Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN INFORMATICS

PCIe and DMA in MirageOS

Fabian Bonk

Technical University of Munich

Department of Informatics

Master's Thesis in Informatics

PCIe and DMA in MirageOS

PCIe und DMA in MirageOS

Author:	Fabian Bonk
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Paul Emmerich
Date:	June 15, 2020

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, June 15, 2020

Location, Date

Signature

Abstract

MirageOS is a unikernel operating system targeting hypervisors. MirageOS generates standard executables for Unix-like operating systems as well as standalone virtual machine images for hypervisors. One of the hypervisors MirageOS targets is Linux's KVM which is supported through the Solo5 unikernel execution environment.

In this thesis we have enabled MirageOS unikernels to communicate with PCIe devices such as network cards by creating a framework for MirageOS targeting Linux as well as Solo5.

The framework extends two of MirageOS' target platforms and allows unikernels to request PCIe devices from their host. Unikernels that include hardware drivers can take control of these devices and communicate with them directly via DMA instead of going through the host system.

On the Solo5 unikernel execution environment we mapped PCIe device regions and DMA-ready memory directly into the unikernel's virtual machine, whereas for MirageOS' Unix backend we created a library for setting up such mappings directly from the unikernel's process. In both cases we made use of Linux's VFIO subsystem for accessing PCIe devices from userspace.

We have also ported the existing userspace network driver *ixy.ml* which is written in OCaml to our new framework. By wrapping this driver in a MirageOS-compatible network interface, we were able to move the driver into the unikernel and improve networking performance significantly compared to previous MirageOS deployments relying on TAP-networking via the unikernel's host. We measured 3.25 Gbit/s of TCP throughput; a twelve-fold increase on MirageOS' Unix backend and a two-fold increase on MirageOS' Solo5 backend. Through this wrapper ixy.ml can be used with any existing MirageOS program requiring a networking device.

Contents

List of Figures

Chapter 1

Introduction

1.1 Outline

Stripping away unnecessary code is a good way of reducing a program's attack surface. Fewer lines of code usually correlates with fewer bugs. Fewer bugs lead to fewer externally exploitable security flaws (aka remote holes). However, most programs require an operating system to run on top of. The operating system provides necessary functionality, such as drivers, file systems, and networking.

The idea of library operating systems (like MirageOS) is to strip away any unnecessary operating system code. A static web server, for example, likely doesn't need an audio driver or a file system. Library operating systems move OS components into standalone libraries that can be linked into the application at compile-time [6] [12].

MirageOS goes one step further by rewriting OS components in OCaml, a memory-safe, statically typed, functional, and compiled programming language. MirageOS applications are also written in OCaml and use MirageOS' components as libraries. These applications are deployed as virtual machines on hypervisors like Xen or Linux's KVM. Such applications are known as unikernels [12] [11].

Unfortunately, by recreating components from scratch, MirageOS also throws out all actually required functionality that is not yet implemented in OCaml, including hardware drivers. To communicate with the outside world, MirageOS defines a set of interfaces for block devices, network devices, etc. The hypervisors, on which MirageOS programs run, implement these interfaces using their own drivers. These drivers are not written in OCaml but rather in the unsafe C programming language, thereby posing a security risk [3].

1.2 Goal of the thesis

We implement a framework for both MirageOS and Solo5's hvt, one of the hypervisors MirageOS targets, that enables programmers to create their own drivers from scratch in OCaml. Thereby, we are eschewing the unsafe C drivers of yesteryear, further reducing application deployments' attack surface.

Our framework enables MirageOS unikernels to take control of PCIe devices connected to the host computer. MirageOS can communicate with a PCIe device through the device's registers and via main memory by way of Direct Memory Access (DMA). As driver bugs or misconfigurations might compromise an application's security, our framework limits a device's view of main memory to only specific areas through the CPU's built-in IOMMU. Incidentally, this also eases driver programming significantly by translating memory accesses from both unikernel and PCIe device to the same physical memory addresses.

We also port our network driver *ixy.ml* [7] to the new framework to prove the framework's functionality. ixy.ml is a network driver for the *ixgbe* family of 10GbE network cards written in OCaml. Currently, it only exists as an OCaml library running as a userspace driver on Linux. Users of ixy.ml have to write network applications targeting its specific programming interface. With our port, any MirageOS application may use ixy.ml (and the network card controlled by ixy.ml) as a network device.

Chapter 2

MirageOS

MirageOS' website states that "MirageOS is a library operating system that constructs unikernels." [25]. Its structure is fundamentally different from most modern operating systems. There is no concept of multiple users, multiple processes, etc. Features typically supplied by an operating system such as networking and file systems are instead provided by MirageOS' libraries. MirageOS applications are single OCaml programs embedded into a statically linked, bootable virtual machine image. Such an image is called a unikernel [11] [12].

2.1 Compilation Targets

MirageOS targets a wide variety of host operating systems and hypervisors. It can generate unikernels for the Xen hypervisor or the Solo5 unikernel runtime [27]. It can also generate standard executables for Unix-like operating systems such as Linux, FreeBSD or macOS. Developers can easily test their applications on their local machines; once the code is ready for deployment, it simply needs to be recompiled for the production target.

2.2 MirageOS programming

Users write MirageOS applications in the OCaml programming language. A MirageOS application differs from typical OCaml programs in that it does not contain a normal entrypoint. The equivalent of an entrypoint is a functor (called `Main` by convention) supplied by the programmer. OCaml functors are explained in Section 2.2.1. In MirageOS'

```
module type Predicate = sig
  type t
  val predicate : t -> bool
end

module Quantifiers (X : Predicate) = struct
  let rec for_all = function
    | [] -> true
    | x :: xs -> X.predicate x && for_all xs

  let rec exists = function
    | [] -> false
    | x :: xs -> X.predicate x || exists xs
end
```

Figure 2.1: Example functor

case the modules given to the `Main` functor contain the operating system functionality provided by MirageOS' libraries.

2.2.1 OCaml functors

A functor in OCaml differs from functors in other languages: it can be thought of as a function mapping an OCaml module onto a new module or functor. A module is a collection of OCaml types, values, nested modules, and module types. A module type assigns a name to a module signature; a module signature describes a module's contents.

Consider the (admittedly very contrived) example in Figure 2.1. It defines a module type `Predicate` and a functor `Quantifiers`. Modules satisfying `Predicate`'s signature define a type `t` as well as a predicate on `t`. When applied to such a module, `Quantifiers` generates a new module containing implementations of universal ($\forall$) and existential ($\exists$) quantification for lists of `t`.

2.2.2 MirageOS functors

Suppose a MirageOS program requires an IPv4 network stack. In this case the program's `Main` functor would be parameterized over a module of type `Mirage_stack.V4` — that is, a module containing UDP/TCP send and receive functionality.

Programming in this way makes switching implementations easy. Depending on the target platform, MirageOS supplies an appropriate module implementing an IPv4 network tack that satisfies the `Mirage_stack.V4` signature — that is, a library calling the appropriate platform-specific functions for sending and receiving data.

```
open Lwt.Infix

module Main (S : Mirage_stack.V4) = struct
  let rec echo flow =
    S.TCPV4.read flow >>= function
    | Error _
    | Ok `Eof -> S.TCPV4.close flow
    | Ok (`Data buf) ->
      S.TCPV4.write flow buf >>= function
      | Error _ -> S.TCPV4.close flow
      | Ok () -> echo flow

  let start s =
    S.listen_tcpv4 s ~port:8080 echo;
    S.listen s
end
```

Figure 2.2: MirageOS echo server

Consider the example in Figure 2.2. It defines a functor `Main` that requires a module fulfilling the signature `Mirage_stack.V4`. When applied to such a module, `Main` generates a new module containing a `start` function. This function will be called after MirageOS' platform-specific initialization code has run. In this example, `start` installs a listener on TCP port 8080 that calls `echo` when receiving a new connection.

The `echo` function tries to read data from the TCP connection (called `flow` in MirageOS) and sends it back. When there is a connection error or the connection is closed by the other side, `echo` also closes the connection. Of note is the required explicit error handling enforced by MirageOS' libraries and OCaml's type system.

To build a unikernel from the source code in Figure 2.2, the programmer must also provide a configuration. The configuration tells MirageOS which modules to apply in which order to which functor. Additionally it may take care of runtime argument parsing (this is beyond the scope of this example). The configuration is also written in OCaml, but it is only executed at compile time. MirageOS uses a metaprogramming library called Functoria [15] to generate appropriate functor applications from programmer-provided configurations.

The example configuration in Figure 2.3 lets MirageOS decide on an appropriate network stack (`generic_stackv4 default_network`).

Lastly the user needs to compile the unikernel by selecting a target platform and optionally selecting details, such as which network stack on the target platform to use.

```
open Mirage

let main = foreign "Unikernel.Main" (stackv4 @-> job)

let stack = generic_stackv4 default_network

let () =
  register "echo" [
    main $ stack
  ]
```

Figure 2.3: Example echo server configuration

```
$ mirage configure -t macosx --net socket
$ make depend
$ make
```

Figure 2.4: Example `mirage` invocation targeting macOS

They do this by invoking the `mirage` [16] utility. The example invocation in Figure 2.4 tells the `mirage` utility to target the macOS operating system (by generating a standard Mach-O executable) and to use macOS' network stack (via the POSIX socket API) instead of a MirageOS-provided one. `mirage` generates a Makefile that installs the appropriate system and opam packages (`make depend`) and builds the unikernel (`make`).

2.3 Why unikernels?

The MirageOS authors cite both security and efficiency as reasons to consider unikernels [12].

2.3.1 Security

Unikernels generally contain only the code necessary for implementing their functionality. A typical Linux server deployment contains millions of lines of code that are not strictly necessary for a job such as serving static webpages. This unnecessary code likely still contains security flaws that may be exploited even though the code is not contributing to the deployment's functionality.

OCaml is a type-safe and memory-safe language. Its compiler statically checks all types at compile-time and its runtime dynamically checks memory accesses. Thus many common programming mistakes can be prevented.

Since the unikernel is statically linked and there is absolutely no dynamic loading of code, code injection attacks can be prevented entirely.

2.3.2 Efficiency

In cloud environments applications are oftentimes deployed using virtual machines. It is not uncommon to deploy an entire virtual machine running an operating system like Linux for a single server process. This type of deployment has a lot of memory and processor overhead due to each virtual machine requiring an entire operating system. Some techniques like same-page merging might lessen this effect, at the cost of increased CPU usage.

Unikernel deployments on the other hand only contain the application code and its dependencies. Deploying a unikernel is much more efficient than an equivalent deployment using virtual machines [12].

Another popular way of deploying applications is OS-level virtualization, an approach commonly known as containerization. OS-level virtualization runs an application as a standard process on the same OS kernel as the deployment's host but with a different userspace. MirageOS' spt target (which is part of Solo5) uses OS-level virtualization to deploy a unikernel as an isolated process instead of in a virtual machine [28].

2.4 Drivers in MirageOS

MirageOS provides no drivers for hardware devices; all hardware communication must be done by the host system. MirageOS itself communicates with the host system via system-specific stubs. Our ixy.ml userspace driver for Intel ixgbe network cards has been ported to MirageOS' Linux target. We have additionally ported ixy.ml to our newly created framework as detailed in 5.

2.5 MirageOS structure

MirageOS' protocol implementations are structured using explicit layering. A protocol is implemented as a functor mapping a lower-level protocol implementation to a higher level protocol implementation. For example, assembling a MirageOS network stack is done by sequentially applying functors to some base implementation of a network device. Strict interfaces, that the modules need to adhere to, guarantee interchangeability of components.

Suppose a unikernel requires a network device for communication and a block device for storage. It declares dependencies on `Mirage_net.S` and `Mirage_block.S`, but does not specify implementations for these signatures. Since all MirageOS network devices satisfy the `Mirage_net.S` module signature and all MirageOS block devices satisfy the

`Mirage_block.S` module signature, at compile time any suitable network and block device can be plugged into the unikernel.

Figure 2.5 shows the structure of such a unikernel using MirageOS' Unix backend on Linux. The Unix backend runs unikernels as standard processes without any special isolation. Components drawn in gray are part of Linux, and those drawn in blue are part of MirageOS.

The user-supplied application contains the `Main` functor and all the actual application code.

MirageOS provides *mirage-net-unix* [21] which connects to a TAP device on Linux and forwards traffic between the TAP device and the unikernel. MirageOS' *mirage-block-unix* [18] similarly wraps Linux's block devices. Both mirage-net-unix and mirage-block-unix provide modules satisfying the signatures `Mirage_net.S` [19] and `Mirage_block.S` [17] respectively.

Usually there exists at least one implementation of each low-level module signature for each MirageOS compilation target, e.g. *mirage-net-solo5* [20] implements `Mirage_net.S` on any of Solo5's backends (see Chapter 3), *mirage-net-xen* [22] implements `Mirage_net.S` on Xen.

Each component of MirageOS' protocol stacks in turn just transforms some lower-level modules to a module conforming to a higher-level module signature, e.g. applying `Ethernet.Make` to a module satisfying `Mirage_net.S` creates a module satisfying `Mirage_protocols.ETHERNET`, the signature of MirageOS Ethernet stacks.

2.5.1 Platform libraries

MirageOS supplies basic platform libraries for each of its target platforms. These libraries contain the unikernel's actual main function on each platform that initiates the unikernel's setup procedure and runs its cooperative multithreading scheduler. Cooperative multithreading in MirageOS is described in more detail in section 2.5.2.

MirageOS platform libraries also contain a sleep function that instructs the unikernel runtime to wake the unikernel once a given sleep timer runs out.

2.5.2 Cooperative multithreading in MirageOS

MirageOS makes use of *Lwt* [13], a cooperative multithreading library written in OCaml. Lwt's basic type is the promise, a type whose value will be determined at some time in the future. A promise that at some point in the future will resolve to a value of type `int` has type `int Lwt.t`.

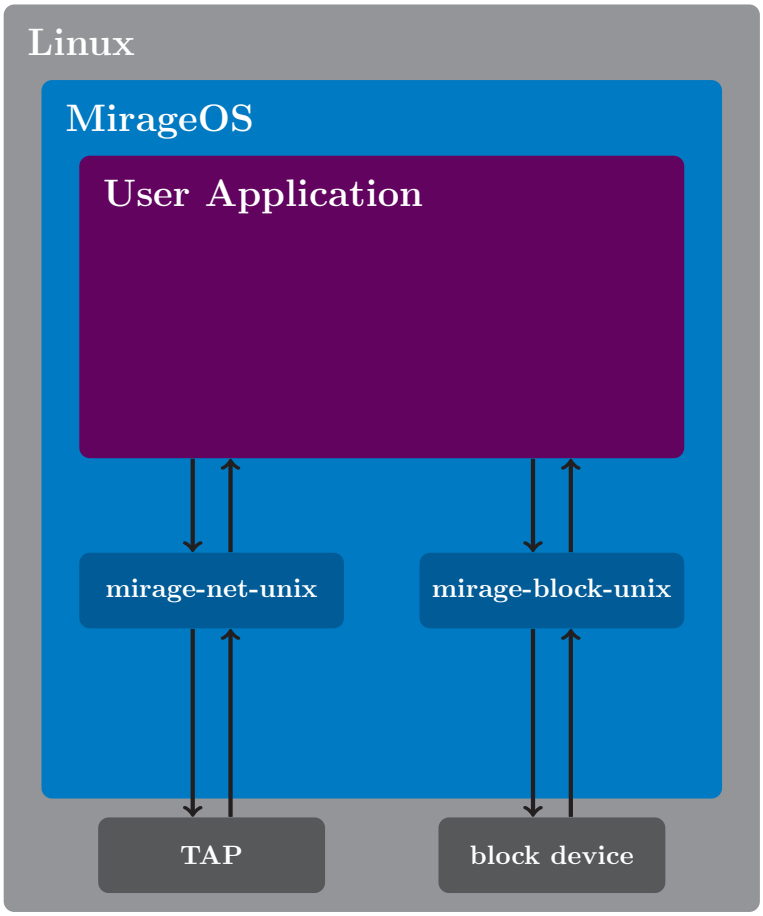

Figure 2.5: Overview of a MirageOS unikernel (containing an application) running on Linux

Lwt provides a monadic interface for chaining promises together, i.e. waiting for some promise to evaluate and then taking its value and creating a new promise from it by applying some function.

Chaining operations is done using a function called `bind` (usually called using the `>>=` operator). Whenever bind is used to chain promises together and one promise resolves to a value, Lwt may interrupt the chained computation and schedule a different promise to be evaluated.

Promises can also voluntarily give up their CPU time and yield to another promise by calling `Lwt.pause`. They are added to a set of paused promises and woken up once there are no other promises ready to run.

MirageOS platform libraries run Lwt's "engine", i.e. a loop that attempts to evaluate promises and wakes up paused promises if there is nothing to evaluate.

Chapter 3

Solo5

Solo5 calls itself a "sandboxed execution environment for unikernels" [27]. Effectively it is an interface between a unikernel and a host operating system's virtualization environment. Each virtualization environment is served by a so-called tender: a small program setting up the unikernel's required environment, loading the unikernel and communicating with it at runtime. There are two tenders: spt (sandboxed process tender) runs unikernels in a seccomp sandbox, whereas hvt (hardware virtualized tender) runs unikernels in a virtual machine.

We will focus only on hvt which interfaces with Linux's, FreeBSD's and OpenBSD's native hypervisors. We will limit ourselves further to Linux's popular KVM hypervisor on the x86_64 architecture.

Solo5 is written mostly in C with some additional assembly as an entrypoint. At the time of writing the hvt tender's bindings that run inside the x86_64 KVM virtual machine consist of around 1340 lines of C code with an additional 360 lines of C code in header files and around 90 lines of assembly, a far cry from the millions of lines of code of a typical OS kernel running inside a VM.

3.1 MirageOS on hvt

Figure 3.1 shows an overview of a MirageOS unikernel deployment on Solo5's hvt. Components drawn in gray are part of Linux, components drawn in green are part of hvt, and those drawn in blue are part of MirageOS. Solid arrows indicate application data transfer; dashed arrows indicate configuration data transfer.

The shown deployment is structurally similar to the one shown in Figure 2.5 from Section 2.5. It also contains a user application that requires access to both a block device and a network device.

3.2 hvt on KVM

hvt's operation can be divided into two phases: setup and runtime. During setup, hvt creates a KVM virtual machine, configures its parameters and loads the unikernel. Once hvt starts the previously created virtual machine, it enters its runtime phase. During this phase hvt facilitates communication between the unikernel and the host.

3.2.1 Setup phase

To begin the setup, hvt parses the manifest; a file typically generated by the unikernel's operating system that states the unikernel's requirements, such as a network device or a storage device for example. It is included in the unikernel's ELF binary and is read by hvt before initializing the unikernel. Figure 3.2 shows an example manifest of a unikernel requesting a network device and a block device, both managed by hvt.

A module handles each device category (currently only networking and storage), e.g. `hvt_module_net.c` handles networking and `hvt_module_block.c` handles block devices.

After the manifest has been parsed, hvt creates a KVM virtual machine with a single virtual CPU and a fixed amount of memory. It then proceeds to load the unikernel (from the ELF binary) and starts the virtual machine. The Manifest is mapped into the unikernel's memory so both hvt's bindings running inside the virtual machine as well as hvt running outside the virtual machine see device configurations.

3.2.2 Runtime phase

During the runtime phase, hvt forwards data between the host and the unikernel. A unikernel requiring network access, for example, communicates with a TAP device on the Linux host through so-called hypercalls. Hypercalls are the hypervisor equivalent of system calls. At the time of writing, hvt supports eight hypercalls; among them are `HVT_HYPERCALL_NET_WRITE` and `HVT_HYPERCALL_NET_READ` for transmitting and receiving network packets, respectively, as well as `HVT_HYPERCALL_BLOCK_READ` and `HVT_HYPERCALL_BLOCK_WRITE` for reading from and writing to block devices, respectively. We have previously explored hvt's existing networking more closely [2].

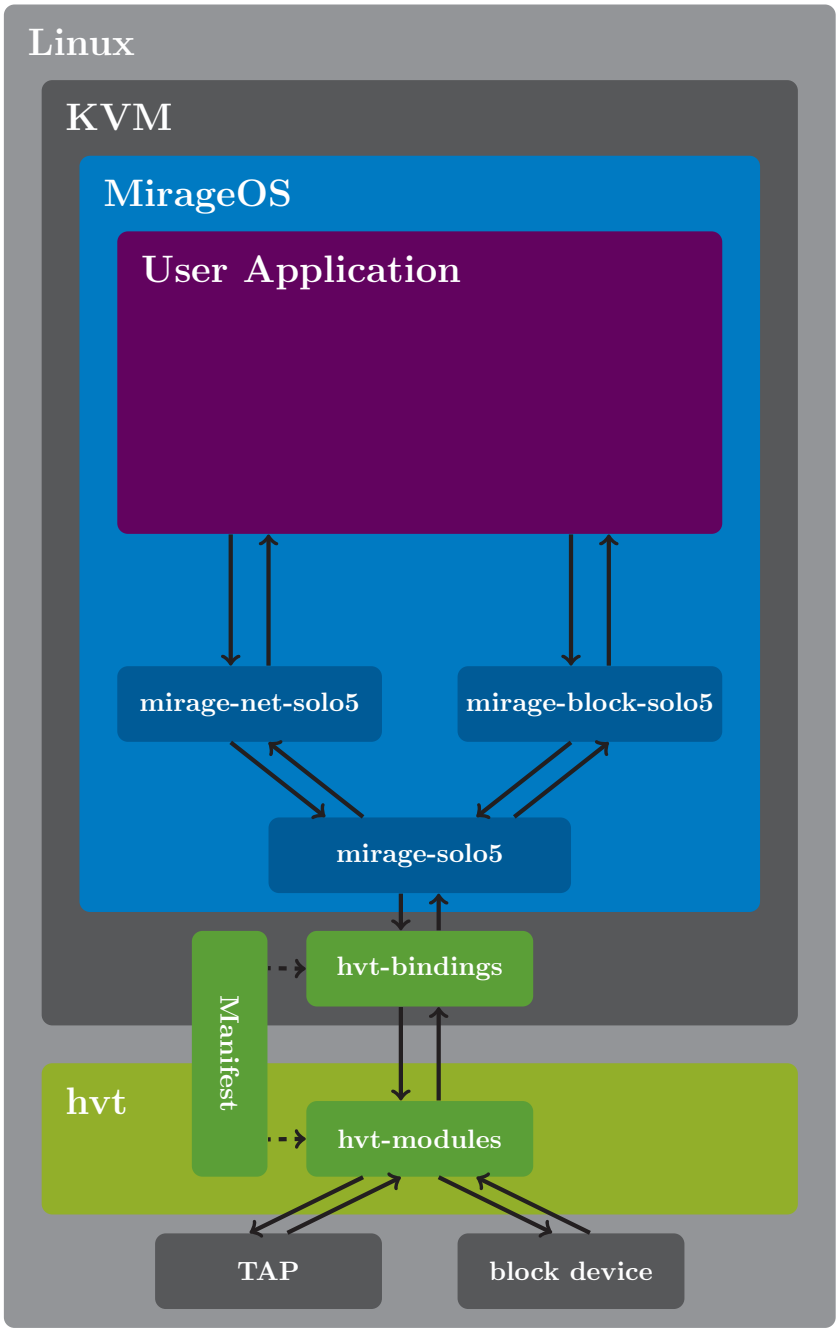

FIGURE 3.1: Overview of a MirageOS unikernel running on Solo5's hvt

```
{
    "type": "solo5.manifest",
    "version": 1,
    "devices": [
        {
            "name": "service0",
            "type": "NET_BASIC"
        },
        {
            "name": "block0",
            "type": "BLOCK_BASIC"
        }
    ]
}
```

Figure 3.2: Example hvt manifest file of a unikernel requesting a network device and a block device

A unikernel targeting Solo5 must call hvt's hypercalls to communicate. MirageOS does this via the *mirage-net-solo5* [20] and *mirage-solo5* [23] libraries: The former implements the `Mirage_net.S` signature by calling the latter's C bindings. *mirage-block-solo5* implements the `Mirage_block.S` signature in the same way. The C bindings are used to call Solo5's C functions (also called bindings) from OCaml via OCaml's foreign function interface (ffi). The bindings are shown as "hvt-bindings" in Figure 3.1.

3.3 Hardware Access Modifications

To enable unikernels to communicate with hardware devices that are attached to the host, we will modify Solo5's hvt in two ways: we will map a PCIe device's configuration registers into the unikernel's address space, and we will add support for mapping DMA-ready memory (Direct Memory Access) into both the unikernel's and the device's address space. Finally, some additional "plumbing" will connect those additions to the manifest.

To make PCIe devices accessible to a unikernel and to facilitate DMA we make use of Linux's VFIO framework. VFIO is part of the Linux kernel and allows access to PCIe devices from userspace through the CPU's IOMMU [1]. The IOMMU maps PCIe devices' memory accesses from virtual addresses to phyical addresses and ensures memory safety by preventing rogue devices from accessing memory that they are not explicitly permitted to access. By binding the device to the `vfio-pci` driver and then opening the corresponding `/dev/vfio/vfio` file, a device can be controlled from userspace and the CPU's IOMMU can be configured.

We make use of ixy's *libixy-vfio* [9], a library that implements VFIO setup and offers facilities for mapping a device's registers and configuration space into memory. ixy [5] is a userspace network driver targeting ixgbe NICs written in C; the previously mentioned ixy.ml driver is based on ixy.

3.3.1 Structure

Figure 3.3 shows an overview of a MirageOS unikernel running on hvt with PCIe support. Components drawn in gray are part of Linux, components drawn in green are part of hvt, components drawn in blue are part of MirageOS, and those drawn in black are hardware. Solid arrows indicate application data transfer; dashed arrows inidcate configuration data transfer. Network and block devices controlled by hvt still work the same way as in figure 3.1, but have been omitted for clarity.

Roughly speaking, the newly added PCIe module fetches the unikernel's required PCIe devices and DMA setup from the provided manifest and sets up mappings in both MMU and IOMMU. The MMU mappings are created via KVM, whereas the IOMMU mappings are created via VFIO. Finally, the unikernel's page table needs to be configured to reflect the mappings set up in the MMU.

Note the lack of crossings of the hypervisor's guest-host boundary between hvt and the KVM virtual machine running the unikernel.

When mapping PCIe regions and DMA-ready memory into the unikernel's address space, its page table needs to be modified accordingly. Section 3.3.2 explains x86_64 paging in more detail.

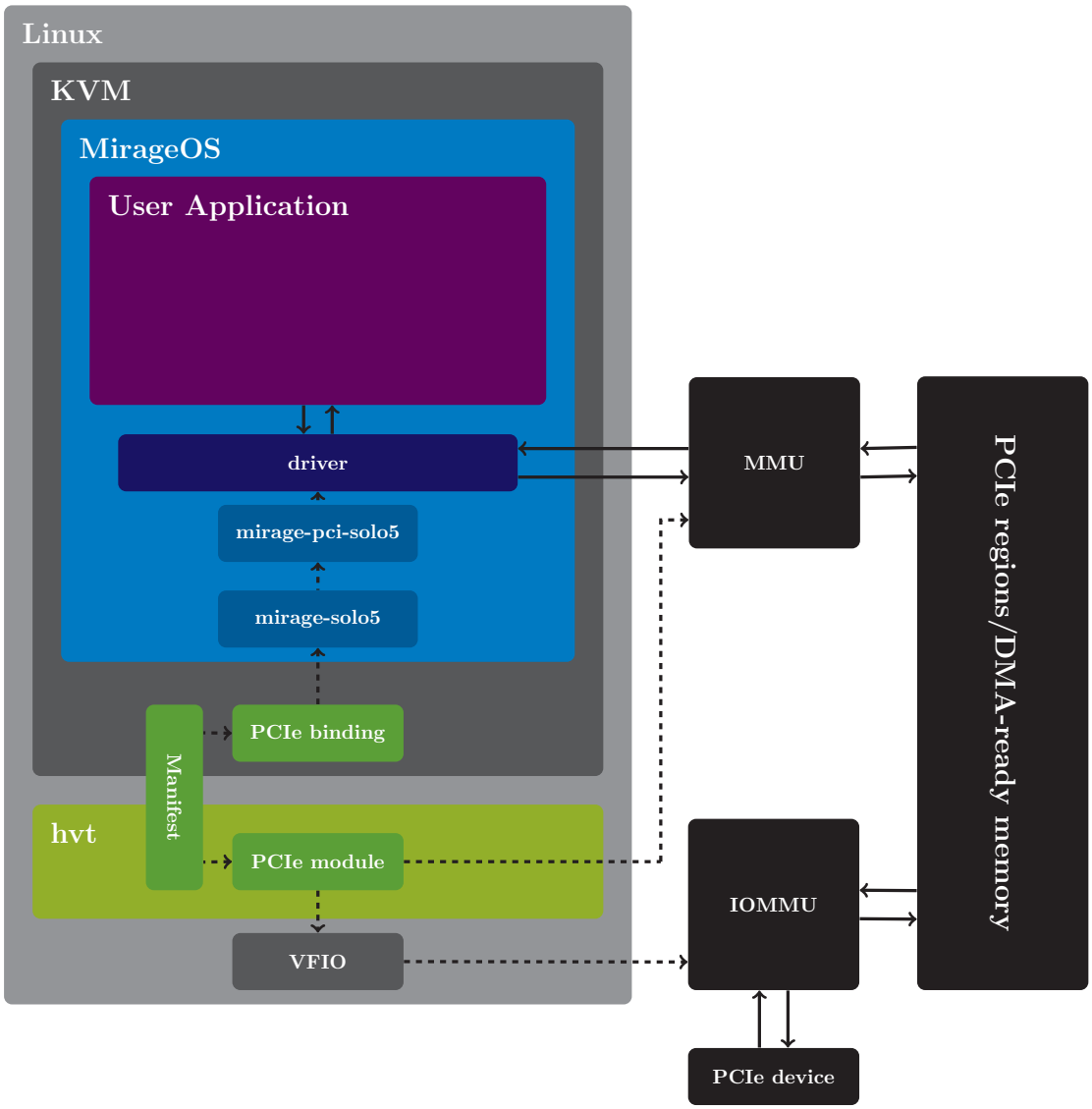

Figure 3.3: Overview of a MirageOS unikernel communicating with PCIe devices on Solo5's hvt

3.3.2 x86_64 paging

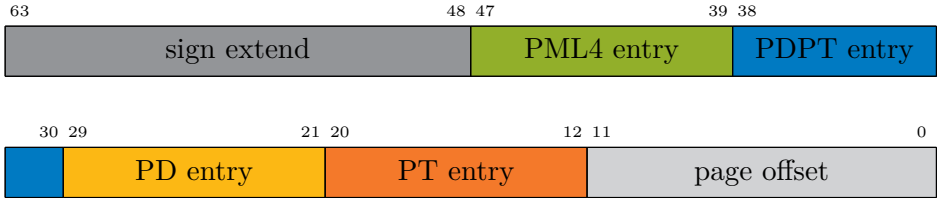

Figure 3.4: x86_64 long mode virtual address structure (4 KiB page)

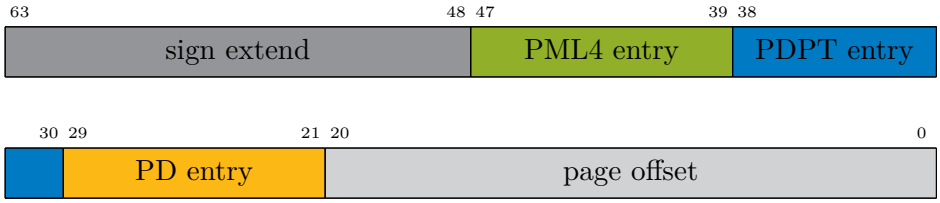

Figure 3.5: x86_64 long mode virtual address structure (2 MiB huge page)

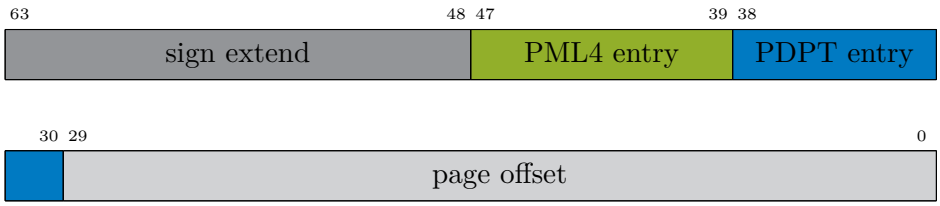

Figure 3.6: x86_64 long mode virtual address structure (1 GiB huge page)

x86_64 processors running in long mode (64-bit mode) employ 4 levels of page tables [10]. The levels of the page table are PML4 (page map level 4), PDPT (page directory pointer table), PD (page directory), and PT (page table). Figure 3.4 shows the structure of a virtual address. When looking up a virtual address's corresponding physical address, the virtual address is split into six sections: sign extend, four page table offsets, and the physical-page offset. The PML4, PDPT, PD, and PT entries are used as indices into the respective levels of the page table.

Each level is an array of pointers to the next level's tables. Using the respective entries as indices into these arrays, each successive table is determined. Each table's entry also contains some status bits; of note are `P` (present) and `PS` (page size). If an entry's `P` bit is not set, a page fault is triggered and the page table is not traversed further. If an entry's `PS` bit is set, the page table is not traversed further and the entry is directly interpreted as the physical address of a page. This bit may be set in levels 2 (PD) and 3 (PDPT). In this case the remaining part of the address will be interpreted as the phyiscal-page offset, adding 9 or 18 bits to the offset's size, respectively. This is used

to implement huge pages (2 MiB or 1 GiB). Figure 3.5 shows the structure of a virtual address pointing into a 2 MiB huge page; Figure 3.6 shows the structure of a virtual address pointing into a 1 GiB huge page.

3.3.3 PCIe

A unikernel must declare the PCIe devices it needs access to in its manifest. A PCIe device's entry in the manifest contains an identifier (a name like `pci0`), some metadata to ensure hvt will map the correct device (vendor and device identifier, PCIe class, subclass and programming interface), and whether or not the device needs access to main memory (by becoming a bus master). Additionally the unikernel needs to state the amount of DMA-ready memory that should be mapped into its and its devices' address spaces. Figure 3.7 shows the manifest of a unikernel requesting two PCIe devices, a network device (managed by hvt), and 16 MiB of DMA-ready memory.

hvt maps the requested regions into the unikernel's address space at predetermined addresses. For convenience we introduced a new function `solo5_pci_acquire()` to Solo5's unikernel-facing API that calculates the correct addresses and stores them in a structure `solo5_pci_info`. This structure is shown in Figure 3.8. `solo5_pci_acquire()` also retrieves device metadata (device ID, vendor ID, etc.) from the manifest.

Each PCIe device's requested regions are mapped to a fixed address based on the device's position in the manifest (starting at 0, the n-th PCIe device appearing in the manifest has position n) and the resource's number (`BAR`i has number i) according to the formula $2^{39} + n * 2^{34} + i * 2^{30}$.

Figure 3.9 shows the memory map of a unikernel that requested `BAR0` and `BAR2` of the first PCIe device in the manifest, `BAR0` of the second PCIe device in the manifest, and some amount of DMA-ready memory. The green areas are PCIe device regions; the blue area is the DMA-ready memory area. A possible manifest file creating this memory map is shown in Figure 3.7.

The base offset 2^{39} was chosen as such to make its index in the top level (`PML4`) of x86_64's page table 1.

3.3.4 DMA

DMA-ready memory is also supplied by libixy-vfio [9] using the VFIO framework. A single area of DMA-ready memory is allocated on huge memory pages (2 MiB) and mapped into the unikernel's address space at the fixed offset 2^{40}. This offset is chosen similarly to the PCIe region offset; its index in the top level page table is 2.

```
{
    "type": "solo5.manifest",
    "version": 1,
    "dma_size": 16777216,
    "devices": [
        {
            "name": "pci0",
            "type": "PCI_BASIC",
            "class": 2,
            "subclass": 0,
            "progif": 0,
            "vendor": 32902,
            "device_id": 4347,
            "bus_master_enable": true,
            "map_bar0": true,
            "map_bar1": false,
            "map_bar2": true,
            "map_bar3": false,
            "map_bar4": false,
            "map_bar5": false
        },
        {
            "name": "service0",
            "type": "NET_BASIC"
        },
        {
            "name": "pci1",
            "type": "PCI_BASIC",
            "class": 2,
            "subclass": 0,
            "progif": 0,
            "vendor": 32902,
            "device_id": 4347,
            "bus_master_enable": true,
            "map_bar0": true,
            "map_bar1": false,
            "map_bar2": false,
            "map_bar3": false,
            "map_bar4": false,
            "map_bar5": false
        }
    ]
}
```

Figure 3.7: Example hvt manifest file of a unikernel requesting two PCIe devices, a network device and 16 MiB of DMA-ready memory

```
struct solo5_pci_info {
    uint16_t vendor_id;      /* This device's PCI vendor. */
    uint16_t device_id;      /* This device's device ID. */
    uint8_t class_code;      /* This device's class code. */
    uint8_t subclass_code;   /* This device's subclass code. */
    uint8_t progif;          /* This device's programming interface. */
    bool bus_master_enable; /* This device is a bus master. */
    uint8_t *bar0;           /* This device's BAR0 or NULL. */
    size_t bar0_size;        /* This device's BAR0 size or 0. */
    uint8_t *bar1;           /* This device's BAR1 or NULL. */
    size_t bar1_size;        /* This device's BAR1 size or 0. */
    uint8_t *bar2;           /* This device's BAR2 or NULL. */
    size_t bar2_size;        /* This device's BAR2 size or 0. */
    uint8_t *bar3;           /* This device's BAR3 or NULL. */
    size_t bar3_size;        /* This device's BAR3 size or 0. */
    uint8_t *bar4;           /* This device's BAR4 or NULL. */
    size_t bar4_size;        /* This device's BAR4 size or 0. */
    uint8_t *bar5;           /* This device's BAR5 or NULL. */
    size_t bar5_size;        /* This device's BAR5 size or 0. */
};
```

Figure 3.8: Structure `solo5_pci_info`

Unikernels running on hvt do not use virtual memory; the page table maps each virtual address to an identical physical address inside the KVM virtual machine. Programming the IOMMU in the same way grants both the unikernel and its PCIe devices an identical view of the DMA-ready memory. Thus, unikernel driver developers need not keep track of virtual and physical addresses as there is no difference between them.

We created another convenience function `solo5_dma_acquire()` that checks whether there is DMA-ready memory available and, if so, returns its address and size.

3.4 PCIe on other backends

It should be possible to integrate PCIe support into Solo5's spt. libixy-vfio's PCIe and DMA setup code is independent of KVM and can be reused in spt. With appropriate bindings and seccomp rules unikernels running on spt could also communicate with PCIe devices.

hvt's aarch64 backend should only require a different page table setup from x86_64; KVM and VFIO support this architecture natively.

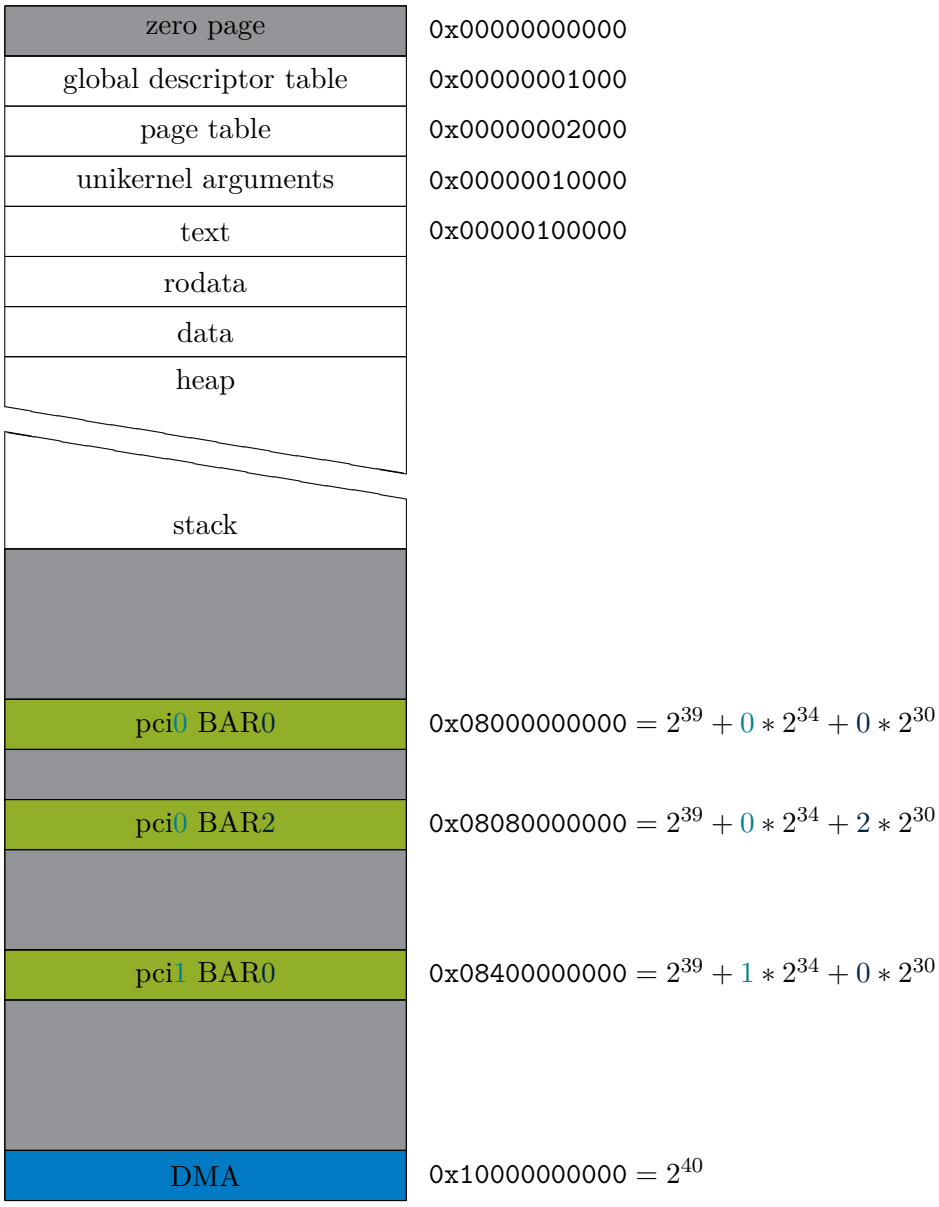

Figure 3.9: Example hvt memory map on x86_64 using the manifest from Figure 3.7

FreeBSD and OpenBSD require different IOMMU setup code; libixy-vfio cannot be used on these platforms.

Chapter 4

Mirage-pci

4.1 mirage-pci

Interfaces between devices and protocol implementations are represented as module types in MirageOS. For example, there is a module type `Mirage_net.S` that defines the types and functions a MirageOS network device must offer. Similarly the module signature `Mirage_protocols.IPV4` defines the types and functions a MirageOS-compatible IPv4 implementation must offer.

To introduce PCIe devices to this system of module types and modules, we define a new module type `Mirage_pci.S` in a new library *mirage-pci* [8]. Figure 4.1 shows the entire module type.

MirageOS' signatures follow the functional programming style of separating data and logic, i.e. each module declares a type `t` (imported from `Mirage_device.S` in this case) that contains the module's main data structure, and a number of functions to extract information from `t` and to control the device represented by `t`.

Programmers familiar with the object-oriented style of programming might initially find this style odd; where one would usually call a method on an object, instead an object is passed to a function. This pattern is necessary due to MirageOS' compilation style (as detailed in 2.2).

4.2 Building unikernels with PCIe support

Besides creating the mirage-pci library, we also added support for building unikernels that require PCIe devices to the `mirage` utility. It needs to select an appropriate

```
(** A PCIe device. *)
module type S = sig
  (* error handling omitted *)
  include Mirage_device.S

  (** The PCIe device's vendor ID. *)
  val vendor_id : t -> int

  (** The PCIe device's device ID. *)
  val device_id : t -> int

  (** The PCIe device's class code. *)
  val class_code : t -> int

  (** The PCIe device's subclass code. *)
  val subclass_code : t -> int

  (** The PCIe device's programming interface. *)
  val progif : t -> int

  (** The PCIe device's BAR0 region. *)
  val bar0 : t -> Cstruct.t option

  (** The PCIe device's BAR1 region. *)
  val bar1 : t -> Cstruct.t option

  (** The PCIe device's BAR2 region. *)
  val bar2 : t -> Cstruct.t option

  (** The PCIe device's BAR3 region. *)
  val bar3 : t -> Cstruct.t option

  (** The PCIe device's BAR4 region. *)
  val bar4 : t -> Cstruct.t option

  (** The PCIe device's BAR5 region. *)
  val bar5 : t -> Cstruct.t option

  (** The DMA memory allocated to this device. *)
  val dma : t -> Cstruct.t

  (** This device's identifier. *)
  val name : t -> string
end
```

Figure 4.1: Mirage_pci.S module type

```
type device_info =
  { bus_master_enable : bool
  ; map_bar0 : bool
  ; map_bar1 : bool
  ; map_bar2 : bool
  ; map_bar3 : bool
  ; map_bar4 : bool
  ; map_bar5 : bool
  ; vendor_id : int
  ; device_id : int
  ; class_code : int
  ; subclass_code : int
  ; progif : int
  ; dma_size : int
  }
```

Figure 4.2: Type representing a PCIe device's metadata in MirageOS

implementation of `Mirage_pci.S` on each target platform. Currently it supports PCIe devices on Solo5 and Unix, and, thus, will fail when selecting other target platforms.

On Solo5 our *mirage-pci-solo5* library is selected; see Section 4.3. On Unix our *mirage-pci-unix* library is selected; see Section 4.4.

Besides selecting an implementation, `mirage` also generates Solo5's manifest (see 3.3.3). Since we require a unikernel to declare its desired PCIe device and amount of DMA-ready memory beforehand, we added a type `device_info` to `mirage`. It contains device metadata similar to what is found in `struct solo5_pci_info` (see figure 3.8). Its definition is shown in Figure 4.2.

To generate Solo5's manifest, all devices declared in a unikernel's `config.ml` file are converted to entries in the new manifest. Since our implementation of DMA-ready memory in Solo5 only supports a single region of memory visible to all devices, the manifest generation simply sums up all devices' DMA requirements.

4.3 Mirage-pci-solo5

To support our additions to Solo5 (see Section 3.3) in MirageOS, we created a library *mirage-pci-solo5*. mirage-pci-solo5 wraps the output of the previously introduced (sections 3.3.3 and 3.3.4) functions `solo5_pci_acquire()` and `solo5_dma_acquire()` in OCaml-compatible types: `int` for integers and `Cstruct.t` for memory regions. `Cstruct.t` is part of *ocaml-cstruct* [26], a library commonly used in MirageOS to access external memory, i.e. memory not managed by the OCaml runtime.

```
type solo5_pci_info =
  { vendor_id : int
  ; device_id : int
  ; class_code : int
  ; subclass_code : int
  ; progif : int
  ; bus_master_enable : bool
  ; bar0_buffer : Cstruct.buffer
  ; bar1_buffer : Cstruct.buffer
  ; bar2_buffer : Cstruct.buffer
  ; bar3_buffer : Cstruct.buffer
  ; bar4_buffer : Cstruct.buffer
  ; bar5_buffer : Cstruct.buffer
  }

type t =
  { id : string
  ; mutable active : bool
  ; info : solo5_pci_info
  ; bar0 : Cstruct.t
  ; bar1 : Cstruct.t
  ; bar2 : Cstruct.t
  ; bar3 : Cstruct.t
  ; bar4 : Cstruct.t
  ; bar5 : Cstruct.t
  ; dma : Cstruct.t
  }
```

Figure 4.3: Type representing a PCIe device in mirage-pci-solo5

Figure 4.3 shows the representation of PCIe devices on Solo5. The type `solo5_pci_info` represents the structure `solo5_pci_info` we added to Solo5's unikernel-facing API (see Section 3.3.3).

4.3.1 Scheduling

MirageOS' platform libraries for Solo5 [23] work somewhat differently from the ones used on other backends like Unix (see Section 2.5.1). Instead of only waking up sleeping Lwt promises, mirage-solo5 also calls `solo5_yield()` to check whether packets have been received by any of the network devices controlled by hvt. Additionally MirageOS' sleep function is also implemented using `solo5_yield()`.

Unfortunately this behavior clashes with poll mode drivers like ixy.ml. ixy.ml's implementation of the `Mirage_net.S` signature needs to run in a tight loop and repeatedly

check whether the NIC has received packets. MirageOS' API requires a network interface to call a callback function for each received packet in a separate promise from the one running the network driver.

Figure 4.4 shows ixy.ml's `listen` function. It runs the recursive `aux` function (line 5) in a loop. Functional programming languages oftentimes implement loops using tail-recursive functions. In every iteration `aux` polls the driver (line 14) and, if there are new packets, calls `recv` on each (line 19). If there are no new packets, the promise that runs the function must explicitly yield to other promises, otherwise it could not be unscheduled until a new packet is received. Lwt can only schedule other promises at certain points, such as a bind (`>>=` operator, line 20). While the driver receives no new packets, no such point is hit and, therefore, an explicit call to `Lwt.pause` must be inserted. `Lwt.pause` inserts the calling promises into a set of paused promises that may be woken up again at any time. Lwt's engine takes care of waking these promises up in its main loop.

```
let listen t ~header_size cb =
  if header_size > 18 then
    Lwt.return_error `Invalid_length
  else
    let rec aux () =
      let recv pkt =
        let open Ixy_core.Ixy_memory in
        let buf = Cstruct.create pkt.size in
        Cstruct.blit pkt.data 0 buf 0 pkt.size;
        Ixy.Memory.pkt_buf_free pkt;
        Lwt.async (fun () -> cb buf);
        Lwt.return_unit in
      if t.active then
        let batch = Ixy.rx_batch t.dev 0 in
        begin
          if Array.length batch = 0 then
            Lwt.pause ()
          else
            Array.fold_left
              (fun acc v -> acc >>= fun () -> recv v)
              Lwt.return_unit
              batch
        end >>= aux
      else
        lwt_ok_unit in
    aux ()
```

Figure 4.4: ixy.ml's implementation of `Mirage_net.S`' `listen` function

Since mirage-solo5 incorporates `solo5_yield()` into its main loop, when there are no network devices managed by hvt or these devices do not receive data, the main loop becomes stuck until a deadline supplied by mirage-solo5 to `solo5_yield()` runs out. This deadline depends on the presence of promises that called MirageOS' sleep function. If there are no such promises, the deadline may be as long as 24 hours.

To ensure poll mode drivers work properly, we modified mirage-solo5 to check whether there are paused promises that can be awoken at any time and, if so, reduce the deadline to 100 µs. If there are no promises waiting for network devices controlled by hvt to receive packets, we can skip the call to `solo5_yield()` entirely.

```
type t =
  { id : string
  ; fd : Unix.file_descr
  ; info : device_info
  ; mutable active : bool
  ; bar0 : Cstruct.t
  ; bar1 : Cstruct.t
  ; bar2 : Cstruct.t
  ; bar3 : Cstruct.t
  ; bar4 : Cstruct.t
  ; bar5 : Cstruct.t
  ; dma : Cstruct.t
  }
```

Figure 4.5: Type representing a PCIe device in mirage-pci-unix

4.4 Mirage-pci-unix

For MirageOS' Unix backend, we again used *libixy-vfio* [9] to create a library *mirage-pci-unix*. Its behavior is almost identical to the PCIe supporting code we added to Solo5; it simply lacks the KVM configuration since the unikernel is not running in a virtual machine but rather as a standard process on Linux. Where previously the unikernel was given access to PCIe devices and DMA-ready memory at fixed addresses, now we provide it directly with virtual addresses as MirageOS' setup code and the unikernel itself share the same address space. Thus we can fulfill the `Mirage_pci.S` signature on Linux.

Figure 4.5 shows the representation of PCIe devices on Unix. The field `fd` stores the VFIO file descriptor that represents this device.

Figure 4.6 shows an overview of a unikernel running on Linux using MirageOS' Unix backend with PCIe support. Components drawn in gray are part of Linux, components drawn in blue are part of MirageOS, and those drawn in black are hardware. Solid arrows indicate application data transfer; dashed arrows inidcate configuration data transfer. Network and block devices controlled by mirage-net-unix and mirage-block-unix still work the same way as in Figure 2.5, but have been omitted for clarity.

mirage-pci-unix instructs VFIO to configure the IOMMU to map PCIe regions into main memory visible to the unikernel. It also installs the same virtual-physical address mapping into the IOMMU as is installed in the MMU so the unikernel and the PCIe device can access the same area of memory under the same virtual address. The driver fetches these memory areas from mirage-pci-unix in form of `Cstruct.t`'s.

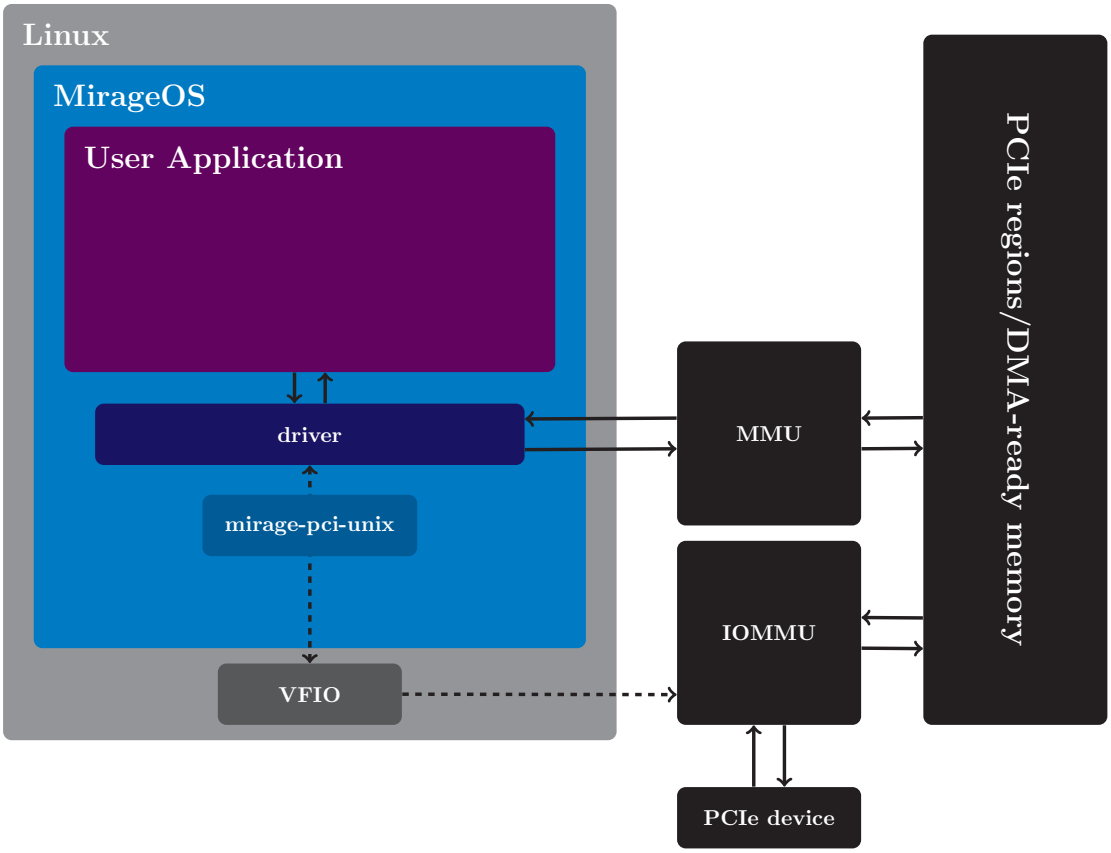

Figure 4.6: Overview of a MirageOS unikernel communicating with PCIe devices on Linux

Chapter 5

Performance

We evaluated our additions to Solo5 and MirageOS using a number of tools. All evaluations were driven using ixy.ml. ixy.ml was initially created as a userspace network driver for Intel ixgbe-compatible NICs written in OCaml. For this thesis we parameterized the driver over an OS-agnostic PCIe device implementation that closely matches the previously detailed `Mirage_pci.S` module signature. This allowed us to split the driver into two versions.

A Linux-based version (*ixy-freestanding*) that does not use an IOMMU and is reliant on Linux's hugetlbfs and sysfs for DMA-ready memory and PCIe device access contains most of ixy.ml's old setup code.

A newly created version for MirageOS, *mirage-net-ixy*, fits into MirageOS' system of functors in that it maps a `Mirage_pci.S`-compatible module (i.e. a module wrapping an ixgbe-compatible NIC) onto a module fulfilling the `Mirage_net.S` signature (i.e. the signature of a MirageOS network device).

Both versions internally use *ixy-core* that contains the actual driver logic dealing with the NIC configuration and transmitting and receiving packets.

Through mirage-net-ixy any MirageOS unikernel requiring a network device may use ixy.ml to directly control an ixgbe-compatible NIC. No modification of the unikernel is necessary, instead only its configuration will have to be adjusted to tell the `mirage` utility to map an ixgbe-compatible NIC into the final unikernel.

5.1 Raw PCIe and DMA performance

We created a simple unikernel using ixy.ml (without going through mirage-net-ixy) and an Intel 82599ES NIC that listened for packets and sent them back on the same connection without changes to measure packet rates and packet latency. Its configuration is shown in Figure 5.4; its source code is shown in figure 5.5. We ran the unikernel on both hvt and Linux directly (via MirageOS' Unix backend). The unikernel ran on an AMD Threadripper 1950X 16-core CPU. 60-byte packets were generated at line rate (14.88 million packets per second) using another Intel 82599ES NIC and an Intel i7-3770K CPU running MoonGen [4]. The unikernel was able to transmit at a rate of 13 million packets per second on both hvt and Linux.

Figure 5.1 shows the structure of the benchmark unikernel running on hvt. Figure 5.2 shows the structure of the benchmark unikernel running on Linux. Figure 5.3 shows the structure of the program running on Linux.

We compared this unikernel's performance with an equivalent program using the older Linux standalone version of ixy.ml (now renamed to ixy-freestanding) that does not use an IOMMU and does not run as part of MirageOS. Its source code is almost identical and shown in Figure 5.6.

Latency measurements for a variety of data rates are shown as CCDF plots in Figure 5.7. Above 6500 Mbit/s both versions were overloaded and latency spiked. We were not able to measure any significant difference between both versions and were able to conclude that our modifications do not hinder performance.

5.2 MirageOS networking performance

To evaluate TCP performance we used MirageOS' *tcpip* [24] network stack running on top of ixy.ml using our newly created mirage-net-ixy library.

We measured TCP throughput using the popular `iperf` utility and the MirageOS unikernel implementation *mirage_iperf* [14] of `iperf`'s protocol.

ixy.ml is somewhat incompatible with MirageOS' network signature in that it requires the use of polling. Polling refers to repeatedly querying the driver for newly arrived packets as fast as possible and is commonly used in high-performance low-latency networks. MirageOS instead relies on external notifications indicating arrived packets. When using Solo5's integrated TAP network interface, MirageOS essentially sleeps until it is notified by Solo5 of an arriving packet.

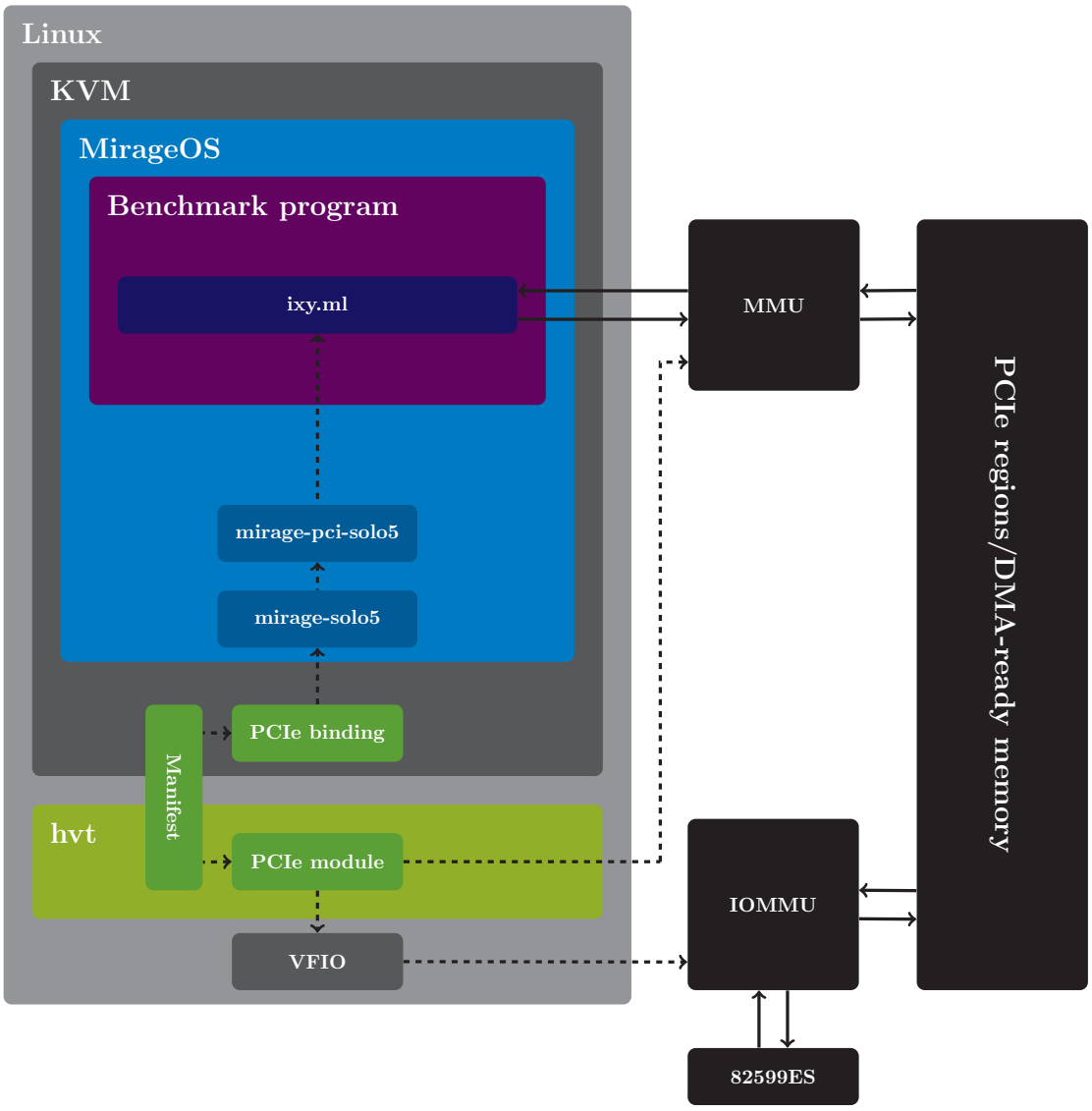

Figure 5.1: Overview of the hvt benchmark unikernel

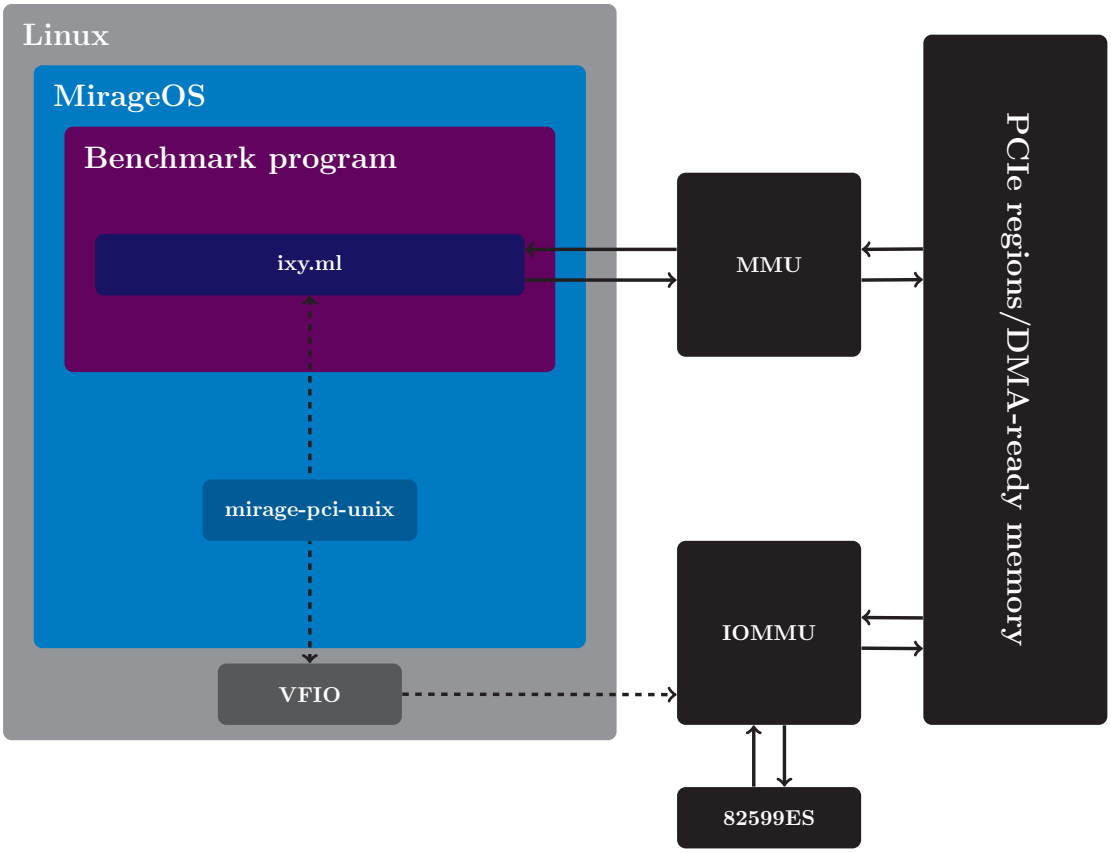

Figure 5.2: Overview of the Linux benchmark unikernel

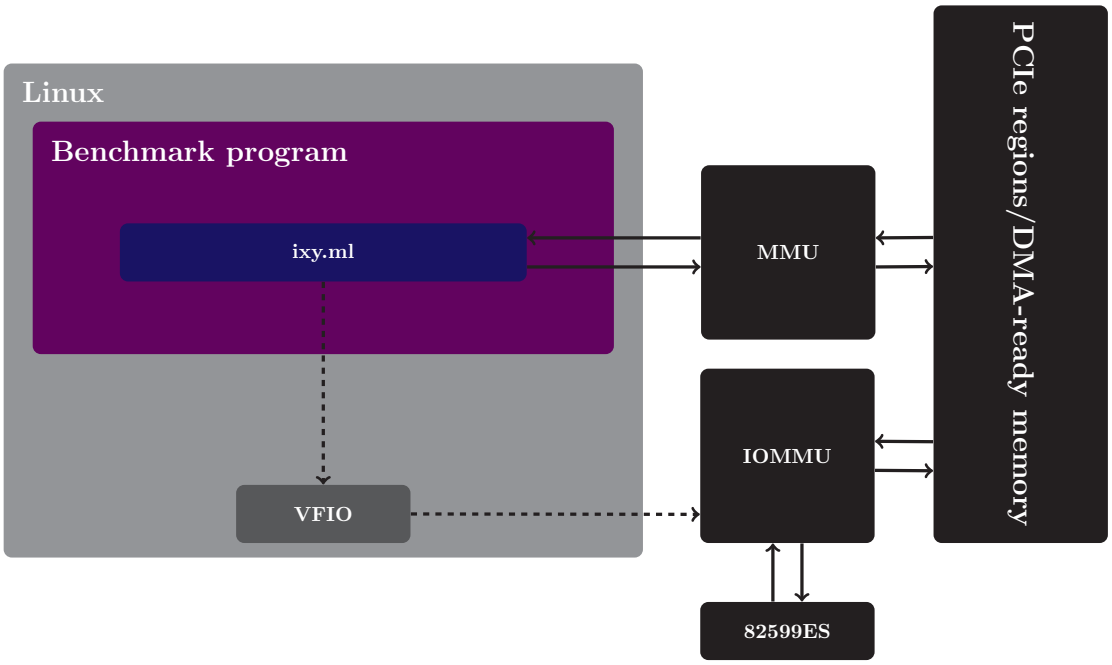

Figure 5.3: Overview of the Linux benchmark program

Without the scheduling fixes detailed in Section 4.3.1 this setup only managed a lackluster 3 MiB/s of TCP throughput on hvt using mirage-pci-solo5. With mirage-pci-unix these scheduling issues went away and MirageOS' network stack on top of ixy.ml managed around 3.25 Gbit/s of TCP throughput. Once we applied the modification to mirage-solo5, we achieved the same peformance on hvt.

Running the same benchmark with hvt's built-in networking and a TAP device on Linux yields 1.58 Gbit/s of TCP throughput. In this case moving the driver achieved twice the networking throughput at the cost of CPU usage stemming from ixy.ml being a poll mode driver.

MirageOS's Unix backend can use both MirageOS' built-in network stack via a TAP device as well as Linux's network stack via the POSIX socket API. The former was able to achieve 272 Mbit/s, yielding a twelve-fold increase in throughput from moving the driver into the unikernel.

Using Linux's network stack via the POSIX socket API shifted most of the load from the unikernel to Linux, which was able to schedule the driver and its network stack on a different core. This lead to a TCP throughput of 9.41 Gbit/s.

Figure 5.8 compares the throughput measurements of the various MirageOS backends. Green indicates results that have been made possible through our framework.

```
open Mirage

let main = foreign "Unikernel.Main" (pci @-> job)

let pci0 =
  let device_info =
        { vendor_id = 0x8086
        ; device_id = 0x10fb
        ; class_code = 0x2
        ; subclass_code = 0x0
        ; progif = 0x0
        ; dma_size = 16777216
        ; bus_master_enable = true
        ; map_bar0 = true
        ; map_bar1 = false
        ; map_bar2 = false
        ; map_bar3 = false
        ; map_bar4 = false
        ; map_bar5 = false
        } in
  pcidev device_info "pci0"

let () =
  register "pci" [
        main $ pci0
  ] ~packages:[ package "ixy-core"; package "mirage-net-ixy" ]
```

Figure 5.4: Configuration for the unikernel from Figure 5.5

```
module Main (S: Mirage_pci.S) = struct
  module Ixy = Ixy_core.Make (Pci_mirage.Make (S))

  let start pci0 =
    let dev = Ixy.create ~pci:pci0 ~rxq:1 ~txq:1 in
    while true do
      let rx = Ixy.rx_batch dev 0 in
      Ixy.tx_batch_busy_wait dev 0 rx;
    done;
    Lwt.return_unit
end
```

Figure 5.5: Unikernel that retransmits all received packets to the sender

```
let usage () =
  Ixy_core.Log.error "Usage: %s <pci_addr>" Sys.argv.(0)

let () =
  if Array.length Sys.argv <> 2 then
    usage ();
  let pci =
    match Ixy.of_string Sys.argv.(1) with
    | None -> usage ()
    | Some pci -> pci in
  let dev = Ixy.create ~pci ~rxq:1 ~txq:1 in
  while true do
    let rx = Ixy.rx_batch dev 0 in
    Ixy.tx_batch_busy_wait dev 0 rx
  done
```

Figure 5.6: Program that retransmits all received packets to the sender

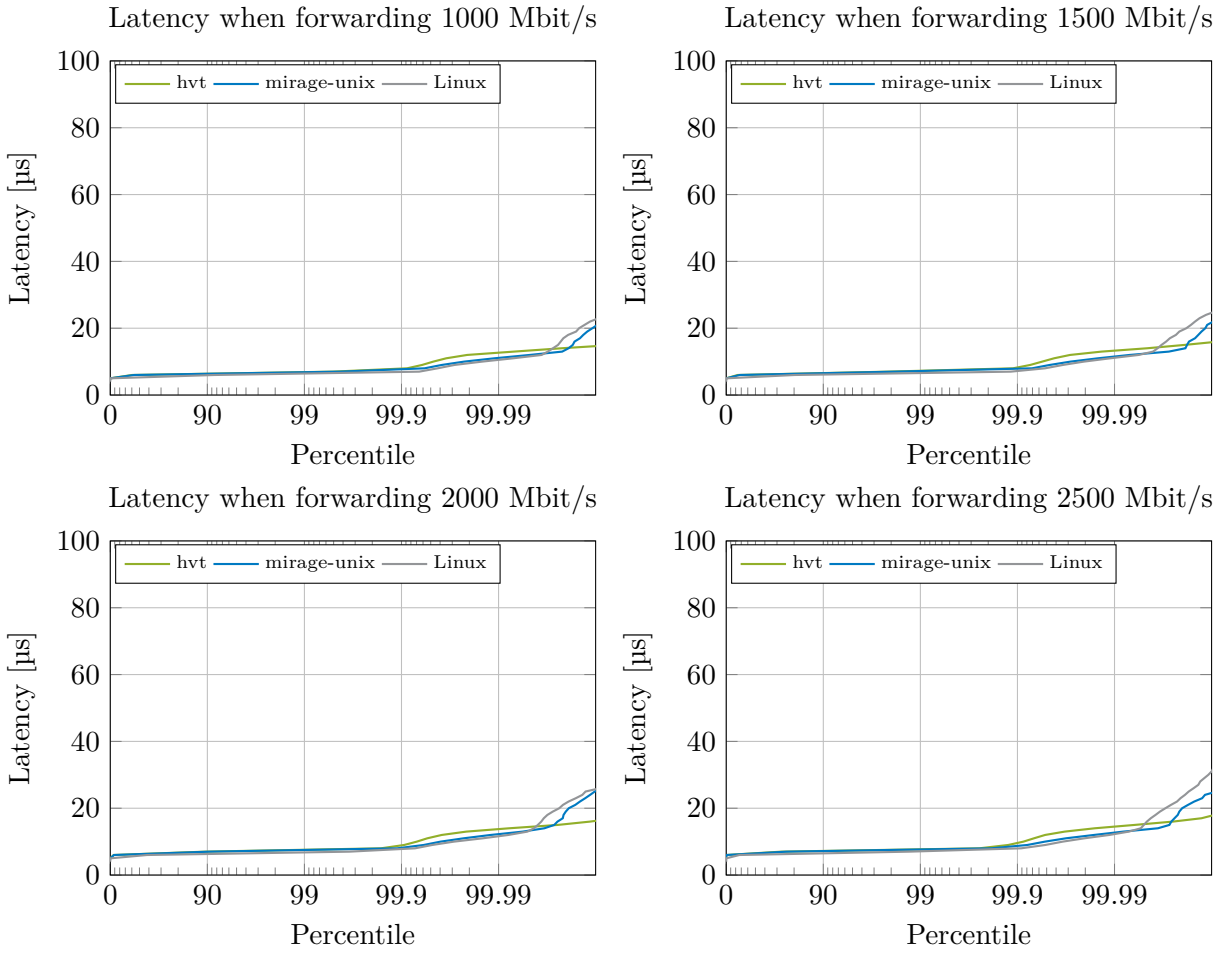

Figure 5.7: Latency measurements using ixy.ml on hvt, mirage-unix and Linux

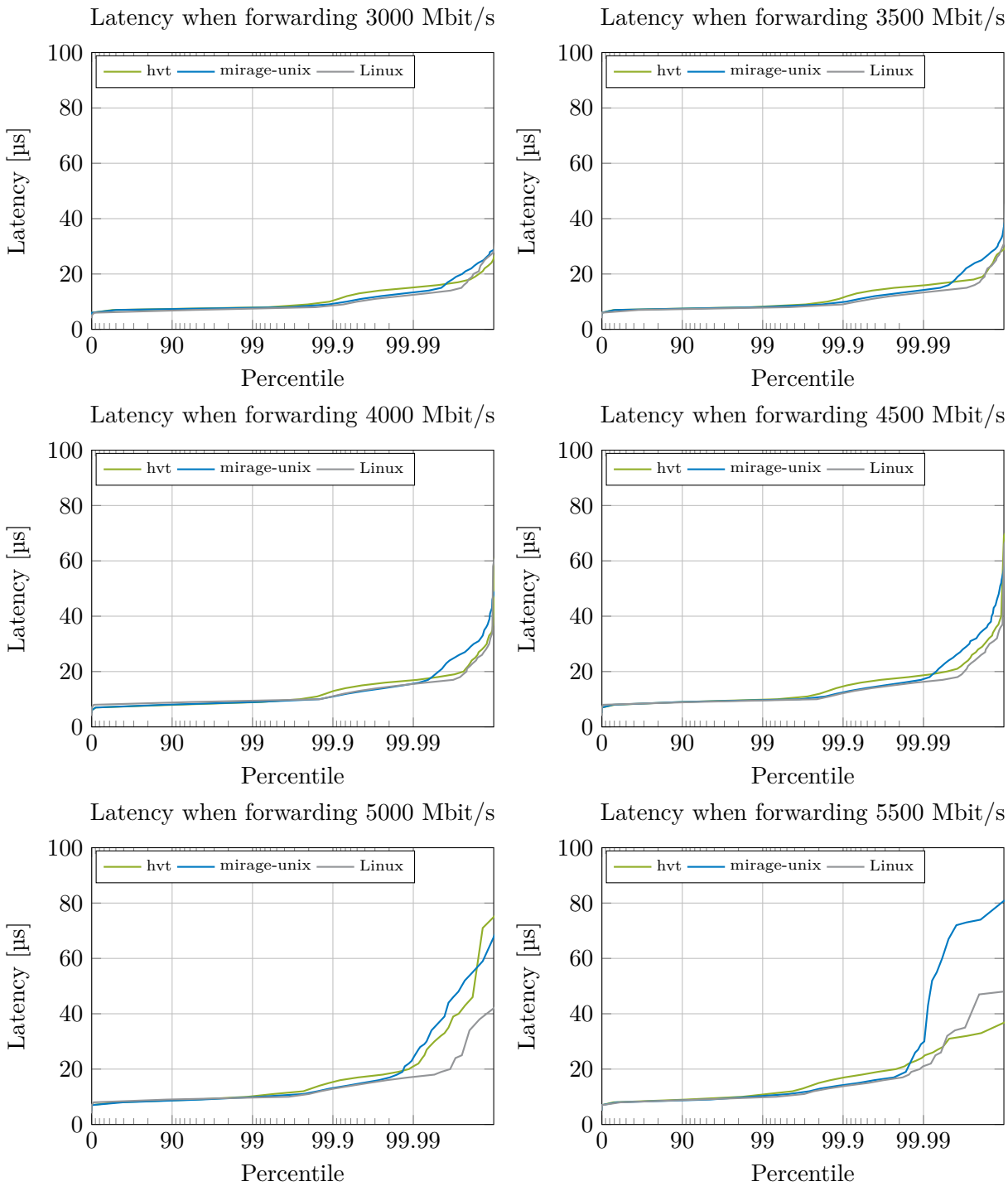

Figure 5.7: Latency measurements using ixy.ml on hvt, mirage-unix and Linux

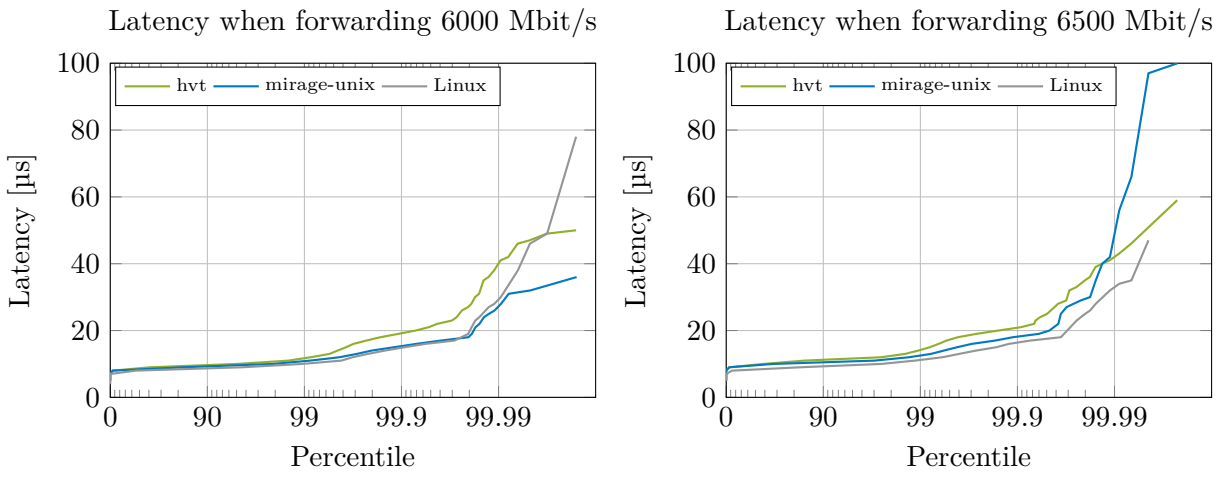

Figure 5.7: Latency measurements using ixy.ml on hvt, mirage-unix and Linux

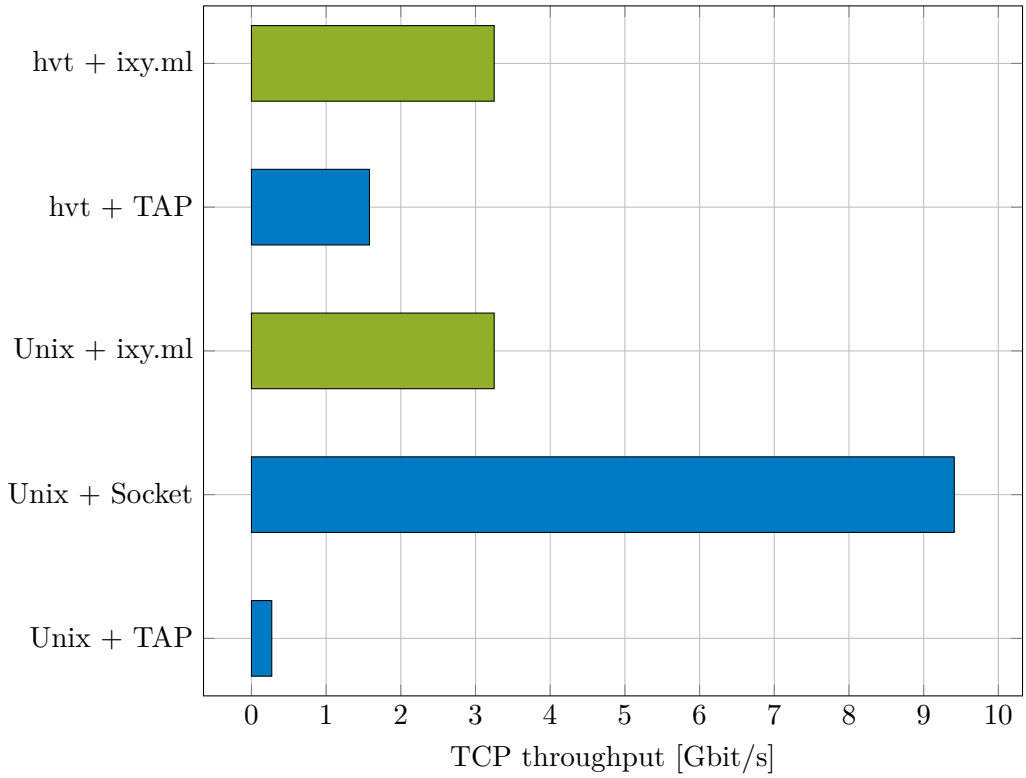

Figure 5.8: TCP throughput measurements using mirage_iperf

MirageOS' networking model is not well suited to ixy.ml's DMA-based packet buffers as there is no concept of freeing old buffers. MirageOS' receive callbacks assume perpetual ownership of incoming packet buffers, requiring mirage-net-ixy to perform a full copy of every incoming packet from DMA-ready memory into memory managed by OCaml's runtime, lest it runs out of DMA buffers. Even if MirageOS were to establish an ownership model for packet buffers, care would have to be taken to not keep any references to a freed packet buffer anywhere in the network stack or pass such references to unikernels using the network stack. Since DMA buffers will be overwritten after enough packets have arrived, the underlying memory pointed to by these references will change. This clashes with MirageOS' safety-first approach and will likely not be adopted.

Chapter 6

Conclusion

In this work we have established the foundation of PCIe support in MirageOS and one of Solo5's unikernel tenders. Additionally we added PCIe support to MirageOS' Unix backend, a backend executing a unikernel as a standard process on Linux.

MirageOS unikernels, as well as other frameworks targeting Solo5's hvt, are now able to benefit from secure and low-overhead access to PCIe devices and DMA-ready memory.

Currently on hvt each unikernel is limited to a single mapped PCIe device, a limit imposed by hvt's bare-bones page table implementation.

Our newly created libraries can be found here:

- mirage-pci - `https://github.com/Reperator/mirage-pci`
- mirage-pci-unix - `https://github.com/Reperator/mirage-pci-unix`
- mirage-pci-solo5 - `https://github.com/Reperator/mirage-pci-solo5`

The required modifications to existing libraries can be found here:

- mirage - `https://github.com/Reperator/mirage/tree/pci-support-stable`
- mirage-solo5 - `https://github.com/Reperator/mirage-solo5/tree/pci-support`
- solo5 - `https://github.com/Reperator/solo5/tree/pci-support`
- ixy.ml - `https://github.com/ixy-languages/ixy.ml`

6.1 Future work

There are a number of directions for future development.

Additional hardware drivers targeting MirageOS can now be written. There is a multitude of popular network adapters that are not supported by ixy.ml, and that could be valuable targets to drive MirageOS adoption. Additionally NVMe drives connected via PCIe could potentially be represented as MirageOS-compatible block devices, either directly or with an underlying file system.

Besides creating new drivers, there are a number of MirageOS target platforms still lacking PCIe support. MirageOS' Xen target still lacks PCIe support entirely. Adding PCIe support to hvt's aarch64 KVM target might be of interest; on Linux it is natively supported by the VFIO framework. By adapting hvt's page table implementation on aarch64 to additionally map PCIe regions and DMA-ready memory besides the unikernel's main memory, it should be possible to use our framework on this architecture as well. hvt's FreeBSD and OpenBSD targets are currently not supported by our framework because it relies on Linux's VFIO framework. Lastly, hvt's page table implementation may be refactored to allow for greater flexibility and more fine-grained memory mappings. This should lift the current limitation of one PCIe device per unikernel.

Besides hvt, it should be possible to extend Solo5's PCIe support to spt, the sandboxed process tender. Unlike hvt, spt executes unikernels in a seccomp sandbox [28]. Solo5's other backends will likely require more work since the VFIO framework our implementation uses is limited to Linux.

Literatur

[1] The Linux authors. *VFIO - "Virtual Function I/O"*. URL: https://www.kernel.org/doc/html/latest/driver-api/vfio.html.

[2] Fabian Bonk und Paul Emmerich. *Networking in MirageOS*. https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2019-06-1/NET-2019-06-1_10.pdf.

[3] Cody Cutler, M. Frans Kaashoek und Robert T. Morris. "The benefits and costs of writing a POSIX kernel in a high-level language". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Okt. 2018, S. 89–105. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/cutler.

[4] Paul Emmerich u. a. "MoonGen: A Scriptable High-Speed Packet Generator". In: *Internet Measurement Conference (IMC) 2015, IRTF Applied Networking Research Prize 2017*. Tokyo, Japan, Okt. 2015.

[5] Paul Emmerich u. a. "User Space Network Drivers". In: *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS 2019)*. Sep. 2019.

[6] D. R. Engler, M. F. Kaashoek und J. O'Toole. "Exokernel: An Operating System Architecture for Application-Level Resource Management". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: Association for Computing Machinery, 1995, 251–266. ISBN: 0897917154. DOI: 10.1145/224056.224076. URL: https://doi.org/10.1145/224056.224076.

[7] Fabian Bonk. *ixy.ml*. https://github.com/ixy-languages/ixy.ml.

[8] Fabian Bonk. *mirage-pci*. https://github.com/Reperator/mirage-pci.

[9] Stefan Huber und Paul Emmerich. "Using the IOMMU for Safe and Secure User Space Network Drivers". M.Sc. Thesis. 2019. URL: https://www.net.in.tum.de/fileadmin/bibtex/publications/theses/2019-ixy-iommu.pdf.

[10] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. 2020. URL: https://www.amd.com/system/files/TechDocs/24593.pdf.

[11] Anil Madhavapeddy und David J. Scott. "Unikernels: Rise of the Virtual Library Operating System". In: *Queue* 11.11 (Dez. 2013), 30–44. ISSN: 1542-7730. DOI: 10.1145/2557963.2566628. URL: https://doi.org/10.1145/2557963.2566628.

[12] Anil Madhavapeddy u. a. "Unikernels: Library Operating Systems for the Cloud". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems, asplos 2013, Irvine, California, USA, March 16-20, 2013*. 2013. DOI: 10.1145/2999572.2999602. URL: http://unikernel.org/files/2013-asplos-mirage.pdf.

[13] The lwt authors. *lwt*. https://github.com/ocsigen/lwt.

[14] The mirage-iperf authors. *mirage-iperf*. https://github.com/TImada/mirage_iperf.

[15] The MirageOS authors. *Functoria*. https://github.com/mirage/mirage/tree/master/lib/functoria.

[16] The MirageOS authors. *mirage*. https://github.com/mirage/mirage.

[17] The MirageOS authors. *mirage-block*. https://github.com/mirage/mirage-block.

[18] The MirageOS authors. *mirage-block-unix*. https://github.com/mirage/mirage-block-unix.

[19] The MirageOS authors. *mirage-net*. https://github.com/mirage/mirage-net.

[20] The MirageOS authors. *mirage-net-solo5*. https://github.com/mirage/mirage-net-solo5.

[21] The MirageOS authors. *mirage-net-unix*. https://github.com/mirage/mirage-net-unix.

[22] The MirageOS authors. *mirage-net-xen*. https://github.com/mirage/mirage-net-xen.

[23] The MirageOS authors. *mirage-solo5*. https://github.com/mirage/mirage-solo5.

[24] The MirageOS authors. *mirage-tcpip*. https://github.com/mirage/mirage-tcpip.

[25] The MirageOS authors. *MirageOS*. https://mirage.io/.

[26] The MirageOS authors. *ocaml-cstruct*. https://github.com/mirage/ocaml-cstruct.

[27] The Solo5 authors. *Solo5*. https://github.com/Solo5/solo5.

[28] Dan Williams u. a. "Unikernels as Processes". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '18. Carlsbad, CA, USA: Association for Com-

puting Machinery, 2018, 199–211. ISBN: 9781450360111. DOI: 10.1145/3267809.3267845. URL: https://doi.org/10.1145/3267809.3267845.