy our requirements of being able to use Hake/-Make for building the rump kernel libraries and of being able to use rump libraries in Hakefiles of other applications. However, one limitation is that changing the rump kernel sources does not propagate down to applications using the rump libraries, i.e. Make will not realize that a dependency of those applications has changed and needs to be recompiled. However, in practice, changes to the rump kernel sources are infrequent enough that manually recompiling the libraries after source code changes is not burdensome to the programmer.

4.4.2 Barrelfish application integration

As mentioned earlier, we need a way for Barrelfish applications to communicate with a rump kernel running inside the address space of another application. In Subsection 3.2.1, we mentioned a mechanism for other applications on POSIX platforms to use rump kernel services using BSD sockets and system call hijacking. Because Barrelfish does not provide BSD sockets and because it does not support dynamic linking, these features are unusable on Barrelfish. However, because Barrelfish is a multikernel, it already provides excellent primitives for inter-domain communication, as we have seen in Subsection 2.1.4. We will see how we can use these to easily implement an interface to allow full integration of rump kernels with Barrelfish applications.

As with the POSIX implementation, the interface we will expose to Barrelfish applications is the system call interface. The reason for this is threefold: first, regular applications on NetBSD would also use this interface to interact with the kernel and therefore it is expressive enough to perform any required tasks, second, this interface is general enough to be suitable for interacting with a broad range of OS components, and third, the system call interface is very stable and unlikely to change.

The implementation consists of two parts: a server part and a client part. Each of them uses a Flounder interface to send messages to the other part. This Flounder interface is basically just the system call interface translated into Flounder syntax. The client exposes an interface nearly identical to the POSIX system call interface and translates these into RPCs using the Flounder interface. The server receives RPCs from the client, executes the relevant system calls on the rump kernel, and returns the result to the client, which then returns it to the caller. In order to distinguish between different wrappers responsible for different OS components and to allow clients to find the right server, the server part registers itself with the name service using a unique name. This paradigm is very similar to regular Barrelfish services such as the SKB, PCI service, or interrupt routing service, which allows rump kernel components to be seamlessly integrated with Barrelfish as first-class citizens of the ecosystem.

Despite the advantage of good integration, this approach does have some drawbacks. The main one is that the large amount of system calls makes translating each one to Flounder syntax very tedious, but the syntax of Flounder is not quite rich enough to easily automate this task. One might think that a workaround would be to implement the syscall(2) system call as an RPC to avoid defining each system call in Flounder, but this does not work, as Flounder does not support passing arguments that could be either integers or pointers to data. We illustrate the problem with an example: consider the read(2) system call and the open(2) system call. The read(2) system call takes a file descriptor (an integer), a path to a file (a string), and a parameter indicating how many bytes to read (an integer). It then returns an integer indicating how many bytes were read. The open(2) call takes a file path (a string), an integer representing some flags, and an optional `mode` parameter used to determine the mode a file should be created in. Here are the Flounder definitions for these two system calls:

```
rpc read(in int d, out uint8 buf[readbytes, 4096], in uint64 nbytes);
rpc open(in String path[4096], in int flags, out int fd);
```

These two system calls present the following challenges to automatic generation. For read(2), the return value is the size of the returned buffer, whereas for open(2), it is a regular integer. It is difficult to automatically decide based on the C interface what the return value is and how to specify it in Flounder (as the size of a buffer or a separate `out` parameter). Also, pointers to memory must be viewed as arrays of bytes (meaning that a lot of ugly casting must be done by the client and server), and one must specify a maximum size for this array, which can be difficult to do automatically. Finally, the `mode` parameter is completely absent from the `open()` RPC, as (1) it cannot be inferred from the C implementation, where it is hidden by the variadic declaration of the function (as "`...`") and (2) Flounder does not support variadic RPCs. Because of these difficulties, all system calls must currently be implemented manually in the Flounder interface, server part, and client part. Because this is a lot of monotonous work, only certain interesting system calls related to files and sockets have been implemented. However, implementing others if needed is easy.

4.4.3 Putting everything together: rump kernel wrappers

Now that we have seen how the rump kernel is implemented and how other Barrelfish applications can communicate with it, let us see how to provide a NetBSD OS component to other Barrelfish applications. The idea is that the NetBSD component will be provided as part of a rump kernel. A special Barrelfish application called a *wrapper* will be responsible for initializing the rump kernel and any other necessary libraries. It then uses the rump kernel server API to expose its rump kernel's system calls to other applications (of course, it can also define its own RPC interface in case the system call interface is not sufficient). In the case of OS components that do not require any additional capabilities, this is easy: just implement the wrapper and run it. For OS components that need additional capabilities (in our case, PCI drivers), we need a method to provide the capabilities. We use Kaluga for this.

Starting PCI driver wrappers with Kaluga

PCI drivers present two challenges: (1) they require capabilities to properly access their devices and (2) they cannot be started if their corresponding device is not present in the machine. We therefore need a mechanism to start wrappers for PCI drivers that solves these two challenges. An obvious choice is Kaluga and Barrelfish driver domains/modules (as discussed in Subsection 2.1.6), since Kaluga automatically only starts driver domains for devices that are present in the system and provides drivers with all the necessary capabilities to their devices. This has the additional benefit that drivers running in the rump kernel are identical to regular Barrelfish drivers from the point of view of Kaluga and other Barrelfish applications, making them first-class citizens of the Barrelfish ecosystem.

The architecture of a PCI driver wrapper is slightly different from a regular wrapper because it must interact with Kaluga, dispatch interrupts, and possibly perform some additional device configuration (e.g. use DHCP to obtain an IP address for network card driver wrappers). As is typical for device drivers started by Kaluga, the wrapper is structured as two parts: a driver domain and a driver module.

As usual, the driver domain is linked with `driverkit` and dispatches events on its default waitset in order to communicate with Kaluga. In addition to this, the driver domain also dispatches events on a separate interrupt waitset. The reason for this separate waitset is that when performing additional configuration within the driver upon initialization (such as using DHCP), the driver may expect interrupts (e.g. for incoming packets), but since it is still in the process of being initialized (i.e. the driver module creation message from Kaluga is still being processed), no events are dispatched on the default waitset and therefore, using the default waitset for dispatching interrupts in such a situation results in a deadlock.

The driver module must implement all the necessary functions required by a driver module, although all of them except the `init()` function simply return `SYS_ERR_OK` without doing anything. The `init()` function is responsible for initializing the PCI hypercall library with the appropriate `struct bfdriver_instance` (which it is passed from Kaluga), bootstrapping the rump kernel, initializing the rump server API, and performing any other necessary configuration, as shown in Figure 4.3.

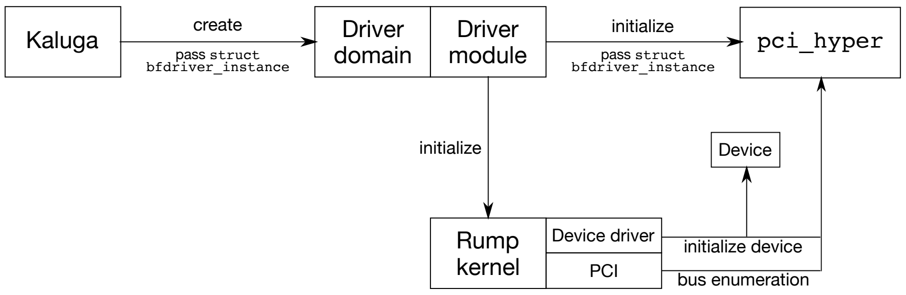

Figure 4.3: Simplified diagram of driver start procedure

Finally, a new entry should be created in Barrelfish's PCI device database `device_db.pl` to make sure that Kaluga knows the wrapper provides a driver for a PCI device with a specific vendor/device ID, similarly to native Barrelfish PCI drivers.

Chapter 5

Performance Evaluation

In this chapter, we will attempt to evaluate the performance of the Barrelfish port of the NetBSD rump kernel. It is important to note that code running in the rump kernel which does not rely on hypercalls is running directly on Barrelfish just like a normal Barrelfish application. Therefore, the only difference between such components running on another host as compared to Barrelfish is the way the scheduling policy of the host affects the execution order of the code. It is beyond the scope of this work to perform a detailed analysis of how the scheduling policy of various operating systems affects the execution time of non-hypercall-based OS components, but Kantee's *The Design and Implementation of the Anykernel and Rump Kernels* [12] mentions that using a rump kernel-based network stack (which uses very few hypercalls except at the bottom of the network stack) as the backend for the Firefox web browser results in no perceptible loss of performance for day-to-day browsing. Instead, we will focus on those OS components that rely on host-specific hypercalls to perform their tasks, which entails measuring the performance of various hypercalls and attempting to establish which of them may be bottlenecks. Due to a lack of time and for reasons documented below, not all of the hypercalls were extensively benchmarked. In addition, we were unable to perform benchmarks related to the network stack, since there is currently a bug that makes the network stack unusable on real hardware (although it works in QEMU). However, since the network stack uses very few hypercalls beyond those performed at the network card driver level, we do not expect its speed to differ significantly from the speed of implementations on other platforms.

5.1 Storage Overhead of PCI Driver Wrappers

Let us commence by analyzing the difference in binary size between native Barrelfish drivers and equivalent ones provided by the rump kernel. Since drivers using a rump kernel must contain a non-negligible portion of the NetBSD kernel within their binaries due to the fact that Barrelfish is statically linked, almost any driver using a rump kernel will be bigger than an equivalent native Barrelfish one. The precise amount of NetBSD kernel code required to be included in the binary depends on how many and which components are required to allow the driver to run and be used.

As an example, we consider the driver for the Intel e1000 network card, since there is a native Barrelfish driver available for it and it is also a large, complex driver that requires several rump kernel components to be able to be run and used properly. The rump kernel-based version of this driver must contain components necessary for the driver itself and for the network stack[1], since there is no easy way to communicate with the driver other than going through the network stack at the system call level. Because each wrapper must include several components, but the linker is able to save space by removing any unnecessary symbols, it cannot be predicted based on the sizes of the individual components how big a wrapper is going to be. For example, the static library for the rump kernel base is over 13 MB (13'125'228 bytes) in size, but the wrapper for the e1000 driver, which contains many more components than just the base, is only 6.99 MB (6'994'576 bytes) in size. For comparison, the native Barrelfish driver is 2.74 MB (2'738'320 bytes). As expected, the rump kernel-based driver is larger, but surprisingly not excessively so, considering the rump kernel driver packs a complete basic network stack in addition the NetBSD e1000 driver, which is significantly more feature-rich than the Barrelfish one. We can conclude that if the number of components linked into a wrapper is kept minimal, rump kernels are typically small enough that the additional size of the wrapper binaries is kept within very reasonable bounds.

5.2 Performance of the `rumpuser` Hypercalls

In Section 4.2, we discussed eight categories of hypercalls for the `rumpuser` library: initialization/termination, memory allocation, files and I/O, clocks, parameter retrieval, randomness, threads, and mutexes. Some of these categories, such as parameter retrieval and randomness, are used very infrequently by the rump kernel and therefore benchmarking and optimizing them has little benefit. Most of the hypercalls are very primitive and are direct wrappers for primitives offered by the host, so benchmarking them would be equivalent to benchmarking the host. Examples of this are the hypercalls for memory allocation (especially `rumpuser_malloc()`, `rumpuser_anonmmap()` is not actually used by any components tested as part of this work), files and I/O (the `posixcompat` functions we use for this part are themselves wrappers around equivalent functions of Barrelfish's VFS layer), and threads. The hypercalls in the mutex category are combinations of wrappers around host mutexes and pre-optimized, host-independent algorithms also used in the POSIX implementation.

It is also worth noting that many of these hypercalls are directly used to implement equivalent functions inside the rump kernel. For example, the `rumpuser_malloc()` call is directly used to implement NetBSD's `kmem_alloc()` (kernel memory allocator) function. Benchmarking such rump kernel functions is therefore equivalent the benchmarking the hypercalls, which in turn is equivalent to benchmarking host primitives. The only hypercalls which do not fall under this category are the ones for initialization and for file I/O. Therefore, it is interesting to see how much overhead is incurred by using

[1]List of factions and components: `rump`, `rumpuser`, `rumpnet`, `rumpdev`, `rumpvfs`, `rumpnet_net`, `rumpnet_netinet`, `rumpnet_netinet6`, `rumpnet_config`, `rumpdev_bpf`, `rumpdev_pci`, `rumpdev_pci_hyper`, `rumpdev_miiphy`, `rumpdev_pci_if_wm`

components that use these hypercalls compared to doing the same thing on Barrelfish.

5.2.1 Initialization

To test how much overhead is incurred by an application initializing the rump kernel in its address space, the execution time of `rump_init()` was tested with various different configurations using different factions and components. Figure 5.1 shows the results.

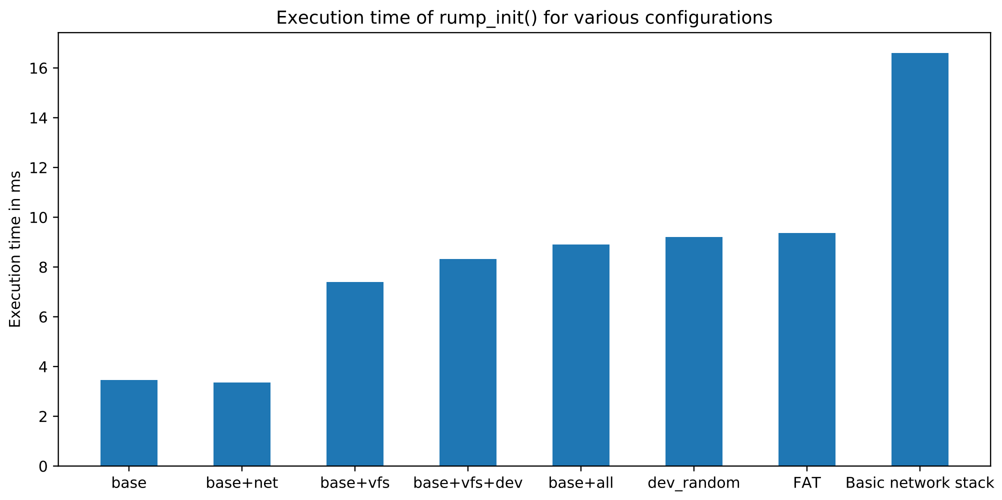

Figure 5.1: Execution times of `rump_init()` using several different configurations

Here, the base configuration is the rump kernel base without anything else linked in, the base+net configuration includes the `net` faction, the base+vfs configuration includes the `vfs` faction, the base+vfs+dev faction includes the `dev` and `vfs` factions (there seems to be a bug in the rump kernel that causes the `dev` faction to require the `vfs` faction), the base+all configuration includes all the factions, the dev_random configuration includes the `vfs` and `dev` functions as well as a component for the Unix `/dev/random` driver, the FAT configuration contains the factions and components necessary for supporting the FAT filesystem (base, `vfs`, `dev` and two other components `rumpdev_disk` and `rumpfs_msdos`), and the basic network stack configuration contains all the components necessary for a network stack that supports TCP/IPv4 and IPv6 (base, `net`, and four other components `rumpnet_net`, `rumpnet_netinet`, `rumpnet_netinet6`, and `rumpnet_config`).

As expected, the more components we add into a rump kernel, the longer it takes to initialize them. It is particularly noteworthy that initializing the `vfs` faction seems to take much more time than the other two factions, and initializing various network protocols also takes a lot of time compared to initializing basic drivers for `dev_random` and the FAT filesystem.

5.2.2 File I/O

It was previously mentioned that file I/O does not directly use the relevant hypercalls. This is because the rump kernel implements its own VFS layer, so performing operations on a file on the host requires going through the VFS layer of the rump kernel as well as the VFS layer of the host. Table 5.1 and Figure 5.2 show how much overhead is incurred by going through two VFS layers.

Table 5.1: Execution times for opening files

	Barrelfish VFS	Rump kernel VFS	Both VFS layers
time for `open()` (μs)	10	18	212

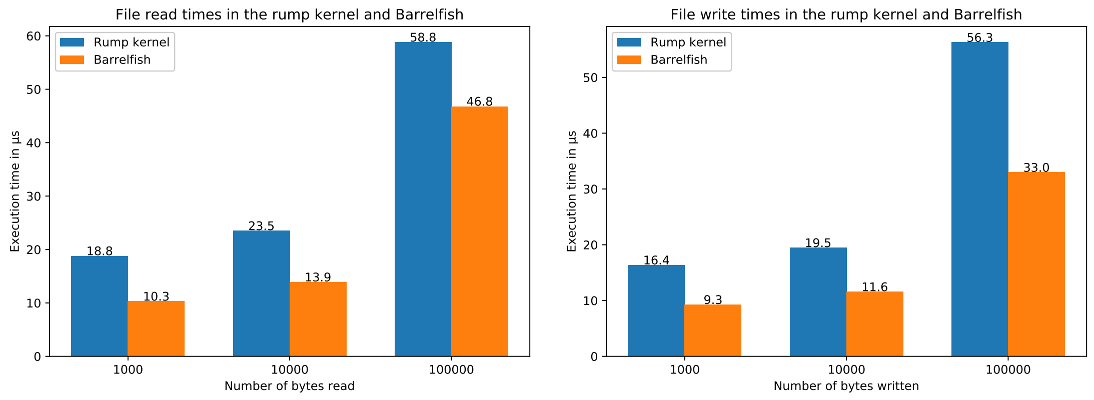

Figure 5.2: Execution times of file read/write operations on rump kernel vs. Barrelfish

We can see that for opening files, remaining within only one VFS layer (of either Barrelfish or the rump kernel) incurs similar overheads, although this must be taken with a grain of salt because the underlying file systems are not the same in each case (on Barrelfish, we open a real file, whereas we open the `/dev/random` pseudo-device on the rump kernel). However, going through both VFS layers incurs a large amount of overhead, far more than simply the sum of the overheads of both VFS layers. This may be due to the fact that the rump kernel is using a special file system[2] to access files on the host which incurs its own overhead.

For reading and writing files, we refrain from using only the rump kernel VFS layer because, as mentioned earlier, it cannot directly be compared to the Barrelfish VFS layer. We see that for reading host files from the rump kernel, there is a fairly constant overhead corresponding to the time necessary to traverse the rump kernel VFS layer. For writing host files, the overhead depends on the amount of data written. This could be an indication that the rump kernel VFS layer performs its own copying of the data before executing the hypercall to the Barrelfish VFS layer.

5.2.3 Conclusions

We are able to conclude that rump kernel initialization is negligible for most applications (less than 50 ms), meaning that users should not see a significant difference in launch time between applications that use a rump kernel and applications that do not. We can also see that applications that execute components that stay within the rump kernel or only use hypercalls (such as `rumpuser_malloc()`) that directly implement NetBSD kernel routines should not incur significant overhead. However, attempting to control

[2]The Extra-Terrestrial File System (ETFS), see rump_etfs(3)

host resources such as files from a rump kernel incurs some overhead and should be avoided if possible.

5.3 Performance of the PCI Hypercalls

Using host files from a rump kernel incurs some overhead, but can the same be said about host devices? This is clearly not the case, as our implementation discussed in Section 4.3 allows the rump kernel access devices directly, almost completely bypassing the host. Only interrupts must be routed through the host kernel to the rump kernel. In the following subsections, we review and assess all possible sources of additional overheads.

5.3.1 Initialization

One area that may incur overhead is initialization of PCI devices drivers. In Subsection 2.4.7, we discussed that the rump kernel performs a bus enumeration to discover devices and start drivers. Figure 5.3 shows how long it takes to initialize various rump kernels containing support for PCI and other components.

In Figure 5.3, the vfs+dev+pci configuration describes a configuration with the rump kernel base, `vfs` and `dev` factions, and the PCI component. The pci+basic network stack configuration is the same, but with the same basic network stack as in Subsection 5.2.1, and the e1000 configuration adds a component for a driver for the Intel e1000 network card. We can see from the execution time for the vfs+dev+pci configuration that the PCI bus enumeration does not add a significant amount of overhead compared to just using the `vfs` and `dev` factions. Adding the network stack adds the expected amount of overhead, but adding the e1000 driver adds a significant amount of overhead. This is most likely due to the fact that the e1000 is very complex and initializing it requires accessing the device registers several times as well as establishing an interrupt routine.

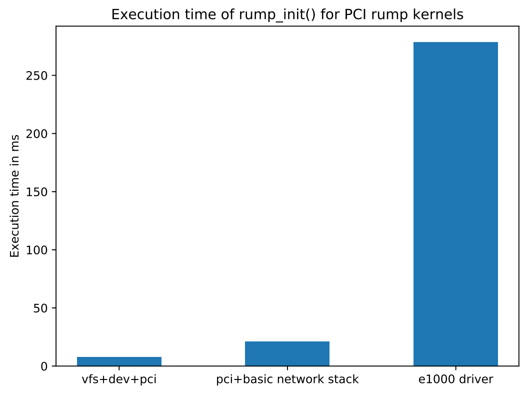

Figure 5.3: Execution times of `rump_init()` for rump kernels using the PCI component

5.3.2 Direct memory access

Subsection 4.3.4 discussed hypercalls for allocating, mapping, and freeing DMA memory as well as translating virtual addresses to physical addresses. The first three hypercalls directly use Barrelfish's facilities and are typically only used when the driver is initializing, and it therefore makes little sense to test them. The latter is used very frequently, and Kent McLeod mentions in his bachelor's thesis [17] that optimizing this hypercall in the seL4 implementation of the rump kernel led to significant improvements in efficiency of the network stack. The Barrelfish implementation has the unfortunate circumstance that due to a bug in the pmap `lookup()` function, we have to use the workaround presented in Listing 4.7. To evaluate the efficiency of this hypercall, we compare three approaches: (1) the initial approach using the pmap `lookup()` function that could not be used due to the bug (referred to as the optimal approach), (2) the approach presented in Listing 4.7 (without caching, referred to as the naive approach), and (3) the same approach with caching (referred to as the cached approach).

Table 5.2 shows the average execution time for a single lookup. For the cached approach, we consider the execution time on a cache miss and on a cache hit.

Table 5.2: Execution times for virtual address translation using various approaches

	optimal	naive	cached (cache miss)	cached (cache hit)
Execution time (μs)	0.64	100.90	102.14	0.48

The optimal approach requires a single system call and a call to the `lookup()` function, which is comparable to the cached approach with a cache hit, which performs a single system call and a cache lookup. On a cache miss, the cached approach is similar to the naive approach (in fact, slightly worse because it must also perform a cache lookup). Both the cached (cache miss) and naive approaches are much worse than the optimal approach. The hope is that the cached method will have minimal cache misses, making it comparable to the optimal approach. We validate this by measuring the average execution time for several translations. The data is generated by `malloc`'ing several addresses and performing address translation on all of them, similarly to what happens when using the network stack with the e1000 driver.

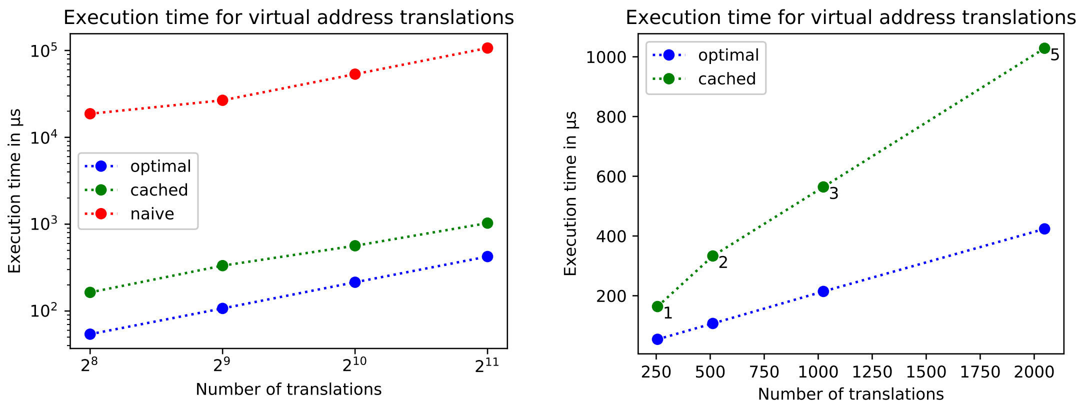

Figure 5.4: Execution times for several translations. The plot on the left shows the cached and naive approaches compared to the naive approach. The plot on the right compares the cached and optimal approaches. The numbers next to the cached values represent the size of the cache.

Figure 5.4 shows that, as expected, the naive approach is several orders of magnitude slower than the cached and optimal approaches. The execution time of the cached approach (with a mix of cache hits and cache misses) is slightly slower than the optimal approach, but kept within reasonable bounds. The size of the cache is equal to the number of cache misses, so we can see that the number of cache misses for most scenarios is kept fairly small even for large amounts of translated addresses, as stated in Subsection 4.3.4. Besides cache misses, a reason why the cached approach is still slower than the optimal approach is that the cache must be safe against concurrent use by multiple rump kernel LWPs, so it must be protected with a mutex. One optimization would be to use reader-writer locks to protect the cache, since it is very seldom updated. However, there was no time to test this idea.

5.3.3 Interrupts

Especially for interrupt-intensive devices such as network cards, it is important that the interrupt handling latency be kept small to increase performance. A potential source of overhead is that interrupts must be routed from the host kernel to the PCI hypercall library and then scheduled onto a rump virtual CPU. We therefore measure the time between the instant the interrupt arrives from the host kernel at the PCI hypercall library and the instant the rump kernel interrupt handler is called. There was not much time to take a large amount of measurements, but the measurements that were taken (of which some examples are shown in Table 5.3) demonstrate that this overhead is typically around 5 ms, but could be larger, presumably due to either the interrupt handler thread not being scheduled in a timely fashion or the fact that no virtual CPU was available. One way to attempt to reduce this would be to provide the rump kernel with more virtual CPUs, but due to time constraints, this was not tested.

Table 5.3: Some samples of measured interrupt latencies

Times in ms											
5.83	11.97	5.63	5.60	5.61	5.62	27.47	12.00	5.63	5.60	5.60	5.64

Chapter 6

Limitations and Future Work

We have shown a working and complete implementation of the rump kernel on Barrelfish. However, several limitations exist and much could be improved and extended. In this chapter, we will review the implementation and demonstrate these limitations as well as discuss possible extensions.

More in-depth performance evaluation Due to a lack of time, we have only performed very basic benchmarks of the rump kernel. To gain a deeper understanding of the efficiency issues, it would be necessary to perform more in-depth benchmarks. This would allow identifying bottlenecks and fine-tuning the hypercalls to avoid them as much as possible. In fact, a more in-depth performance evaluation may even reveal that the approach we have taken to follow the POSIX implementation as closely as possible when implementing the `rumpuser` hypercalls (for the sake of correctness) is a huge performance drain on Barrelfish. Moreover, a detailed performance evaluation would provide a more in-depth understanding of how what could be considered a POSIX compatibility layer performs on Barrelfish.

Fix bugs with RPCs There is currently an unresolved bug where using RPCs to communicate with a rump kernel through system calls sometimes causes interrupts to completely stop arriving. Since using RPCs to communicate with rump kernels is an important aspect of being able to use NetBSD OS components on Barrelfish, fixing this would be an important part of future work and is a major limitation of the current implementation.

Fix bugs on real hardware There is another unresolved bug related to using network drivers and the network stack while using real hardware (it works fine on QEMU). When trying to communicate over a network through a gateway, the bug causes the ARP reply from the gateway to not be recognized by the network stack, although the driver receives the packet, which means that no packets can be sent. MSI-X interrupts also currently do not work on real hardware, although this is likely a limitation of the Barrelfish MSI-X service and not specific to the rump kernel. Also, more testing on real hardware would be beneficial, since there was not much time to do this.

Testing of other drivers and OS components The work done on the rump kernel in this thesis placed a heavy focus on network card drivers. While those now seem to be working properly, more testing of other types of PCI drivers is needed in order to verify that the PCI hypercall implementation as well as the current architecture of PCI driver wrappers work correctly for their intended purposes. Testing other types of OS components would also be a good test of the `rumpuser` hypercall interface. Due to time constraints and since there are a very large variety of different drivers and OS components available for NetBSD, testing a multitude of different drivers and OS components was not feasible within the time allotted for this project.

Decouple network stack from network card drivers The rump kernel provides a mechanism (the `virtif` interface) to use the NetBSD network stack, but actually send out packets over the network using the host's facilities. Currently, it is impossible to use the NetBSD network stack without also using a NetBSD network card driver. Implementing the `virtif` interface would therefore allow decoupling the network stack from NetBSD network card drivers and using it as a drop-in replacement for the current Barrelfish network stack, which would be an interesting feature to have on Barrelfish, especially considering that the NetBSD network stack is more feature-complete, more heavily tested, and more heavily fine-tuned and optimized than the current lwIP-based network stack.

Decouple network card drivers from network stack The inverse decoupling, i.e. decoupling NetBSD network card drivers from the network stack, would also be interesting, since it would allow using the native Barrelfish network stack together with NetBSD drivers instead of having to use the Barrelfish network stack for Barrelfish drivers and NetBSD network stack for NetBSD drivers. Unfortunately, this is not as easy as decoupling the network stack from network drivers. Since NetBSD network card drivers rely on certain link-layer functions in the NetBSD network stack, doing this would involve emulating those link-layer functions while also presenting an interface compatible with the Barrelfish network stack.

Add more system calls to Flounder interface As mentioned in Subsection 4.4.2, only certain interesting system calls related to file I/O and BSD sockets were implemented using the Flounder-based RPC mechanism. Interacting with arbitrary NetBSD OS components would require implementing all or most of the NetBSD system calls as RPCs.

Support for multiple drivers per rump kernel Another feature that would be useful to have is the ability to run multiple PCI drivers within a single rump kernel. This would be more efficient compared to running a separate rump kernel for each driver, although it does come with the drawback that a crash of the rump kernel would affect all the drivers running in it. This is currently not possible because Kaluga only hands us capabilities for one device and always starts one driver domain per device. This would be particularly useful in case there are multiple copies of the same device installed in a system. For example, using a rump kernel-based driver on many of the ETH Zürich Systems Group's machines currently results in an error, since they contain multiple e1000 network cards and the rump kernel-based driver does not support multiple instances of the driver module.

Support scatter-gather DMA The current version of the PCI hypercall interface does not support scatter-gather DMA. This means that devices and device drivers that rely on this function cannot be used on the rump kernel. It is therefore in the interest of compatibility with a broad range of PCI devices and device drivers to extend the PCI hypercall library to support this.

Support devices using I/O BARs It is currently not possible to use PCI devices whose BARs only point to I/O space. However, this is not a limitation of the rump kernel or PCI hypercall interface but a limitation of Barrelfish's PCI support. There is currently no method to get capabilities for specific I/O ports in Barrelfish and Kaluga does not pass drivers capabilities to I/O port BARs. This is especially important for supporting legacy devices that do not support memory-mapped I/O.

Port Rumprun to Barrelfish Another interesting item of future work would be to port the Rumprun framework [13] to Barrelfish, which would allow running unmodified POSIX applications on top of a rump kernel running on Barrelfish. This is especially interesting considering the wide availability of POSIX software as opposed to Barrelfish software and since the main building block (the kernel itself) is already ported. Furthermore, it would interesting to compare this approach with other approaches currently available on Barrelfish, e.g. Graphene [32].

Support for multi-dispatcher rump kernels The rump kernel is currently limited to running on a single core and cannot take advantage of parallelism to speed up tasks. Modifying the implementation to run within a single domain but over several dispatchers would therefore be an interesting extension potentially leading to large increases in efficiency of the rump kernel.

Chapter 7

Conclusion

The goal of this thesis, as presented in the introduction, was to port the NetBSD rump kernel to Barrelfish in order to support running a broad range of NetBSD OS components on Barrelfish which can be integrated into the Barrelfish ecosystem. We saw how this goal can be achieved by implementing two sets of hypercalls: the `rumpuser` hypercalls, which provide the main support for the rump kernel, and the PCI hypercalls, which provide support specifically for running NetBSD PCI drivers on Barrelfish.

The `rumpuser` implementation was heavily inspired by the equivalent implementation for POSIX platforms in order to provide some measure of insurance that the rump kernel operates as expected on Barrelfish. We illustrated how using Barrelfish's `posixcompat` library enabled us to reuse large portions of the POSIX implementation, sometimes even without modification. Where needed, implementations were also possible using functions and libraries equivalent to POSIX libraries.

The PCI hypercall implementation was made particularly easy by the fact that all drivers in Barrelfish run in userspace. By using the capabilities provided to us by Kaluga and the functions provided to all PCI drivers by various Barrelfish services, we were easily able to implement all the required features and were even able to extend the hypercall library to provide support for MSI-X interrupts.

We also saw how regular Barrelfish applications could interact with rump kernel components. We discussed a client/server, Flounder RPC-based interface which provides Barrelfish applications with easy access to the NetBSD system call interface, which should in theory grant them the same abilities as applications running on a regular NetBSD system. Furthermore, we mentioned how wrappers could be used to encapsulate NetBSD OS components inside regular Barrelfish applications, and demonstrated which special features were necessary in the case of wrappers for PCI device drivers to have them started automatically by Kaluga.

Finally, we saw how the current implementation has some bugs and has the potential to become even more powerful if additional work is invested in implementing new features.

However, all in all, we would say that the goal of this thesis has been met and a solid foundation for using NetBSD OS components on Barrelfish has been laid.

References

[1] About the NetBSD Project. `https://www.netbsd.org/about/`. Accessed: 2019-08-15.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 95–109, Pacific Grove, California, USA, 1991. ACM.

[3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, Bolton Landing, NY, USA, 2003. ACM.

[4] Barrelfish Project. Barrelfish Architecture Overview. Barrelfish Technical Note 000, Systems Group, ETH Zürich, 2013.

[5] Barrelfish project. Device Drivers in Barrelfish. Barrelfish Technical Note 019, Systems Group, ETH Zürich, 2017.

[6] Andrew Baumann. Inter-dispatcher communication in Barrelfish. Barrelfish Technical Note 011, Systems Group, ETH Zürich, 2011.

[7] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, Big Sky, Montana, USA, 2009. ACM.

[8] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. Why isn't your OS? In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems*, Monte Verita, Switzerland, May 2009.

[9] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, Copper Mountain, Colorado, USA, 1995. ACM.

[10] Info: Who Uses Rump Kernels. `https://github.com/rumpkernel/wiki/wiki/Info:-Who-Uses-Rump-Kernels`, 2016. Accessed: 2019-08-15.

[11] Intel Corporation. Intel® 82574 GbE Controller Family Datasheet (Revision 3.4), June 2014.

[12] Antti Kantee. *The Design and Implementation of the Anykernel and Rump Kernels*. Second edition, 2016.

[13] Antti Kantee. Rumprun. `https://github.com/rumpkernel/rumprun`, 2018. Accessed: 2019-08-13.

[14] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, Big Sky, Montana, USA, 2009. ACM.

[15] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[16] Jochen Liedtke. Improving IPC by Kernel Design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, Asheville, North Carolina, USA, 1993. ACM.

[17] Kent McLeod. Usermode OS Components on seL4 with Rump Kernels. Bachelor's thesis, School of Electrical Engineering and Telecommunication, University of New South Wales, 2016.

[18] Microsoft Corporation. Windows Subsystem for Linux Documentation. `https://docs.microsoft.com/en-us/windows/wsl/about`, 2016. Accessed: 2019-09-02.

[19] Microsoft Corporation. Introduction to Hyper-V on Windows 10. `https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/`, 2018. Accessed: 2019-09-02.

[20] Microsoft Corporation. Introduction to NDIS 6.0. `https://docs.microsoft.com/en-us/windows-hardware/drivers/network/introduction-to-ndis-6-0`, 2019. Accessed: 2019-08-13.

[21] NDISulator. `https://github.com/NDISulator/ndisulator`, 2015. Accessed: 2019-08-13.

[22] ndiswrapper. `http://ndiswrapper.sourceforge.net/wiki/index.php/Main_Page`, 2010. Accessed: 2019-08-13.

[23] Oracle Corporation. Oracle VM Virtualbox. `https://www.virtualbox.org`. Accessed: 2019-08-13.

[24] PCI-SIG. PCI Local Bus Specification (Revision 3.0), 2002.

[25] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. *SIGARCH Comput. Archit. News*, 39(1):291–304, March 2011.

[26] Project UDI. `http://udi.certek.com`, 2010. Accessed: 2019-08-14.

[27] Octavian Purdilla, Lucian Adrian Grijincu, and Nicolae Tapus. LKL: The Linux Kernel Library. In *Proceedings of the 9th RoEduNet IEEE International Conference*. IEEE, 2010.

[28] Luigi Rizzo. Building Linux Device Drivers on FreeBSD. `http://www.iet.unipi.it/~a007834/FreeBSD/linux_bsd_kld.html`, 2008. Accessed: 2019-08-13.

[29] Timothy Roscoe. Hake. Barrelfish Technical Note 003, Systems Group, ETH Zürich, 2017.

[30] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic Device Driver Synthesis with Termite. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 73–86, Big Sky, Montana, USA, 2009. ACM.

[31] Akhilesh Singhania, Ihor Kuz, Mark Nevill, and Simon Gerber. Capability Management in Barrelfish. Barrelfish Technical Note 013, Systems Group, ETH Zürich, 2017.

[32] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 9:1–9:14, Amsterdam, The Netherlands, 2014. ACM.

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Using NetBSD Kernel Components on Barrelfish Through Rump Kernels

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Horné	Leo

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, September 4, 2019	[illegible]

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.