

Forside

Eksamensinformation

NDAB12006E - Bachelorprojekt i datalogi, Datalogisk Institut (Axel & Svante)

Besvarelsen afleveres af

Axel Prütz Kanne
jqc157@alumni.ku.dk

Svante Geisshirt
pcj655@alumni.ku.dk

Eksamensadministratorer

Eksamen Nørre
eksamen-noerre@adm.ku.dk

Bedømmere

Martin Elsman
Eksaminator
mael@di.ku.dk
☎ +4535335683

Patrick Bahr
Censor
paba@itu.dk

Besvarelsesinformationer

Titel, engelsk: Unikernels with Region Inference

Tro og love-erklæring: Ja

Indeholder besvarelsen fortroligt materiale: Nej

Må besvarelsen gøres til genstand for udlån: Ja

Må besvarelsen bruges til undervisning: Ja



Bachelor thesis in Computer Science

Unikernels with Region Inference

Axel Prütz Kanne	jqc157
Svante Geisshirt	pcj655

Supervised by Martin Elsman

June 2024



Abstract

Unikernels are single-purpose operating systems designed to run on a hypervisor, fusing the application and operating system code into one kernel, where only required layers are left, improving security, performance, and size. Unikernels can be created entirely in higher-order languages, which improves memory safety and developer productivity. However, using a higher-order language will often include using a garbage collector, which relieves the developer of managing memory but can affect the application’s performance. An alternative is region-based memory management with region inference, which can be used to manage and infer how values should be stored inside regions. Using this alternative for developing unikernels can potentially avoid the system-wide performance overhead associated with garbage collection while keeping the same level of developer productivity. To explore whether this is the case, a library for creating services as unikernels that use only region-based memory management has been developed, which provides the essential functionality for answering network requests. While the implementation of the library is somewhat unstable, it can be used to create basic services running as unikernels, and thereby demonstrates that it is feasible to use region-based memory management for unikernel development. Profiling of the library shows that while region inference, in some cases, can manage memory efficiently, an application that is not optimized for region usage can result in memory consumption that grows proportionally with the running time. To avoid this, one can either optimize the application for region usage, be more explicit about the use of the regions, or use a garbage collector in conjunction with region-based memory management.

Contents

1	Introduction	3
1.1	Reading guide	4
2	Background	5
2.1	Unikernels and cloud computing	5
2.1.1	Cloud computing and virtualization	5
2.1.2	The limitations of virtualization methods	6
2.1.3	Unikernels	7
2.1.4	Unikernels in practice	7
2.2	Region-based memory management	8
2.2.1	Concept	8
2.2.2	Benefits	10
3	Design	11
3.1	Unikernel libraries	11
3.1.1	Network protocols	12
3.2	Architecture design	14
3.2.1	Language runtime	15
3.2.2	Platform support	15
3.2.3	Network stack	17
3.2.4	Differences between platforms	17
4	Implementation	19
4.1	Network Library	19
4.1.1	User layer	19
4.1.2	Handling packets	20
4.1.3	Sending packets	23
4.1.4	Network profiling	23
4.2	Network protocols	24
4.3	Platform implementations	28
4.3.1	Linux	29
4.3.2	Xen	31
4.4	Runtime minimization	33
4.4.1	Platform related changes	33
4.4.2	Runtime	33

4.4.3	Basis library	34
5	Evaluation	37
5.1	Testing the implementation	37
5.2	Example services	38
5.3	Running services on Xen	41
5.4	Evaluating thesis	42
5.4.1	Use of regions	42
5.4.2	Developer productivity and memory safety	46
5.4.3	Significance for unikernels	47
6	Future developments	48
7	Conclusion	49
A	Running the services on Xen	53
A.1	facfib	53
A.2	sort	54
A.3	monteCarlo	55
B	Region-annotated code from profiling	56
B.1	listen function in network-library	56

Chapter 1

Introduction

Today, cloud computing relies on virtualization to utilize servers in data centers to their fullest by running multiple operating systems as virtual machines. Different solutions for virtualization exist, such as virtualizing the whole operating system or isolating processes through containerization. This enables developers to rent resources from cloud providers and run their web service inside a virtual machine without having to manage servers on their own. However, running a general-purpose operating system for a single application adds many layers that are not necessarily needed by the service, introducing performance overhead and a larger attack surface.

Unikernels is the idea of combining the application and underlying operating system, to create one single-purpose kernel that can be run inside a hypervisor. This removes the unnecessary layers, and keeps the parts that are used by the service. Unikernels can be developed almost entirely in higher-level languages, which can offer type safety and memory safety. Using a garbage collector relieves the developer of thinking about managing memory efficiently and correctly, thereby increasing developer productivity. However, garbage collection also introduces program halts and performance-hampering memory traversals, leading to a decrease in performance, which in unikernel development will be system-wide.

Region-based memory management offers an alternative way to manage memory, which can lead to fewer halts and memory traversals. Region-inference is a way to infer regions in the program for the developer, such that region-based memory management can be used without changing the program. This project aims to explore whether this is a feasible way to develop unikernels in a higher-order language with the following thesis statement:

Using region-based memory management with region inference is a feasible approach for developing unikernels that do not rely on dynamic garbage collectors, which may introduce interruptions and unnecessary memory traversals. Subsequently, this approach does not compromise memory safety or developer productivity and thus maintains the possibility of using higher-order programming languages for programming unikernels in a cloud computing setting.

To ascertain whether the thesis statement holds, the project's goal is to create a library for developing unikernels that use region-based memory management with region inference.

The library should be able to support the development of services that can answer network requests. To do this, a minimal networking stack will be implemented, which allows for communication over a network. This aims to demonstrate the feasibility of using region-based memory management in unikernels.

1.1 Reading guide

This report will include the following chapters, and what their focus will be:

- **Background** aims to give an understanding of the concepts used throughout this project, from the unikernels in a cloud computing setting to region-based memory management.
- **Design** will describe the design goals for the project implementation and a discussion of why each module is necessary for the project.
- **Implementation** here the implementation details of the project will be described, both for implementing networking protocols, but also implementation details regarding the construction of the service on different platforms.
- **Evaluation** will discuss the results of the project, both in terms of testing and to what degree the thesis statement holds.
- **Future developments** will give suggestions for how the project could be improved and what further additions could be made to the project.
- **Conclusion** will conclude the project findings as a reflection on the thesis statement

Chapter 2

Background

This chapter aims to give some background information about the technologies used in this project. The first section will concentrate on what cloud computing and virtualization are and what they aim to solve, the limitations of virtualization, and then the solution to some of the limitations of unikernels. Furthermore, how and where unikernels are used in practice. The second section will focus on what region-based memory management is, how it works in broad strokes, and what benefits it has over an explicit method and a more traditional memory management method such as a garbage collector.

2.1 Unikernels and cloud computing

This section presents an overview of cloud computing and virtualization and how these concepts are related to unikernels. Additionally, the section includes how unikernels can improve performance and reduce overhead in cloud computing. Furthermore, some existing projects for developing unikernels are also mentioned, as their project goals and implementation language.

2.1.1 Cloud computing and virtualization

When creating an application that needs to communicate through the internet, the application needs to run and be managed on hardware such as a server. However, setting up and managing a self-hosted server can be rather insecure and costly. *Cloud Computing* refers to the idea that software and hardware resources can be rented out over the internet through the *cloud* - a collection of hardware resources and software. These can either be rented out to the public in a pay-as-you-go manner as a *public cloud* or used internally by a business or organization as a *private cloud* [2]. This means that rather than hosting and managing the resources for your application yourself, cloud providers like Google or Amazon can rent out the resources as needed and manage them in a data center for you. Cloud computing provides other benefits, such as low variable costs. Instead of potentially wasting resources, hosting several servers in advance that might need to be used, either to set up for a future project or for surges of requests, with Cloud Computing, you only pay for the resources you currently use [5]. It is also easier for an application to scale *smoothly* since an application

can start out small and gradually pay for more resources if needed [2].

However, if a dedicated server was used for each application, or the applications were installed directly on the servers, the servers would end up being underutilized since running different applications would then, in some instances, require different servers with different operating systems. Instead, virtualization is used to host *virtual machines* (VM), which is a technique to replicate/emulate real hardware architectures, and where the code responsible for managing virtual machines is called a *hypervisor* or *virtual machine monitor* [24]. This enables cloud providers to host several services inside separate virtual machines, each running an instance of an operating system and thereby utilizing their servers better since more applications can be packed into a single host without modifications to the application [6]. An example is the Xen hypervisor, which supports concurrent execution of up to a hundred operating systems [3].

Another way to use virtualization is to use *containers*, which is a method for isolating processes on a shared kernel [24]. A container consists of an isolated package that contains the application and its dependencies, and multiple applications can then share the same underlying kernel and common utilities isolated from each other [6].

2.1.2 The limitations of virtualization methods

However, there are some problems with using virtual machines. Normally, a general-purpose operating system like Windows or Linux is used where the application is run, and this generic software is initialized on each startup by using configuration files from storage [19]. Because the OS running is for general purpose, it will contain many utilities and binaries that are necessary for general use, for example, support for backward compatibility with other existing applications [20], multi-user authentication, advanced math libraries, support for multiple processes and virtual memory management [22].

One aspect that is negatively affected by this is boot time. To scale a service, one method is spawning new VMs when the load is high and then using load balancing to balance the requests out on several VMs. Once the load is low again, the VMs can be closed as they are no longer in use. However, general-purpose operating systems are not optimized for this, as all the generic software needs to be initialized each time by reading configuration files from each the VM boots up, which means that a load balancer might need to keep idle VMs running to deal with load spikes [19]. This means that the elasticity of using a cloud solution is worsened since there might be idle VMs that are running in the background to prepare for large increases and decreases in the number of requests.

Another concern is security. A general-purpose operating system will have wider attack surface, since there are many libraries and utilities to exploit, one such utility is the shell [6]. This enables the attacker to have many possibilities to exploit libraries and utilities of the VM, even though the components that might be exploited are not used or unnecessary for the application running in the VM. Another part of this is that the configuration of the VM might also be insecure, and it might also be difficult to account for all the generic software

that is part of the VM [22].

Even though the use of containers can counteract the problem with boot-time, since a new container with the application can be booted instead of an entire virtual machine, the problem of security is still an issue. First of all, it is still important to configure the container correctly, just as with virtual machines, or else there can be utilities that could be exploited. Second, exploits that are aimed at the kernel space will also be a bigger problem for Containers than VMs, as containers running on the same system will share the same kernel. Therefore, all containers running on the same machine will be compromised if the kernel is compromised through one of them; an example of such an exploit is the *Meltdown* exploit [18]. Here, virtual machines offer another layer of security.

2.1.3 Unikernels

To resolve this problem, another approach is the *unikernel* - a specialized single-purpose library operating (libOS) system that runs directly on a hypervisor [20]. The idea is that only the needed libraries and related configuration will be linked at compile-time, creating immutable and lightweight VMs that only use what is necessary for the application to run [6]. The library-operating system design consists of implementing libraries like device drivers and the network stack, which can then be linked directly with the application [20]. With this approach, the application and operating system are fused together to form a specialized single-purpose VM that has a smaller attack surface, a faster boot time, smaller binary size, and better runtime performance than other virtualization methods [19].

This concept is further supported by the fact that many services today are single-purpose, e.g., a database, or serving a website, etc. [19]. This means that the unikernel solution is a good fit in many cases since a general-purpose operating system's generic functionality is often unnecessary. Another way that this is made possible is that it is difficult for a libOS to support a wide range of real-world hardware. However, this is not a problem with unikernels since device drivers need only be implemented for the targeted hypervisor [20]. Unikernels, in that sense, take advantage of the fact that they are supposed to run inside of a hypervisor.

2.1.4 Unikernels in practice

An example of a unikernel project is MirageOS [21], which primarily uses the programming language OCaml [20]. MirageOS enables the development of creating unikernels in a high-level language, which enables programming in different paradigms, especially functional, and also provides type and memory safety, with a strong static type system and a garbage collector [20]. Using a modern high-order language has several benefits [19]:

- It gives a higher level of security, as many memory errors like integer and buffer overflow can be avoided through static type checking.
- It has automatic memory management, such that the programmer is relieved of the burden of managing memory correctly, which can lead to crashes and security issues.

- Functionality can be encapsulated into different modules, allowing the codebase to scale better, and the compiler can check the use of modules together.

This is very relevant when developing systems, such as unikernels with large code bases. In general, it allows for a higher level of developer productivity and better safety guarantees.

There are also other notable projects for unikernel development:

- *Unikraft* [32], an open-source unikernel development kit that supports many different languages. It aims to provide a highly configurable unikernel codebase and supports many languages and applications.
- *IncludeOS* [14] which uses C++ as the primary language for development.
- *Hermit-rs* [12] which uses Rust as the primary language. It is possible to develop applications in Rust, C/C++, Go and Fortran

2.2 Region-based memory management

Memory management is necessary for any programming language since computers have limited memory and thus must be recycled. Memory is allocated and deallocated to ensure that all the memory isn't used up and values are kept for as long as needed but not much longer. Typically, programming language memory will consist of a stack for statically sized values, e.g., ints, booleans, etc. Having just a stack is rather restrictive, hence the use of a dynamically sized memory management method. Such values include arrays, trees, etc. These values are independently allocated and deallocated when needed. Some languages offer explicit memory allocation and deallocation, e.g., C where the programmer may use `malloc` or `calloc` to allocate memory and `free` to deallocate it again. Such memory management is known to be difficult for the programmer and is error-prone. Other languages, e.g., Python, offer automatic memory management by a *garbage collector*. A garbage collector is a separate procedure from the application itself managed by the runtime to collect any values in the memory that are not in use. However, this carries an additional performance overhead.

2.2.1 Concept

An alternative to garbage collection is region-based memory management. In this memory management method, the values in a given program whose size exceeds a machine word (e.g., 8, 16, or 32 bits) are stored in *regions*. Each region is implemented as a linked list of large blocks of memory known as *region pages* [11]. Such a page should be large enough to serve multiple allocations, and its values include but are not limited to function closures and values of recursive types, such as lists and trees [29]. Regions are kept in a *region stack*, as seen in figure 2.1, where the current region maintains a pointer to the next free position. When a new region or expansion of an existing region is needed, a region page will be taken from the global list of free regions pages known as the *free list*. If a new region is required the region page will be put in a new region that is pushed onto the region stack, and if a

region should be expanded then the region page is appended to the list of region pages in the region. Once a region stack is no longer in use, it may be *popped* off the region stack and returned to the free list; thus, the region is deallocated.

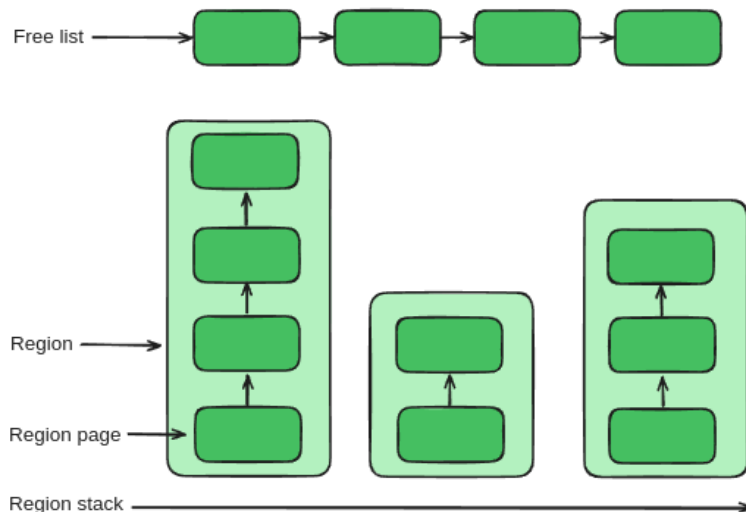


Figure 2.1: Overview of regions with pages in a region stack and the global free list

This greatly surpasses the efficiency of a garbage collector because the deallocation in region-based memory management only has to do simple pointer arithmetic to free an entire page instead of the individual freeing of each value in a garbage collector [16].

This, however, poses a problem of unsafe behavior. Suppose a region, r_0 , contains a region page with a pointer to a region page in the region r_1 , and now r_1 is deallocated. This will leave a dangling pointer in r_0 , and we risk dereferencing it and crashing. To solve this problem, each region will maintain a *reference count*, and it will only be possible to deallocate the region if and only if no other regions have any pointers to the region itself.

Two different types of regions exist to handle the allocation and lifetimes: finite regions and infinite regions [30]. Finite regions are defined as regions that have a finite upper bound of one value, and infinite regions are all other regions. Furthermore, the size and lifetime of finite regions can be determined at compile time rather than at runtime of infinite regions. An infinite region can grow dynamically, and an arbitrary number of regions can exist at runtime [29].

In region-based memory management, some regions are known as *global*, a specific type of region [28]. This region often lives for the entire lifetime of a program. It is typically used for some data that is used throughout the entire program, e.g., global states. Since global regions have just one allocation, they reduce the overhead with frequent allocation, thus improving efficiency some cases.

However, the allocation and deallocation of regions are determined at compile time by a type-based analysis of the program known as *region inference* [28]. This works by first anno-

tating the program with region annotations. This could, for instance, be a variable of some type, and that type should belong to a certain region. Secondly, a set of constraints on how the regions are related to each other is constructed. This includes which regions should outlive which. These constraints are then solved, meaning the assignment of regions is created to satisfy all the constraints previously created. Finally, allocation and deallocation points are created based on the solved constraints. This ensures that regions are correctly allocated and released at the correct times.

It is worth noting that this project uses the MLKit compiler. This compiler is a compiler to the Standard ML programming language (SML), including SML with region-based memory management and region inference [31].

2.2.2 Benefits

Region-based memory management has multiple benefits over explicit and implicit garbage collection memory management. These include:

- Region-based memory management is not explicit with the use of region-inference, meaning simpler code that is easier to read and write and has fewer errors since it is done automatically and not by a human programmer.
- Since regions use pointer arithmetic to expand and free an entire region, deallocation can be made in constant time. This removes the overhead of freeing individual values like in garbage collection. An example is if a program is to free a tree data structure. Here garbage collection would have to traverse and free each node and leaf in the tree individually but region-based memory management could just free the region the tree is stored in.
- Garbage collection will have to pause the program to clean up each value, which could lead to potential long halts, which greatly reduce execution speed. Region-based memory management, on the other hand, can avoid long halts since deallocation is moving pointers such that the region returns to the free list.
- Region-based memory management is beneficial in concurrent programs because the memory exists in different regions, and the concurrent threads can access different regions without interfering with each other.
- Improved locality (spatial and temporal) due to values and objects in the same region being allocated close to each other, thus fewer and shorter traversals of the heap than with garbage collection.

Chapter 3

Design

This chapter describes the project’s design. The first section will explain how the network library was designed in terms of structure and what protocols and parts of them have been chosen for implementation. Next, the architecture design for the library’s underlying support on each platform will be described, including which platforms will be supported and how, both in terms of networking and porting the language runtime.

3.1 Unikernel libraries

The code for this project has been created in modules for several reasons.

- **Modularity** makes it easier to understand the code and/or swap parts. For instance, if one wants to swap the implementation of a network protocol in the network stack, it is easier to do so if each of the protocols has its own module.
- **Abstraction**, which hides some functionalities’ complexity. This helps make it easier for other developers to use the code without fully understanding a rather large codebase.
- **Reusability**, which allows other developers to use code from the modules across the project, i.e., minimizes the amount of duplicate code.
- **Maintainability** allows for having parts of the code encapsulated and abstracted away from the application itself, which will ease the task of maintaining each library without affecting the application itself (assuming no interface changes).

Each of the libraries has a signature of publicly accessible functions. However, it is likely the unikernel developer will only ever use the functions declared in the `network` library as this is where the port binding and listen functions are declared (these methods will be discussed in detail in section 4.1). The `network` library is dependent on the other libraries to further abstract away some of the complexity and make it easier for the unikernel developer. The parts that are abstracted away are decoding, encoding, and sending packets, which are all packaged into functions that are easy to call. Figure 3.1 is an overview of the libraries, and here, arrows indicate which libraries depend on which. The arrow direction means that it

receives one or more functions from the library of where the arrow came from, e.g., IPv4 depends on **Utilities**.

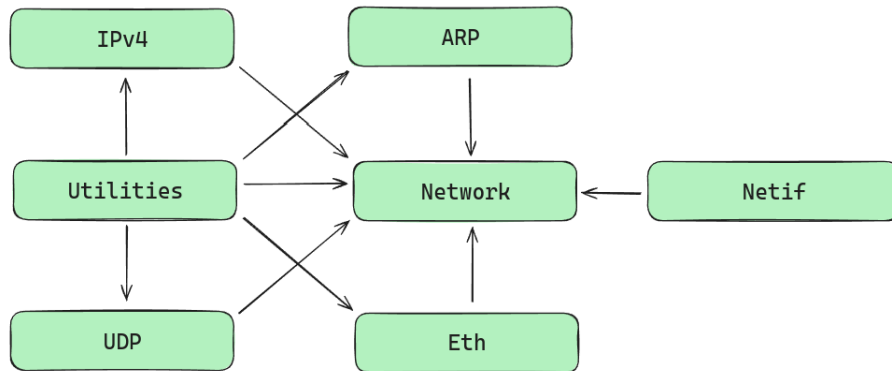


Figure 3.1: Dependency graph of the network libraries

The modules are made with signature files (`.sig`) and ML basis files (`.mlb`), which allow for some functions to be public and some to be private to only the module itself. To use the public function declared in some module A’s signature file, one must include the `.mlb` file of A in the `.mlb` file of B as a dependency.

3.1.1 Network protocols

A network stack is needed for the unikernel to handle incoming messages and reply to them over the network. Figure 3.2 is an overview of the layers needed in an internet stack [1] as well as some protocols that are commonly used to implement a layer. In this project, the protocols filled in with solid green have been chosen to be used, and the protocols filled with a cross-hatch pattern are other commonly used protocols that could be implemented in the future (see section 6). The goal with this internet stack is for the unikernel to be able to send and receive packets over a network. As mentioned in section 3.1 and seen in figure 3.1, each protocol resides within its own module, and in this section, we will go over each of the chosen protocols.

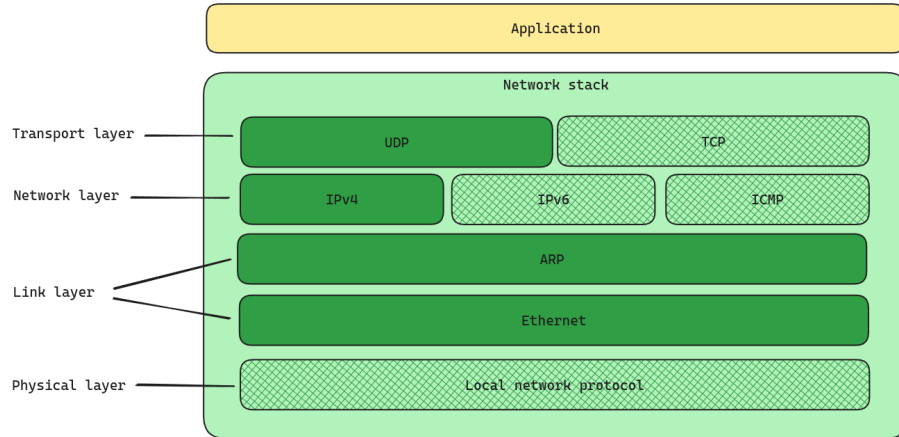


Figure 3.2: four layer network stack

Ethernet

The ethernet protocol is a link layer protocol, which means it can transport data on an ethernet link with the frame itself being the payload, i.e., it ties the physical layers together with the software. This protocol is necessary to communicate with the physical layer in the network stack. Furthermore, IEEE ensures that devices from different manufacturers can communicate with each other, thus making it an immensely interoperable choice for the link layer [13].

ARP

The Address Resolution Protocol (ARP) is also a link layer, although it is slightly above the ethernet in the internet protocol stack (as seen in figure 3.2). The ARP protocol provides mechanisms to connect the IP (internet protocol) addresses to the MAC (Medium Access Control) address on a local network. This is necessary for a network stack as the IP addresses are used in the network layer of the network stack, and the MAC address is used in the link layer. The ARP then acts as the bridge between the two layers. The only notable alternative to ARP is the Neighbor Discovery Protocol (NDP) [26], but since this project is primarily going to be based on IPv4, there is no need for a more complicated protocol such as NDP; hence, ARP was chosen.

IPv4

The Internet Protocol Version 4 (IPv4) implements the network layer in the network stack [15]. The protocol is used to identify other machines on the internet using a 32-bit address system. An address is made of 4 2^8 numbers often separated by a '.' when written down by humans, e.g., 127.0.0.1. The protocol is made up of a 20-byte header divided into 12 fields and a 65536-byte payload [1]. The payload will often be a protocol that implements the transport layer with a large payload, e.g., a file. If a payload size greater than 65536 is needed, the protocol features a mechanism to break a packet up into the appropriate size and send it with a legal size - this is known as fragmentation. In this project, the IPv4 protocol

has been chosen as the network layer on account of its widespread adoption throughout the internet, meaning most machines support the protocol. Furthermore, it is fairly simple to implement as opposed to its successor IPv6.

UDP

The User Datagram Protocol (UDP) is a widely used communications protocol that implements the transport layer in the network stack. The protocol is made up of an 8-byte header and divided into 4 fields and a payload size of 1500 bytes. This protocol allows for one machine to send data to another. Here, the protocol can send packets that carry a wide range of different types of data, e.g., a DNS query, online music streaming data, etc. [10]. The header for the UDP is only 8 bytes and contains 4 fields. The rest of the packet is the payload [23]. UDP implements a *fire and forget* strategy, which means the protocol forgets all about it as soon as a packet has been sent. This could be fatal if a packet is lost in transit. At the transport layer, TCP and UDP are the commonly used protocols, and TCP is generally considered the safer option as it uses a three-way handshake to ensure no packets get lost (in the event of a lost packet, a recovery packet will be sent with the missing data - this is known as retransmission). However, UDP has been chosen for this project as it provides a simpler design, meaning it is easier to implement. Furthermore, it has reduced overhead as opposed to TCP since it does not require a three-way handshake to ensure every packet gets delivered.

3.2 Architecture design

This section will go through how the network library will be supported and built for different platforms, and describe how it will be supported on a local Linux machine for testing the application, and for building it as a unikernel to run on the Xen hypervisor.

The overall goal for this design is to support the network library so that it can be compiled into a unikernel and run on a target hypervisor. The design has the following goals:

1. **Minimal platform-dependent codebase:** The platform dependent code should be minimal. This means that, if possible, functionality will be implemented platform-independently. This will mean it will be easier to reason about and debug the platform-specific code.
2. **Simple and portable:** It should be easy to support different platforms by only supporting the essential functionality needed for creating the services. This means that functionality might be restricted.
3. **Consistent across platforms:** The behavior on different platforms should be as similar as possible, such that the application can be developed in a development environment without running into problems when it is eventually compiled to a unikernel and run on the target hypervisor.

4. **Local development:** Although the aim is for the services to run on a target hypervisor as a unikernel, the service should be able to be run in a local development environment as well, such that it is possible to test the functionality of the service without the need to run it on the target hypervisor for each iteration.

3.2.1 Language runtime

The chosen language for this project is Standard ML (SML), and the compiler that will be used is the MLKit compiler. The MLKit compiler supports the development of SML applications using region-memory management and region inference [31]. The MLKit compiler-backend currently supports the x86 architecture, and this project will also target x86.

MLKit supports both tools for profiling the use of the regions and also offers support for incremental garbage collection. However, the runtime will be minimized, so these and other parts of MLKit will not be supported in the minimized version. While support for garbage collection could be a nice addition, the project will use region inference as the only memory-management method to better illustrate the viability of using regions in unikernels. Thus, garbage collection will not be supported by the minimized runtime. Many other parts of the runtime that will not be supported will be related to OS-specific functionality, like reading files.

Besides minimizing the runtime, each platform will also have separate versions of the minimized runtime. The intention is that these versions of the runtime should be very similar in behaviour to follow the goal of consistency. However, it might be difficult to translate directly in some cases.

Because of the minimization of the runtime, some parts of the SML basis library will also not be supported. However, the minimization should support the parts of the library that are essential for creating services.

3.2.2 Platform support

Overview

To make it easier to develop unikernels, the design provides support for developing the service in a local development environment, which can be done on a Linux machine, and then compile it to a unikernel for the target hypervisor, in this case, Xen (see section 3.2.2). This design enables the developer to quickly debug and test the unikernel on their local Linux machine without the need to set up a target environment. The unikernel can then be created and transferred to Xen when the service is ready. This is similar to Mirage-OS' developing environment, where the application is tested and developed on a local machine, and it can then be compiled to a standalone unikernel to run on a hypervisor [19].

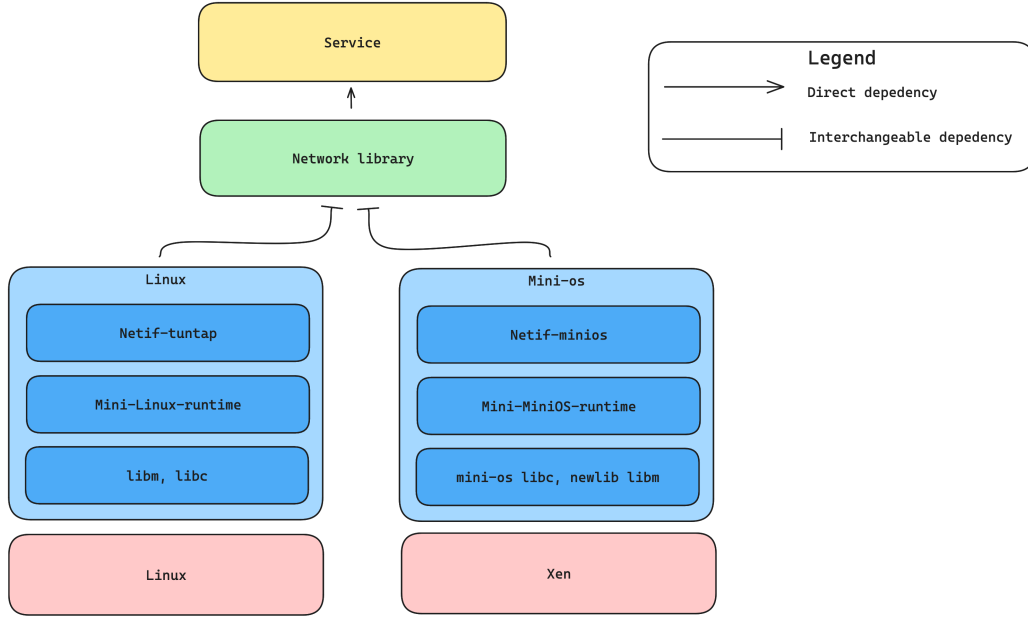


Figure 3.3: The service at the top uses the network library to establish connections. The Network library can then either use the code for running on Linux or to create a unikernel for running on the Xen hypervisor.

As shown in figure 3.3, the network library as well as the application will stay the same. This will include the code for the service and all the code for the network protocols. Then, the platform-dependent code for Linux and Xen will contain the functionality for receiving and sending ethernet frames on that platform, as well as a platform-specific language-runtime.

Linux

The chosen development platform is Linux. Linux supports many network-related development tools and runs on x86 architecture. However, the main choice for using Linux is that the kernel module TUN/TAP could be used for the project, which provides support for receiving and transmitting IP packets with tun or ethernet frames with tap to userspace programs on Linux, working like a virtual ethernet device [17]. In this case, it is used in the project to read ethernet frames with tap. Using TUN/TAP instead of unix sockets gives the library code more control over the network stack. The network stack, which will be written entirely in SML, will behave the same on both platforms, assuming that the runtimes for the platforms behave similarly, such that there are fewer inconsistencies when moving the application from Linux to Xen.

Xen

The Xen hypervisor is the target for this project, as it supports x86 and many other unikernel projects, for example MirageOS. It also provides the Mini-OS kernel as a starting point for unikernel development [36].

Mini-OS is a tiny OS kernel for Xen distributed with the Xen hypervisor sources that can be used to develop unikernels [34]. Mini-OS, among other things, provides networking support, a memory and page allocator, and minimal libc support [35]. The kernel is linked together with the application code to compile a unikernel that can be run on Xen. This is done by changing the MLKit runtime, such that it includes code from Mini-OS, instead of relying on shared libraries like libc and libm. When Mini-OS is run, it will call the main function defined in the runtime, and the SML code is then executed.

3.2.3 Network stack

The networking protocol implementations are entirely developed with standard ML. The application can use the code to listen and respond to connections. As explained in section 3.1, it provides implementations of the Ethernet, ARP, IPv4, and UDP protocols. The application code can freely use these implementations and they are entirely independent of the underlying code on each platform that reads ethernet frames, which will have different implementations depending on the platforms. Implementing the network stack with SML, instead of using either sockets or an external implementation of the network stack, will make it easier to port to different systems and minimize the amount of the code that is platform-dependent. In this way, the only part that needs to be implemented for a platform is how to receive and send ethernet frames.

An underlying network interface, which is the code that will be used to receive and send ethernet frames, will be designed to support this, and its implementation will depend on the platform. This will mean that the service can call SML functions to receive and send ethernet frames, but the different implementations of the networking interfacing code can be changed, depending on the platform. These implementations will utilize the C-interop that is available in MLKit, such that implementations of the network interface will be implemented in C but can be called from SML.

3.2.4 Differences between platforms

While the intention is that the behavior on different platforms should mostly be identical and consistent, there are some unavoidable differences.

Firstly, the way that ethernet frames are received and sent will be different. This will mean that the underlying networking C-code receiving and sending ethernet frames will not have the same behavior on the two platforms. However, the intention is that a developer can be fairly certain that if it works in their local Linux machine, then if anything unexpected happens while running on Xen, there is a high probability that it is because of the target-dependent networking code since this is where the differences are most visible.

The runtimes for the two environments will also be somewhat different. The two mini-runtimes for Linux and Xen are different since when compiling to Linux, it will use shared libraries like libc and libm, while the build for Xen will use equivalent calls from Mini-OS, and in this case also an external implementation libm. The external implementation of libm

is from [33], which implements static version of libc and libm. There might be deviations in the implementations, but replaced calls are expected to behave similarly, such that there is a lower likelihood of inconsistencies between the two platforms regarding the runtime.

Chapter 4

Implementation

This chapter outlines the implementation of the project, and it aims to explain how the main areas intend to function. The first section covers the network library and how the network protocols are utilized to implement an entire network stack. Next, the network protocols in the project's network stack are covered at each layer. Next, the differences of the supported platforms (Unix and Xen) are covered. Finally, the minimization of the runtime system will be explained in greater detail. Throughout the entire implementation, functional design has been kept in mind to streamline all processes. This includes consistency in the libraries, clarity in their purposes, and a simple pipe operator to streamline the code by chaining operations together. The syntax for the pipe operator is `|>`. The code is publicly available on GitHub [here](#).

4.1 Network Library

To create a unikernel, one can write regular functions and combine them with the libraries in this project to communicate over a network. This will be referred to as a *service*. To create such a service, one must bind a callback function to a port on the IP address and call a function that will listen to incoming messages. In this section, these functions will be discussed.

4.1.1 User layer

A user in this context is not the end user who makes requests to the unikernels but rather the developer who implements the unikernel service.

Binding

A function called `bindUDP` binds a port to a callback function using UDP on the transport layer. Figure 4.1 is a simple echo service, and here, the port 8080 is bound to the identity function, i.e., it returns what it is given as input (see section 5.2 for other service examples). Here, the service is the identity function.

This function is in the `network` library, and it prepends a tuple of the port with the callback function to a global list, which the `listen` function (see below) will use appropriately. It is

```

val _ = (
  bindUDP 8080 (fn data => data);
  listen ()
)

```

Figure 4.1: Example echo service

possible to call the bind function on an arbitrary number of ports, thus serving many ports with many services (NB only one call to listen is sufficient for any number of port-to-function bindings).

Callback

The callback function (as seen in line two of figure 4.1) is the service of the unikernel itself. It receives a string as input extracted from a UDP packet sent to the port to which the callback function is bound. The function will also return a string; hence the type annotation for `bindUDP` is:

$$\text{int} \rightarrow (\text{string} \rightarrow \text{string}) \rightarrow \text{unit}$$

Listen

The `network` library also includes a `listen` function, which continuously reads and handles internet packets from the `netif` interface. This is done in the following order:

1. Read the raw ethernet frame from the `netif` interface using the `netif` library and convert it to a string.
2. Decode the ethernet frame into a tuple of the ethernet header and the ethernet payload.
3. Determine the ethertype. The incoming packet may either be an ARP packet or an IPv4 packet, and the appropriate handle function for the ethertype is called (see sections below)
4. Recursively calls itself, allowing for more packets to be handled.
5. Should an error occur, it is handled by printing an error message and the `listen` function is recursively called regardless to continue the handling of more packets.

4.1.2 Handling packets

This section describes the functionality of handling incoming packets at the layers of the network stack.

Handling ARP

To handle the incoming ARP packets appropriately, the `handleArp` function is used within the `network` library. This function takes an ethernet frame as well as a header. Firstly, it decodes the ARP packet within the ethernet frame using the decode function defined in the ARP module. Secondly, it pattern matches the decoded ARP packet to see if it was decoded successfully, i.e., the received data was an ARP packet. Thirdly, it creates a new ARP packet to use as the reply packet. This packet has some hardcoded fields, e.g., the hardware type, hardware length, etc. The rest of the fields are extracted from the header, which was the argument for the function. Finally, it sends the ARP reply packet back to the sender using the `ethSend` function, which is also defined in the `network` module (see below).

Handling IPv4

To handle the incoming IPv4 packets appropriately, the `handleIPv4` function is used within the `network` library, and this function also takes an ethernet payload as well as a header as its arguments. Much like the `handleARP` function above, it starts by decoding the IPv4 packet into the header, and here, it constructs a tuple containing the header and the payload of the IPv4 packet. As mentioned in 3.1.1 the IPv4 protocol allows for fragmentation and this has to be handled properly. One of the fields, `flags`, in the IPv4 header is responsible for representing if the packet is fragmented, and another, `fragment offset`, is responsible for representing the offset, i.e., where the packet belongs. The `flags` are made of 3 bits, and the meanings them are as follows [15]:

bit 0: is a reserved bit and must always be zero

bit 1: represents not fragmented (DF)

bit 2: represents more fragments (MF)

If the `flags` field is equal to 2 and the `fragment offset` is equal to 0, it is trivial that the packet is not fragmented and the payload is handed over to the `handleUDP` (see below) function along with the destination MAC address and IPv4 header. If the `flags` field, on the other hand, indicates that the message is fragmented, this is handled by inserting the payload into the `fragmentBuffer`. This is done by calling the `addFragment` function (see below) on the header and payload. If the `flags` of that header is 0, i.e., there are no more fragments, and the `initAssembling` function can be called followed by the `assemblePacket` function. If the `flags` indicate more fragments are coming, it can return none and wait for the remaining fragments. Some fields are hardcoded and never checked to simplify the implementation, e.g., IP address.

The utilized data structures and variables are the following:

1. `packetID` is a datatype consisting of the IP address, the `identity` field of the IPv4 header and the protocol e.g. ICMP.
2. `fragment` is a datatype consisting of the payload as a string and the offset of the particular fragment.

3. `fragmentBuffer` is a list of type `(packetID * (fragment list ref)) list ref` and this is used to buffer/hold the incoming IPv4 fragments.
4. `assemblingList` is a list of type `(packetID * (char array)) list ref` and this is used to assemble the packets. Since fragments won't necessarily arrive in order, the payloads are kept in a list until the last fragment is received and the payload size is known. Once the last fragment is received the assembling can begin.

And the helper functions behave as follows:

1. `pktIDCmp` is a function which compares the IP address, identity, and protocol fields within the IPv4 header. This function returns an appropriate boolean value.
2. `addFragment` adds the IP fragment to the `fragmentBuffer`, which is used to assemble the fragments at a later time. This is done via `pktIDCmp` function, and if the fragments of the same identity already exist in `fragmentBuffer` and if so, the fragment is appended to the list that contains the other related fragments as it belongs there. If it doesn't exist in the fragment buffer list, it moves on to check if it exists in `assemblingList`, and if it does, the `updatePacketArray` function is called on the incoming packet. If the identity of the fragment hasn't been seen before, a new entry for fragments is created in the `fragmentBuffer`.
3. `initAssembling` initializes the assembly of a fragmented IP packet by first initializing a packet's identity from the header. Second, it creates an array with an appropriate size to fit all the fragments and checks if the `fragmentBuffer` contains any fragments of the same packet. If fragments of the same packet are found, it copies the fragments to the new array and removes them from the `fragmentBuffer` list. Finally, the new list is added to the `assemblingList`.
4. `updatePacketArray` recursively updates an array containing the fragmented payloads of a message.
5. `assemblePacket` assembles a whole packet from its fragmented pieces, which is done by first initializing a `packetID` from its given argument, which is an IPv4 header. Then, it searches through the `assemblingList` for the array containing the fragments of the same packet ID. If such an array is found, it is removed from the `assemblingList`, and the char array is converted into a string, which is now the entire packet payload made of the fragmented payloads.

Handling UDP

The `handleUDP` function is used within the `network` module to handle the UDP packets appropriately. This function takes a destination MAC address, an IPv4 header, and a payload as its arguments. The payload of the UDP packet is what the callback function in the `bindUDP` will receive. This is extracted by decoding the IPv4 payload with the `decode` function in the UDP module. If the port in the UDP header is mapped to a callback function,

that function is executed with the UDP payload as its argument, and the result is encoded as a UDP packet using the `encode` function in the UDP module. If the port is not mapped to a callback function, a UDP packet is still encoded, but the payload is *"Port is not mapped to a function."* to ensure the request gets an appropriate reply. Once encoded, the packet is sent via the `ipv4Send` function (see below).

4.1.3 Sending packets

This section will describe the functionality that handles outgoing packets at the IPv4 and ethernet levels.

Sending IPv4 packets

To send IPv4 packets, the `ipv4Send` function is used. Much like the `handleIPv4` function, packets may be fragmented. It must fragment if the payload is greater than the Maximum Transmission Unit (MTU) of 1500 bytes. If the payload is less than the MTU, there is no need to fragment, and the IPv4 packet can be encoded using the `encode` function in the IPv4 module and sent with the `ethSend` function as previously mentioned. If it is, however, greater than the MTU, it must be fragmented by recursively calling a helper function `sendFragments` by creating an IPv4 header for the current fragment (with appropriate id, offset, flags, and payload), extracting a substring of the payload that's 1500 bytes and sending the fragments with `ethSend`. Some of the fields in the IPv4 header can be inferred from the received packet, e.g., `source address`, and other fields can be calculated with the payload, e.g., `total length`. The rest of the fields are hard coded, e.g. `time to live`, which is set at 128 seconds.

Sending ethernet packets

The `ethSend` function constructs and sends an ethernet frame. This is done by firstly encoding the header using the `encode` function in the `ethernet` module, then converting the header into a byte list, and finally calling `netif`'s `send` function to send the packet on its way. The header is converted using the `toByteList` function in the `utility` library.

4.1.4 Network profiling

A separate implementation of the network library called `networkProflib` replaces `networklib`, which has also been made for use with region profiling. This implementation introduces functions to set variables for generating data that can be used to simulate networking. This includes the following functions:

1. `setProfData` is used to set what data should be used for profiling.
2. `setRuns` is used to set how many times the `listen` function should run.
3. `setPort` is used to set which port should be used.

4. `generateProfData` after the relevant variables have been set, this function can be called to generate the data to use. It will function similarly to when a request is handled in `networklib`; instead of sending the encoded data to a network interface, the fragments generated from the packet will be placed inside the `simPktArr` array.

The normal `receive` call from `netiflib` is replaced, such that the fragments from the `simPktArr` array are read instead, by taking one element at a time from the array, and when the end is reached, it wraps around and reads the array again.

For region profiling, the minimized version of the runtime is not used. Instead, the normal MLKit runtime is used, which contains the runtime for profiling, and the flag `-no_gc` is used to ensure that garbage collection is not used, just like in the minimized runtime. This means that it will not be the same runtime that is used for profiling, however if the service works on the minimized version, then the behavior should be similar to the one with profiling, since the functionality has not been changed, but rather some functionality has been stripped away.

One can then import `networkProflib` instead of `networklib` in the `.mlb`-file of the service and then set up profiling inside the main file of the service. The implementation here is a copy of the current version of `networklib` with additional changes and, therefore, is a temporary solution.

4.2 Network protocols

To promote the use of functional design, the protocols in the network stack are implemented with similar interfaces and implementations, thus creating uniformity across them. This uniformity is possible because the protocols all need to be decoded (to separate header into fields and payload) and encoded (combine header fields and payload) in a similar fashion to comply with how the `tap` reads and writes to the network. Below is an overview of how similar functions are implemented for the `ARP`, `ethernet`, `IPv4`, and `UDP` protocols and then how the protocols are implemented besides the similar functions.

Header

Each protocol has a header with fields, and these fields are kept in a user-defined SML datatype. Some of the fields are *non-conventional* datatype, e.g., the ethernet frame ether type is `IPv4`, which is another user-defined datatype to ease matching on such fields. Some of the protocols have implemented helper functions to convert between strings, ints, and the custom datatype, e.g., in the `IPv4` protocol, the protocol `UDP` has the integer value of 17 and the string value of "UDP". Figure 4.2 is the `header` datatype for the `UDP` (see section 4.2 for header details).

```

datatype header = Header of {
    source_port: int,
    dest_port: int,
    length : int,
    checksum: int
}

```

Figure 4.2: Header datatype for the UDP

Decode

When the `tap` reads an incoming message, it will be received as a string in SML. The string is passed to the appropriate protocol library where a `decode` function is residing, and this function will put the appropriate fields and datatype into the header datatype of the specific protocol. The three implemented protocols that carry a payload, `Ethernet`, `IPv4`, and `UDP`, will return a tuple consisting of the header and the payload as a string. Figure 4.3 is the `decode` function for the UDP.

```

fun decode s = (Header {
    source_port = String.substring (s, 0, 2) |> convertRawBytes,
    dest_port = String.substring (s, 2, 2) |> convertRawBytes,
    length = String.substring (s, 4, 2) |> convertRawBytes,
    checksum = String.substring (s, 6, 2) |> convertRawBytes
}, String.extract (s, 8, NONE))

```

Figure 4.3: `decode` function for the UDP

Encode

When the `tap` writes an outgoing message, it must be a string of raw bytes. Each protocol then receives the appropriate header datatype and uses one or more helper functions from the utility library, which can convert ints, int lists, strings, etc, into a raw byte string. In the `IPv4` module, the encode function also calculates the checksum with the `ipv4Checksum` function defined in its module (see section 4.2). Figure 4.4 is the `encode` function for the UDP.

```

fun encode (Header{length,source_port,dest_port,checksum}) data =
    (intToRawbyteString source_port 2) ^
    (intToRawbyteString dest_port 2) ^
    (intToRawbyteString (String.size data + 8) 2) ^
    (intToRawbyteString 0 2) ^
    data

```

Figure 4.4: `encode` function for the UDP

toString

Each of the protocols has a function to convert the fields within the header to a string with the name of the field so it is easy to print for debugging purposes. When making an application, one can turn on the printing of the incoming and outgoing messages by calling the function `logOn()`. Figure 4.5 is the `toString` function for the UDP.

```
fun toString (Header {
    source_port,
    dest_port,
    length,
    checksum
}) =
    "\n-- UDP INFO --\n" ^
    "Source port: " ^ Int.toString source_port ^ "\n" ^
    "Destination port: " ^ Int.toString dest_port ^ "\n" ^
    "UDP length: " ^ Int.toString length ^ "\n" ^
    "Checksum: " ^ Int.toString checksum ^ "\n"
```

Figure 4.5: `toString` function for the UDP

Ethernet

The header of the ethernet frame is seen in table 4.1. Its header consists of three fields and a total size of 8 bytes. Besides the functions described in section 4.2, the ethernet frame module contains functions to convert the ether type to strings and integers, as well as integers to ether types, e.g., 2054 is ARP.

Field	Data type	Size
Ether type	ARP \vee IPv4 \vee IPv6	2 bytes
Destination MAC address	int list	6 bytes
Source MAC address	int list	6 bytes

Table 4.1: Ethernet frame header

ARP

The ARP header is seen in table 4.2, and it consists of nine fields and has a size of 28 bytes. Besides the functions described in section 4.2 the ARP module contains functions to convert the ARP operation to strings and integers, as well as integers to ether types e.g. 1 is `Request`.

Field	Data type	Size
Hardware type	int	2 bytes
Protocol type	int	2 bytes
Hardware address length	int	1 byte
Protocol address length	int	1 byte
Opcode	Request ∨ Reply	2 bytes
Sender IP address	int list	6 bytes
Sender MAC address	int list	4 bytes
Target IP address	int list	6 bytes
Target MAC address	int list	4 bytes

Table 4.2: ARP header

IPv4

The IPv4 header is seen in table 4.3, and it consists of 13 fields 23 bytes. Besides the functions described in section 4.2, the IPv4 module contains functions to convert the protocol type (e.g. `ICMP`) to strings and integers, as well as integers to protocols, e.g. 1 is `ICMP`.

Field	Data type	size
Version	int	1 byte
Internet header length	int	1 byte
Differentiated services code point	int	1 byte
Explicit congestion notification	int	1 byte
Total length	int	2 bytes
Identification	int	2 bytes
Flags	int	1 byte
Fragment offset	int	2 bytes
Time to live	int	1 byte
Protocol	ICMP ∨ TCP ∨ UDP	1 byte
Header checksum	int	2 bytes
Source address	int list	4 bytes
Destination address	int list	4 bytes

Table 4.3: IPv4 header

Furthermore, it contains a function, `ipv4Checksum`, to calculate the checksum by summing over all the fields and adding the *carry* (overflow from sum) and then bitwise inverting the result to the final checksum.

To detect if a packet is fragmented, a helper function, `isFragmented`, is defined. Here, the `flags` field in the IPv4 header is inspected and returns a boolean value appropriate to whether the packet is fragmented.

UDP

The header of the UDP is seen in table 4.4, consisting of four fields of eight bytes. This module contains no additional functionality besides the functions described in section 4.2.

Field	Data type	Size
Source port	int	2 bytes
Destination port	int	2 bytes
Length	int	2 bytes
Checksum	int	2 bytes

Table 4.4: UDP header

4.3 Platform implementations

Each platform (Unix and Xen) has its own version of the minimized runtime, and besides this, they also have different netifs, as discussed in section 3.2. In figure 4.6, the dependencies can be seen between the different components of the library. The reason the runtime for local development is named the *Unix* runtime, is that there is potential for it to work on other Unix systems, however in this project it has only been tested on Linux. The runtimes and the implementations of netif will be affected since these will need to replace calls to libc and libm with calls to either Mini-OS' libc implementation or the newlib libm implementation. The SML basis library will be affected by the change of runtime, and the parts that can be implemented here depend on what can be implemented using Mini-OS. The network library will then depend on the SML basis library and the netif implementation. The intention is that the application itself is then only dependent on the network library.

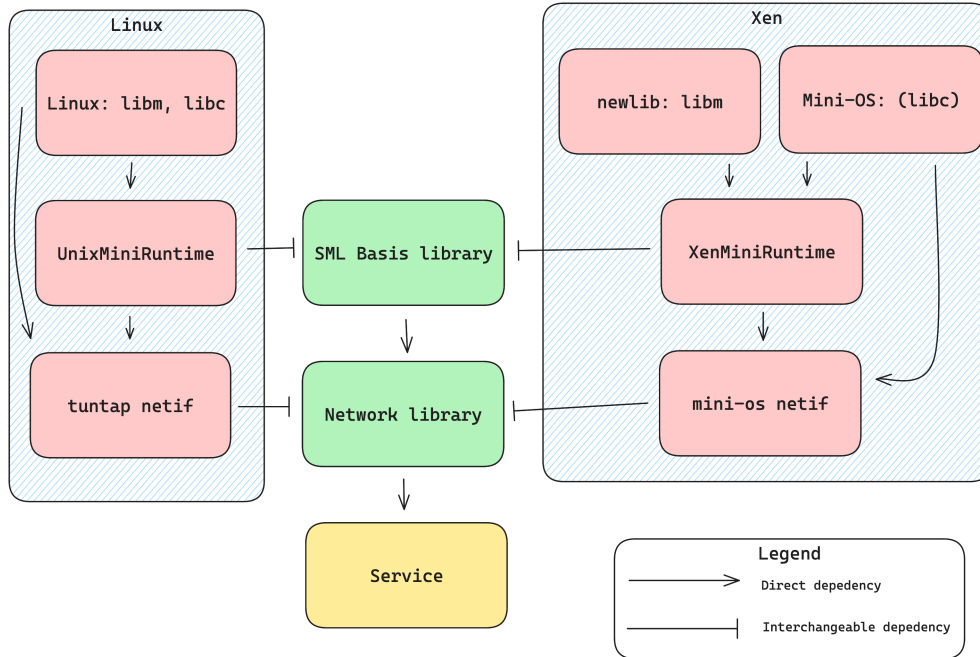


Figure 4.6: This figure shows that the SML basis library and the network library will be the same for both platforms but will use different runtimes and networking functions.

4.3.1 Linux

Building

When building for Linux, the end result is an executable that can be run directly on a local Linux machine. To build an application for Linux, the command

```
$ make t=unix application_name-app
```

can be used. This will first build the Unix runtime, then build the object file `libtuntaplib.o` that contains the netif code, and then finally compile the application to an executable using the MLKit compiler.

Running

Before running the application on a local unix machine, the network must be configured correctly. To do this, the `$ make setup` command can be used. This will add the tun kernel module and then set up a tap device that can be interacted with using the IP-address `10.0.0.2`. The executable can now be run, and communicated with using a networking utility such as netcat.

Network interface

When reading for the first time with the `receive` call from the application, if the global `tapfd` file descriptor has not been set yet, as can be seen on line 4 of figure 4.7, then it

will call function to set up the file descriptor. First, the `tun_alloc` function opens a tap-device, and a file descriptor is returned that will be used for communicating through the tap device. Second, the `setup` function is run to ensure that the interface is configured correctly

```

1  int tapfd = -1;
2  String Receive(int addr, Region str_r, Context ctx) {
3
4      if (tapfd == -1) {
5          char tap_name[IFNAMSIZ];
6
7          strcpy(tap_name, "tap0");
8          tapfd = tun_alloc(tap_name, IFF_TAP);  /* tap interface */
9
10         if (tapfd == -1) return NULL;
11
12         setup(tap_name);
13     }
14
15     char buf[MTU]; // MTU + 18 (the 18 bytes are header and frame check
16     ↪ sequence)
17     ssize_t bytesRead = read(tapfd, buf, MTU);
18
19     // Null-terminate the buffer
20     buf[bytesRead] = '\0';
21
22     return toMLString(str_r, buf+4, bytesRead-4);
23 }

```

Figure 4.7: The code for receiving ethernet frames with tuntap.

When the tap device is set up, and `tapfd` has been configured, the application is ready to read and write ethernet frames. To read from the tap device, the system-call `read` is used on `tapfd`, with a maximum of the Maximum Transmission Unit (MTU), and the output is placed into `buf`. To return the buffers content to SML, the `toMLString` function is called, which allocates a `StringDesc` struct, copies the contents of the buffer into the `data` member of the `StringDesc` struct, which can now be returned to the original call to `receive` from SML. One note is that reading the etherframe includes 4 bytes at the start of the frame, which are bytes added by tuntap. These include 2 bytes for flags, and 2 bytes for the protocol (the ethertype) [17]. These bytes are not used, and to make the interface more simple these are skipped, as can be seen on line 21 of figure 4.7, it adds 4 to the `buf` pointer when calling `toStringML`, so only the raw frame data is returned.

However, for writing frames back to the device, the implementation is different, since there were difficulties converting an ML string to a char array in C, the function for writing `send` instead accepts a list of integers, that represents the bytes of the string. Functions from the runtime like `isCons` are then used to go through the list, convert the SML-integers into c integers, and then copy them over to a buffer starting from the fifth index, as the first 4 bytes are reserved for tuntap related fields. Then, after the raw frame data has been copied over, the first two bytes of the buffer are left to be zero for the flag field in tuntap, and the ethernet bytes are copied over for the protocol field in tuntap. Then, the buffer is ready to be written to the tuntap device with a `write` call on `tapfd`.

4.3.2 Xen

Building

For Xen, the goal is to create one file that contains the kernel code, with the SML application statically linked inside, such that the kernel will call the runtime and begin execution of the SML application.

To compile the application that will later be statically linked to the Mini-OS kernel, the command

```
$ make t=xen application_name-app
```

First, the runtime that is aimed at Mini-OS is compiled. Next, the Mini-OS C-code is compiled, containing the code that will set up networking and call the runtime main function. The last step is then to compile the specific application. In this step, the MLKit compiler is called with the flags `-objs` and `-no_delete_target_files`, which will produce a file that contains paths to all the object files that are used in the SML application, including object files for each of the libraries, the minimized SML basis library and the runtime archive file. All related object files are then placed inside a `build` directory, and all related archive files are also extracted into the `build` directory, this includes `runtime.a` and `libm.a`. The last step is to create an archive containing all the relevant object files, which is named `app.a`.

The next is to build the Mini-OS, and linking in the `app.a` archive. The Mini-OS sources need to be installed, and the make variable needs to be set to the path to the Mini-OS directory. To make it easier to ensure that the linking is done correctly, the command `make configure` can be used, which will change the relevant parts of the Mini-OS files such that `app.a` is linked correctly. If all prerequisites are fulfilled, this will be the kernel, which is `mini-os.o`, which is the final unikernel with the SML application linked into it that can be run on the Xen hypervisor.

Running

Two files are needed to run the unikernel. The first is the unikernel itself, and then a `domain-config` file will be used to give the correct configurations when running it inside

Xen.

The domain that is used to control instances on Xen is called the control domain or Domain0 [38]. From the control domain, the `xl` tool can be used to start and stop guest domains [39]. First, the network needs to be configured in the control domain. This entails creating a network bridge `xenbr0`, which can then connect the physical ethernet device `eth0` to the virtual network device of the guest domain, which will be called `vifDOMID.DEVID`, where DOMID is the id, and DEVID is the index of the guest domain [37].

To run the unikernel the command `$ xl start -c domain_config` can be used, which will use the previously mentioned `domain_config` to configure, among other things, resources and the name of the guest domain. The `-c` flag will open a terminal where the console output from the guest domain can be seen. The guest domain created can then be shut down with `xl shutdown DOMAIN_NAME`.

Implementation of netif

The code for starting the SML application and reading ethernet frames with Mini-OS reside in the same file `Libs/netiflib/netif-miniOS.c`. Mini-OS will call the function `app_main` which will be the entry point for an application. In this case `app_main` creates two threads, `netfront` in which the networking code is run, and the second `sml` calling the `main` function from the MLKit runtime, which will start the SML application.

When the `netfront` thread is run, it will initialize the network properly by calling `init_netfront`. Here there is some ambiguity of how the internals of netfront works, but the second thread `sml` will need to wait on the `netfront` code to be initialized, so, therefore, it starts out by using a semaphore to wait for the network to be initialized. Then when the network is setup, the SML application will begin, and ethernet frames will be read using the `netif_rx` function, which will be called every time a packet is received.

The `netif_rx` function will continuously receive ethernet frames, so the pointer to the data of the packet is written to a `dataPtrs` array, and the length is written to a `dataLengths` array. When it reaches the end of the array it will wrap around to the start. This data can then later be retrieved when `receive` is called. To stop `receive` from reading data from the array that is currently used, it will run in an endless loop if it sees that the index it needs to read from is used by `netif_rx`. This is not an ideal solution, since `netif_rx` can overwrite ethernet frames, however, due to the ambiguity of how scheduling and thread-management works in Mini-OS, this solution seems to work as expected.

Both reading and writing will use a similar method as with tuntap, as described in section 4.3.1. The only notable difference is that reading is done through the buffers that `netif_rx` will write to, and writing is done with `netfront_xmit` function from Mini-OS.

4.4 Runtime minimization

In this section, the minimized runtime will be explained. First, what changes were needed for running on each platform, and how they might differ? Next, it will explain what parts of the original MLKit runtime have been excluded and the reason. The last section will go through what parts of the basis library of Standard ML that is supported.

4.4.1 Platform related changes

There are two minimized runtimes, one that uses Mini-OS for running on Xen and one to use with Unix. Although they both have had the same parts of the runtime removed, calls to `libc` and `libm` on Unix have been replaced with equivalent calls from Mini-OS. For example, all `printf` calls have been replaced with Mini-OS' `printk` calls. `printf` is used throughout the MLKit runtime to communicate what errors happened while running the program, which would normally be printed to `stderr` on Unix platforms; however, this distinction is not made in Mini-OS. Other relevant replacements are `malloc`, type definitions, and string functions like `strlen`.

One place where the runtimes differ is when setting an unlimited stack size. This could not be supported with Mini-OS, however it has been left in for the Unix runtime, as it did not seem to make a difference, so to ensure that behavior was similar on other machines when testing the service locally, this was not removed from Unix.

Because Mini-OS provides only limited `libc` supported, it is not entirely clear what is not supported. One such example is that formatting floats with formatting directives such as `%f` is not supported with Mini-OS, which did not give an error but printed out the literal formatting directive. In this case, a temporary solution was made to support formatting floats. However, other functions might also lack functionality in this way.

The math functions in the runtime were not fully supported by Mini-OS, therefore an external `libm` implementation is needed such that math functionality is supported. In this case, `newlib` was used as a static `libm` [33], since `libm` is normally linked dynamically as a shared library, but it needed to be linked statically into the kernel. This is where there are some problems with the Xen runtime, as the `floatSetRoundingMode` and `floatGetRoundingMode` in the MLKit use functions that are not supported in `newlib`.

One other important change is that the runtime on Xen will never exit but instead continue running in an endless loop. This is for debugging purposes since it will lead to an error for Mini-OS if the runtime is exited, which overwrites the output just before exiting. This also allows it to check the guest domain to see what happens if it crashes.

4.4.2 Runtime

The parts of the runtime that are left after minimization are:

- **Runtime.c:** this is the entry point for the runtime, which contains the main function that runs the external `code` function that will be run the actual SML application. It also contains the code for exiting the runtime, and for catching exceptions.
- **Region.c:** this contains the functionality that manages regions, including allocating and deallocating regions and getting new pages to the free-list.
- **String.c:** this contains functions related to ML-strings, for example converting between C and ML-strings, concatenating strings and printing strings. This is especially useful in the projects, since the raw packet is worked with as strings, and it is also helpful for printing to the console and debug.
- **Math.c:** this contains essential math functions for integers and floats, for example division, modulo and rounding functions. Many of the services that were designed used functions that were math related.
- **Other:** there were also header files that were left in, both those related to the source files, but also `List.h`, that contained macro definitions for working with lists and `Exception.h` that contained definitions for working with exceptions.

The parts of the runtime that were left out were partly because they were not useful to the project or because it was not immediately obvious how easy they would be to support on Xen. These are some of the parts that were left out of the minimized runtime:

- **Time.c:** this contains functions for working with time. Mini-OS already contains functionality for this, so it should be possible to implement, however, for this project, it was left out. This could be useful for logging utilities, and pseudo-number generators.
- **GC.c:** this contains functions and definitions for using the garbage collector. This project's focus is mainly on using region inference in unikernel, so garbage collection is not part of the runtime. However, this could be useful for handling cases where region inference is not sufficient.
- **Socket.c:** this provides support for sockets. This was left out, since it is not guaranteed that sockets were supported on the target platform, and also there would be a higher level of consistency moving from the development environment to the target environment since all parts of the network stack would be implemented in SML.
- **Spawn.c:** this provides functionality for working with threads. This was not essential to the project, as the application could be implemented single-threaded. However, this could be useful to answer multiple requests at a time.

4.4.3 Basis library

There were also parts of the SML basis library that were left out. Table 4.5 shows what required signatures of and structures of the SML standard library [25], which can still be used after the minimization of the runtime. Most of the missing parts are either OS-related, such as the `OS` signature and implementation, or the IO-related signatures. There are also

signatures related to time like `Time` and `Date` that is not supported, as discussed in section 4.4.2 because it is not part of the runtime.

As also discussed before, one of the parts where it was not clear what was supported, was regarding the formatting of floats and reals. The temporary implementation does work, however it is unclear whether other parts of the runtime might seem to have been implemented, but does not work correctly as a result of the limitation of the libc Mini-OS implementation. Therefore figure 4.5 shows what seems to be supported, but it is not fully known if the replaced function call might lead to other behavior that is not correct.

Some optional signatures and structures also are available. For instance, `INT_INF`, for working with arbitrarily large integers, and `PACK_REAL` and `PACK_WORD` for packing and unpacking floating point numbers and words into Word8 vectors and arrays [25].

Signature	Implementation(s) of signature	Is covered
ARRAY	Array	Yes
ARRAY_SLICE	ArraySlice	Yes
BIN_IO	BinIO	No
BOOL	Bool	Yes
BYTE	Byte	Yes
CHAR	Char	Yes
COMMAND_LINE	CommandLine	Yes
DATE	Date	No
GENERAL	General	Yes
IEEE_REAL	IEEEReal	Yes
IMPERATIVE_IO		No
INTEGER	Int, LargeInt, Position	Yes
IO	IO	Yes
LIST	List	Yes
LIST_PAIR	ListPair	Yes
MATH	Math	Yes
MONO_ARRAY	CharArray, Word8Array	Yes
MONO_ARRAY_SLICE	CharArraySlice, Word8ArraySlice	Yes
MONO_VECTOR	Word8Vector	Yes
MONO_VECTOR_SLICE	Word8VectorSlice	Yes
OPTION	Option	Yes
OS	OS	No
OS_FILE_SYS		No
OS_IO		No
OS_PATH		Yes
OS_PROCESS		No
PRIM_IO	BinPrimIO, TextPrimIO	No
REAL	LargeReal	Yes
STREAM_IO		No
STRING	String	Yes
STRING_CVT	StrintCvt	Yes
SUBSTRING	Substring	Yes
TEXT	Text	Yes
TEXT_IO	TextIO	No
TEXT_STREAM_IO		No
TIME	Time	No
TIMER	Timer	No
VECTOR	Vector	No
VECTOR_SLICE	VectorSlice	No
WORD	LargeWord	Yes

Table 4.5: The required signatures and implementation of the signature for SML [25]. The `COMMAND_LINE` line signature is marked yellow since it is part of the runtime, but there are no arguments passed to it when running it on Xen.

Chapter 5

Evaluation

This chapter includes an evaluation of the project. First, it includes a section on how the project has been tested continuously through the development process. Second, it includes some example unikernels that showcase the project's potential. Third, a section will include the results of running the project on Xen. Finally, an evaluation of the regions within the unikernels will be included.

5.1 Testing the implementation

To test the implementation of the project, several unit tests and an integration test has been implemented. Furthermore, a Github workflow has been setup and these will be discussed in this section.

Unit tests

The project includes 67 unit tests to test the implementation of the code. The unit tests have been made as *black box* tests, meaning the public functions in each module have been given a specific input where a specific output is expected. A small test module has been implemented with a function to assert another function, `assert (name, f, expected, toString)`, and its type is:

$$\text{string} * (\text{unit} \rightarrow \alpha) * \alpha * (\alpha \rightarrow \text{string}) \rightarrow \text{unit}$$

Here `name` is the name of the function to test, `f` is the function to be tested on some input, `expected` is the expected output, and `toString` is a function that converts the result type, α , to a string such that it can be printed. Figure 5.1 is an example of the definition of a unit test. The particular unit test tests the `encode` function in the `IPv4` module (NB: `testRaw` is a hard-coded tests header defined in the test file. Both `rawByteString` and `toByteList` are functions defined in the `Utilities` module).

Integration testing

To ensure that packets are properly fragmented, the project includes an integration test that sends the first 60000 digits of π to the `echo` service. The `echo` service is ideal for testing


```

assert ("encode",
      (fn () => IPv4.encode (IPv4.Header header) payload),
      testRaw,
      (rawBytesString o toByteList)
);

```

Figure 5.1: Example of unit test

fragmentation as it replies to any message with the exact message it received. Since 60000 digits are larger than the MTU for UDP, the payload must be fragmented on both receiving and sending.

Github actions (CI)

To automatically test the code in every commit pushed to the project's GitHub, a GitHub actions workflow has been set up to

1. Build a service (**echo**)
2. Run unit tests
3. Run integration test

This ensures that the functionality is as intended and that no subtle bugs sneak into the code base upon every commit to any branch.

5.2 Example services

Four example services are included in this project to showcase some potential use cases of the project.

- **Echo** is the classic *hello world* example, which simply returns the string it was given untouched.
- **Fibonacci and factorial** includes two services on two different ports, showcasing that the unikernel can serve multiple services and how to define such. The functions here are the factorial function bound to port 8080 defined as:

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 3 \times 2 \quad (5.1)$$

and the Fibonacci function bound to port 8081 is defined as:

$$f_n = f_{n-1} + f_{n-2} \quad \text{for } n > 1 \quad (5.2)$$

- **Merge sort** utilizes an implementation of mergesort in SML to sort a string of space-separated numbers e.g. `2 7 1 8 2 8` will result in `1 2 2 7 8 8`. This example

service has been chosen to show the capabilities of the region inference as the implementation of merge sort will generate many temporary arrays through its recursion steps. The mergesort implementation code is seen in figure 5.2.

```

1  fun merge ([], 1) = 1
2    | merge (1, []) = 1
3    | merge ((h1::t1), (h2::t2)) = if h1 < h2 then h1 :: merge (t1, (h2::t2))
4                                   else h2 :: merge ((h1::t1), t2)
5
6  fun split l =
7    let fun split [] xs ys = (xs, ys)
8          | split (x::[]) xs ys = (x::xs, ys)
9          | split (x::y::t) xs ys = split t (x::xs) (y::ys)
10   in split l [] [] end
11
12 fun mergesort [] = []
13   | mergesort [x] = [x]
14   | mergesort l = split l |> (fn (x, y) => (mergesort x, mergesort y)) |>
    ↪ merge

```

Figure 5.2: Merge sort in SML

- **Monte Carlo estimation of π** utilizes a Sobol sequence of repeated quasi-random sampling to obtain the estimation of π . The Sobol sequence is generated by a Standard ML package for generating Sobol sequences [8] thus showing via this example that it is possible to include external packages in the services. The estimation works by generating 2D points within a square with a length size of $2r$. If we then imagine a circle inside the square with a radius r we can calculate the number of points inside the circle and the number of points outside the circle in the square. Since the area of the square is $4r^2$ and the area of the circle is πr^2 , the ratio of the points is:

$$\frac{\text{circle area}}{\text{square area}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4} \quad (5.3)$$

With a large amount of randomly generated numbers, we can estimate π with:

$$\frac{\pi}{4} = \frac{\text{points inside the circle}}{\text{total points}} \quad (5.4a)$$

$$\Rightarrow \pi = 4 \times \frac{\text{points inside the circle}}{\text{total points}} \quad (5.4b)$$

where the points inside the circle are determined by:

$$x^2 + y^2 \leq 1 \quad (5.5)$$

Figure 5.3 is the code for the service that serves port 8080 with the Monte Carlo estimation of π . Its input is how many data points to use.

```

1  open Network
2
3  structure Sobol = Sobol(val D = 2
4                          structure SobolDir = SobolDir50)
5
6  (* UC is short for unit circle. *)
7  fun monteCarlo n =
8      let
9          fun loop 0 UC = UC
10             | loop i UC =
11                 let
12                     val v = Sobol.independent i
13                     val x = Sobol.frac(Array.sub(v,0))
14                     val y = Sobol.frac(Array.sub(v,1))
15                     val new_UC =
16                         if (x * x + y * y <= 1.0) then
17                             UC + 1
18                         else
19                             UC
20                 in
21                     loop (i - 1) new_UC
22                 end
23             val UC = loop n 0
24         in
25             4.0 * (Real.fromInt UC) / (Real.fromInt n)
26         end
27
28  val _ = (
29      bindUDP 8080 (
30          fn data =>
31              case Int.fromString data of
32                  SOME n => monteCarlo n |> Real.toString
33                  | NONE => "Invalid input"
34      );
35
36      listen ()
37  )

```

Figure 5.3: Example service with a Monte Carlo estimation of π

5.3 Running services on Xen

In this section the results of running the services as unikernels on Xen will be discussed. Each service mentioned in section 5.2 has been tried on Xen. This was done by running Xen inside virtualbox, with the control domain running Ubuntu.

An example of running the echo service on Xen can be seen in figure 5.4. Here the netcat tool is used to send a request over an UDP connection, containing the first 60,000 digits of pi. This ran successfully on Xen and it can also be seen in figure 5.4 in the bottom-right corner that the package is fragmented, splitting up the response in multiple fragments.

```
axel@axel-VirtualBox: ~/...  
axel@axel-VirtualBox: ~$ sudo xl create -c domain_config  
Parsing config from domain_config  
Xen Minimal OS (pv):  
  start_info: 0x129000(VA)  
  nr_pages: 0x5400  
  shared_info: 0x0d24000(VA)  
  pt_base: 0x12c000(VA)  
  nr_pt_frames: 0x5  
  mfn_list: 0xf7000(VA)  
  mod_start: 0x0(VA)  
  mod_len: 0  
  flags: 0x0  
  cmd_line:  
    stack: 0xd33a0-0xf33a0  
MM: Init  
  _text: 0(VA)  
  _etext: 0x8944b(VA)  
  _erodata: 0x90000(VA)  
  _edata: 0x934c(VA)  
  stack_start: 0xd33a0(VA)  
  _end: 0xf6310(VA)  
  stack_ptr: 131  
  max_ptr: 6400  
Mapping memory range 0x131000 - 0x6400000  
setting 0x90000 readonly  
skipped 1000  
MM: Initialise page allocator for 161000(161000)-6400000(6400000)  
  Adding memory range 162000-6400000  
MM: done  
Demand map pfn at 100000000000-100000000000.  
Initialising console... done  
gnttab table mapped at 0x100000000000.  
Initialising scheduler  
Thread 'idle': pointer: 0x0x164078, stack: 0x0x170000  
Thread 'xenstore': pointer: 0x0x1640d0, stack: 0x0x180000  
xenbus initialised on irq 1  
Thread 'shutdown': pointer: 0x0x164138, stack: 0x0x190000  
Thread 'netfront': pointer: 0x0x164198, stack: 0x0x1a0000  
Thread 'sm1': pointers: 0x0x1641f0, stack: 0x0x1b0000  
***** NETFRONT for device/vif/0 *****  
  
net TX ring size 256  
net RX ring size 256  
backend at /local/domain/0/backend/vif/25/0  
mac is 00:16:3e:2b:be:43  
*****  
Thread 'netfront' exited.  
[]
```

```
axel@axel-VirtualBox: ~/mini-os  
axel@axel-VirtualBox: ~/mini-os$ cat pi.txt | nc -u 10.0.0.2 8080  
3.1415926535897932384626433832795028841971693993751058209749445923078164062862089966280848253421170  
679821480865132838664709384466953058223172535940812846117420284102701938521105559464629489549303  
81964428810957566593344612847564823378678316527120190914564856692346034661045432648213393672602491  
4127372458700660631558817488152092096282925409171536436789259036001133953054885284662513841469519415  
116094330572703657595919530921861173819326117931051185480744623799627495673518857527248912279381830  
11949129833673624406566430860213949643952247371907821798609437027705392171762931767552346748184676  
694051320005681271452635608277857713427577896091736371787214684409012249534301465495853710507922796  
8925892354201995611212902196086403441815981362977477130996051870721134999999379704009510597317328  
160963185956424594553490830262523200153244685035261911801710100031378375280587532083814260171  
7760914703598254904204546873115956286388235370537519578183577005321712268066110019278766111959  
692164201989389525720185485863278865936153381827968230301952035301852968995736225994138912497217  
75283479131515574857424541566959508295331168617278558890750983817546374649393192550660400977016711  
3900984882401285836160356730766010471018194295559619894676783744944825537977472684718040475346462880  
466842590694912933136770289891521047521620569660240580381501935112533824300355876402474964732639141  
992726042699227967823547816360093417216412199245863150302861829745557067498385054945885869269596909  
2721079509302953211653449872027559602364806654991198818347977535663698074265425278625518184175746  
72890977772793800808167060016145249192173217214772350141441973568548161361157352552134757418494684  
38523323907394143345477624168605189035694855620992192221842725502542568876717904946016534680049886  
2723279178608578438382796797668145418095388378636956800642251252051173929848960841284886269456842  
419652850222106611863067442786220391949450471237137869609563643719172874677646575739624138908658326  
459958133904780275900994657640789512694683983525957098258226205224894077267194782684826014769990926  
401363944374553050682034962524517493996514314298091906592509372216964615157098583874105978859597729
```

```
axel@axel-VirtualBox: ~  
108 28.722149154 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=11840, ID=2734)  
109 28.722141905 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=13328, ID=2734)  
110 28.722144048 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=14880, ID=2734)  
111 28.722143927 10.0.0.2 → 10.0.0.1 UDP 146 8080 → 60016 Len=16384  
112 28.722146357 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=0, ID=2735)  
113 28.722159253 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=14880, ID=2735)  
114 28.722151965 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=2960, ID=2735)  
115 28.722153688 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=4440, ID=2735)  
116 28.722153270 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=5920, ID=2735)  
117 28.722156744 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=7400, ID=2735)  
118 28.722158508 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=8880, ID=2735)  
119 28.722160434 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=10360, ID=2735)  
120 28.722162364 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=11840, ID=2735)  
121 28.722164071 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=13320, ID=2735)  
122 28.722165928 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=14800, ID=2735)  
123 28.722167793 10.0.0.2 → 10.0.0.1 UDP 146 8080 → 60016 Len=16384  
124 28.722169595 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=0, ID=2735)  
125 28.722171816 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=14880, ID=2736)  
126 28.722173450 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=2960, ID=2736)  
127 28.722175092 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=4440, ID=2736)  
128 28.722176748 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=5920, ID=2736)  
129 28.722178175 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=7400, ID=2736)  
130 28.722179804 10.0.0.2 → 10.0.0.1 IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=8880, ID=2736)  
131 28.722181329 10.0.0.2 → 10.0.0.1 UDP 531 8080 → 60016 Len=10849
```

Figure 5.4: An example of running the echo service as a unikernel on Xen. 3 terminal windows are visible. The right terminal window is where the unikernel is started with the *xl* command-line tool. The upper-right window shows sending a request to the echo service through netcat with pi as the input. Finally in the bottom-right window the package going through the network interface used by the unikernel is shown.

All the other services also worked for the most part on Xen, the runs of the other services can be seen in the appendix A. However crashes occurred during some requests. This was especially the case with services, that used a considerable amount of memory, like the facfib service. In figure 5.5 shows an example where fibonnaci of 2000 is requested on port 8081, but this fails with the error "Page fault and linear address" followed by "GPF rip". When requesting on 8081, the service saves all intermediate fibonacci results, but also uses intermediate vectors, which is not optimal. However it also successfully answered the same request other times.

lists and region inference would, therefore, come into effect. A string of 10000 numbers was used as input for this example with a size of 29202 bytes, which should result roughly in $\frac{29202}{1480} \approx 20$ fragments. The `listen` function was run 1000 times and therefore completed processing 1000 fragments. This will mean that since the data will be read in a loop, and the list results in 20 fragments, the list will be processed $\frac{1000}{20} = 50$ times. Since the data is more than one fragment long, the IPv4 fragmentation implementation of the networking library will be activated. The result of this can be seen in figure 5.6 (note there are 50 spikes):

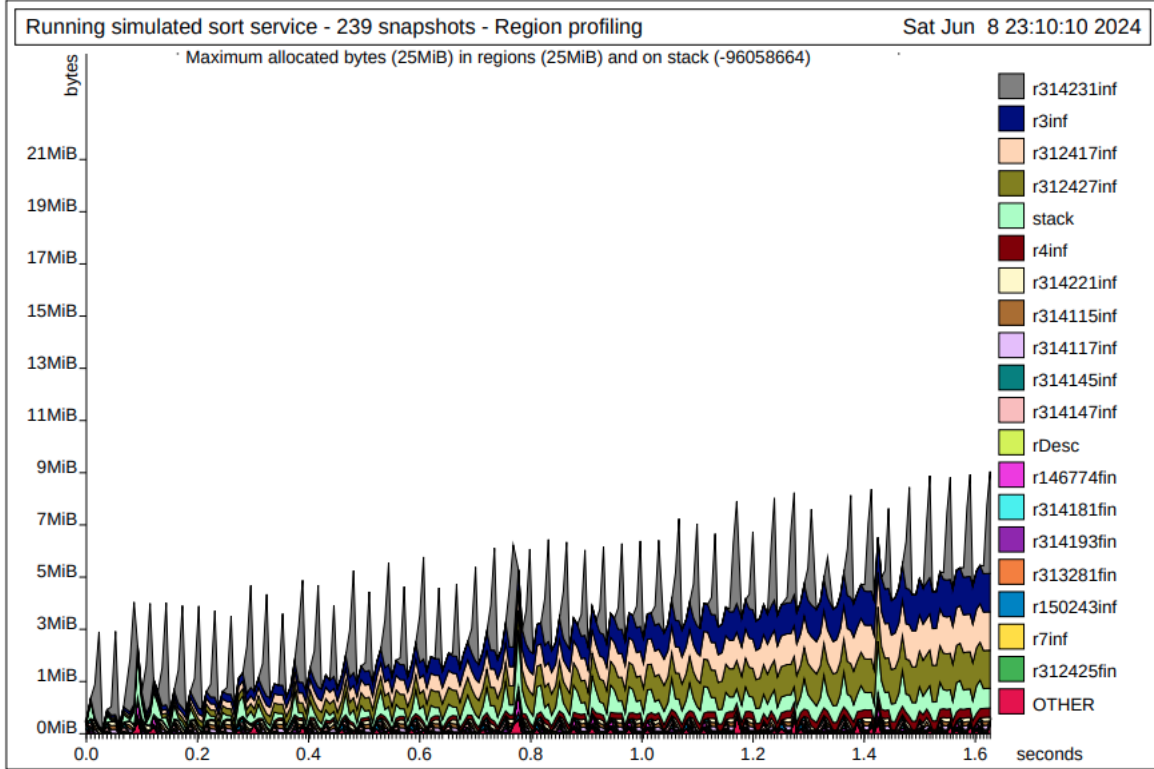


Figure 5.6: This figure shows how the sort service uses regions by using the region-profiling tool in MLKit.

As can be seen in figure 5.6, it seems that some regions, more specifically the regions `r3inf`, `r312417inf` and `r312427inf`, are growing over time. The region `r3` is a global region that holds string values [31], while the other two are used in the `listen` function in the call to `String.extract` and `Eth.decode`. The region annotated code for the `listen` function can be seen in appendix B.1. This seems to suggest that there are strings that are never freed, possibly because the `fragmentBuffer` and `assemblingList` lists are in a global scope. The values, which include the payload strings of the fragments, are available during the entire run of the service, and there is no mechanism to free the resources used for these strings since they are contained within a global region. Without a garbage collector present, these values will not be collected, resulting in a linear increase in memory usage of the running time.

The following is parts of the `mergesort` function from figure 5.2, with region-annotations:

```

1 fun mergesort attop r1 [r314049:inf] (var5) =
2   (* ... Function pattern matching *)
3   let region r314133:2;
4   val v324 =
5     let region r314115:inf, r314117:inf, r314119:2;
6     val v326 =
7       let region r314059:0;
8       fun split atbot r314059 [r314079:2, r314077:inf,
9         ↪ r314075:inf] (var2, var3, var4) =
10         (* ... *)
11         in funcall split[atbot r314119,atbot r314117,atbot
12           ↪ r314115] <var5, nil, nil>
13         end;
14       val v228 = #0 v326; (* First list *)
15       val v229 = #1 v326 (* Second list *)
16     in (funcall mergesort[sat r314049] v228,
17       funcall mergesort[attop r314049] v229)atbot r314133
18     end
19   in funcall merge[sat r314049] <#0 v324, #1 v324>
20 end
21 (* ... *)

```

Figure 5.7: Merge sort with region annotations. The formal region parameter that `mergesort` is given to store the result is highlighted, and some parts of the code have been left out to simplify.

As shown in figure 5.7, many local regions are used throughout the function. The region that has been highlighted is a *formal region parameter* [31], and this will be used to store the sorted version of the list given to `mergesort` (throughout this section, the region highlight colors will match the color region seen on figure 5.6). `atbot`, `attop`, and `sat` refer to which storage mode should be used when storing values inside regions. `atbot` is for resetting the region and then storing the value, `attop` is for storing the value without destroying the region, and `sat` is when the storage mode should be determined at runtime [31].

In the region-annotated program for `mergesort`, the formal region parameter `r314049` is used to store the final sorted list. This region is also used to store intermediate results; for example, when sorting each half on lines 14-15, the lists are stored inside the `r314049` region. For the call to `mergesort` on line 14, the storage mode for the first half of the list is `sat`, while the last half uses `attop`. For the last half, it makes sense that `attop` is used, as there are two intermediate results that need to be stored simultaneously, and therefore, storage of this list should not reset the region, or else the first half of the list will not be available. However, the reason for using storage mode `sat` for the first half might be because

it depends on which call to `mergesort` it is.

To demonstrate this, suppose we are sorting 100 numbers; at some point, we need to sort the first sublist of 25 numbers from the original list. In this case, if `atbot` was used on line 14, then the region that holds the entire list, including the other 75 numbers, would be reset, and the other 75 numbers would be lost. Therefore, using `attop` would make sense. However, if we have returned to the top-level `mergesort` call, then all numbers are contained in the two halves, and the `r314049` region can be reset, since the intermediate results of splitting the lists are stored in the local regions on line 5. A similar situation arises on line 17, if it is the last `merge`. Here it also makes sense that the region parameter is reset when given to `merge`. This demonstrates how region-inference effectively and safely can transform a function, such as `mergesort`, into using region-based memory management.

```

1  (* ... *)
2  let val _ =
3      funcall bindUDP[]
4      <8080,
5      fn attop r1 data =>
6          (* ... *)
7          let (* ... *)
8              region r314231:inf;
9              (* v241 contains the list of integers*)
10             val v330 = funcall mergesort[atbot r314231] v241;
11             region r314267:inf, r314269:inf;
12             val strL =
13                 let region r314233:1;
14                     fun map atbot r314233 [r314245:inf, r314243:inf] (var20)
15                         ↪ =
16                             (* ... *)
17                             in funcall map[atbot r314269,atbot r314267] v330
18                             end
19                             in (* concat the list of strings *)
20                                 funcall concatWith[atbot r3] <" "attop r314267, strL>
21                                 end
22                             (* ... *)
23             >
24             in funcall listen[] ()
25             end
26         (* ... *)

```

Figure 5.8: Binding callback function using `mergesort` to port 8080 with region annotations.

As seen in figure 5.8, the highlighted local region `r314231` is given as an actual region pa-

parameter to `mergesort`. Looking back at the region-profiling figure 5.6, we can see that this region is the grey-colored region that continuously spikes and then is freed immediately after, which means that the sorted list is freed after it is no longer in use. This suggests that region inference is very effective for the `mergesort` service, which also makes sense since it primarily uses local regions instead of global regions.

This poses the question of how the benefits of using region inference can be retained, as seen with the `mergesort` function, while still ensuring that the memory usage does not grow proportionally to the running time, as seen with networking library. One method would be to try to write more region-optimized code. This might include avoiding the use of global regions, as well as analyzing the region-annotated version together with the use of region profiling. Another related method would be to use regions explicitly, for example, using *ReML*, a Standard ML extension implemented in MLKit, that allows programs to be explicit about using regions [7]. This could help give better control over the usage of regions. Minor changes might affect the output of using region inference to a considerable degree, and by using *ReML*, such changes to the properties of the program could be avoided to a larger extent.

Another way would be to integrate a garbage collector that could take care of allocations that are not reclaimed well with region inference, like with global regions. The use of region inference can then minimize the number of times the garbage collector is run [30]. This would allow the developer to use only region inference or a combination with garbage collection, depending on what fits the program best. In the case of the sort service, this could collect the values stored inside the global regions, as these are often the regions that need to be collected with garbage collection [9].

5.4.2 Developer productivity and memory safety

Since region-based memory management offers automatic memory management via region inference, it greatly benefits the developer's productivity. Because this system automatically determines when and what to allocate and deallocate, the developer does not have to think about memory management as much as with garbage collection. Furthermore, region inference is likely better than most developers at figuring out the lifespan of objects and when it would be best to clean up the memory.

Although this lowers the burden of thinking about regions, as discussed beforehand, if the developer is not thinking about the use of regions, it can end up becoming a potential problem since the program might end up being poorly optimized, as described in section 5.4.1. However, it still gives the developer the opportunity to design and implement libraries and services that work without the need to think deliberately about regions and then afterward have the opportunity to optimize for region usage.

The project is an example of this, as the focus has not been on optimizing region usage but rather on implementing the essential parts for communicating through a network. By using region inference, the project has not explicitly used regions, and in this way, the focus can be entirely on getting the implementation to work. Therefore, by using region inference, the

usage of region memory management has not hampered the developer productivity of this project.

Moreover, region inference offers memory safety, just like garbage collection. Like with MirageOS, services can be developed reliably without worrying about potential memory leaks or premature deallocations. This is in contrast to languages like C, where you need to explicitly allocate and deallocate memory, leading to the aforementioned issues.

Other languages exist that could fulfill the same need of memory safety while also having performance closer to that of languages that use explicit memory management. One such language is Rust, which does not use a garbage collector but instead also uses region-based memory management among other methods [4]. However, in Rust, there is a high mental load where the developer needs to think actively about the introduced concepts, which results in a language that is harder to learn and use [4]. Region inference, therefore, offers an alternative way where the developer is not required to actively think about the underlying memory management methods but still has the potential to manage memory in a more efficient and safe way.

5.4.3 Significance for unikernels

For the MirageOS project, using OCaml as the language for almost all of the code of the unikernel provides type safety and memory safety and improves developer productivity. In this case, developing the project with SML and MLKit, region-based memory management with region inference can offer the same benefits, but the developer is provided with a way to manage memory in a more optimal way.

If the libraries and operating system functionality of the unikernel are all developed in the same high-order language, like with MirageOS, then nearly all parts of the system will benefit if memory management costs of the language can be minimized. This is where one might argue that it would be better to use a language like Rust, however developing new libraries and operating system functionality in Rust would be a more time consuming task as discussed in the previous section. Having region-inference as an alternative to garbage-collection is therefore a viable option when developing unikernels in higher-level languages, and further maintains the viability of using higher-level languages for unikernel development, without the need to turn to lower-level languages like C or Rust for performance gains.

Chapter 6

Future developments

Possible implementations to further develop this project may include the following:

- **Further evaluations.** To further evaluate the project various benchmarks could be implemented. An example of this is to compare the differences with only garbage collection compared to using only region-based memory management.
- **Network protocols.** To ensure every packet gets received and delivered a sensible future development is to implement the Transmission Control Protocol (TCP) which ensures every packet is delivered even in the event of a packet loss on the network anywhere. To further complete the network stack IPv6 should be implemented to succeed the project's current network layer, IPv4.
- **Multithreading.** As mentioned in section 4.4.2 `spawn.c` has been left out in this project. However, in the future, it could be useful to include it to be able to work with threads. A neat implementation would be each binding of a callback function to a port would be its own thread.
- **Combine miniruntimes.** Currently, the project includes two different runtimes as mentioned in section 4.3. In the future, this should be combined into one combined runtime perhaps using `ifdef` to include the appropriate parts of the runtime.
- **Xen port.** The current implementation of porting the code to Xen is unstable. This could be improved in the future by either understanding the Mini-OS codebase better and improving that or finding a different way to support Xen.
- **Add support for other platforms.** Other platforms could include the Kernel-based Virtual Machine (KVM) that is built into Linux and allows Linux to be turned into a hypervisor. This could be done via the solo5 project [27].
- **Expand basis support.** These expansions could include using time within the services, allowing for reading and writing to files, etc., and in general have better libc support.

Chapter 7

Conclusion

The goal of creating a unikernel library that supports the development of services that can answer requests from a network was successful. While some parts of the implementation for the Xen platform have issues with crashing, in general, the developed services work as unikernels on Xen and do not use garbage collection.

While this demonstrates that region-based memory management with region inference is feasible for the development of unikernels, services developed without optimizing for region usage can lead to space leaks and a large amount of memory usage. As seen with the merge sort service, while the sorting itself works well with the use of regions, the underlying networking implementation used global regions, resulting in memory growing over time. Several methods could be used to counteract this. One way is to analyze the result of region inference and thereby try to write a more region-friendly program. Another way is to be explicit about how the regions should be used with extensions like ReML. The final way would be to integrate garbage collection, which could take care of parts, including the use of global regions, that are not optimized for region usage.

For the development of unikernels, region-based memory management with region inference offers a way for high-order languages to gain performance benefits without the need to turn to lower-order languages, which either compromise developer productivity or memory safety. This allows for system-wide performance gains in unikernels and a high degree of developer productivity when implementing new libraries and operating system functionality.

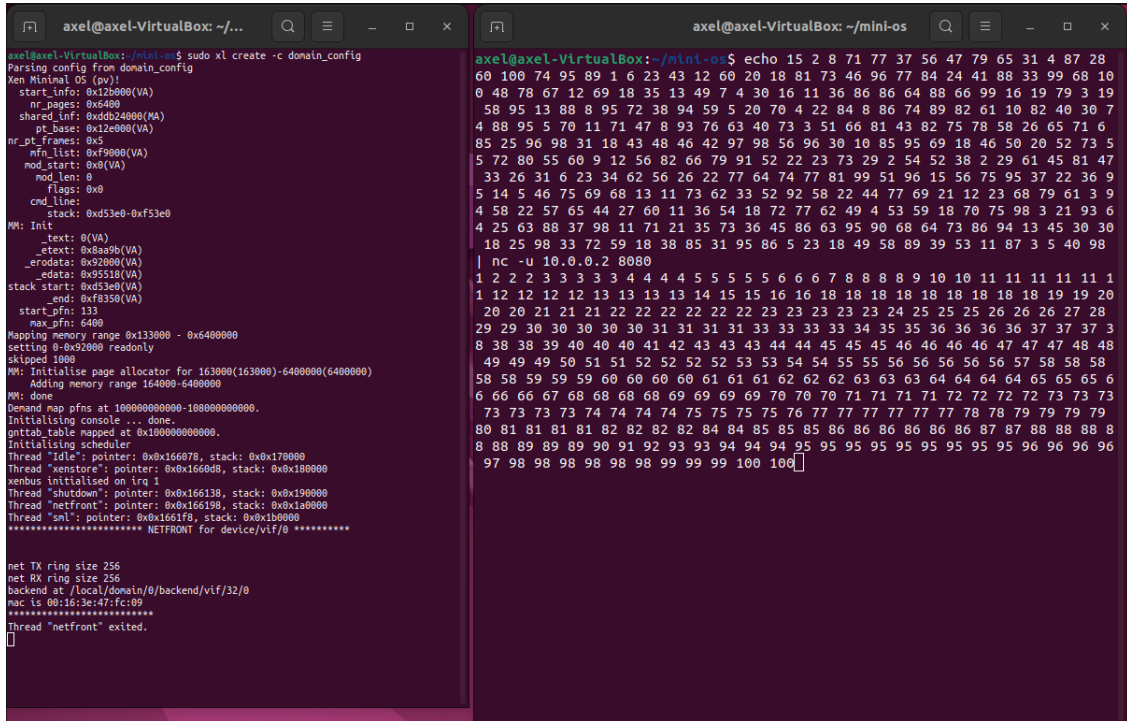
Bibliography

- [1] James F. Kurose et. al. *Computer Networking, A Top-Down Approach*. 8th edition. Pearson education, 2022.
- [2] Michael Armbrust et al. “A view of cloud computing”. In: *Commun. ACM* 53.4 (Apr. 2010), pp. 50–58. ISSN: 0001-0782. DOI: 10.1145/1721654.1721672. URL: <https://doi.org/10.1145/1721654.1721672>.
- [3] Paul Barham et al. “Xen and the art of virtualization”. In: *SIGOPS Oper. Syst. Rev.* 37.5 (Oct. 2003), pp. 164–177. ISSN: 0163-5980. DOI: 10.1145/1165389.945462. URL: <https://doi.org/10.1145/1165389.945462>.
- [4] Michael Coblenz, Michelle L. Mazurek, and Michael Hicks. “Garbage collection makes rust easier to use: a randomized controlled trial of the bronze garbage collector”. In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE ’22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1021–1032. ISBN: 9781450392211. DOI: 10.1145/3510003.3510107. URL: <https://doi.org/10.1145/3510003.3510107>.
- [5] Mache Creeger. “Cloud Computing: An Overview: A summary of important cloud-computing issues distilled from ACM CTO Roundtables”. In: *Queue* 7.5 (June 2009), pp. 3–4. ISSN: 1542-7730. DOI: 10.1145/1551644.1554608. URL: <https://doi.org/10.1145/1551644.1554608>.
- [6] Nabil El Ioini et al. “Unikernels Motivations, Benefits and Issues: A Multivocal Literature Review”. In: *Proceedings of the 3rd Eclipse Security, AI, Architecture and Modelling Conference on Cloud to Edge Continuum*. ESAAM ’23. , Ludwigsburg, Germany, Association for Computing Machinery, 2023, pp. 39–48. ISBN: 9798400708350. DOI: 10.1145/3624486.3624492. URL: <https://doi.org/10.1145/3624486.3624492>.
- [7] Martin Elsman. “Explicit Effects and Effect Constraints in ReML”. In: *Proc. ACM Program. Lang.* 8.POPL (Jan. 2024). DOI: 10.1145/3632921. URL: <https://doi.org/10.1145/3632921>.
- [8] Martin Elsman. *Standard ML package for generating Sobol sequences*. URL: <https://github.com/diku-dk/sml-sobol>.
- [9] MARTIN ELSMAN and NIELS HALLENBERG. “Integrating region memory management and tag-free generational garbage collection”. In: *Journal of Functional Programming* 31 (2021), e4. DOI: 10.1017/S0956796821000010.

- [10] Fortinet. *What is Transmission Control Protocol TCP/IP?* URL: [https://www.fortinet.com/resources/cyberglossary/user-datagram-protocol-udp#:~:text=User%20Datagram%20Protocol%20\(UDP\)%20is,destination%20before%20transferring%20the%20data](https://www.fortinet.com/resources/cyberglossary/user-datagram-protocol-udp#:~:text=User%20Datagram%20Protocol%20(UDP)%20is,destination%20before%20transferring%20the%20data).
- [11] David R. Hanson. “Fast allocation and deallocation of memory based on object lifetimes”. In: (1990). DOI: <https://doi.org/10.1002/spe.4380200104>.
- [12] *Hermit-rs*. URL: <https://github.com/hermit-os/hermit-rs>.
- [13] “IEEE Standard for Ethernet”. In: *IEEE Std 802.3-2022 (Revision of IEEE Std 802.3-2018)* (2022), pp. 1–7025. DOI: 10.1109/IEEESTD.2022.9844436.
- [14] *IncludeOS*. URL: <https://github.com/includeos/IncludeOS>.
- [15] *Internet Protocol*. RFC 791. Sept. 1981. DOI: 10.17487/RFC0791. URL: <https://www.rfc-editor.org/info/rfc791>.
- [16] Nicholas Jacek et al. “Optimal Choice of When to Garbage Collect”. In: *ACM Trans. Program. Lang. Syst.* 41.1 (Jan. 2019). ISSN: 0164-0925. DOI: 10.1145/3282438. URL: <https://doi.org/10.1145/3282438>.
- [17] Maxim Krasnyansky, Maksim Yevmenkin, and Florian Thiel. *Universal TUN/TAP device driver*. 2002. URL: <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/networking/tuntap.rst?id=HEAD>.
- [18] Moritz Lipp et al. *Meltdown*. 2018. arXiv: 1801.01207 [cs.CR].
- [19] Anil Madhavapeddy and David J. Scott. “Unikernels: Rise of the Virtual Library Operating System: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework?” In: *Queue* 11.11 (Dec. 2013), pp. 30–44. ISSN: 1542-7730. DOI: 10.1145/2557963.2566628. URL: <https://doi.org/10.1145/2557963.2566628>.
- [20] Anil Madhavapeddy et al. “Unikernels: library operating systems for the cloud”. In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’13. Houston, Texas, USA: Association for Computing Machinery, 2013, pp. 461–472. ISBN: 9781450318709. DOI: 10.1145/2451116.2451167. URL: <https://doi.org/10.1145/2451116.2451167>.
- [21] *MirageOS*. URL: <https://github.com/mirage/mirage>.
- [22] R. Pavlicek. *Unikernels: Beyond Containers to the Next Generation of Cloud*. O’Reilly Media, 2016. ISBN: 9781491959244. URL: <https://books.google.dk/books?id=qfDXuQEACAAJ>.
- [23] J. Postel. *User Datagram Protocol*. 1980. URL: <https://datatracker.ietf.org/doc/html/rfc768>.
- [24] Allison Randal. “The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers”. In: *ACM Comput. Surv.* 53.1 (Feb. 2020). ISSN: 0360-0300. DOI: 10.1145/3365199. URL: <https://doi.org/10.1145/3365199>.
- [25] John Reppy. *The Standard ML Basis Library*. 2004. URL: <https://smlfamily.github.io/Basis/overview.html>.

- [26] William A. Simpson et al. *Neighbor Discovery for IP version 6 (IPv6)*. RFC 4861. Sept. 2007. DOI: 10.17487/RFC4861. URL: <https://www.rfc-editor.org/info/rfc4861>.
- [27] *Solo5*. URL: <https://github.com/Solo5/solo5>.
- [28] M. Tofte and L. Birkedal. *A Region Inference Algorithm*. 1998. URL: <https://elsman.com/mlkit/pdf/toplas98.pdf>.
- [29] Mads Tofte. “A brief introduction to regions”. In: (1998), pp. 186–195.
- [30] Mads Tofte et al. “A Retrospective on Region-Based Memory Management”. In: *Higher Order Symbolic Computation* 17.3 (2004), pp. 245–265.
- [31] Mads Tofte et al. “Programming with Regions in the MLKit”. In: (2022).
- [32] *Unikraft*. URL: <https://github.com/unikraft/unikraft>.
- [33] Corinna Vinsche and Jeff Johnston. *info*. URL: <https://sourceware.org/newlib/>.
- [34] Xenproject. *Mini-OS*. 2018. URL: <https://wiki.xenproject.org/wiki/Mini-OS>.
- [35] Xenproject. *README*. Version <http://xenbits.xenproject.org/gitweb/?p=mini-os.git;a=blob;f=README;hb=HEAD>. 2024. URL: <http://xenbits.xenproject.org/gitweb/?p=mini-os.git;a=tree;hb=HEAD>.
- [36] Xenproject. *Unikernels*. 2021. URL: <https://wiki.xenproject.org/wiki/Unikernels>.
- [37] Xenproject. *Xen Networking*. 2016. URL: https://wiki.xenproject.org/wiki/Xen_Networking.
- [38] Xenproject. *Xen Project Beginners Guide*. 2020. URL: https://wiki.xenproject.org/wiki/Xen_Project_Beginners_Guide.
- [39] Xenproject. *XL*. 2020. URL: <https://wiki.xenproject.org/wiki/XL>.

A.2 sort



```
axel@axel-VirtualBox: ~/...
axel@axel-VirtualBox:~/mini-os$ sudo xl create -c domain_config
Parsing config from domain_config
Mem Minimal OS (ov):
start_info: 0x12b000(VA)
nr_pages: 0x6400
shared_inf: 0x0db24000(VA)
pt_base: 0x12e000(VA)
nr_pt_frames: 0x5
mfn_list: 0xf9000(VA)
mod_start: 0x0(VA)
mod_len: 0
flags: 0x0
cmd_line:
stack: 0xd53e0-0xf53e0
MM: Init
_text: 0(VA)
_etext: 0x8aa9b(VA)
_erodata: 0x92000(VA)
_edata: 0x95518(VA)
stack_start: 0xd53e0(VA)
_end: 0xf8350(VA)
start_pfn: 133
max_pfn: 6400
Mapping memory range 0x133000 - 0x6400000
setting 0-0x92000 readonly
skipped 1000
MM: Initialise page allocator for 163000(163000)-6400000(6400000)
Adding memory range 164000-6400000
MM: done
Demand map pfns at 100000000000-1000000000000.
Initialising console ... done.
enttab table mapped at 0x100000000000.
Initialising scheduler
Thread "idle": pointer: 0x0x166078, stack: 0x0x170000
Thread "xenstore": pointer: 0x0x1660d8, stack: 0x0x180000
vmbus initialised on irq 1
Thread "shutdown": pointer: 0x0x166138, stack: 0x0x190000
Thread "netfront": pointer: 0x0x166198, stack: 0x0x1a0000
Thread "sml": pointer: 0x0x1661f8, stack: 0x0x1b0000
***** NETFRONT for device/vif/0 *****

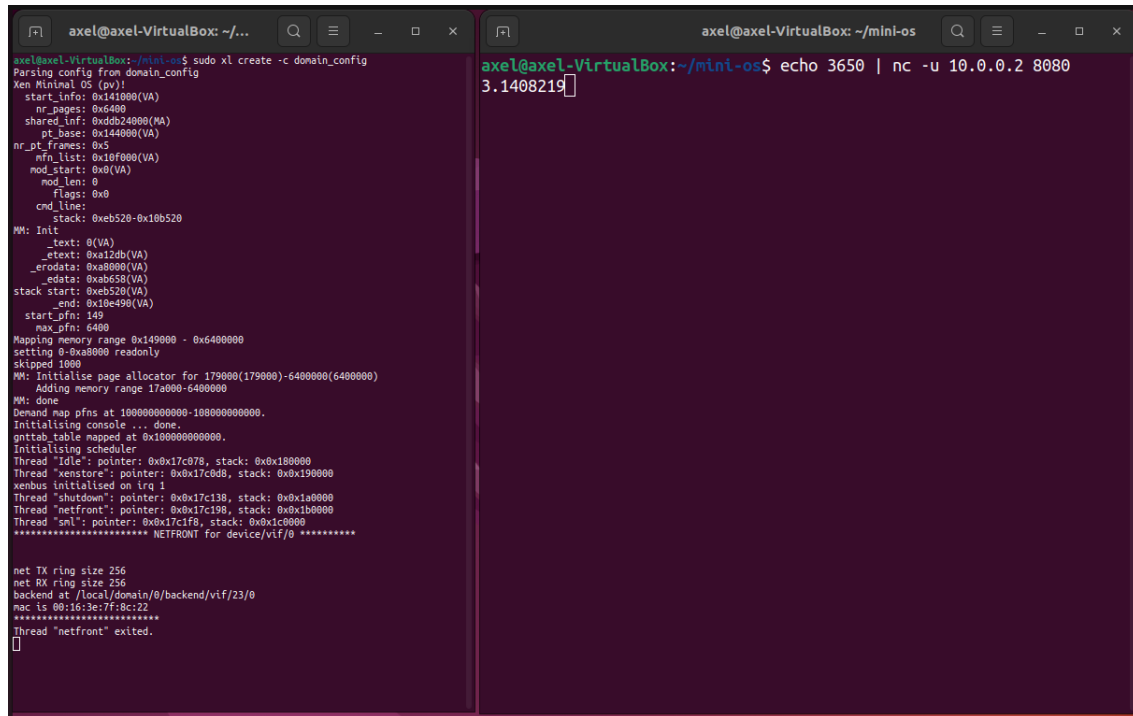
net TX ring size 256
net RX ring size 256
backend at /local/domain/0/backend/vif/32/0
mac is 00:16:3e:47:fc:09
*****
Thread "netfront" exited.

```

```
axel@axel-VirtualBox: ~/mini-os$ echo 15 2 8 71 77 37 56 47 79 65 31 4 87 28
60 100 74 95 89 1 6 23 43 12 60 20 18 81 73 46 96 77 84 24 41 88 33 99 68 10
0 48 78 67 12 69 18 35 13 49 7 4 30 16 11 36 86 86 64 88 66 99 16 19 79 3 19
58 95 13 88 8 95 72 38 94 59 5 20 70 4 22 84 8 86 74 89 82 61 10 82 40 30 7
4 88 95 5 70 11 71 47 8 93 76 63 40 73 3 51 66 81 43 82 75 78 58 26 65 71 6
85 25 96 98 31 18 43 48 46 42 97 98 56 96 30 10 85 95 69 18 46 50 20 52 73 5
5 72 80 55 60 9 12 56 82 66 79 91 52 22 23 73 29 2 54 52 38 2 29 61 45 81 47
33 26 31 6 23 34 62 56 26 22 77 64 74 77 81 99 51 96 15 56 75 95 37 22 36 9
5 14 5 46 75 69 68 13 11 73 62 33 52 92 58 22 44 77 69 21 12 23 68 79 61 3 9
4 58 22 57 65 44 27 60 11 36 54 18 72 77 62 49 4 53 59 18 70 75 98 3 21 93 6
4 25 63 88 37 98 11 71 21 35 73 36 45 86 63 95 90 68 64 73 86 94 13 45 30 30
18 25 98 33 72 59 18 38 85 31 95 86 5 23 18 49 58 89 39 53 11 87 3 5 40 98
| nc -u 10.0.0.2 8080
1 2 2 2 3 3 3 3 3 4 4 4 4 5 5 5 5 5 6 6 7 8 8 8 8 9 10 10 11 11 11 11 11 1
1 12 12 12 12 13 13 13 13 14 15 15 16 16 18 18 18 18 18 18 18 18 19 19 20
20 20 21 21 21 22 22 22 22 22 22 23 23 23 23 24 25 25 25 26 26 26 27 28
29 29 30 30 30 30 31 31 31 31 33 33 33 33 34 35 35 36 36 36 37 37 3 3
8 38 38 39 40 40 40 41 42 43 43 43 44 44 45 45 45 46 46 46 47 47 47 48 48
49 49 49 50 51 51 52 52 52 53 53 54 54 55 55 56 56 56 56 57 58 58 58
58 58 59 59 59 60 60 60 60 61 61 61 62 62 62 63 63 63 64 64 64 64 65 65 6
6 66 66 67 68 68 68 68 69 69 69 69 70 70 71 71 71 71 72 72 72 72 73 73 73
73 73 73 73 74 74 74 74 75 75 75 75 76 77 77 77 77 77 78 78 79 79 79
80 81 81 81 81 82 82 82 82 84 84 85 85 85 86 86 86 86 86 86 87 87 88 88 88
8 88 89 89 89 90 91 92 93 93 94 94 94 95 95 95 95 95 95 95 95 96 96 96 96
97 98 98 98 98 98 98 99 99 99 100 100
```

Figure A.2: The sorting service, where it can be seen that a list of numbers is requested, and the sorted version is returned.

A.3 monteCarlo



```
axel@axel-VirtualBox: ~/...
axel@axel-VirtualBox:~/mini-os$ sudo xl create -c domain_config
Parsing config from domain_config
Xen Minimal OS (pv):
  start_info: 0x141000(VA)
  nr_pages: 0x6400
  shared_inf: 0xddb24000(VA)
  pt_base: 0x144000(VA)
  nr_pt_frames: 0x5
  mfn_list: 0x10f000(VA)
  mod_start: 0x0(VA)
  mod_len: 0
  flags: 0x0
  cmd_line:
    stack: 0xeb520-0x10b520
MM: Init
  _text: 0(VA)
  _etext: 0xa12db(VA)
  _erodata: 0xa8000(VA)
  _edata: 0xab658(VA)
  stack_start: 0xeb520(VA)
  _end: 0x10e490(VA)
  start_pfn: 149
  max_pfn: 6400
Mapping memory range 0x149000 - 0x6400000
setting 0-0xa8000 readonly
skipped 1000
MM: Initialise page allocator for 179000(179000)-6400000(6400000)
Adding memory range 17a000-6400000
MM: done
Demand map pfns at 100000000000-100000000000.
Initialising console ... done.
gnttab table mapped at 0x100000000000.
Initialising scheduler
Thread "idle": pointer: 0x0x17c078, stack: 0x0x180000
Thread "xenstore": pointer: 0x0x17c0d8, stack: 0x0x190000
xenbus initialised on irq 1
Thread "shutdown": pointer: 0x0x17c138, stack: 0x0x1a0000
Thread "netfront": pointer: 0x0x17c198, stack: 0x0x1b0000
Thread "sml": pointer: 0x0x17c1f8, stack: 0x0x1c0000
***** NEIFRONT for device/vif/0 *****

net TX ring size 256
net RX ring size 256
backend at /local/domain/0/backend/vif/23/0
mac is 00:16:3e:7f:8c:22
*****
Thread "netfront" exited.

```

```
axel@axel-VirtualBox: ~/mini-os
axel@axel-VirtualBox:~/mini-os$ echo 3650 | nc -u 10.0.0.2 8080
3.1408219
```

Figure A.3: The monteCarlo service, where a request with 3650 is sent, resulting in an estimation of pi of 3.1408219.

Appendix B

Region-annotated code from profiling

B.1 listen function in network-library

```
1 fun listen attop r1 [] (v774) =  
2 (* .. *)  
3   let (* ... *)  
4     region r312417:inf;  
5     val ethFrame =  
6       let region r312415:1 in funcall extract[atbot r312417] <rawTap,  
        ↪ 0, NONE atbot r312415> end;  
7     region r312425:3, r312427:inf, r312429:2;  
8     val v809 = funcall decode[atbot r312429,atbot r312427,atbot  
        ↪ r312425,attop r4] ethFrame;  
9 (* ... *)
```

Figure B.1: Here the listen function is shown, simplified to show only the part where regions referenced in figure 5.6 is used. These regions, r312417 and r312427 has been highlighted and match the color on figure 5.6.