Optimizing Boot Times and Enhancing Binary Compatibility for Unikernels

Daniel J Chiba

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Changwoo Min
Robert P. Broadwater

May 08, 2018
Blacksburg, Virginia

Keywords: Virtualization, Unikernel, Hypervisor

Optimizing Boot Times and Enhancing Binary Compatibility for Unikernels

Daniel J Chiba

(ABSTRACT)

Unikernels are lightweight, single-purpose virtual machines designed for the cloud. They provide enhanced security, minimal resource utilisation, fast boot times, and the ability to optimize performance for the target application. Despite their numerous advantages, unikernels face significant barriers to their widespread adoption. We identify two such obstacles as unscalable boot procedures in hypervisors and the difficulty in porting native applications to unikernel models. This work presents a solution for the first based on the popular Xen hypervisor, and demonstrates a significant performance benefit when running a large number of guest VMs. The HermiTux unikernel aims to overcome the second obstacle by providing the ability to run unmodified binaries as unikernels. This work adds to HermiTux, enabling it to retain some of the important advantages of unikernels such as fast system calls and modularity.

Optimizing Boot Times and Enhancing Binary Compatibility for Unikernels

Daniel J Chiba

(GENERAL AUDIENCE ABSTRACT)

Cloud computing provides economic benefits to users by allowing them to pay only for the resources that they use. Traditional virtual machines, so far the mainstay of cloud computing, come with a large number of features that are unnecessary for most cloud applications. Unikernels are specialised virtual machines that are compiled with only the features required to run the target application on top of a hypervisor. They have reduced memory requirements, short boot times, fast system calls, enhanced security and greater customizability. Despite these advantages, unikernels have not gained significant traction in industry. One reason is that existing hypervisors were not designed with unikernels in mind. Specifically, we show that for the Xen hypervisor, boot times rise exponentially with the number of VMs running on the system. The small size of unikernels allows us to run a much larger number of guest VMs than was previously possible, but these rising boot times present a major bottleneck. This thesis analyses the cause of this overhead and presents a solution that leads to a 4x reduction in the overall time required to boot 500 unikernels at once. Another reason for the slow adoption of unikernels is the difficulty involved in porting legacy applications to unikernel models. The HermiTux unikernel aims to remove this effort by allowing users to run unmodified, statically compiled executables compiled for Linux. In doing so, however, we lose the ability to modularise the unikernel for the application concerned, and also reintroduce a major source of overhead from regular applications - namely system calls. This thesis presents techniques based on binary analysis and binary rewriting that enable us to regain these advantages of unikernels in HermiTux.

Dedication

This thesis is dedicated to my parents.

Acknowledgements

There are many people without whom this thesis would not have come to fruition, and I would like to thank the following people here:

My advisor, Dr. Binoy Ravindran, for providing me with the opportunity to work with him and the amazing people at the Systems Software Research Group, and for his valuable guidance throughout.

My committee members for taking the time to review and provide valuable feedback on my thesis.

Dr. Pierre Olivier, for having the patience to share some of his immense knowledge and amazing ideas with me, and for all the brainstorming and debugging sessions. It would not have been possible without him.

All the members of the Systems Software Research Group, who were always ready to provide help and advice when needed.

My parents and brother, for their perennial love and support.

The sponsors - this thesis was supported in part by ONR under grant N00014-16-1-2104.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last two decades, innovations in virtualization technologies, such as the hypervisors introduced by VMware [55] and Xen [13], have enabled the rise of cloud computing. Under this paradigm, cloud providers owning a large number of high-powered physical servers rent out these resources to their clients, who pay the provider for only the resources that they use. These resources include computing time, computing power and memory. This scheme provides several benefits to cloud users. First, it presents a low initial investment, obviating the need to buy expensive hardware. It also frees the client from having to maintain that hardware, which gets especially costly during times of low demand when the hardware sits idle. Instead, it is more beneficial to the client if they only rent hardware based on their need at any given time. In addition, due to economies of scale, the low running costs that a cloud provider could achieve would be difficult for a client to attain.

Hypervisors, also known as virtual machine monitors or VMMs, are one of the indispensable technologies responsible for the success of cloud computing. A hypervisor enables the running of multiple operating systems in isolation on a single host machine. A high-powered server computer could therefore run a large number of "virtual machines" (VMs), wherein the user space applications are unaware of this virtualization. The hypervisor is responsi-

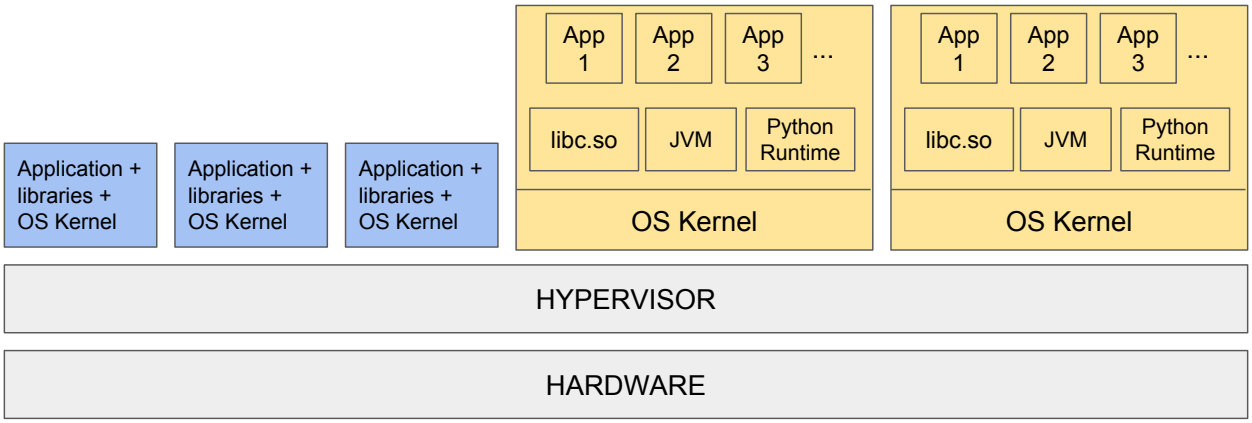

Figure 1.1: A cloud computing scenario where traditional virtual machines (yellow) are shown running alongside unikernels (blue) on top of a hypervisor on the same host.

ble for effectively multiplexing hardware resources among the virtual machines, as well as maintaining security between them.

Under the most popular cloud computing model, labelled infrastructure as a service (IaaS) [7], clients rent the cloud provider's resources by running virtual machines on their servers. This sort of service is provided, for example, by Amazon's EC2 and Google's Compute Engine. The client can then run whatever application they want within this virtual machine, such as a web server or a database. Traditionally these virtual machines (also known as guests) run full-fledged general purpose operating systems, such as Linux distributions, which are capable of running a wide variety of software.

However, a lot of the features that come with these guest operating systems are not required to run the client's target applications. Cloud users usually only desire to run a single application within their virtual machines [37]. These insights led to the rise of *unikernels* - lightweight virtual machines that contain only those features of the operating system and standard libraries required to run the target application. This leads to unikernels having significantly smaller image sizes, which provides several benefits - a lower memory and disk footprint, leading to lower costs for the client and a higher per-host density for the provider; faster boot times (on the order of milliseconds [43]); the ability to tailor libraries and the kernel for the target application; and enhanced security by virtue of a reduced attack surface.

Further, since unikernels only run a single application, they can make do without some of the security features present in traditional operating systems, such as the separation between user space and kernel space. This means that system calls become regular function calls, thereby improving performance.

Despite the advantages presented by unikernels, there are a number of barriers to their widespread adoption. The first barrier tackled in this work is that current hypervisors were designed to run traditional VMs. As a result, they cannot scale to support the large number of guest VMs that the small size of unikernels allows. The second barrier that we aim to solve is the difficulty involved in running legacy applications as unikernels. This includes the general difficulty faced in porting the build infrastructure and code base of existing software, as well as the inability to run proprietary software as unikernels due to the absence of source code (existing unikernel models require the application's source code to create a unikernel). HermiTux, described in 2.3, aims to alleviate these problems by enabling unmodified binaries to be run as unikernels.

1.1 Contributions

With the aim of removing some of these hurdles and helping to accelerate the adoption of unikernels, this thesis makes the following contributions:

1. Current hypervisors were not designed with unikernels in mind. The small memory requirements of unikernels means that a single host machine is capable of running a much larger number of guests (> 1000) compared to regular VMs. In particular, we show that the boot process of Xen does not scale for such a scenario: the boot time of a unikernel increases super-linearly with the number of running VMs. In Chapter 4 we analyse the causes of this bottleneck and propose a solution that brings the boot time for the 500th unikernel back down to millisecond range (from several seconds).
2. The HermiTux unikernel was designed to run unmodified, statically compiled binaries as unikernels. Existing unikernels require the source code of the target application to

create a unikernel. This enables them, at compile time, to include only the services required by the application to run. However, since the target applications of HermiTux have already been compiled, this is not an option. Chapter 5 presents a solution to this which involves static analysis of the binary to determine exactly what kernel facilities it would need from the unikernel, and hence exclude whatever is not required from the final unikernel image. For this purpose we focus on statically identifying which system calls are made by the binary.

3. The statically compiled binaries run by HermiTux have been compiled for Linux and will therefore request kernel services by making system calls according to the Linux convention. This involves moving the system call arguments to the appropriate registers and executing the `syscall` assembly instruction on x86_64 hardware. HermiTux is able to capture system calls made in this way. This is in contrast to existing unikernels, wherein the application code is compiled against libraries provided by the unikernel that simply make function calls to request kernel services. The latency of system calls in HermiTux is therefore much higher compared to the function calls on existing unikernels. In Chapter 6 we use binary rewriting techniques to replace the `syscall` instruction with a function call, thereby regaining this benefit of unikernels.

1.2 Limitations

We note that this thesis has a few limitations in its scope. First, the boot procedure optimizations presented in Chapter 4 are specific to the Xen hypervisor, and applying them to other hypervisors is not possible. Xen is, however, a very popular hypervisor (it is used in Amazon's EC2 cloud service), and a number of unikernels are able to run on Xen.

Similarly, the techniques presented in Chapters 5 and 6 are specific to the context of HermiTux. This is primarily due to the fact that the problems we're trying to solve are specific to unikernels that provide binary compatibility, and HermiTux is the only one that does so. If other unikernels were to provide binary compatibility by running statically compiled

Linux executables, these would also make system calls according to the Linux convention. Therefore, the system call identification technique in Chapter 5 would be useful to determine the kernel functionalities required by the user application. Similarly, the binary rewriting technique in Chapter 6 would be useful in alleviating the overhead incurred by the `syscall` assembly instruction.

1.3 Thesis Organisation

The remainder of this thesis is organised as follows:

Chapter 2 presents a comparison of various operating system models (Section 2.1), and a more in-depth look at unikernels and their benefits (Section 2.1.4). This is followed by a more detailed description of the Xen hypervisor (Section 2.2), which is the focus of Chapter 4. Finally, we provide an overview of HermiTux (Section 2.3), which is the basis for the optimizations in Chapters 5 and 6.

Chapter 3 presents related work. In particular, this includes prior work that has been done to reduce the boot time of virtual machines on Xen, as well as work related to the binary analysis and rewriting techniques that are employed in Chapters 5 and 6.

Chapter 4 demonstrates the scalability issues of the existing boot process of the Xen hypervisor (Section 4.1). We then present an analysis of the causes of this overhead and how we alleviate them (Section 4.2). Section 4.3 evaluates the speedup achieved by our solutions.

Chapter 5 shows how we modularise HermiTux based on the system calls being made by the target application. Section 5.1 explains the benefits of modularisation and why the methods used by existing unikernel models will not work for HermiTux. Section 5.2 presents the techniques used to identify each system call. Section 5.3 presents an evaluation based on the number of successfully identified system calls and the reduction in code size of HermiTux for various programs.

Chapter 6 shows why system calls are a major source of overhead that we would like to remove (Section 6.1). In Section 6.2 shows how we remove the overhead associated with the `syscall` assembly instruction, while Section 6.3 compares the resulting latency with that of the existing system.

Chapter 7 concludes the thesis and also presents future research opportunities that can further enhance the appeal of unikernels.

Chapter 2

Background

This chapter provides the background information necessary for the rest of the thesis. Section 2.1 briefly describes various operating system models in order to help locate unikernels in this classification, before going on to describe unikernels themselves in Section 2.1.4. Since Chapter 4 focuses on the boot time scalability of the Xen Hypervisor, Section 2.2 describes Xen along with its boot process. Section 2.3 describes the HermiTux unikernel, which is the basis for the optimizations presented in Chapters 5 and 6.

2.1 Operating System Models

This section first provides a brief overview of various operating system architectures, which aids in understanding the differences between them and unikernels. Section 2.1.4 then introduces unikernels, their advantages over traditional virtual machines, and provides a few examples of existing unikernel models.

2.1.1 Monolithic Operating Systems

This is the most popular operating system architecture, wherein all of the operating system's functionalities are contained within the kernel, and the entire kernel runs at the highest privilege level. The entire kernel runs in the same address space, which means that every procedure or variable within the kernel is visible throughout the kernel. Applications running in user space request services from the OS via system calls, causing a world switch from user space to kernel space. Since the entire kernel runs in the same address space, monolithic kernels have generally higher performance than microkernels. This is because there is no need for communication between disaggregated components as in microkernels. The downside of having all OS functionality within the kernel is that a bug in a single part of the kernel could potentially cause the entire system to crash, which can only be recovered from via a hard reboot. In addition, having such a large trusted computing base has negative implications for security.

2.1.2 Microkernels

Microkernels attempt to tackle some of the disadvantages of monolithic kernels by putting a minimum amount of OS functionality within the kernel (which is called, in this case, a microkernel). The majority of OS functionality runs in user space as separate applications. User programs then request services from the these applications through inter-process communication (IPC). Since each component of the OS is isolated from the other, a fault in one component will only crash that component and not the entire system. The same goes for security flaws - a flaw in one of the user space components is less likely to compromise the system than a flaw in a monolithic kernel, while the small code base of the kernel minimises the probability of security flaws within the kernel itself. The result is a highly modular and fault-tolerant operating system model, which is widely used in scenarios that require high reliability, such as real-time, industrial, and military applications [52]. In addition, the small size and modularity of microkernel systems allows them to be formally verified, as is the case with the seL4 microkernel [29].

2.1.3 Library Operating Systems

Library operating systems (LibOS), pioneered by Exokernel [22], were the initial inspiration behind unikernels [38]. Library operating systems were introduced as a way to move functionality from the kernel to user space on a per-application basis. The original goal of this scheme was to improve performance by enabling applications to manage resources according to their own needs, thereby allowing a high level of customisability. Under this model, a minimum amount of OS functionality would be handled by the kernel, while the remainder would be handled by 'libraries' in user space.

Exokernels also left hardware management to user space. Due to the difficulty involved in supporting a large variety of hardware devices, the LibOS model never really took off. As we explain in the following section, this drawback does not apply to unikernels. Drawbridge [44] and Graphene [53] are more recent library operating systems that focus on application compatibility and, by leaving low-level hardware management in the kernel, overcome some of the obstacles that prevented the exokernel model from flourishing.

Note that while microkernels [12, 23], also move OS functionality to user space, this functionality is still shared by all applications on the system. This means that inter-process communication (IPC) is required for applications to request OS services in a microkernel. In a LibOS, on the other hand, the libraries exist within the same address space as the applications they serve, eliminating the need for IPC.

Hypervisors, like exokernels, are only responsible for scheduling, memory management, and interrupt handling. However, in hypervisors, hardware devices can be managed by the host operating system or by special privileged guest VMs, freeing them of the need to support the myriad hardware devices available.

2.1.4 Unikernels

The traditional cloud computing model involves running an application, say a web server, within a virtual machine and deploying this virtual machine on servers managed by the cloud provider. This provides cost benefits to the client, since they pay only for the resources (memory, time, compute power) that they need, when they need them. So a client requiring 10 GB of memory would pay more than a client using 2 GB, assuming other metrics remain the same. The virtual machine in question is usually a general purpose Linux distribution capable of running a wide variety of software, providing a myriad utilities, and supporting a wide variety of hardware.

However, most cloud clients only run a single application within their virtual machine [37], thereby rendering a large number of the features provided by such operating systems unused. For example, a regular Linux distribution includes a large number of drivers to enable support for a diverse set of hardware. In a virtualized environment, drivers are usually provided by the hypervisor, and communication with the hypervisor is sufficient to retain wide hardware support. These distributions also come with a number of utility applications out-of-the-box, such as text editors, package managers, etc. Finally, in order to support a wide variety of software, they include various libraries and runtime environments, such as LibC, the Java Virtual Machine, etc. All of these superfluous features lead to unnecessary bloat in the virtual machine, meaning that the cloud client is paying for resources that aren't required to run their application.

Unikernels, on the other hand, are lightweight virtual machines wherein a single user application is statically compiled along with only the libraries and kernel features needed to run that application. Unikernels can therefore be considered as a form of library operating system for the cloud [38]. This leads to significantly smaller image sizes, as well as a reduced attack surface, which improves security. Figure 2.1 shows the basic architecture of a unikernel as compared to a traditional, full-fledged virtual machine.

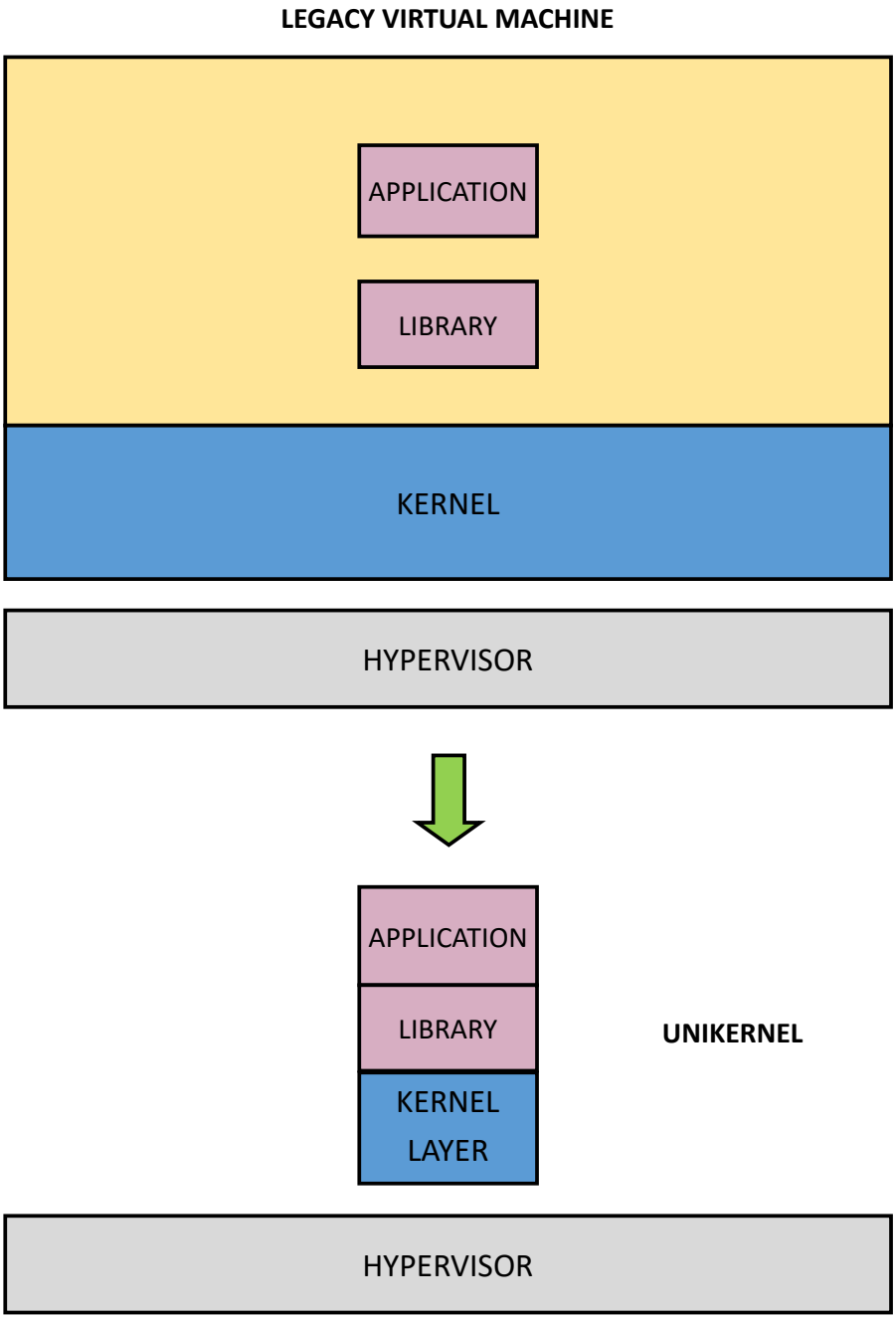

Figure 2.1: Unikernel Structure vs. Traditional VM

We now discuss some of the features from a general purpose OS kernel that become expendable in a unikernel. Drivers are usually provided by the host operating system or special privileged virtual machines, with the guest virtual machine really only needing to implement virtual device drivers that provide easier abstractions than actual hardware. This trait of virtualization was one of the enabling factors of unikernels, whereas the LibOS model did not gain traction due to lack of driver support.

Since unikernels are only designed to run a single application, there is no need for the kernel to protect different applications from each other. In fact, there is also no need to protect the kernel from user space either, since the effects of a malicious application will be restricted its own unikernel (protection between different unikernels/VMs is provided by the hypervisor). There is therefore no need to separate kernel space from user space, and all the code within a unikernel runs at the highest privilege level within a single address space.

Advantages of Unikernels

The minimalist nature of unikernels leads to lower resource utilisation in terms of memory, disk space and processing power. This not only provides cost savings for the user, but also allows cloud providers to increase the density of clients they can support on a single host.

In addition to the monetary advantages, unikernels also provide performance advantages. Their small size means that they have fast boot times, making them well-suited to scenarios where a quick ramp-up is required to meet a sudden increase in demand. Since all code runs within a single address space, this eliminates the costs incurred by context switches between applications and mode switches between different privilege levels (system calls).

Finally, the significantly lower amount of code present within a unikernel means that their attack surface is much smaller than that of a traditional VM, making them more secure. This property also makes them more secure compared to another popular lightweight virtualization solution - containers [40, 42, 60]. Containers provide isolation through the operating

system, which means that the trusted computing base is the entire host operating system. Hypervisors, on the other hand, have a smaller code base, and are known to provide stronger isolation than operating systems [17, 34].

Existing Unikernel Models

The first unikernel to be introduced was MirageOS [37, 38], which further enforced security by using a type safe functional programming language, namely OCaml. Other unikernel models that followed this philosophy include HaLVM [58], which uses Haskell, and Erlang on Xen [4]. Due to the relatively lower popularity of functional programming languages, most applications would need to be completely rewritten from scratch in order to be compatible with these unikernel models. On the other hand, there are unikernels such as IncludeOS [18], Mini-OS [1], HermitCore [33], etc. that support legacy programming languages such as C and C++. OSv [28] can run applications written in C, C++, Java, Ruby, Javascript and others, and can be run on a variety of hypervisors including Xen, KVM, VMware and VirtualBox.

Apart from cloud computing [11, 18, 28, 38], there are several other application scenarios for unikernels. They can be used for edge computing, where computation is moved closer to the end user to overcome network latency [31, 39]. They are also light enough to be used for IoT applications [20, 21], while at the same time providing the essential security that these applications require.

2.2 The Xen Hypervisor

The Xen hypervisor [13] is one of the most popular hypervisors, and is used by Amazon's EC2 cloud services. Xen is a type-1 hypervisor, which means that it runs directly on top of the hardware, taking the traditional place of the operating system. In Xen, guest virtual machines are known as *domains*. There is one domain with elevated privileges known as

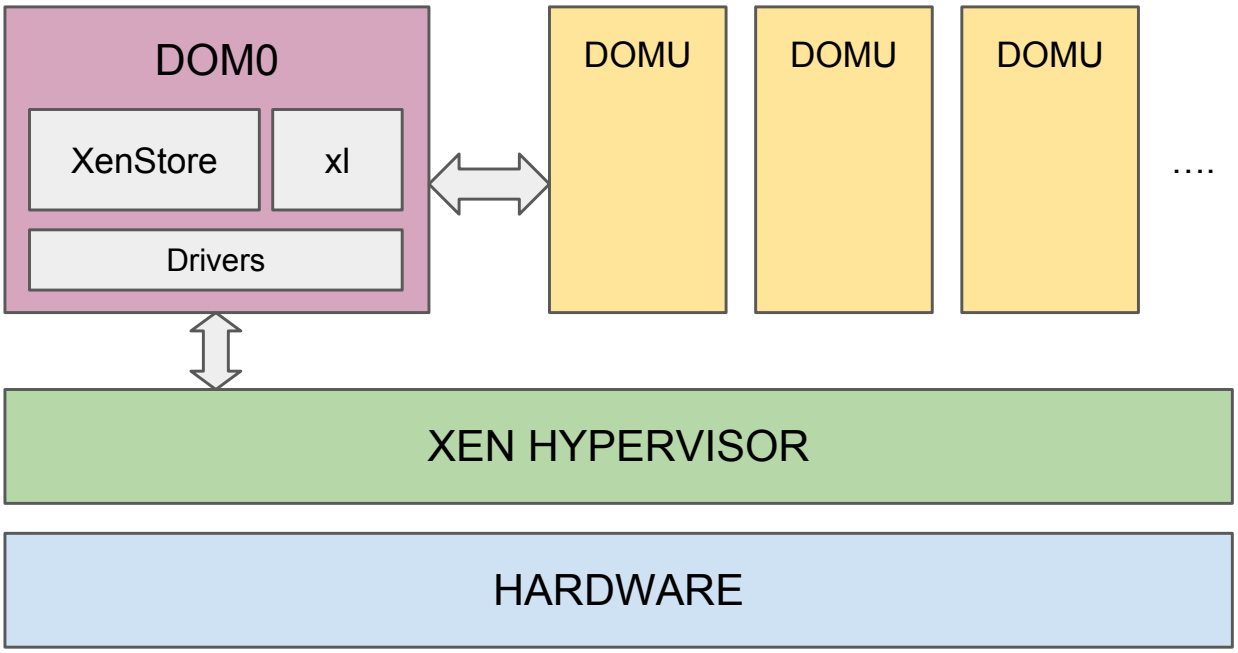

Figure 2.2: Basic Xen Architecture

Domain0, (or dom0). The remaining domains are known as *domU*s, where 'U' stands for 'user'. In addition to these two basic types of guests, Xen also supports stub domains, which are a special type of system domain used to offload certain Dom0 functionalities and run them within dedicated VMs. Due to its elevated privileges, within a cloud computing environment Domain0 (along with any stub domains) is under the control of the cloud provider.

The basic architecture of Xen is shown in Figure 2.2.

As a privileged guest, Domain0 is responsible for a number of functions within Xen. Since it runs a Linux distribution, it contains all of the drivers required by the guests to run. This saves Xen from having to implement every single driver that might be required. The Xen toolstack, which is used to control user domains and set their features, is also contained within Domain0. The toolstack is called `xl`, and among other things, it is responsible for controlling (creating, destroying, pausing, migrating, etc.) domUs. Since Chapter 4 focuses on optimizing the boot process of Xen, we now briefly describe the VM boot process executed by `xl`.

The creation of a new guest domain, initiated via the `xl create` command in Dom0, consists of the following steps:

1. `xl` checks the properties of the VM to be created to ensure that they are consistent.
2. Data structures used to manage the VM are set up. This involves interactions with the XenStore (described below).
3. Resources (e.g. memory) are allocated to the guest VM by the hypervisor.
4. Control is passed to the guest kernel, which carries out its boot process.

As we demonstrate in Section 4.2, the XenStore presents a major source of overhead for the boot process. We therefore briefly describe its functions here.

The XenStore is a key-value store hosted within Domain0. It is shared by all guest domains, and is used for communication between them, but mostly for configuration data as opposed to large transfers. Data within the XenStore is stored in a hierarchical namespace based on the path to each domain, similar to the `/proc` pseudo filesystem in Linux [9]. Among the information contained in the XenStore is the name of each domain, as well as whether the domain is a stub domain or not. As described in Section 4.2, this information is accessed multiple times during the boot process. Communication with the XenStore is possible via a Unix socket in Dom0 (default), a kernel-level API, or an `ioctl` interface [8].

2.3 HermiTux

2.3.1 Drawbacks of Existing Unikernels

Despite the various advantages of unikernels mentioned in Section 2.1.4, their adoption has been relatively slow. This is largely due to the difficulty in porting applications to unikernels. For legacy applications, running them in a unikernel that supports only modern type-safe languages would require a full application rewrite [16], which is often infeasible, since, in

addition to time, this requires in-depth knowledge of both the application as well as the target unikernel model [6].

Even for unikernels that employ legacy programming languages, there is some difficulty involved in porting them [6, 19, 40]. First, these applications might require certain libraries that are not supported by the unikernel being considered. Since unikernels generally come with their own modified versions of standard libraries, creating/adapting all such libraries that an application might require represents a non-trivial amount of effort on the part of the application developers. Second, existing build infrastructures, which could possibly consist of multiple Makefiles, configure scripts, auto configuration tools, Cmake scripts, etc., would need to be modified to compile the application as a unikernel. This, once again, represents a significant effort. Finally, the source code for the application to be ported might not be available, as is the case for proprietary applications. All current unikernel models require the source code for the target application in order to create a unikernel.

HermiTux was developed to alleviate some of these difficulties, in part by allowing unmodified, statically compiled Linux binaries to be run as unikernels. We first briefly describe HermitCore, which is the unikernel on top of which HermiTux was developed.

2.3.2 HermitCore

HermitCore [33] is a unikernel originally targeted at High Performance Computing (HPC) applications. Its primary design goal was performance isolation in order to achieve a reduction in OS noise and predictable running times.

HermitCore includes a version of the Newlib C library, modified to call HermitCore's internal functions where it would normally make system calls. User applications are then compiled against this library as well as the kernel, effectively creating a library operating system. While most system calls are serviced internally by HermitCore's kernel, file-system related ones are forwarded to the host, freeing HermitCore from having to implement a file-system internally.

HermitCore also includes networking functionality through the lightweight internet protocol (LWIP) stack.

HermitCore is capable of running on two hypervisors, namely Qemu/KVM and Uhyve. Uhyve, (short for unikernel hypervisor), as the name suggests, is a lightweight hypervisor optimized to run unikernels. It is based on the Ukvm hypervisor [59], and was developed as part of the HermitCore project. Similar to unikernels themselves, Uhyve's small size and code base allows it to enjoy the benefits of a small attack surface and higher performance for the type of guests it is targeting (unikernels). This is in contrast with Qemu, which supports a large number of hardware, host operating systems as well as guests.

One such advantage is more straightforward memory sharing between the host and guest. On startup, Uhyve allocates a large memory buffer that will be used as host (pseudo) physical memory. This effectively represents shared memory between the guest and host, eliminating the need to copy data. This in turn makes the forwarding of file-system related system calls to the host much faster than in Qemu. Like unikernels, Uhyve's initialization is also much faster than Qemu. This, combined with HermitCore's fast boot process, yields boot times on the order of milliseconds for a HermitCore unikernel running on Uhyve.

2.3.3 HermiTux

HermiTux (a portmanteau of HermitCore and Linux) enables us to run unmodified, statically compiled Linux executables within a unikernel. This required modifications to Uhyve's bootloader as well as the addition of a system call handler to vanilla HermitCore. At boot time, Uhyve first loads the HermitCore kernel into memory, followed by the loadable sections from the ELF file of the Linux executable. Then, after the kernel's initialization, it sets up the stack according to the Linux convention, which is what the application would expect in order to be able to run. The kernel then jumps to the entry point address of the program as indicated by its ELF header. The basic architecture is shown in Figure 2.3.

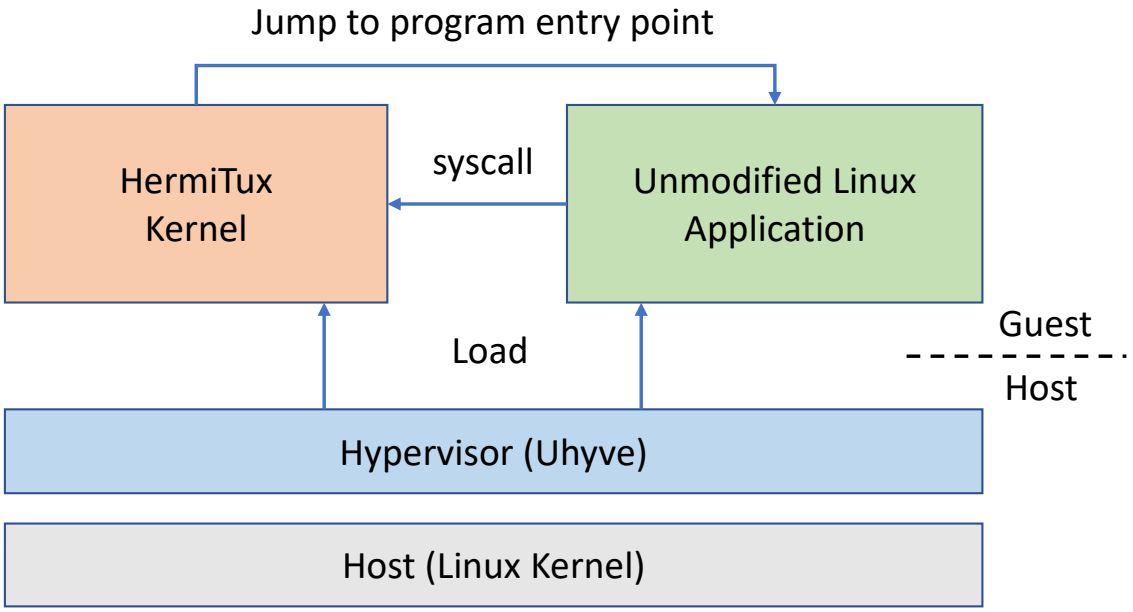

Figure 2.3: Overview of HermiTux

HermiTux is targeted at applications statically compiled for Linux. The C library implementation within these executables will make system calls according to the Linux convention. This involves putting the system call arguments into registers as dictated by the System V ABI [36] and executing the `syscall` x86_64 assembly instruction [10]. In addition to the arguments, the system call number is placed in the RAX register.

HermiTux therefore needs to be able to capture these system calls when made by the application. To do so, HermiTux registers a system call handler by writing its address to the appropriate model specific register (MSR) at startup. This handler includes a snippet of assembly code that first saves the registers on the stack, copies the stack pointer to RDI (which is the register used to pass the first argument to a function), and then calls a C function. This C function reads the register values off the stack, and based on the system call number present in RAX, passes control to the internal implementation of that system call.

This ability to capture system calls written in the Linux convention is one of the key features that makes HermiTux binary compatible with Linux applications. Chapter 6 describes how

we further optimize the system call handling process by actually modifying the Linux binary while maintaining application correctness.

In order to run a wide variety of applications, HermiTux was required to implement a number of system calls that were not supported by HermitCore. While this number is not nearly as large as the total number of system calls in Linux (368 in version 4.14 for x86_64), system call implementations currently represent around 20% of the HermiTux code base. This percentage is expected to increase as support for more applications is added over time. Furthermore, it is desirable to stick with the unikernel principle of only including OS features required by the application. Chapter 5 describes how we achieve this with HermiTux.

Chapter 3

Related Work

The related work presented in this chapter is divided into two sections. The first (Section 3.1) describes works related to optimizing boot times, which is the contribution of Chapter 4. After that, Section 3.2 describes work that is similar to that performed in Chapters 5 and 6 for the purpose of optimizing HermiTux.

3.1 Hypervisor Boot Process

In Chapter 4 we show that the boot time of guest domains on Xen does not scale well. Since VM boot time is a critical metric for assessing the performance of a cloud computing service, a number of works have studied and optimized this process. Studies such as those in [24, 41] analyse VM boot time in the cloud. They don't, however, perform such experiments for the large number of guest VMs possible using unikernels, nor do they attempt to analyse the reasons behind the observed performance.

In Section 4.2, we determine that a major source of overhead in the guest boot procedure on Xen is interactions with the XenStore. Jitsu [39] and LightVM [40] also make the same

observation as us. In Jitsu, the authors modify the XenStore to enable fast parallel transactions. While they achieve fast boot times, their work focuses on deploying unikernels in embedded environments. This prevents them from evaluating their optimizations on the large number of unikernels that could be run on high-powered server machines. As a result the scalability of their solutions is unclear. However, we note that the solutions presented by Jitsu are complementary to ours - while they optimize the XenStore itself, the solution we present in Section 4.2 simply reduces the number of interactions with the XenStore.

In LightVM [40], the authors remove the XenStore completely. Their solution is aimed specifically at lightweight virtual machines developed by them. While this scheme would work well for scenarios where only such lightweight guests are intended to be run on the system, it would cause issues for existing VMs that might use the XenStore. The optimized toolstack we present in Section 4.2 maintains the XenStore API, and can therefore run unikernels alongside legacy virtual machines without any issues. Further, we note that while it would be a one time operation, our solution does not require a system reboot, since only the toolstack, which operates in userspace, is modified. Unlike LightVM the hypervisor itself remains untouched.

Finally, multiple previous works [30, 32, 62] focus on optimizing the cloning and migration process of VMs. They then take advantage of these optimizations combined with the fact that new VMs will share the same initial state. This helps to convert the boot process to a cloning process for a fixed initial state. These solutions can be used in conjunction with ours, since the XenStore interactions that we remove are also required for restoring a VM.

Jitsu [39] and LightVM [40] focus on lowering the boot times of lightweight virtual machines. They both focus on Xen and the XenStore. Jitsu modifies the XenStore while LightVM removes it altogether. These solutions are ideal for scenarios where only unikernels/VMs of the intended type are going to be run on the system. The solution we presented in Section 4.2, on the other hand, does not modify the XenStore or its API. It is therefore fully compatible with existing VMs that may make use of the XenStore.

3.2 HermiTux Optimization Techniques

In Chapters 5 and 6, we present techniques that enable HermiTux to regain some of the advantages of unikernels - advantages that are lost in its quest to provide binary compatibility. These include modularisation and fast system calls. Since HermiTux is the only unikernel that we know of that provides binary compatibility, there aren't any works related to optimizations in this particular context. However, the techniques that we used to achieve these have been explored, so this section presents works related to these techniques.

In Chapter 5 we use system calls as the basis for our modularisation. This necessitates identifying all system calls that a statically compiled program can make. Tsai et al. presented a study on the usage of the Linux API [54]. As part of this study they statically identified the system calls being made by applications and libraries in the Ubuntu software repositories. The technique that they used for this analysis is not clear, but they mention that they cannot identify the system calls being made at 4% of the call sites. Since we intend to exclude unused system calls from a HermiTux image compiled for the target application, we cannot afford to miss out on any system calls, as doing so could potentially result in the application failing at runtime. We therefore provide a solution in Section 5.2 that, although slightly vulnerable to changes in the C library, manages to identify 100% of system calls for a large variety of applications.

Chapter 6 aims to reduce the overhead of system calls in HermiTux. This is caused by the fact that the target applications for HermiTux are statically compiled for Linux platforms, and will therefore make system calls according to the Linux convention. This involves execution of the `syscall` x86_64 assembly instruction, which causes a processor interrupt and is a major source of overhead. In Section 6.2 we describe a method that employs a binary rewriting technique to replace this instruction and hence turn it into a regular function call as is the case with traditional unikernels.

In our case, we only want to replace a single instruction (`syscall`) with another (`call`). De-

buggers implement a simple binary rewriting technique which, at first glance, could address our problem. They insert breakpoints by replacing the target instruction with the `INT 3` instruction [14]. This is a single-byte instruction (`0xCC`), so it can effectively replace any other assembly instruction without overflowing onto the next instruction. On the other hand, we need to replace `syscall` (two bytes long) with a `call` instruction, which is (at least) 5 bytes long. Since we are focused on improving performance, making use of an interrupt is out of the question, since the associated overhead is no better than that of a system call. We therefore need to come up with a slightly more involved technique.

Modifying compiled executable binaries is known to be a significant challenge [27, 50, 56]. Smithson et al. describe this issue in [50]. Simply put, if we want to add instructions to a binary (or replace an instruction with a bigger one), that would change the resultant address of every instruction after that. We would then need to change the target of every branch instruction that could transfer control to one of these modified addresses. However, finding all such instances is not easy [56]. While there have been multiple works such as [50] that try to address this, we instead make use of a 'trampoline' as described by Hollingsworth et al. in [26]. The technique described in [26] cannot insert such trampolines at system call locations, which is something we overcome with the techniques described in Section 6.2. Finally, we note that dynamic binary rewriters would add a runtime performance overhead and are therefore undesirable for our use case, especially when it is possible to rewrite them statically.

Since the overheads associated with system calls have been an issue for a long time, there have been numerous works that aim to mitigate this overhead. Most such schemes involve 'batching' system calls together, thereby replacing multiple system calls with a single system call [47, 48, 51]. FlexSC [51] takes this idea step further by turning regular system calls into exception-less system calls that are executed asynchronously, allowing for flexibility in their scheduling. Cassyopia [47] achieves batching statically through compiler optimizations. We note that all of these works were targeted at systems where system calls are still indispensable.

Unikernels, on the other hand do not require system calls. Section 6.1 explains in more detail why this is the case. We therefore adopt an approach that removes system calls entirely. Also, while FlexSC achieves exception-less system calls, that scheme involves runtime overhead. In addition to the temporal cost involved, a runtime solution would simply add unnecessary bloat to the final unikernel image.

Chapter 4

Enhancing Guest Initialization Scalability in Unikernel Environments

This chapter describes how we optimize the Xen toolstack to swiftly boot VMs in scenarios where we might have large numbers of guest domains running on a single host machine, such as unikernel environments. Section 4.1 shows that the current boot process in Xen does is not scalable in such scenarios. In Section 4.2 we analyse, at a fine granularity, the sources of overhead that result in this poor scalability, and provide a solution for the same. Section 4.3 evaluates the work by providing the resultant boot times of unikernels after our optimizations, as well as showing that the speedup is a result of these optimizations.

4.1 Motivation

Hypervisors, the enabling technology of cloud computing, allow multiple virtual machines to run on a single host. Traditional virtual machines, which run full-fledged commodity operating systems, require at least a gigabyte of memory. Unikernels, on the other hand, require memory as low as a few megabytes. This allows us to run a much higher number

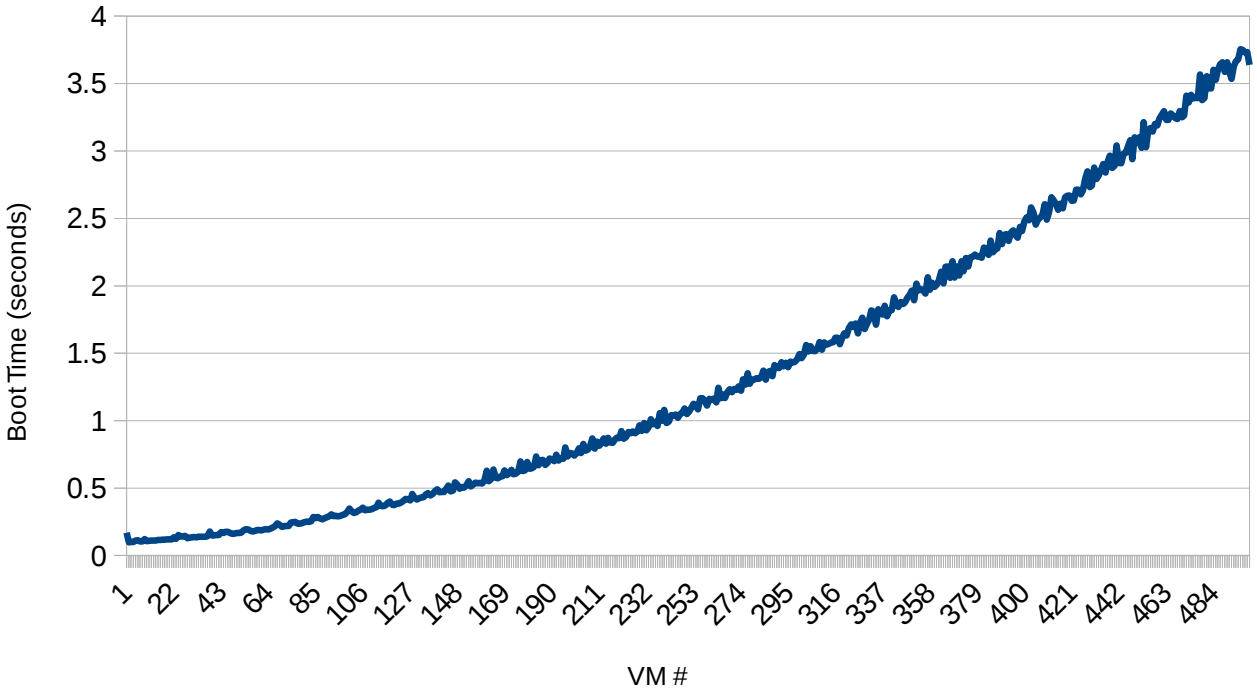

Figure 4.1: Boot time on Xen for the nth VM to be booted with (n-1) idle VMs running in the background

of guest VMs on the same host. However, as we will show, guest boot time on the Xen hypervisor does not scale well when running a large number of guest domains.

The elastic nature of today's cloud provides multiple benefits to both cloud providers and their clients. Clients only need to pay for the resources that they use, when they need to use them. This allows them, for example, to quickly scale up/down their services (by starting more virtual machines) to meet an increase/decrease in demand. This kind of on-demand scaling is highly dependent on boot time. A slow boot time would negatively affect the latency of the services provided and decrease overall quality of service. Also, the time required to boot a virtual machine is usually not billed to the customer [57], so a long boot time would result in a loss of revenue for the cloud provider. Due to unikernels' small size, their boot time is much lower than that of a traditional VM, but once again, scalability of the hypervisor is the issue.

Figure 4.1 demonstrates the boot time scalability of Xen. The guest domains are all instances

of the Mini-OS unikernel that go to sleep immediately after booting. This reduces the interference from running VMs on the boot process of the new VMs, ensuring that maximal processor resources are dedicated to the boot process. The VMs are booted sequentially, and the Y-axis indicates the boot time for the nth VM, with n - 1 idle VMs running in the background. We can see that the boot time increases super-linearly with the number of already running guests. While the maximum boot time of 4 seconds does not seem like much for a regular VM, it is significant for a unikernel; indeed it is 40 times that of the first unikernel to be booted. This analysis also reveals another problem - since it is difficult for a cloud client to know how many virtual machines are running on a particular host, the boot time is non-deterministic, which could lead to unpredictable behaviour in certain scenarios [43].

4.2 Implementation

In order to successfully alleviate this issue, we first need to identify the major sources of overhead. Through manual instrumentation of the `xl` source code, we observed that the percentage of time spent within the XenStore increases significantly with the number of background VMs. While this is 23% for two VMs, it shoots up to 96% for 512 VMs. The XenStore therefore represents a major bottleneck during the boot process.

In order to break this down further, we instrumented the various XenStore functions invoked by `xl` during the boot process with timestamps at the beginning and end of each. We notice that two functions - `xs_read` and `xs_get_domain_path` dominate the time spent within XenStore functions (see Figure 4.2). Further, we observe that (1) not only are they being invoked a higher number of times with each successive boot, but (2) the execution time of each invocation also increases, as shown in Figure 4.3. Both of these metrics increase linearly, which is what leads to the super-linear increase in overall boot time observed in Figure 4.2. In contrast, since the number of calls to `xs_write` remains constant, we see that its influence on the overall boot time actually reduces with an increase in the number of VMs

despite dominating earlier on. We therefore focus on `xs_read` and `xs_get_domain_path` for our optimizations.

Solving the problem of increasing execution times of these functions is straightforward. We notice that this is caused by Domain 0 creating a background process for every running guest that maintains a connection to the XenStore for the entire life of that guest. The performance of this daemon decreases with every additional connection due to the way the server event loop is written. The solution is simply to disable this daemon - an ability that Xen provides but does not enable by default, as it is not an issue with the low density of regular VMs.

However, we still want to reduce the other metric, namely the number of calls made to the XenStore during the boot process. For that, we first need to understand when the XenStore is used and what the purpose of these function calls are.

While there are multiple points during the boot process when each of the aforementioned functions is invoked, there are two particular call sites of interest. Both of these are within a `for` loop that iterates over each running domain, which explains the increase in the number of invocations seen in Figure 4.3.

The first loop is in the function `libxl_name_to_domid`, where `libxl_domid_to_name` (which in turn calls `xs_read`) is called for each domain in order to get that domain's name. This is done to ensure that the domain being booted does not have the same name as an existing domain, hence ensuring uniqueness of domain names in the system. If this rule is violated, domain instantiation fails.

The second `for` loop, in the function `libxl_list_vm`, contains a call to `libxl_is_stubdom`, which in turn makes one call to `xs_read` and one call to `xs_get_domain_path`. As the name suggests, this function is used to find out if any of the existing domains are stub domains. Stub domains are a special type of guest domain in Xen that are used for offloading certain functions performed by Domain 0 [3].

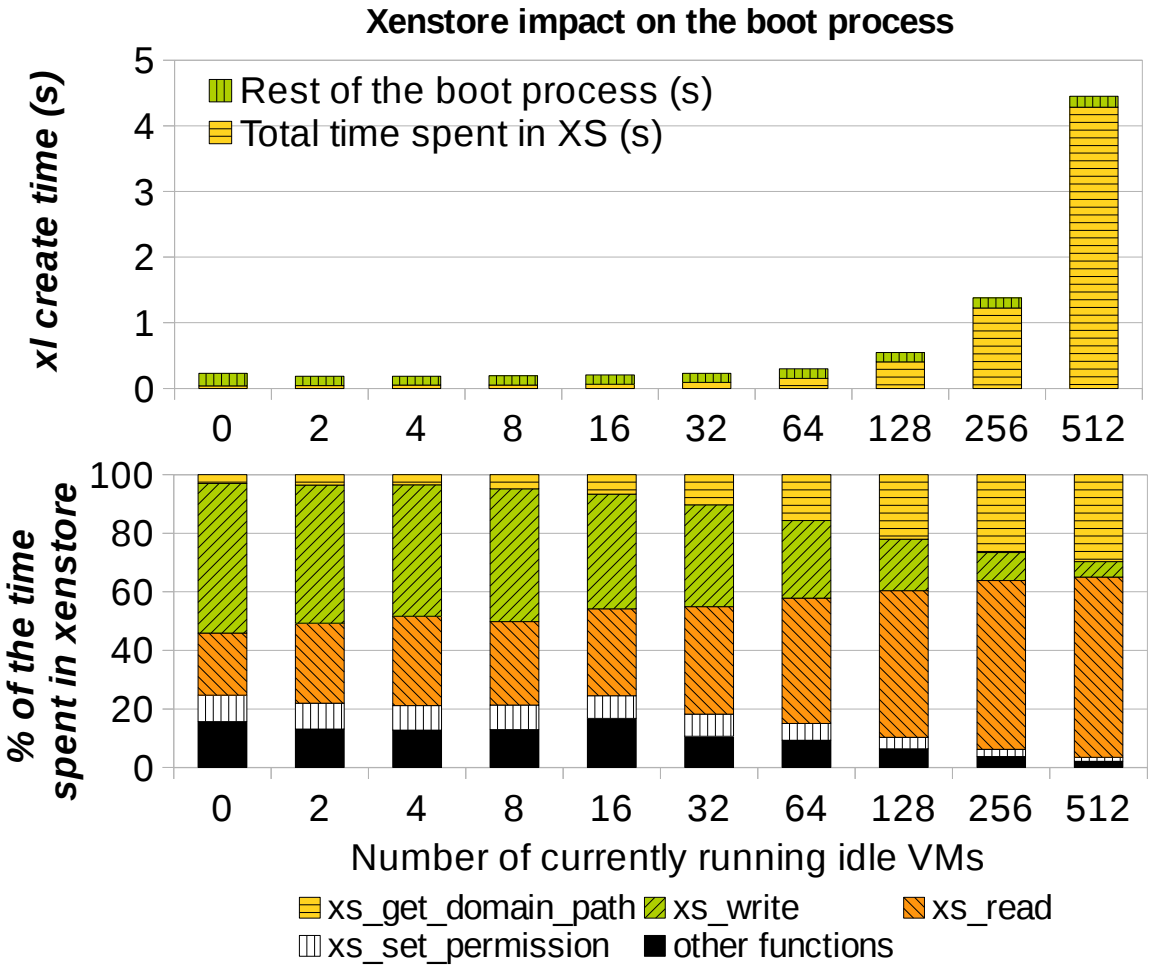

Figure 4.2: Percentage of time spent in the XenStore during the boot process (top) along with a per-function breakdown of the same (bottom).

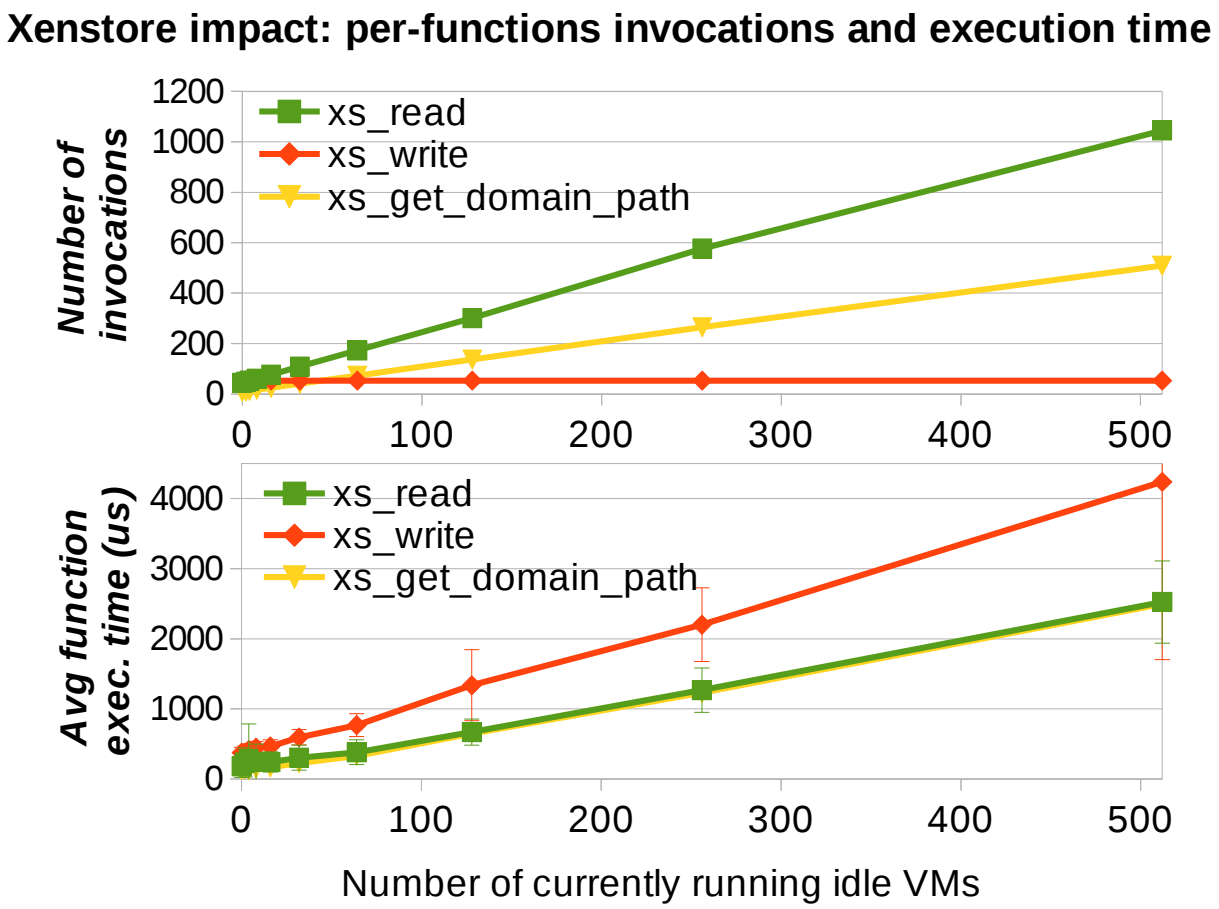

Figure 4.3: Number of calls to XenStore functions (top) along with their average execution time (bottom).

The findings from these two `for` loops are consistent with the graph in Figure 4.3, where, for 500 VMs, the numbers of calls to `xs_read` and `xs_get_domain_path` are roughly 1000 and 500 respectively.

Our solution is to use an in-RAM cache to store the data required by these three function calls. For this, we make use of a technique described in [43] to enable parallel booting of guest domains in Xen[1]. This technique transforms the `xl` toolstack into a daemon that listens for VM creation requests on a socket. At the same time, VM creation requests, each of which was a separate process in the vanilla Xen implementation, are converted to threads that share the same address space with each other and the daemon.

In order to save calls to the XenStore, we cache the data required by the aforementioned functions within a cache in the `xl` process itself. The cache consists of a list of structures with three properties - domain number, domain name, and a boolean value indicating whether the domain is a stub domain or not. Calls to `libxl_domid_to_name` and `libxl_is_stubdom` within `xl` (which call the XenStore) are replaced by calls to our own cache lookup functions. The cache itself is populated lazily, so if the data for a particular domain is not present, the default functions are called and the retrieved values are added to the cache. This effectively means that for every new domain that's booted, there is a constant number of calls to the XenStore. In addition, we also ensure that the cache is updated appropriately when a domain is renamed or destroyed.

The addition of the cache introduces negligible memory overhead. If we consider 4 bytes for the domain number, 64 bytes for the domain name, 1 byte for the boolean value and 8 bytes for the pointer to this structure, that gives us a total of 77 bytes for each VM. For a system running 500 guests, that adds just around 38 KB of additional memory requirements.

[1]Not a contribution of this thesis

4.3 Evaluation

The experiments below were carried out on a server class machine containing an AMD Opteron 6376 CPU with 64 cores and 128 GB of memory. The server consists of 8 NUMA nodes, each with 8 cores and 16 GB of memory. Our solution was implemented on top of Xen 4.7. Domain0 is set up with 4 VCPUs and 4 GB of memory. Dom0 is pinned on physical cores 0-3, while DomUs occupy the remaining cores. For the experiments in this section the guest domains comprise a Mini-OS unikernel running an idle loop. This was chosen so as to minimise the interference of running domains with the boot process carried out by `xl` on Dom0. They are each configured with one VCPU and 32 MB of memory.

To evaluate the XenStore cache, we measure the boot time of a single VM with a variable number of VMs running in the background. The results are shown in Figure 4.4. This figure shows the boot times for regular `xl`, `xl` with the daemon disabled, and `xl` with the daemon disabled and the cache enabled. We see that for 512 background VMs, the boot time reduces from several seconds to 0.3 seconds (note the logarithmic scale of the Y-axis).

Further, we break down the effects of the two optimizations by measuring the percentage of time saved by each, as shown in Figure 4.4. We see that for a small number of VMs, the XenStore cache does not provide a huge benefit; most of the benefit is provided through disabling the daemon. This changes when we get to a higher number of running VMs, with the cache providing reductions of 32% and 46% for 256 and 512 VMs respectively. We also counted the number of calls to the XenStore and confirmed that they remain constant.

Finally, we measured the time taken to boot 512 guest virtual machines in parallel [2]. The vanilla Xen implementation took 13 minutes and 57 seconds while our optimized version took just 1 minute and 51 seconds, giving us a performance boost of 87%.

[2]These measurements include the effect of the fine-grained locking described in [43], which was not a contribution of this thesis.

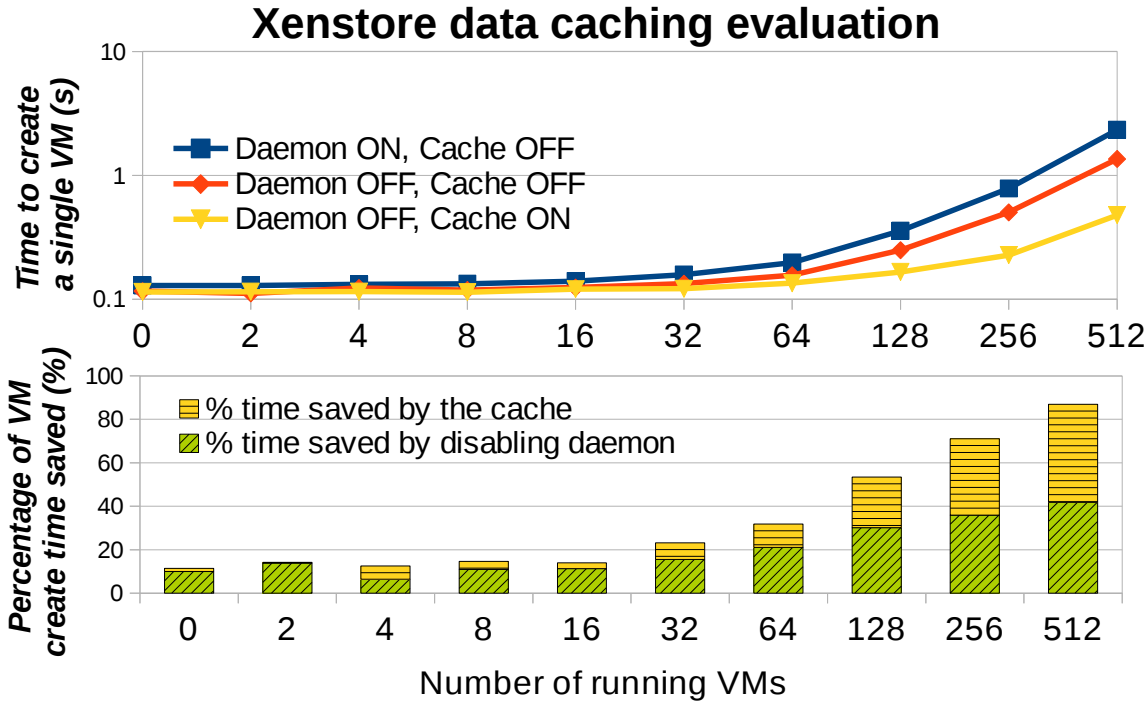

Figure 4.4: Boot times before and after our optimizations to `xl` (top), and the percentage of time saved by each optimization (bottom).

4.4 Discussion

It is important to note that the optimizations presented in this chapter are specific to the Xen hypervisor. This is primarily due to the fact that we focus on interactions with the XenStore, which is unique to Xen.

However, the popularity of Xen limits the impact of this specificity. Xen is the hypervisor used on Amazon Web Services' EC2 platform, which is the most popular IaaS cloud computing service. Further, a number of unikernels are able to run on Xen, including MirageOS, OSv, Rumprun and HalVM.

Chapter 5

System Call Identification for Modularisation of Binary Compatible Unikernels

This chapter starts by justifying our decision of using system calls as the basis to modularise HermiTux (Section 5.1). We then describe our approach to successfully identifying all system calls in a pre-compiled application binary (Section 5.2). Section 5.3 presents an evaluation of both our identification technique and the effects of modularisation on HermiTux.

5.1 Motivation

As mentioned in Section 2.1.4, one of the salient features of unikernel design is the ability to include only those features of the operating system that are needed to run the target application. This contributes to two of the biggest advantages of unikernels, namely a reduced memory footprint and reduced attack surface.

MirageOS [37] makes use of OCaml's module signatures to select the functionality to be included with the operating system image. Unikernels written in C and C++ [18, 33] can similarly make use of compiler and linker flags such as `--ffunction-sections`, `--fdata-sections` and `--gc-sections` to exclude code that isn't called by the user application. In HermitCore [33], the user application would initially be compiled to a relocatable object file, which would then be linked against libc.a from the modified version of Newlib, and libhermit.a. While the original HermitCore was not modular, achieving this was trivial with the help of the compiler and linker flags mentioned above.

HermiTux makes no assumption about the availability of source code, so we can no longer make use of such compiler features. Since our target applications have been statically compiled, they already contain an implementation of the C library within them. Modularisation at the level of the C library is therefore not possible.

We therefore need to focus on how best to select the kernel features that are actually required by the application, given that we only have access to the compiled executable itself. In a regular operating system environment, user mode programs will request such features via system calls. Since the applications run within HermiTux were originally compiled for regular operating system environments (specifically Linux), we can simply base our modularisation on the system calls being made by the application. The following section describes how we identify these system calls given only the binary.

5.2 Implementation

In order to correctly modularise HermiTux, we would need to identify **all** of the system calls that could potentially be made by the application. If our identification technique fails to identify a system call, our compilation process would exclude it from the kernel image. If this particular system call is subsequently invoked at runtime, it would result in failure as the system call handler would be unable to find the given system call implementation.

There are several ways to identify the system calls made by a program. The simplest and most straightforward way is to run the program under the strace utility [5], which prints out all the system calls made by that run of the program. The obvious pitfall of this dynamic approach is code coverage - certain possible execution paths could be skipped in some scenarios. We might therefore miss certain system calls that could be made in a production environment, even if they are rare. In addition, security concerns might discourage the user from running the application outside a sandboxed environment.

We could augment this approach with the help of fuzzing or symbolic execution to increase our code coverage, but such techniques are not guaranteed to have 100% coverage. It also does not address the security concerns mentioned above.

We therefore go with a static analysis approach. This involves searching for each instance of the `syscall` instruction within the disassembled application binary. On the x86_64 architecture, the Linux kernel checks the RAX register for the system call number when the system call is made. We must therefore identify the value contained in this register at the moment the `syscall` instruction is executed. If we can do this for every `syscall` instruction in the binary, that will give us all of the system calls that could be made by the binary at runtime. In addition to helping with modularisation, this method allows us to identify the system call being made at a particular call site, which is something we leverage in Chapter 6.

While observing the object dumps of several executables, it appears as though most of the time, the RAX register is loaded with an immediate value in the instruction immediately preceding the `syscall` instruction. In these cases identifying the system call number is fairly straightforward. However, there were numerous instances wherein the loading of RAX and `syscall` were separated by multiple instructions. In such cases it becomes necessary to analyse the control flow to make sure that we get the correct value that we're seeking. For example, consider the contrived snippet of code in Listing 5.1. Here, there are two possible values that RAX could have when the `syscall` is made, and this value is decided at runtime. In such a scenario we must include both system call implementations in our modularised

Listing 5.1: Assembly code snippet (AT&T format) demonstrating how different system calls could be made by the same instruction

```
1     400535: 83 7d ec 01             cmpl   $0x1,-0x14(%rbp)
2     400539: 75 07                   jne    400542 <main+0x1c>
3     40053b: b8 27 00 00 00          mov    $0x27,%eax
4     400540: eb 05                   jmp    400547 <main+0x21>
5     400542: b8 6e 00 00 00          mov    $0x6e,%eax
6     400547: 0f 05                   syscall
```

kernel. The other scenario we need to be aware of is that it might not necessarily be an immediate value being moved into RAX - it could be the value from another register. In this case we would need to find the value stored in that source register when this instruction is executed.

In order to analyse the control flow of the program, we make use of a tool named Dyninst [25], although this could be easily replaced by other tools that provide such control flow analysis features. The algorithm used to identify system calls is shown in Listing 5.2. The basic idea of this algorithm is as follows: We iterate backward over instructions within the `syscall` instruction's basic block, looking for an instruction that loads the RAX register. If an immediate value is found, we stop there. If no instruction within the basic block modifies RAX, we search recursively through all of the incoming basic blocks in a similar manner. If, at any point, we encounter an instruction that loads RAX with the value from another register, we continue the process starting from that instruction and looking for instructions that modify the register in question (indicated by the `dest_register` argument in Listing 5.2).

We did observe a few corner cases that aren't covered by the algorithm. One such case involves a value being moved into RAX from memory. This was observed in the `_int_free` function of the GNU C Library. We were able to look up the source code alongside the disassembly to determine which system call was being made by that particular `syscall` instruction. We then create a lookup table which is only used when the system call number appears to be coming from memory. This is not ideal, but since system calls are almost

Listing 5.2: Pseudocode for System Call Identification

```
/* Populates possible_syscalls with all possible values that could
   be present in dest_register at assign_address */
function get_value_in_register(dest_register, assign_address,
        current_block, possible_syscalls)
    reversed_block = current_block.reverse()
    foreach instruction in reversed_block
        /* Occurs when value in RAX comes from another register */
        if instruction.address > assign_address
            continue
        endif

        if instruction assigns to dest_register
            if source_operand is immediate
                possible_syscalls.append(immediate_value)
                return
            endif
            if source_operand is a register
                get_value_in_register(source_register,
                        instruction.address, block,
                        possible_syscalls)
            endif
        endif
    endforeach

    /* No assignment was made in the current block; we need to
       check incoming blocks */
    foreach source_block in incoming_blocks
        get_value_in_register(dest_register,
                source_block.last_address, source_block,
                possible_syscalls)
    endforeach
endfunction
```

always made by the C library, it is a viable solution for these rare occurrences. In addition, when backtracing through basic blocks, we limit ourselves to basic blocks within the same function as the `syscall` instruction under consideration, except when we're in the C library's `syscall` function.

Finally, once we have identified the system calls required by the application, we need to include or exclude parts of the kernel code base accordingly. We first placed every system call implementation into its own compile unit (C file), and surrounded their invocations (in the system call handler) with preprocessor macros. The build system, which consists of a number of CMake scripts, will then invoke the system call identification program. Based on the output it will then include/exclude the relevant C files and enable/disable the macros.

5.3 Evaluation

As part of the evaluation, we first look at the effectiveness of the technique described above in terms of the number of executables for which we can successfully identify all system calls being made (Section 5.3.1). After that, Section 5.3.2 examines the impact of modularisation on HermiTux in terms of reduction in kernel code size.

5.3.1 System Call Identification

We used our system call identification technique to determine the system calls being made by a number of applications. These applications were compiled statically against both musl and Glibc. Our findings are shown in Table 5.1. For all of these applications we were able to successfully identify all of the system calls being made with both libraries. Further, the only call site where we needed to use our lookup table was the `_int_free` function in glibc.

In addition to the applications shown in Table 5.1, we also applied our technique to the coreutils suite [2] of applications compiled against Glibc. Out of 106 programs there were

Table 5.1: System call identification on selected programs

Program	`syscall` Instructions		Unique System Calls		Unidentified System Calls	
	musl	**glibc**	**musl**	**glibc**	**musl**	**glibc**
Minimal (return 0)	7	124	7	44	0	0
Hello World	15	124	12	44	0	0
Postmark	92	132	39	51	0	0
Mongoose	97	152	40	62	0	0
netio	26	133	19	51	0	0
Blackscholes	55	124	18	44	0	0
Sqlite	90	138	43	54	0	0

two containing system calls that couldn't be identified. For both of them, the sources were `syscall` instructions in the `__nptl_setxid` and `sighandler_setxid` functions. Digging deeper, we find that the system call numbers in both functions once again come from memory. However, in these functions the system call made depends on the arguments to the functions, which can only be determined at runtime. Looking at the source code tells us that only a specific set of system calls can be called from these spots, namely the `setxid` family of system calls. One option to overcome this problem is to use the lookup table to return the system call numbers for all of them. Since most of these system calls have very small implementations, it shouldn't significantly add to the codebase of the resultant HermiTux image.

5.3.2 Modularisation of HermiTux

In this section we show the reduction in code sizes achieved by modularising HermiTux. In order to get a more accurate picture independent of debugging symbols, stripped binaries, etc., we compare only the size of the `.ktext` sections of the resultant HermiTux executables, which contain the kernel code. The results for selected applications, all compiled against musl, are shown in Table 5.2.

Table 5.2: Effects of modularisation on kernel code size

Program	Kernel Text Size (KB)	Percent Reduction
Full	112.72	0.00
Sqlite	99.94	11.34
Postmark	96.53	14.36
Blackscholes	93.50	17.05
Hello World	90.35	19.84
Minimal	88.06	21.87

In this table *Full* denotes HermiTux compiled with all supported system calls, while *Minimal* represents a minimal program that just returns 0. We note that the difference for these two extreme cases is 22% of the maximum. As the number of system calls supported by HermiTux increases to support a wider variety of applications, the maximum code size will also increase accordingly. System call-based modularisation will therefore play an increasingly influential role as the project progresses.

5.4 Discussion

In Section 5.2, we cite code coverage and security concerns as factors that drove us towards using a static approach to identifying system calls instead of a dynamic approach. We note, however, that in certain scenarios a dynamic approach might actually be better.

A look at Table 5.1 shows, especially for Glibc, a large number of unique system calls present in most binaries. However, we have observed that a lot of these are not actually invoked in practice. If a program comes with a comprehensive test suite that is known to cover all possible execution paths, using this to identify system calls would provide a more accurate indication of which system call implementations should be included. Such an approach would also eliminate the need for a lookup table. In the absence of such a test suite, however, we contend that static identification of system calls is certainly the safer option.

Further, static analysis enables us to determine which system call is being made at a particular address in the binary, which is an important piece of information required for our system call processing infrastructure described in the next chapter. This analysis is not possible when a running a program under `strace`.

Finally, we note that identifying system calls to modularise a unikernel is something that is only required when we want to run unmodified, statically compiled programs within the unikernel. Since HermiTux is the only unikernel that currently provides binary compatibility, this approach is only applicable for HermiTux at the moment. If other unikernel models were to provide binary compatibility, the user applications would still request kernel services via system calls that use the Linux convention. The process for statically identifying the system calls in the binaries would then be identical to the approach described in Section 5.2. However, there would be some engineering effort required to refactor the code and build process of the unikernel in order to enable it to include or exclude parts of the code base based on the system calls identified.

Chapter 6

Efficient System Call Processing for Binary Compatible Unikernels

This chapter describes how we regain the fast system call feature inherent in unikernels while running a Linux executable within HermiTux. We first go over the need for system calls in regular operating systems (Section 6.1.1) and why these don't apply for unikernels (Section 6.1.2). We then goes on to describe the existing system call handling mechanism in HermiTux, and why it is still suboptimal (Section 6.1.3). Section 6.2 then explains how we alleviate this overhead through a binary rewriting technique. Section 6.3 evaluates the performance of this scheme.

6.1 Motivation

6.1.1 System Calls on Traditional Systems

In a regular multiprogramming environment, user space applications request services from the operating system kernel by making system calls. On the x86_64 architecture, this is done

using the `syscall` assembly instruction. The kernel, in turn, uses the `sysret` instruction to return control to user space.

In order to maintain security isolation between a user space application and the kernel, the kernel needs to carry out a number of tasks when a system call is invoked, such as switching to a temporary kernel stack and copying user data into the kernel's address space. The kernel will then look up which routine to call depending on the system call number, execute it and return control back to user space. In addition, the `syscall` instruction causes the processor to trap (similar to an exception or interrupt) and switch from privilege level 3 to privilege level 0, while the reverse is true for the `sysret` instruction. All these operations result in system calls incurring significant overhead every time they are invoked. As a result, there have been numerous works [45, 51] attempting to mitigate the performance impact of system calls.

6.1.2 System Calls in Unikernels

Unikernels, on the other hand are naturally immune to the effects of system calls. Since unikernels are only used to run a single application at a time, there is no need for the kernel to provide isolation or protection of any sort between applications. It also means that we do not need to protect the kernel from the application, since a malicious application would only end up compromising its own unikernel. When a malicious unikernel (or traditional VM) could potentially affect other guests on the system, it is presumed that such protection is provided by the hypervisor.

As a result, all code within a unikernel runs within a single address space at the highest privilege level and there is no separation between user space and kernel space. In such a scenario, the application running within a unikernel requests kernel services using function calls instead of system calls.

6.1.3 System Calls in HermiTux

In vanilla HermitCore [33] user programs are linked against a customized version of the Newlib C library, which is modified to invoke HermitCore kernel functions instead of system calls when kernel services are required. In HermiTux, on the other hand, since we are loading unmodified, statically compiled binaries, these binaries will contain unmodified versions of the C libraries they were compiled against, which will make system calls using the Linux convention. In order to handle this, HermiTux also implements its own system call handler, which redirects control to the specific system call implementation depending on the system call number. This process is described in more detail in Section 2.3.

As a unikernel, HermiTux does not need to perform any of the operations described above that are mandated by security requirements. Also, the processor automatically stores the address of the instruction following the `syscall` instruction in the RCX register. Since all code on HermiTux, including that of the application binary, is executed in ring 0, HermiTux can simply jump to this address after the system call is serviced. We therefore avoid using the more expensive `sysret` instruction, which would have been unavoidable if we needed to switch back to ring 3 execution.

All of these optimizations result in HermiTux system calls incurring significantly less overhead compared to Linux system calls. However, it is still significantly slower than a regular function call, leaving plenty of room for improvement. This is primarily due to the continued use of the `syscall` assembly instruction and the exception it causes. The following section shows how replacing that instruction with a `call` instruction can improve performance. Note that such a replacement is only doable in a unikernel context since there is no need for privilege protection or a mode switch.

6.2 Implementation

Prior works have proposed numerous solutions for alleviating the overhead of system calls [47, 48, 51]. As explained in Section 3.2, these techniques are targeted at traditional multi-programming environments where the protection between user space and kernel space is still required, which is not the case for unikernels.

In order to reduce the overhead of system calls in HermiTux, we attempt to statically rewrite the application binary to replace each occurrence of the `syscall` instruction with a `call` instruction to the system call handler. In x86_64, `call` is the instruction used to make function calls. A straightforward replacement, however, is not possible, due to the difference in instruction sizes - `syscall` is two bytes long, whereas `call` is at least five bytes long. Note that although the `call r32` variant of the `call` instruction (wherein the destination address is in the indicated register) is only 2 bytes long, we would still need an additional instruction to load the destination address into the register, which would bring the total to more than five bytes. We observe that any sort of control flow redirection here - be it a jump, call or return oriented programming-inspired return [49] - would exceed the two bytes occupied by `syscall`.

We could, in theory, simply insert a `call` instruction in place of a `syscall` instruction while moving the remaining instructions down by three bytes. This, however, creates correctness concerns that have perpetually plagued binary rewriting techniques [50]. For example, the target addresses of every branch instruction would need to be modified accordingly, which in turn is complicated by the fact that it is difficult to identify which operands in the assembled code might represent addresses [56]. While there exist tools that claim to get very close to solving these problems [50, 56], they are not perfect [61]. We therefore choose to overwrite a minimal number of instructions (just the five bytes required by a jump) and use a code trampoline [26] as shown below.

Under our chosen technique, for a `syscall` instruction to be successfully replaced while

maintaining correctness, we require the following sequence of events to occur when one is encountered during execution:

- The `syscall` is replaced by a relative `jmp` instruction, which jumps to a snippet of assembly code that is unique for each `syscall` instruction in the application binary.
- The registers, at this point, are set up in accordance with the system call conventions dictated by the System V ABI [36]. Since we now want to make a function call, we need to move them so that they conform to the function call conventions. This simply involves moving R10 into RCX, since the remaining arguments are passed in the same registers in both cases [36].
- We can then call the system call implementation function directly. Note here that we make use of the system call identification techniques described in Chapter 5 to completely bypass the system call handler, thereby eliminating a superfluous function call.
- On returning from the system call, the instruction(s) that were overwritten by the `jmp` instruction must be executed.
- The address of the instruction following the last overwritten instruction is pushed onto the stack.
- A `ret` is executed, thereby returning control to the previously pushed address. The reason for using this return oriented programming-inspired approach is that the x86_64 instruction set does not contain an instruction that can perform a direct jump to an immediate value. We could put the destination address into a register and jump directly to it, but that would necessitate the undesirable overhead of saving and restoring that register.
- If the instructions that were displaced by `jmp` do not occupy exactly three bytes (five bytes for `jmp` minus two for `syscall`), the remaining bytes are occupied by `nop` instructions. Note that these `nop` instructions are just used for padding - they are never executed and hence do not add any temporal overhead.

Listing 6.1: Original Executable

```
    404e6e: 4c 63 d3                movslq %ebx,%r10
    404e71: 4d 63 c0                movslq %r8d,%r8
    404e74: 48 63 d5                movslq %ebp,%rdx
    404e77: b8 09 00 00 00          mov    $0x9,%eax
    404e7c: 0f 05                   syscall
    404e7e: 48 83 f8 ff             cmp    $0xffffffffffffffff,%rax
    404e82: 75 16                   jne    404e9a <__mmap+0x9b>
    404e84: 48 85 ff                test   %rdi,%rdi
```

Listing 6.2: Modified Executable

```
    404e6e: 4c 63 d3                movslq %ebx,%r10
    404e71: 4d 63 c0                movslq %r8d,%r8
    404e74: 48 63 d5                movslq %ebp,%rdx
    404e77: b8 09 00 00 00          mov    $0x9,%eax
    404e7c: e9 42 84 e1 ff          jmpq   21d2c3 <exit-0x1e2e2d>
    404e81: 90                      nop
    404e82: 75 16                   jne    404e9a <__mmap+0x9b>
    404e84: 48 85 ff                test   %rdi,%rdi
```

Listing 6.3: System Call Trampoline Code

```
    syscall_404e7c_destination:
    mov     %r10,%rcx
    call    sys_mmap
    cmp     $0xffffffffffffffff,%rax
    push    $0x00404e82
    ret
```

Listing 6.1 shows an example snippet of assembly code (printed using `objdump`) from a user program with a system call that needs to be replaced. Listings 6.2 and 6.3 show the resultant code from within the user program and kernel respectively.

The assembly code shown in Listing 6.3 is automatically generated and compiled in with the HermiTux kernel. After compilation of the kernel, we search for the address of the `syscall_404e7c_destination` symbol within the kernel's ELF file. The destination (relative address) for the `jmp` in Listing 6.2 is then filled in accordingly. Note that this address does not exist within the application binary, which is why the object dump displays a label that does not make any sense.

Note that there are certain situations where we cannot overwrite a system call, such as when the `syscall` instruction is near the end of a function. In such cases, our `jmp` instruction would cross the function boundary within the binary, and the first instruction of the following function would end up at a different address (within the trampoline). While we could, in theory, modify the destination operand of every `call` instruction that calls this function, this gets complicated when function pointers are used, as the destination address is decided at runtime.

Rewriting the binary dynamically at runtime could potentially solve this issue. While this is possible, it would add a significant runtime overhead. We conclude that it is better to pay the runtime cost of executing a *small number* of `syscall` instructions, rather than incur the runtime cost of replacing *all* such instructions. In addition, binary rewriting tools such as Dyninst [15] can run into several megabytes in size, which would add undesirable bloat to the unikernel.

6.3 Evaluation

The evaluation of our system call handling infrastructure consists of a microbenchmark designed to specifically measure system call latency.

The experiments were carried out on a server class machine with an Intel Xeon E5-2695 CPU with 24 cores and 94 GB RAM. The Linux experiments were carried out on Ubuntu 16.04 with both versions 4.9.0 and 4.9.86 of the Linux kernel, the latter being patched to protect against the Meltdown vulnerability.

In order to measure the latency of a system call invocation (including the return path), we implemented a system call in HermiTux, HermitCore, and Linux that does nothing but return zero to the caller. The HermitCore implementation includes a corresponding function in the modified version of Newlib that converts the system call to a function call. For each platform, we invoke the system call 100,000 times in a loop, taking time stamps before and after the loop. Since the system call body itself is minimal, this gives us an accurate representation of the time taken to both enter the system call as well as return from it. The results are shown in Figure 6.1.

Both versions of HermiTux system calls are faster than either version of Linux. HermiTux system calls that make use of the `syscall` instruction are 42.2% faster than the unpatched version of Linux. This speedup is primarily due to the optimizations discussed in Section 6.1.3. HermiTux with overwritten system calls is a further 71.3% faster than this, which is due to the removal of the `syscall` instruction and is the focus of the work in Section 6.2. Combining these two sets of optimizations gives the resultant HermiTux system calls an 83.4% speedup over unpatched Linux.

Another striking feature of this graph is the large latency encountered on newly patched versions of Linux. These patches were necessitated by the recent revelations of the Meltdown vulnerability [35]. We expect that in the long term, the latency of Linux system calls will be closer to that of the unpatched version, both due to new hardware that addresses these vulnerabilities as well as optimizations to the current fixes.

We can see that even with the `syscall` instruction removed, HermiTux is still around 3x slower than a function call on HermitCore. This is because of the extra assembly instructions added to the call path of an overwritten system call, which include branch operations that are

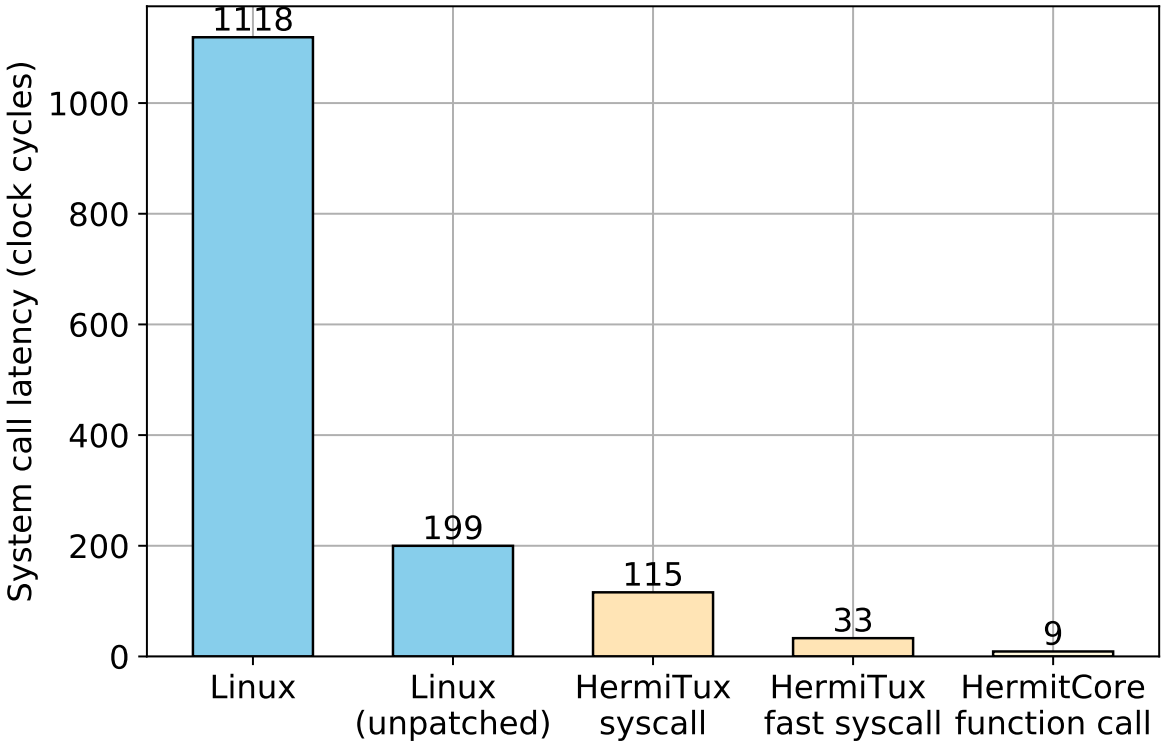

Figure 6.1: System call latency on various platforms, for a system call that just returns an integer.

generally more expensive. Conversely, on HermitCore, each of these function calls involves simply a `call`, `mov` (to put the return value in RAX) and a `ret`.

6.4 Discussion

One drawback of the binary rewriting mechanism described above is that a small number of `syscall` instructions cannot be overwritten. This happens primarily when the `syscall` instruction is towards the end of the function. In such cases, we may not have three extra bytes (required for overwriting with a `jmp`) between `syscall` and the start of the next function. In such cases, while we could modify the target address of every `call` instruction invoking the following function, we choose not to since this would break any function pointers being used.

Similar to Chapter 5, the techniques presented here only apply to unikernels that aim to run unmodified, statically compiled Linux binaries, of which HermiTux is the only one that we are aware of. However, if other unikernel models were to support unmodified binaries, this technique could be applied to them as well, since the binaries being run would still make use of the `syscall` instruction.

Chapter 7

Conclusion and Future Work

Unikernels are lightweight, single purpose virtual machines that contain only the components needed for their target application to run. They present several advantages, such as low memory and disk footprint, fast boot times and improved security through a reduced stack surface.

Despite these benefits, unikernels have not witnessed widespread adoption in industry. In this thesis we identified two issues preventing this from happening. The first is the inability of current hypervisor boot procedures to scale in order to efficiently support the large number of guests that the small size of unikernels allows. The second is the difficulty involved in porting legacy applications to unikernels. In fact, in the case of proprietary software this becomes impossible due to the unavailability of the source code. This thesis presented techniques that aim to solve the two problems mentioned.

The first problem is solved in the context of the Xen hypervisor (Chapter 4). We show that the boot time of guest domains increases exponentially with the number of guests running on the system. We determined that a major bottleneck in the boot process was interactions with the XenStore. Using a caching mechanism within the `xl` toolstack, we ensure that the

number of these interactions remains constant with an increase in the number of guests. Doing so resulted in a performance improvement of 4x when booting a large number (≈ 500) of unikernels at once. This should allow us to reap the benefits of the inherently small boot times of unikernels themselves. It should also allow cloud providers to efficiently pack a large number of unikernels onto a single host machine, something which is not possible with full-fledged guest VMs.

The second issue is solved in the context of HermiTux, which is able to run unmodified executables statically compiled for Linux. This effectively transforms the porting effort of application developers into a supporting effort on the part of the unikernel developers. However, in doing so, HermiTux loses some of the benefits of unikernels - namely modularity and fast system calls. Chapters 5 and 6 demonstrate the techniques we used to help HermiTux retain these benefits.

In Chapter 5 we use binary control flow analysis to statically identify which system calls are made by the target application. We then modularise HermiTux by only including those system call implementations in the final image of the HermiTux kernel. We demonstrate that we can successfully identify all system calls in most of the application binaries that we tested. We also analyse the reduction in the code size obtained when compiling HermiTux for different applications.

Finally, Chapter 6 demonstrated a binary rewriting technique that turns system calls in the target application binary into simple function calls, just as in a regular unikernel. Further, we leverage our ability to identify system calls in order to call the system call implementation directly and bypass the system call handler. According to our microbenchmark, this technique achieves a 71% speedup over the original system call handling mechanism of HermiTux, and approaches the performance of a function call as in vanilla HermitCore.

The techniques presented in this thesis are specific to Xen (Chapter 4) and HermiTux (Chapters 5 and 6). While the boot procedure optimizations for Xen cannot be applied to other hypervisors, the system call identification and replacement techniques for HermiTux could be

applied with minimal (or zero) modifications to other unikernels that might want to provide binary compatibility. In this case, of course, it is assumed that such unikernels would run the unmodified binaries by loading them into their own address space. Other possible techniques for providing binary compatibility, such as decompiling and recompiling the binaries would not be aided by the techniques described here.

It is our hope that the optimizations presented in this thesis will encourage developers and systems administrators to make the switch to unikernels. In addition to the monetary benefits for service providers, this transition could have a more widespread positive impact. All end-users that rely on services deployed on the public cloud would benefit from the increased security that they provide. Due to their lower resource consumption, unikernels could save energy in data centres, which would add up when considering the amount of energy that data centres around the world collectively consume [18], benefiting the environment as well.

7.1 Future Work

There are a few avenues for future work. The most obvious is to increase the number of system calls supported by HermiTux. This would increase the number of applications that could be run unmodified as unikernels. The results presented by Tsai et al. in [54] would be a good guide to decide which ones to use start with. That study analysed which system calls from the Linux API are most commonly used, as well as which system calls must be implemented in any new system so that it can support a maximum number of applications.

Some of the applications that we would like to see supported by HermiTux, such as Apache and Memcached, are multi-process applications. As discussed in Section 2.1.4, the unikernel model advocates that only a single process should be run within a unikernel. This requires that traditional multi-process applications be rewritten to spawn a new unikernel instead of forking a process, which involves significant effort. Further, well-known inter-process communication (IPC) mechanisms would also need to be converted to inter-VM communication. Providing binary compatibility for such applications would greatly reduce the effort required

to port them, and would also vastly increase the number of applications that can be run on HermiTux. Techniques introduced by Graphene [53], which achieved the same with a library operating system, could be adapted for HermiTux.

Given our observations with Xen (Section 4.1), as well as the higher performance of Uhyve compared to Qemu (Section 2.3), we believe that further exploration is required to assess, and possibly improve, the suitability of hypervisors for unikernel environments. Indeed, a lot of work has already focused on these issues [39, 40], and there are even some hypervisors besides Uhyve that are dedicated to running unikernels [59].

While unikernels remove bloat from the OS kernel, running statically compiled programs introduces another source of bloat - standard libraries. Using lightweight C libraries such as musl would help in this regard. Indeed, in Table 5.1, we observe the difference between the number of system calls made by musl as compared to Glibc. This is naturally accompanied by a corresponding difference in the size of the programs. While selecting a smaller library goes a long way, we still notice that some of the system calls in the programs compiled against musl are never used. Quach at al. performed a study [46] that determined that only 65% of code in standard libraries was used by user programs on average. Using binary debloating techniques would hopefully alleviate this issue, and maintain the minimal nature of unikernels.

Bibliography

[1] A Tour of Mini-OS Kernel. URL https://www.cs.uic.edu/~spopuri/minios.html.

[2] Coreutils - GNU Core Utilities. URL https://www.gnu.org/software/coreutils/coreutils.html.

[3] Device Model Stub Domains - Xen. URL https://wiki.xenproject.org/wiki/Device_Model_Stub_Domains.

[4] Erlang on Xen - at the heart of super-elastic clouds. URL http://erlangonxen.org/.

[5] strace(1): trace system calls/signals - Linux man page. URL https://linux.die.net/man/1/strace.

[6] Unikraft. URL https://www.xenproject.org/help/wiki/80-developers/207-unicore.html.

[7] What is IaaS? Infrastructure as a Service — Microsoft Azure. URL https://azure.microsoft.com/en-us/overview/what-is-iaas/.

[8] XenStore Reference - Xen, . URL https://wiki.xen.org/wiki/XenStore_Reference.

[9] XenStore - Xen, . URL https://wiki.xen.org/wiki/XenStore.

[10] Intel® 64 and IA-32 Architectures Software Developers Manual, 2016. ISSN 18650929.

[11] Mohamed Ahmed, Felipe Huici, and Armin Jahanpanah. Enabling dynamic network processing with clickOS. *ACM SIGCOMM Computer Communication Review*, 42(4):

293, 2012. ISSN 01464833. doi: 10.1145/2377677.2377737. URL http://dl.acm.org/citation.cfm?doid=2377677.2377737.

[12] Andrew S. Tanenbaum. The Impact of MINIX0. (July):14–15, 2014.

[13] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03*, page 164, 2003. ISSN 01635980. doi: 10.1145/945445.945462. URL http://portal.acm.org/citation.cfm?doid=945445.945462.

[14] Eli Bendersky. How debuggers work: Part 2 - Breakpoints - Eli Bendersky's website. URL https://eli.thegreenplace.net/2011/01/27/how-debuggers-work-part-2-breakpoints.

[15] Andrew R Bernat and Barton P Miller. Anywhere , Any-Time Binary Instrumentation Categories and Subject Descriptors. pages 9–16.

[16] Sren Bleikertz. How to run Redis natively on Xen. URL https://openfoo.org/blog/redis-native-xen.html.

[17] Bogdan Botezatu. What Is Hypervisor-based Security and Why Is It Important in Stopping Zero-Day Exploits? URL http://www.infosecisland.com/blogview/24966-What-Is-Hypervisor-based-Security-and-Why-Is-It-Important-in-Stopping-Zero-Da
html.

[18] Alfred Bratterud, Alf Andre Walla, Harek Haugerud, Paal E. Engelstad, and Kyrre Begnum. IncludeOS: A minimal, resource efficient unikernel for cloud services. *Proceedings - IEEE 7th International Conference on Cloud Computing Technology and Science, CloudCom 2015*, pages 250–257, 2016. doi: 10.1109/CloudCom.2015.89.

[19] Bryan Cantrill. Unikernels are unfit for production — Joyent. URL https://www.joyent.com/blog/unikernels-are-unfit-for-production.

[20] Vittorio Cozzolino and Aaron Yi Ding. FADES: Fine-Grained Edge Offloading with Unikernels. *Proceedings of HotConNet 17, Los Angeles, CA, USA,*, pages 36–41, 2017. ISSN 16130073. doi: 10.475/123.

[21] Bob Duncan, Andreas Happe, and Alfred Bratterud. Enterprise IoT security and scalability. *Proceedings of the 9th International Conference on Utility and Cloud Computing - UCC '16*, pages 292–297, 2016. ISSN 16130073. doi: 10.1145/2996890.3007875. URL http://dl.acm.org/citation.cfm?doid=2996890.3007875.

[22] Dawson R Engler, M Frans Kaashoek, James O 'toole, , and James O Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. *ACM SIGOPS Operating Systems Review*, 1(212):251–266, 1995. ISSN 01635980. doi: 10.1145/224057.224076. URL http://portal.acm.org/citation.cfm?id=224076.

[23] Gernot Heiser and Kevin Elphinstone. L4 Microkernels : The Lessons from 20 Years of Research and Deployment. *ACM Transactions on Computer Systems*, 34(1):1–29, 2016. ISSN 07342071. doi: 10.1145/2893177. URL https://www.nicta.com.au/publications/research-publications/?pid=8988.

[24] Zach Hill, Jie Li, Ming Mao, Arkaitz Ruiz-Alvarez, and Marty Humphrey. Early observations on the performance of Windows Azure. *Scientific Programming*, 19(2-3): 121–132, 2011. ISSN 10589244. doi: 10.3233/SPR-2011-0323.

[25] Jeffrey K Hollingsworth and Bryan Buck. an Api for Runtime Code Patching. *International Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[26] Jeffrey K. Hollingsworth, Barton Miller, and Jon Cargille. Dynamic program instrumentation for scalable performance tools. *Proceedings of IEEE Scalable High Performance Computing Conference*, 1994(May):841–850, 1994. doi: 10.1109/SHPCC.1994.296728. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=296728%5Cnftp://ftp.cs.wisc.edu/pub/techreports/1994/TR1207.pdf.

[27] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of

computer programs. *Computer Journal*, 23(3):223–229, 1980. ISSN 00104620. doi: 10.1093/comjnl/23.3.223.

[28] Avi Kivity, Dor Laor, Glauber Costa, and Pekka Enberg. OSvOptimizing the Operating System for Virtual Machines. *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 61–72, 2014. URL https://www.usenix.org/system/files/conference/atc14/atc14-paper-kivity.pdf.

[29] Gerwin Klein, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, and Rafal Kolanski. seL4. *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*, page 207, 2009. doi: 10.1145/1629575.1629596. URL http://portal.acm.org/citation.cfm?doid=1629575.1629596.

[30] Thomas Knauth and Christof Fetzer. DreamServer. *Proceedings of International Conference on Systems and Storage - SYSTOR 2014*, pages 1–11, 2014. doi: 10.1145/2611354.2611362. URL http://dl.acm.org/citation.cfm?doid=2611354.2611362.

[31] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. Unikernels Everywhere : The Case for Elastic CDNs. *Vee*, 2017. ISSN 0362-1340. doi: 10.1145/3050748.3050757.

[32] H Andrs Lagar-Cavilla, Joseph A Whitney, Adin Scannell, Philip Patchin, Stephen M Rumble, Eyal De Lara, Michael Brudno, and M Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12, 2009. ISSN 07342071. doi: 10.1145/1925109.1925111.

[33] Stefan Lankes, Simon Pickartz, and Jens Breitbart. HermitCoreA Unikernel for Extreme Scale Computing. *6th International Workshop on Runtime and Operating Systems for Supercomputers*, pages 1–4, 2016. doi: 10.1145/2931088.2931093. URL http://doi.acm.org/10.1145/2931088.2931093.

[34] David Lie and Lionel Litty. Using Hypervisors to Secure Commodity Operating Systems. *Network*, pages 11–19, 2010. ISSN 15437221. doi: 10.1145/1867635.1867639.

[35] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, 2018. URL https://arxiv.org/abs/1801.01207.

[36] H J Lu, David L Kreitzer, Milind Girkar, and Zia Ansari. System V Application Binary Interface. pages 1–141, 2013. URL papers3://publication/uuid/E47921A1-4555-47A5-8590-2EF7C8D132DD.

[37] Anil Madhavapeddy and David J. Scott. Unikernels: The Rise of the Virtual Library Operating System. *Communications of the ACM*, 57(1):61–69, 2014. ISSN 00010782. doi: 10.1145/2541883.2541895. URL http://doi.acm.org/10.1145/2541883.2541895%5Cnhttp://dl.acm.org/citation.cfm?doid=2541883.2541895.

[38] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '13*, 48(4):461, 2013. ISSN 03621340. doi: 10.1145/2451116.2451167. URL http://dl.acm.org/citation.cfm?doid=2451116.2451167.

[39] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu: Just-in-time summoning of unikernels. *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 559–573, 2015. URL https://www.usenix.org/system/files/conference/nsdi15/nsdi15-paper-madhavapeddy.pdf.

[40] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles - SOSP '17*, 2017. ISBN 9781450350853. doi: 10.1145/3132747.3132763.

[41] Ming Mao and Marty Humphrey. A performance study on the VM startup time in the cloud. *Proceedings - 2012 IEEE 5th International Conference on Cloud Computing, CLOUD 2012*, pages 423–430, 2012. ISSN 2159-6182. doi: 10.1109/CLOUD.2012.103.

[42] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: A performance comparison. *Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, pages 386–393, 2015. doi: 10.1109/IC2E.2015.74.

[43] Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Barbalace, Daniel Hagimont, and Binoy Ravindran. Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor. 2017. doi: 10.1145/nnnnnnn.nnnnnnn. URL http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn.

[44] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library OS from the top down. *ACM SIGPLAN Notices*, 47(4): 291, 2012. ISSN 03621340. doi: 10.1145/2248487.1950399. URL http://dl.acm.org/citation.cfm?doid=2248487.1950399.

[45] Amit Purohit, Joseph Spadavecchia, Charles Wright, and Erez Zadok. Improving Application Performance Through System Call Composition. URL http://www.fsl.cs.sunysb.edu/docs/cosy-perf/.

[46] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. A Multi-OS Cross-Layer Study of Bloating in User Programs , Kernel and Managed Execution Environments. pages 65–70, 2017. doi: 10.1145/3141235.3141242.

[47] M Rajagopalan, S K Debray, M Hiltunen, and R Schlichting. Cassyopia: Compiler Assisted System Optimization. *HotOS'03*, pages 1–5, 2003.

[48] Mohan Rajagopalan, Saumya K Debray, Matti A Hiltunen, and Richard D Schlichting. System Call Clustering :{A} Profile-Directed Optimization Technique. Technical report, University of Arizona, 2002. URL https://www2.cs.arizona.edu/solar/papers/multi-call.pdf.

[49] R Roemer and E Buchanan. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security*, 15(1):1–36, 2012. ISSN 10949224. doi: 10.1145/2133375.2133377. URL http://cseweb.ucsd.edu/~hovav/papers/rbss09.html%5Cnhttp://dl.acm.org/citation.cfm?id=2133377.

[50] Matthew Smithson, Khaled Elwazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. *Proceedings - Working Conference on Reverse Engineering, WCRE*, pages 52–61, 2013. ISSN 10951350. doi: 10.1109/WCRE.2013.6671280.

[51] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. *Osdi'10*, pages 33–46, 2010. URL http://www.usenix.org/events/osdi10/tech/full_papers/Soares.pdf.

[52] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. ISBN 9780133591620.

[53] Chia-Che Tsai, Donald E. Porter, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, and Daniela Oliveira. Cooperation and security isolation of library OSes for multi-process applications. *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*, pages 1–14, 2014. doi: 10.1145/2592798.2592812. URL http://dl.acm.org/citation.cfm?doid=2592798.2592812.

[54] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern Linux API usage and compatibility. *Proceedings of the Eleventh European Conference on Computer Systems - EuroSys '16*, pages 1–16, 2016. doi: 10.1145/2901318.2901341. URL http://dl.acm.org/citation.cfm?doid=2901318.2901341.

[55] Carl A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181, 2002. ISSN 01635980. doi: 10.1145/844128.844146. URL http://portal.acm.org/citation.cfm?doid=844128.844146.

[56] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. Ramblr: Making Reassembly Great Again. *Proceedings 2017 Network and Distributed System Security Symposium*, (March), 2017. doi: 10.14722/ndss.2017.23225. URL https://www.internetsociety.org/sites/default/files/ndss2017_10-5_Wang_paper_0.pdf.

[57] Joe Weinman. Time is Money: The Value of " On-Demand " Time is Money: The Value of On-Demand. 2011. URL http://www.joeweinman.com/Resources/Joe_Weinman_Time_Is_Money.pdf.

[58] Adam Wick. The HaLVM: A Simple Platform for Simple Platforms. URL https://www.slideshare.net/xen_com_mgr/the-halvm-a-simple-platform-for-simple-platforms.

[59] Dan Williams and Ricardo Koller. Unikernel Monitors: Extending Minimalism Outside of the Box. *8th USENIX Workshop on Hot Topics in Cloud Computing*, (Vm), 2016. URL https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams.

[60] Bruno Xavier, Tiago Ferreto, and Luis Jersak. Time Provisioning Evaluation of KVM, Docker and Unikernels in a Cloud Platform. In *Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2016*, 2016. ISBN 9781509024520. doi: 10.1109/CCGrid.2016.86.

[61] Kyungjin Yoo and Rajeev Barua. Recovery of object oriented features from C++ Binaries. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 1:231–238, 2014. ISSN 15301362. doi: 10.1109/APSEC.2014.44.

[62] Jun Zhu, Zhefu Jiang, and Zhen Xiao. Twinkle: A fast resource provisioning mechanism for internet services. *Proceedings - IEEE INFOCOM*, pages 802–810, 2011. ISSN 0743166X. doi: 10.1109/INFCOM.2011.5935302.