SERVICE EXTRUSION IN GENERAL PURPOSE KERNEL

A Thesis

by

YIFAN LIU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Riccardo Bettati
Co-Chair of Committee,	Dilma Da Silva
Committee Members,	Paul V. Gratz
Head of Department,	Scott Schaefer

December 2020

Major Subject: Computer Science

Copyright 2020 Yifan Liu

ABSTRACT

General-purpose kernels sometimes fail to provide specialized services that may be required by applications, for example real-time capabilities, low-latency communication, or specialized device access capabilities. Addressing this with the use of specialized kernels loses the generality that one has come to expect from general purpose kernel. This thesis proposes a solution that combines general-purpose and specialized kernels to enable services from both sides while providing performance isolation, in a fashion that is transparent to both programmer and user. The application is provided direct hardware access, and at the same time still has access to Linux system calls without crossing the privilege layer.

DEDICATION

To my parents, Rongjun Liu and Xinwei Zhu, who raised me and taught me to speak.

To all my families, who gave me encouragement and support.

And my deepest gratitude to my advisor, Prof. Riccardo Bettati.

These two years are one of the best two years I've ever had.

ACKNOWLEDGMENTS

I would like to thank Prof. Riccardo Bettati for his advise and review, and my committee members, Prof Dilma Da Silva and Prof Paul V. Gratz, for their comments and feedback.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis (or) dissertation committee consisting of Professor Riccardo Bettati and Professor Dilma Da Silva of the Department of Computer Science and Engineering and Professor Paul V. Gratz of the Department of Electrical Computer Engineering.

All work conducted for the thesis (or) dissertation was completed by the student independently.

Funding Sources

No outside funding was received for the research and writing of this document.

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGMENTS iv

CONTRIBUTORS AND FUNDING SOURCES v

TABLE OF CONTENTS vi

LIST OF FIGURES viii

LIST OF TABLES ix

1. INTRODUCTION 1

2. BACKGROUND 4

2.1 Hardware-Assisted Virtualization 4
2.1.1 x86 CPU Virtualization 4
2.1.2 x86 MMU Virtualization 4
2.1.3 x86 I/O Virtualization 5
2.2 Inter-VM Shared Memory 6
2.3 Xen 6
2.3.1 Grant Table Shared Memory Mechanism 6
2.3.2 Xen Event Channel 7
2.3.3 XenStore 7

3. RELATED WORK 9

3.1 Virtual Interface Architecture (VIA) 9
3.2 TCP Offload 9
3.3 RT Linux and RTAI 9
3.4 DPDK 10
3.5 Arrakis 10
3.6 IX 10
3.7 Cross Call 11

4. SERVICE EXTRUSION 12

4.1 Service Extrusion Overview ... 12
4.2 Design Considerations ... 12
4.2.1 The Hardware ... 13
4.2.2 The Hardware Abstraction Layer ... 13
4.2.3 The Extruded Service and The Unified API ... 14
4.2.4 The Application ... 15

5. REMOTE SYSTEM CALL FRAMEWORK ... 16

5.1 Overview ... 16
5.2 System Architecture ... 16
5.2.1 OSv modification ... 18
5.2.2 Remote System Call API and Library ... 18
5.2.3 Remote System Call Split Driver ... 19
5.2.4 Notification Channel ... 19
5.3 Example: Read/Write a File From Linux ... 20
5.3.1 User Application ... 20
5.3.2 Remote System Call Setup: Config Negotiation ... 21
5.3.3 Remote System Call: Redirection ... 21
5.3.4 Remote System Call: Invocation and Return ... 21
5.4 Challenges and Issues ... 22
5.4.1 Page Fault Handling ... 22
5.4.2 Address Space Collision ... 23
5.4.3 Unsupported System Calls ... 24

6. EXPERIMENTATION AND EVALUATION ... 25

6.1 Experiment Settings ... 25
6.2 Redis NoSQL Store ... 25
6.2.1 Redis Benchmark ... 26
6.3 OS Jitter ... 29
6.3.1 Settings and Results ... 29
6.4 Remote System Call Performance ... 31
6.4.1 NULL System Call Performance ... 31
6.4.2 Read/Write System Call Performance ... 32

7. CONCLUSION AND FUTURE WORK ... 35

REFERENCES ... 36

LIST OF FIGURES

FIGURE Page

4.1 Service Extrusion Architecture 13

5.1 Remote System Call Architecture 17

6.1 Redis performance, 600 concurrent clients, no pipelining 28

6.2 Redis PING_INLINE performance 28

6.3 OS Jitter 31

6.4 Read system call latency 33

6.5 Write system call latency 34

LIST OF TABLES

TABLE Page

6.1 Redis Benchmark Operations 27

6.2 Samples Summary of 2/4 and 4/4 benchmark 30

6.3 OS Jitter Samples Summary 30

6.4 NULL System Call Performance 32

1. INTRODUCTION

With the development of fast I/O devices, the role of general-purpose operating systems in many scenarios has been gradually reduced. Linux, being the most popular general-purpose kernel, sometimes fails to provide predictable, high performance, or customizable services to specialized applications. For example, the authors of [1] analyzed the cost for processing a single network packet, and found that nearly 70% of the time was spent in the kernel network stack.

The cost of general-purpose kernels leads to many attempts to bypass the kernel, such as Intel's Data Plane Development Kit (DPDK) [2] for high performance networking. Many of these attempts give user space applications direct access to unmediated hardware resources, and thus bypass the kernel, in exchange for better performance. Most of these attempts focus on network devices only.

On the other hand, specialized kernels, such as unikernels [3], where an application is linked with a library OS and deployed directly on a virtualized or physical machine, have also gained interest, especially in cloud computing scenarios. Recent unikernels implement single-process applications and single address spaces. This eliminates overheads, such as context switching, and enables whole-system optimization. Unikernels have demonstrated significant advantages over general-purpose kernels (e.g. Linux), including boot time [4], I/O performance [3] and security [5].

Despite these advantages, the unikernel's single-application architecture has a number of shortcomings: First, debugging and management of the unikernel are complicated because of a lack of the ability to run concurrent processes. Many of the tools that are available on a general-purpose kernel, such as shell, debugger, or profiler will not be available on a unikernel. Porting supporting software, such as device drivers, onto a unikernel can be a daunting task, because unikernels usually have different scheduling strategies, memory protection models, or even filesystem APIs from a general-purpose kernel. Next, even performance or latency sensitive applications occasionally need access to a rich set of general-purpose services, such as logging, visualization, etc. Finally,

programmers appreciate to be able to develop and run applications in a familiar environment such as POSIX.

So the question is: Can we have access both to high-performance specialized services, and to general-purpose services and mediated device access such as provided by general-purpose operating systems? More specifically, can we derive a solution that satisfies the following design goals?

- **Hybrid services:** The solution is able to provide access to both a specialized kernel, such as high performance networking, and a general-purpose kernel such as multiple filesystem support, device drivers and IPC with other processes on the same kernel.

- **Performance isolation:** Accessing the general-purpose services should not affect the performance of specialized services. Similarly, accessing the general-purpose services from the specialized kernel should also not affect the applications already running on the general-purpose kernel.

- **Zero privilege boundary crossing:** One straightforward method to access general-purpose services from a specialized kernel is to issue remote invocations of system calls to a general-purpose kernel across kernel boundaries, typically using a network protocol. The cost of such remote invocations is significant because they cross multiple privilege boundaries (e.g., user to kernel, VM to a hypervisor). Rather than sending requests through a network stack, access to either kind of services should be done in the form of function/system calls to reduce call overhead. The calls could be executed locally or, when necessary, offloaded for remote execution, but the offloading process of such calls should not cross privilege boundaries.

- **Transparency:** Any solution should provide the user and application running on specialized kernel with a familiar system call interface. In addition, porting existing (POSIX) applications to the specialized kernel should require no or only minimal code changes.

This thesis will propose and evaluate an architecture to achieve these goals. It will describe one

implementation, which makes use of the Xen hypervisor's Grant Table shared memory mechanism [6]. Our experiments show that our architecture meets all the criteria described above.

2. BACKGROUND

In this section we present background knowledge related to the topic.

2.1 Hardware-Assisted Virtualization

Hardware-Assisted Virtualization is an efficient virtualization technique that uses help from underlying hardware capability. Generally, a hypervisor (or Virtual Machine Monitor) has to provide virtualization for the following components: CPU, MMU and I/O devices. On the x86 architecture, the presence of hardware virtualization extension (VT-x) eliminates the need for binary translation and at the same time maintains high performance. In the following sections we summarize the virtualization approach for the three parts we have mentioned above.

2.1.1 x86 CPU Virtualization

On x86 processors with VT-x support, the processor is able to run in two different modes: *VMX root mode* and *VMX non-root mode*. These two modes are orthogonal to privilege rings (i.e., each mode has its own separate privilege rings). The hypervisor software runs in the VMX root mode and retains control of the physical machine. The VMX non-root mode is privilege-reduced to enable the hosting of a guest OS. The transition from root to non-root mode can be achieved via new VMX-related instructions, such as `VMENTER`, `VMEXIT`. Typically, the CPU runs in non-root mode until an external event (e.g., timer, I/O interrupt) happens, at which point it traps to the hypervisor in root mode. After the handling of an event by the hypervisor, the CPU resumes execution in non-root mode.

2.1.2 x86 MMU Virtualization

Before the advent of architectural support for MMU virtualization, hypervisor software employed a technique called *shadow paging* to keep the mapping of virtual address to host-physical address. This technique is extremely expensive and complex since it relies on memory tracing to keep track of the changes of page table in memory.

With the introduction of hardware-assisted virtualization, such as the EPT (*Extended Page Table*) mechanism, the virtualization of MMU has become much simpler. The EPT is similar structure to a normal page table, except that it translates directly from guest physical address (GPA) to host physical address (HPA). When in VMX non-root mode, all accesses to the guest physical memory will be treated as "guest access", and will be further translated through the EPT by hardware. In this way every guest can have its own VM-level address space, and shared memory between VMs becomes easier to maintain.

2.1.3 x86 I/O Virtualization

I/O virtualization has been one of the most complex parts of a hypervisor. It is usually addressed using one of the two approaches:

The first is *I/O emulation*, where hypervisors provide software emulations of the hardware to the guest OS. This approach is still employed by many modern hypervisors, such as Xen or KVM [7], to virtualize components such as BIOS, VGA adapters, or USB controllers.

The second approach is *I/O para-virtualization*. This is typically used to virtualize devices that have high performance requirements, such as disk or network I/O driver. Such para-virtualized drivers are usually separated into two parts, namely *frontend* and *backend*. The frontend serves as an interface to the virtual machine, while the backend is the actual implementation, usually running in the hosting OS (or privileged VM in bare-metal, or Type-I, hypervisor).

In addition, if a device is DMA-capable, the DMA requests would have to be emulated or para-virtualized because by using DMA, the device is able to access any part of the physical memory, if unrestricted. With the introduction of hardware support, the isolation of devices can be achieved more efficiently by using hardware assisted remapping [8], where hypervisor can create multiple DMA protection domains, and each of the protection domain contains only a subset of the physical memory. This protection model also enables a third way for VMs to interact with devices: direct device assignment. The hypervisor is able to exclusively assign certain devices to specific VM, in a secure and scalable way, for maximum performance and availability.

2.2 Inter-VM Shared Memory

Modern hypervisors will usually provide inter-VM shared memory mechanism, such as Grant Table mechanism [6] on Xen, or inter-VM shared device on QEMU/KVM. In most cases, such mechanism will be employed in one of the following cases:

1. Communication channel for split drivers;

2. Bulk data transfer;

3. Memory footprint optimization. Some hypervisors may merge read-only pages with exact same content (e.g., different VMs may have same kernel code page) to reduce overall memory footprint.

In this thesis, we also employed shared memory but to a different use case scenario: Inter-VM page fault. We will present more details in Section 5.4.1. For that purpose, we decided to use Xen Grant Table mechanism, where the hypervisor provides low-level operations for shared memory manipulation through hypercalls.

2.3 Xen

Xen [9] is an open-source Type-I (or bare-metal) hypervisor originally developed by University of Cambridge and now maintained by Linux foundation. To achieve better VM isolation and security, Xen hypervisor itself does not contain drivers for I/O devices. Instead, a privileged VM (Domain 0) is responsible for providing I/O services, such as network and disk I/O services, to all other VMs. As described in Section 2.1.3, such para-virtualized drivers usually contains a frontend running in guest OS and a backend running in privileged VM. The communication of the front and backend is achieved through Grant Table shared memory and an inter-VM event channel.

2.3.1 Grant Table Shared Memory Mechanism

Xen provides Grant Table mechanism [6] to enable memory sharing and transferring across VMs. Grant Table is a data structure shared between guest OS and hypervisor to keep track of

shared memory. For memory sharing, a VM is able to grant access of certain memory frames to other VM while retaining ownership. A typical process of memory sharing includes:

1. VM1 creates a grant access reference, and transmits the grant ID to VM2.

2. VM2 uses the grant ID to map the granted frame.

3. VM2 accesses the frame.

4. VM2 unmaps the frame.

5. VM1 removes its grant from its grant table.

In addition to sharing, Grant Table also allows a VM to transfer the ownership of certain frames in similar process. The sharing mechanism is widely used in the front-backend split drivers, while the transferring is typically used for bulk data transfer.

2.3.2 Xen Event Channel

Xen Event Channel [10] is the basic primitive for event notification. An event is equivalent to an interrupt. Currently there are four types of event supported by Xen:

1. Inter-domain notifications. This include the events from para-virtualized devices.

2. VIRQs

3. Inter-Processor Interrupts

4. PIRQs (Hardware interrupts)

The notification between split drivers is built on top of this mechanism.

2.3.3 XenStore

XenStore [11] is an information storage maintained by Domain 0 and shared across all VMs. Structurally, it is similar to a `procfs`, where VMs are able to read/write data under its own namespace. XenStore is built on top of very low-level primitives (virtual interrupts and shared

memories) to provide higher-level operations such as read/write a key, enumerate directory or notify upon value change.

It is typically used for configuration and small-sized information exchange for split drivers. It is not meant to be used as persistent datastore or message queue.

In this thesis, we make use of the above features of Xen to develop a solution that satisfy the design goal described in Section 1.

3. RELATED WORK

Over the years, a number of kernel bypass techniques have been proposed, some with the intent to primarily reduce device access latencies, others to increase timing predictability. In this section we present a survey a number of kernel bypass techniques that have been proposed over the years.

3.1 Virtual Interface Architecture (VIA)

Virtual Interface Architecture [12] is a model for user-space zero-copy network. The idea was inspired by Virtual Memory model, where each user process is provided an illusion of private memory with the assistance of hardware such as the MMU. In VIA, each consumer process, called a *VI Consumer*, is provided direct access to an interface to the network hardware, called *Virtual Interface*. In this case, the consumer process will have an illusion that it was provided a "private network", and therefore is able to manage the transmission buffer of its own, while the Virtual Interface provides protection.

3.2 TCP Offload

TCP offload is a technique implemented in the network interface card (NIC) to offload the entire TCP/IP stack from the host CPU for faster packet processing. However, this idea didn't last long[13] because of a lack of scalability and maintainability. Hardwiring the network stack makes it impossible to fix bugs in network stack itself, and the scalability is limited by the capability of the NIC (e.g., memory limit and IP routing tables).

3.3 RT Linux and RTAI

RT Linux [14] is an attempt to make Linux real-time capable. In the RT Linux implementation, Linux runs as a lowest-priority thread, in an emulation layer, on top of a small RTOS. The real-time tasks run alongside Linux in RTOS. The RT-Linux architecture is able to provide low-latency interrupt services, customizable scheduling, high timer precision, and at the same time retain access to a range of general-purpose OS services from Linux.

Similarly, RTAI [15] (Real Time Application Interface) also runs patched Linux as a lowest-priority task on top of a hardware abstraction layer, where Linux is not able to block interrupts or prevent itself from being preempted.

However, one of the disadvantages of these approaches is that in order for application to run on the RT-Linux platform, they must explicitly separate the hard real-time part and non real-time part.

3.4 DPDK

The Intel Data Plane Development Kit (DPDK) [2] is a typical example of kernel-bypass networking. The kernel network stack is completely bypassed, and NICs need to be unbound from the kernel driver before they can be used directly from user space. DPDK implements a run-to-completion model for fast data plane operations, and the devices are accessed via polling to eliminate the overhead of interrupt handling.

3.5 Arrakis

Arrakis [1] is a network server OS that acts as a control plane. Applications are provided direct access to virtualized I/O devices. Data plane operations therefore do not require kernel mediation. Arrakis relies on SR-IOV (Single Root, Input/Output Virtualization) for multiplexing. A single SR-IOV capable NIC can present itself as multiple virtualized PCIe devices. Arrakis can achieve two to five times faster R/W latency and close to 9 times write throughput compared to a well-tuned Linux.

3.6 IX

IX [16] is a data plane OS that provides high performance user-level networking. IX and Arrakis share a number of similarities: Both separate control and data plane for better performance; both utilize hardware virtualization technology to provide unmediated hardware access to user application; both provide a POSIX compatible API; and both achieve a significant performance gain compared to Linux. The major differences between the two are: IX relies on run-to-completion adaptive batching, while Arrakis does not; IX does not support SR-IOV and IOMMU as Arrakis

does; and Arrakis is a fork of BarrelFish [17], while IX is based on Dune [18]/Linux. The system calls in the IX data plane are replaced by `VMCALL`, which requires hypervisor intervention and rerouting.

3.7 Cross Call

Cross Over [19] is a solution for zero ring-crossing cross-VM system calls (i.e., the system call is issued by user application in VM1, and served by the kernel in VM2, without trapping into the hypervisor). The idea is to switch Extended Page Table mapping of the caller machine to directly execute in the VM address space of the callee machine. On the Intel platform, the switching operation can be achieved via the `VMFUNC` instruction without trapping into the hypervisor. To achieve a system call, a universally mapped code page and shared memory is utilized to pass the arguments and return value. This is an appealing idea to our solution, but there is still one fatal problem: The cross call switches the entire Guest Physical Address to Host Physical Address mapping to that of the other machine. This will freeze any other application on the caller machine for the entire duration of the cross call. While the crossing is happening, the CPU cores of both the caller and callee may be executing same piece of code (in particular Linux system call code), and that code may not be lock-protected when Linux detects there should have been only one "available" core.

4. SERVICE EXTRUSION

From the discussion in Section 1 and Section 3, it is evident that the various approaches for control-data plane separation do not satisfy the goals laid out for our work. General purpose services are not limited to control plane services, e.g. access control and rate limiting for network transmission. Rather, they can be any services, usually unrelated and not in critical path. This section provides the architecture of service extrusion and one possible implementation of such architecture.

4.1 Service Extrusion Overview

As an alternative to layering services (typically middleware services) on top of a general-purpose kernel, we propose a compartmentalized deployment of services over a minimal mediating hardware abstraction layer (HAL): Critical services are deployed directly on top of the mediating layer, alongside the general-purpose kernel. Services that are offered beyond the critical part can still be provided by the general-purpose kernel. Figure 4.1 shows an overview of the service extrusion architecture.

The objective of this architecture is to run performance- or safety-critical services separately from the general-purpose kernel and run them directly on the mediating layer, in a fashion that is transparent to the user and the kernel.

4.2 Design Considerations

Since we provide unified API to user applications without crossing the privilege boundary, there will be many choices and challenges in designing a possible implementation, for nearly every component shown in Figure 4.1. The subsequent sections detail the challenges and choices we face.

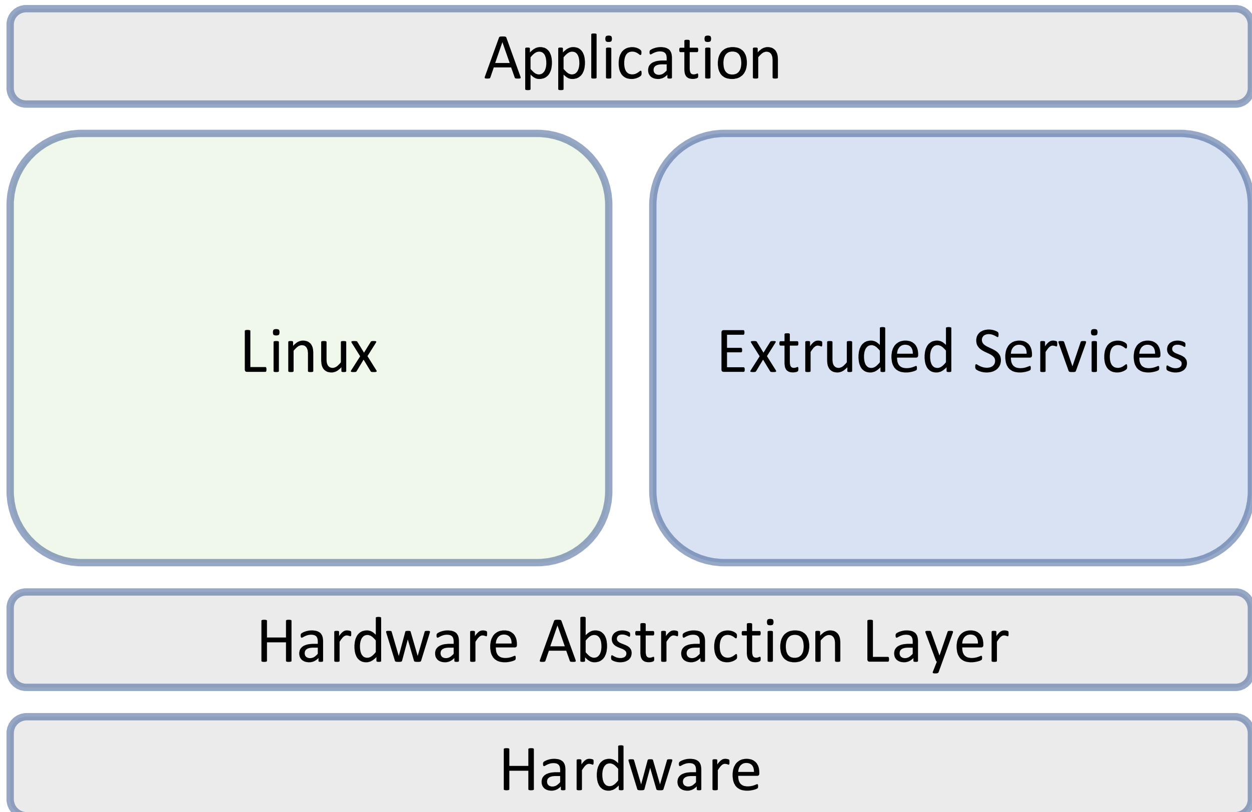

Figure 4.1: Service Extrusion Architecture

4.2.1 The Hardware

The Service Extrusion architecture, by design, should work on any use case scenario including embedded devices, mobile phones, desktop computers or data-center servers. However, one crucial capability we require is the presence of hardware assisted virtualization. As described in section 2, most modern processors, such as x86 or ARM processors, were equipped with hardware assisted virtualization capability [20] [21].

For network interface card, we do not rely on SR-IOV for packet multiplexing.

4.2.2 The Hardware Abstraction Layer

By design, the HAL has to be as lightweight as possible to ensure reasonable performance of the extruded services, and at the same time has the ability to host out-of-the-box Linux system.

Currently there are two approaches to host Linux: hosting Linux based on virtualization technology (either hardware virtualization or para-virtualization), and the HAL runs as hypervisor; or hosting Linux by adding emulation layer without the use of hardware virtualization. The first approach was employed by most of the Type-I hypervisors (or software that contains Virtual Machine Monitor module), such as Xen, ACRN [22], L4Linux [23] or SeL4 [24]. The second approach was adopted primarily by RT-Linux [14] and RTAI [15]. In our case, we have picked Xen for its excellent performance [25], ease of use and built-in support for inter-VM shared memory.

4.2.3 The Extruded Service and The Unified API

As described by design goal, the service extrusion does not affect the performance of Linux. That is, although "extruded", the service can still be accessed by applications running on Linux. The extruded service should be self-contained, high performance and compatible with most Linux applications without modification. In general, unikernel is a good candidate for this task.

However there are many types of unikernels, and not all of them fit in the role because of different designs. Many unikernels, such as MirageOS [3] or HaLVM [26], requires the application itself to be written in certain programming languages (OCaml in MirageOS, Haskell in HaLVM) and compiled with specific toolchain. Some other unikernels, such as Drawbridge [27], are designed to achieve benefits of a virtual machine without having to run on a hypervisor or bare metal.

In addition, in order to provide unified API with minimum modification to the application, the extruded services should support POSIX. To our knowledge, only two unikernels, Rumprun [28] and OSv, satisfy all the criteria. We picked OSv as it is more stable, and supports running unmodified binary directly without recompilation. Note that although OSv claims to be binary compatible [29] with Linux, only a subset (less than 100) of the full Linux system calls were implemented. And due to lack of drivers, running OSv directly on top of bare metal will not be able to fully replace the Linux environment.

4.2.4 The Application

As depicted in Figure 4.1, the application was provided unified API to access the services of both kernels. Essentially, this means we ask the application to have two sets of system call interfaces.

Many challenges arises in designing such an architecture. First, where do we run the application? In general we have two choices: running on Linux or running on unikernel. Since we need to provide maximum performance out of unikernel, we chose to run the application on unikernel, and access Linux services from unikernel.

Second, does application need to be divided into multiple parts? For example, is it possible to separate the application into critical and non-critical part, where each part runs on a different type of kernel while maintaining transparency? This approach seems working in some scenarios, but it has the following disadvantages:

1. The code segment compiled and linked for Linux environment may not properly work (e.g., different library version or different system call semantics) under a different execution context;

2. It may disrupt the execution flow of the original program, where the critical "thread" and non-critical "thread" were originally in the same thread. In this case, the handling of external events such as interrupts becomes complex.

3. The critical and non-critical parts may not be static. Non-critical code path can become critical in some cases, while critical code path may become less critical after, e.g., configuration change. Dynamically adjusting such separation will be very hard under this circumstance.

In the following sections, we detail a possible implementation of the architecture, Remote System Call Framework, to address the design goals laid out earlier.

5. REMOTE SYSTEM CALL FRAMEWORK

5.1 Overview

In this section, we present an implementation of the Service Extrusion, namely Remote System Call framework. We make use of hardware virtualization to provide user-level access to the hardware resources. The system is built on top of the Xen [9] hypervisor and its Grant Table shared-memory mechanism on x86_64 platform. Grant Table mechanism allows a domain to explicitly share or transfer its memory pages to other domain. The Extruded Services component is implemented based on our fork of the OSv unikernel [29]. OSv is a POSIX-compatible, single-address-space, single-application unikernel written in C++. A user application is compiled and linked against the OSv, and most of the specialized services (e.g., high-performance I/O) are provided by the OSv kernel. In order to support Remote System Call, we have partially ported the Xen shared memory driver (`gntalloc`) onto the OSv to enable the use of Xen shared memory. Figure 2 shows an overview of the implementation.

5.2 System Architecture

In addition to the cost of privilege layer crossing, the problem of mediated or re-routed system calls is that any pointer arguments (buffers) of system calls cannot be interpreted as they are since they are of no use outside of the address space of the caller. Either a marshalling/unmarshalling process is required as in RPC, or multiple copying needs to be done. This may result in a complication of various security checks such as a boundary check. More complications can arise when calling memory-related calls such as `mmap`. To address this problem, we have developed the Remote System Call Framework to support zero-copy cross-VM system calls. More specifically, our approach does NOT require the user space application to:

1. Allocate memory in specific region. The memory can be in any place, as long as it is within the limitation of Linux user space upper limit (i.e., 128TB).

2. Use a patched `libc`. Users can use the standard `libc` without worrying about incompati-

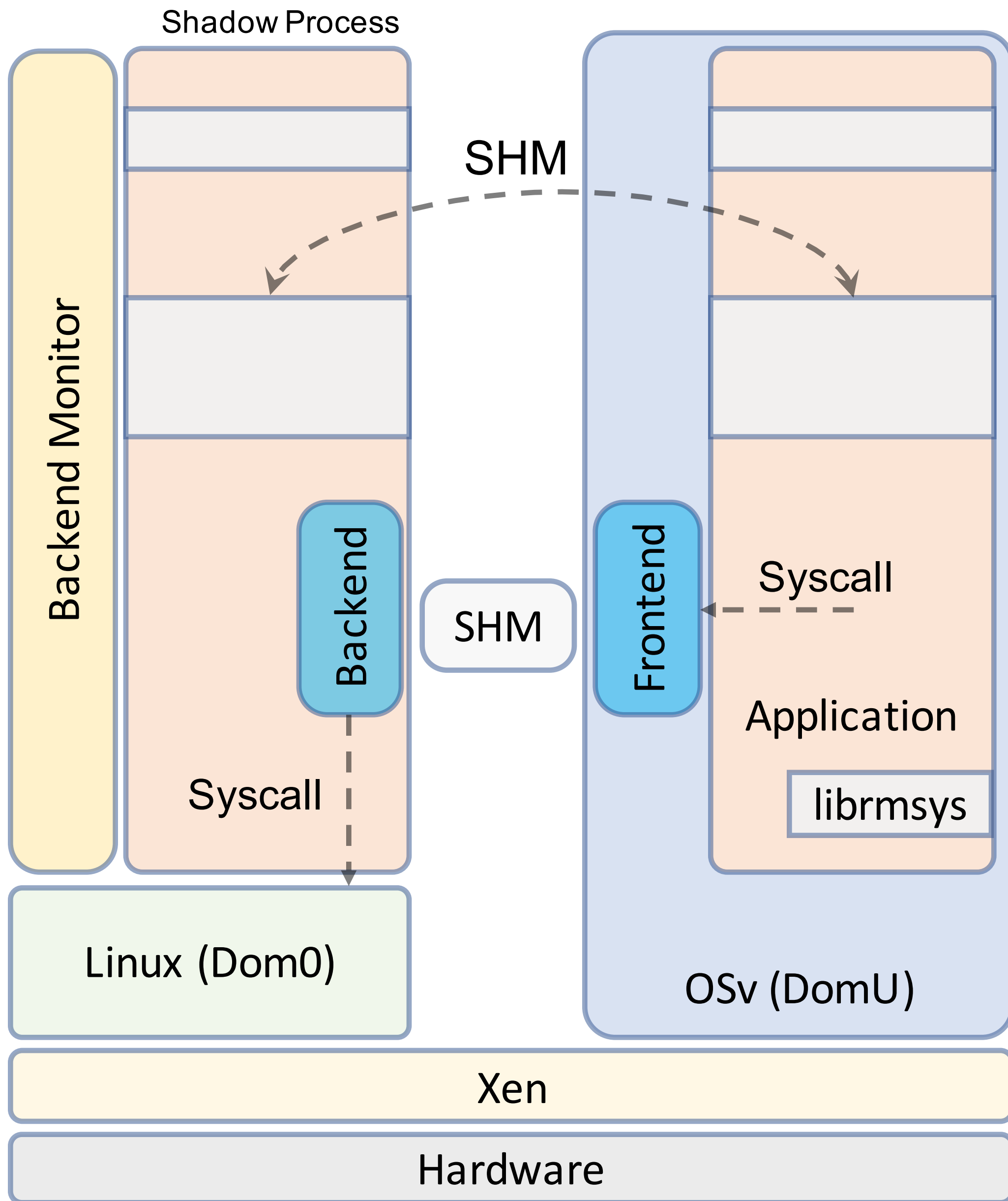

Figure 5.1: Remote System Call Architecture
Remote system calls will be forwarded from frontend to backend via shared memory as-is. Any missing pages will be populated using Xen grant-table shared-memory mechanism after the first `EFAULT` failure. Notice that the unikernel has no notion of kernel space vs user space since it has a single address space, and all code runs in privileged mode (Ring-0 as in x86_64 architecture).

bility issues.

3. Manage two sets of `libc` calls (i.e., remote vs. local). We don't provide additional set of calls that are explicitly remote.

As shown in the Figure 5.1, the framework contains numerous components on each side of the call. We will explain these components in the following sections, and present an example of doing remote system call.

5.2.1 OSv modification

The OSv is modified to support our Remote System Call architecture. To fully enable Xen shared memory and notification, we have partially ported the `gntalloc`[1] and `evtchn`[2] driver onto the OSv. We have also modified the VFS of OSv to enable the support for custom driver. To implement system call redirection, we have added checkpoints before system calls, and we've modified several system call/library implementation to avoid address space collision. We will detail the decision of such modification in the subsequent sections.

5.2.2 Remote System Call API and Library

From the perspective of the application, the API allows it to issue two sets of system calls. A wrapper library `librmsys` is developed to provide APIs to user space. Many of the system calls can be issued locally or remotely, depending on current per-thread state. The state can be switched (on or off), by issuing corresponding special system calls we've added. An example of the use of the API is provided in section 5.3. The issuing of normal system calls are not affected by the use of this library.

Notice that this approach does not distinguish the resources from different kernels. For example, the file descriptor returned from a remote `open` call could be a valid file descriptor at local scale (if there is a filesystem on the unikernel). It is the user (or library)'s responsibility to manage the resources separately. However this does not contradict the transparency objective. The correct

[1] Xen inter-vm shared memory allocation

[2] Xen event channel notification

use of such resources usually only involves calling on/off call pair at correct places.

5.2.3 Remote System Call Split Driver

To avoid privilege layer crossings and to support system calls as-is, we developed a split driver to serve the Linux system calls. Similar to a traditional Xen split driver such as `blk` or `if`[30], the remote system call split driver contains three parts, namely frontend, backend and a backend monitor. The frontend is built into the OSv kernel as a device driver. Upon initialization, the frontend negotiates general configurations with backend monitor, and the backend monitor generates one backend process, for every unikernel requesting services to Linux, according to negotiated configuration. The backend process, or a shadow process, serves as a (almost perfect) clone of the OSv user address space, as well as an execution context, to support system calls and arguments as-is. The system call requests are sent through a shared memory channel established between OSv and shadow process, and the backend will execute the call on behalf of the user. Return value will be put to the same channel when the call returns.

5.2.4 Notification Channel

We implement two different version of notification channels between frontend and backend: interrupt-based and polling-based.

In the interrupt-based implementation, a secure bi-directional Xen event channel was created between the domains at initialization, and notifications were sent in the form of interrupts. Upon the creation of the channel, a tuple of target domain and a port (channel) number were specified by the creating domain for authentication purpose. Only notifications from correct domain and correct port (channel) number were allowed to go through. The interrupt-based implementation is also the standard way of sending notifications between frontend and backend of split drivers such as `blk` or `if`.

In the polling-based implementation, the backend polls the shared communication buffer for incoming new requests. The frontend also polls the shared communication buffer for return value of the system call. This approach sacrifices the CPU time and memory bandwidth for low latency:

It can be up to 15x faster than the interrupt-based approach. We will present more detailed analysis in Section 6.

5.3 Example: Read/Write a File From Linux

We will now present a step-by-step breakdown of the process of reading/writing a file from Linux.

5.3.1 User Application

First, the user application was slightly modified by adding appropriate on/off and enable-disable call pairs at correct places. One possible implementation looks like the following:

```
char buffer[BUFFER_SIZE];
/* error handling ignored */
handle = rmsyscall_enable();

rmsyscall_on(handle);
fp = fopen(file_path, "r");
fread(buffer, size, count, fp);
fclose(fp);
rmsyscall_off(handle);

/* do something with data */

rmsyscall_on(handle);
fprintf(stderr, "this message prints to Linux");
rmsyscall_off(handle);

rmsyscall_disable(handle);
```

In the above code segment, the `file_path` is a relative path with regard to the directory where

the shadow process is running. There is no restriction for pointer arguments such as buffer. It can be either global or local. The expectation is that the `fileno(fp)` (i.e., the file descriptor) is a valid file descriptor on Linux. [3]

5.3.2 Remote System Call Setup: Config Negotiation

In the `rmsyscall_enable` call above, the frontend tries to bootstrap its connection with the backend when there was no shared memory established between them. This can be achieved via XenStore, an information store shared across all VMs. Callbacks and watches can be set on specific keys to enable the backend monitor, running in Domain 0, to be notified of the incoming frontend. XenStore is typically used to store configuration or small-sized information. It is not designed to be a datastore or cache.

The frontend and backend monitor exchange configurations such as application address space layout, notification channel port number, etc., and the backend monitor spawns a shadow process that conforms to the address space layout of the application. The backend running in shadow process then connects to the frontend and establish shared memory channel between them. All communication afterwards are going through the shared memory channel.

5.3.3 Remote System Call: Redirection

To achieve call redirection, we have modified the OSv system call implementation to add checkpoints before system calls[4]. Calling `rmsyscall_on` will change the per-thread state to allow the checkpoint to redirect the call to frontend, which subsequently forwards the call to backend. We present more detailed analysis of the redirection overhead in section 6.

5.3.4 Remote System Call: Invocation and Return

The backend will be notified via the notification channel of the incoming system call request. In above example, the first one will be `open`.

[3] Note that on OSv, if `SYSCALL` instruction is used to explicitly invoke system calls without calling through `libc`, the call would incur an additional context switch cost.

[4] The system call dispatcher in OSv is used to serve `SYSCALL` instruction. All `libc` calls go through a custom `libc` implementation to eliminate the context switch overhead of system calls.

The first attempt of executing `open` from backend will result in `EFAULT` because `file_path` points to a non-mapped region in shadow process. On receiving this error, the frontend will initiate a page sharing request to backend, where the page containing `file_path` string will be shared across VM and mapped at the exact same virtual address with same permission in shadow process.

If the above process is successful, `open` will be retried one more time, and a file descriptor will be returned to the user.

The process for `read` is similar, except that we adopt pre-fault strategy (populate pages before the first attempt) for `read` because it may not return `EFAULT` when only part of the buffer was mapped.

If `EFAULT` persists in the second attempt, or some other errors are returned in the first attempt (e.g., `ENOENT`, No such file or directory), we return the error directly without retrying.

When the user application exits, the shadow process will receive a state change of XenStore entries, and therefore is able to properly finalize and exit.

5.4 Challenges and Issues

There are many challenges in serving remote system calls, especially the ones containing pointers.

5.4.1 Page Fault Handling

The pages in shadow process are populated in a hybrid strategy. Unlike a page fault in user space, system calls do not fault and no signals are generated. We could only identify a fault by examining the return value of the system call.

Most system calls will return `EFAULT` (i.e., Bad address) error when the supplied user space address was inaccessible (i.e., not mapped, or permission denied). When a system call returns with `EFAULT`, a page sharing routine is initiated by frontend to populate the missing pages from OSv to the exact same virtual address in shadow process. The system call will be retried one more time after all missing pages were populated.

In order to identify the missing page address, we perform a static annotation of system calls.

More specifically, we keep a metadata of each system call that includes:

- Which arguments contain pointers, and the R/W property (const vs. non-const) of the pointed memory area;
- Which arguments contain sizes of what buffers;
- If a pointer points to a `struct` with nested pointers (e.g., `struct iovec` in `readv`/`writev`), recursively record the above two points.

Some of the system calls however (e.g., `read`/`write`), will half-finish instead of returning `EFAULT`. For example, if `read` is called to fill a page-aligned buffer of 8KB with only the first half mapped, the `read` will return with the number of bytes successfully written (i.e., 4096) instead of an `EFAULT`. In this case we adopt the pre-fault strategy where we examine the arguments of the system calls to actively populate the buffer before the invocation of the call.

The page mappings will be updated upon memory changes on user application, e.g., after a `munmap` call.

The page sharing is also achieved through Xen grant-table mechanism. The page fault handler could be placed in either frontend or backend, and we choose to place it in frontend to reduce the code size of the backend.

5.4.2 Address Space Collision

There may be address space collision between shadow process and user application. More specifically, two types of address space collision may happen in the process of page sharing. The first type is the collision between shared pages and pages private to shadow process (i.e., the backend code and data page, process stack, etc.) To reduce such potential collision, as described in section 5.3.2, the shadow process was built without shared libraries, and code and data pages was placed appropriately according to the address space layout. During the initialization of shadow process, process stack is also moved to a vacant region. To fully eliminate such collisions, user may reserve enough space (typically several 4K pages) at application-linking stage to explicitly accommodate pages private to shadow process.

The second type is that the unikernel and shadow process address space may not be one-to-one mapped. unikernel owns the entire virtual address space as supported by MMU (typically 64-bit on modern machines), while Linux user process has an virtual address upper limit of `0x7fffffffffff` (i.e., 128TB). Currently we do not support populating pages beyond user space limit, and we've modified OSv to make sure that mapped address ranges of a typical user application, including stack, `mmap`'ed and `malloc`'ed pages are within the 128TB limit.

5.4.3 Unsupported System Calls

Some system calls are not remote executable, in the sense that it may break the cooperation of frontend and backend. Currently we do not support the forwarding of the following system calls:

1. `exit`, `exit_group`, `fork`, `vfork` or `execve`; The `fork` and `execve` are not supported by OSv.

2. Memory related, such as `mmap`, `mremap` and `munmap`. The role of shadow process is to automatically clone the address space of OSv. Manipulating the memory mapping of shadow process is not supported.

3. Shared memory related, such as `shmget`, `shmat`, `shmctl`. Currently we do not support the cross-VM sharing of `shm` object.

However, we plan to add support for `mmap` in our future work to enable mapping of files (and especially, drivers) from Linux VM.

6. EXPERIMENTATION AND EVALUATION

In order to demonstrate the effectiveness of our approach, we plan to evaluate the following characteristics:

1. **Hybrid services**: Is application able to access services from both kernels? What is the performance (e.g., network latency/throughput, OS jitter) of such services?
2. **Performance isolation:** Do remote system calls affect the performance of extruded services (i.e., OSv)? Do remote system calls affect the rest of the Linux?
3. **Zero privilege boundary crossing:** Do remote system calls cross privilege boundaries? What is the latency of the forwarding process?
4. **Transparency:** How many lines of code change is required for an application to access remote system calls?

As a proof of concept, we ported a real-world data-center application, Redis [31], onto our platform. The performance of the application, as well as the performance of remote system calls and OS jitters, will be measured to answer above questions.

6.1 Experiment Settings

All of the following experiments were conducted on a Dell Latitude E5270 Laptop with Intel Core i5-6300U at 2.4GHz, 8GB memory and Gigabit Ethernet Controller. We use Xen version 4.9.2 with Ubuntu Linux 18.04 as Domain 0.

6.2 Redis NoSQL Store

Redis is a single-threaded, in-memory key-value datastore. To achieve the goal of hybrid services, we have slightly modified Redis on OSv (in about 10 lines of code) to enable the reading of configuration file from Linux. Redis implements its own event library to achieve efficient I/O multiplexing. To serve a read request, `epoll_wait` is called to wait for incoming request, followed

by a `read` system call to read the request. The response is then generated after some processing, and sent to the client using a `write` system call. For write requests, Redis will additionally log every write operation into an append-only log, and the log will be persisted to disk at a frequency defined by the `fsync` policy (i.e., perform `fsync` every second, for every request, or not at all). In our test scenario we have disabled the persistence (i.e., never `fsync`) because this task is usually delegated to a forked child, and therefore not feasible in Unikernel scenario, as OSv does not support `fork`.

We run Redis server in three different settings:

1. Redis server runs in OSv with system call access to Linux, as illustrated in Fig 5.1. The OSv was given 2 VCPUs and 1GB memory.

2. Redis server runs in a Ubuntu Linux 18.04 VM (User Domain) with 2 VCPUs and 1GB memory.

3. Redis server runs on bare-metal Linux, and was limited to use only 2 cores and 1GB memory using Linux control group (`cgroup`).

6.2.1 Redis Benchmark

We use the `redis-benchmark` program in the Redis source tree for performance benchmarking. The benchmark client runs on a MacBook Pro early 2016 version. The laptop running the server was connected to a router via a cable, and the MacBook Pro was connected to the same router via wireless network.

The Redis benchmark conducts different operations as listed in Table 6.1.

Redis server supports pipelining, where multiple requests are sent at once. On server side only one `read` system call is needed to read all batched requests, and one `write` call to write all the responses, thus significantly improve the throughput of the server. By default the pipeline batch size of the benchmarking program is set to 1 (i.e., no pipeline, at most 1 request pending per connection).

Operation	Description
PING_INLINE	Ping-Pong echo.
SET	Set **key** to hold string **value**.
GET	Get **value** stored at **key**.
RPUSH	Insert value(s) to the tail of the list stored at **key**.
LPOP	Removes and returns the first element of a list stored at **key**.
SADD	Add the specified members to the set stored at **key**.
HSET	Set *field* stored at **key** to *value*.
SPOP	Removes one or more random elements from set stored at **key**.
LRANGE_n	Get first n elements of the list stored at **key**.
MSET	Same as SET, operates on multiple **key**s.

Table 6.1: Redis Benchmark Operations

Note that Redis frequently invokes `gettimeofday` to generate timestamps for logging. Usually on bare metal Linux, the call to `gettimeofday` will be accelerated by C library using Virtual Dynamic Shared Object (vDSO), to minimize the latency of the call to obtain more precise timing. In this case, calling `gettimeofday` no longer requires a context switch into the kernel, and the latency of such call is usually comparable to a normal function call.

Linux VM on Xen sets the clock source to `xen` by default. This will prevent `gettimeofday` from utilizing vDSO, and therefore incurring the cost of a full system call. This can significantly degrade Redis performance under high pipeline work load. To work around this problem, we set the clock source to `tsc`. Our experiment showed that under pipeline mode with batch size 16, PING_INLINE test has gained 48% performance increase simply by changing clock source from `xen` to `tsc`.

It is worth noting that setting the clock source to `tsc` may itself incur additional cost because by default [32], `tsc` was emulated to ensure the correctness (e.g., time never shifts backwards) of the time keeping.

To fully stress the kernel, we run `redis-benchmark` with 600 concurrent clients and no pipelining (pipeline value 1). Figure 6.1 shows the performance of Redis in requests/sec.

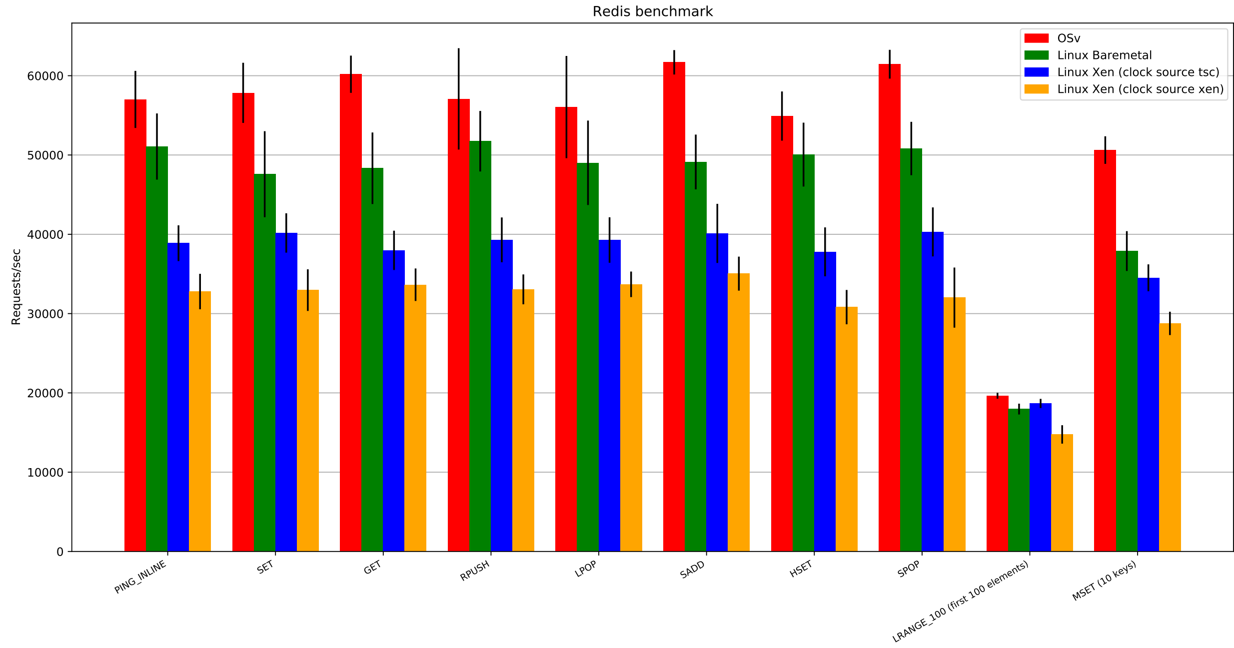

Figure 6.1: Redis performance, 600 concurrent clients, no pipelining

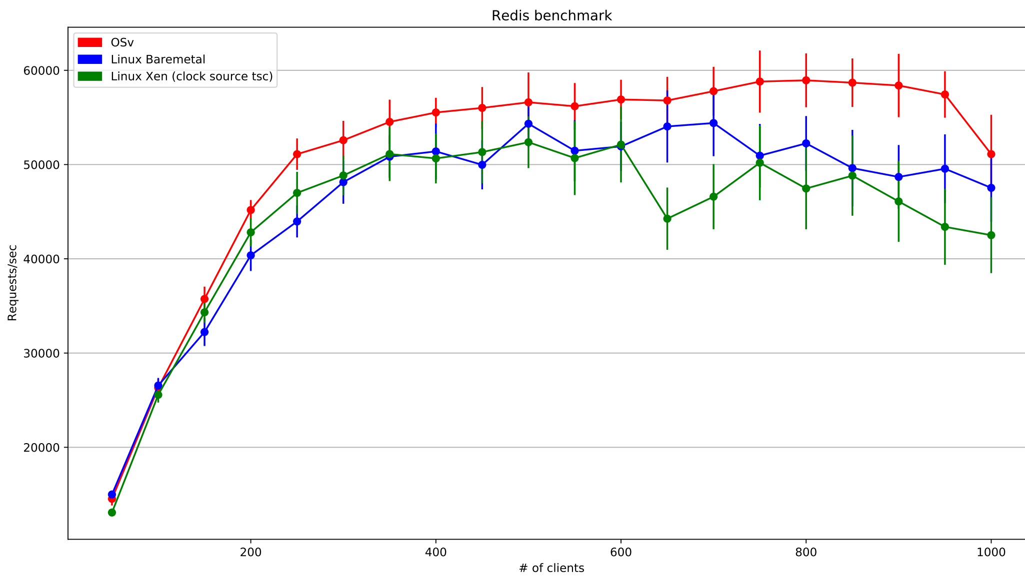

Figure 6.2: Redis PING_INLINE performance

Figure 6.2 shows the performance of PING_INLINE with variable concurrent clients size. As shown in the figure, OSv is able to sustain more concurrent clients because of the low overhead incurred by the system call, i.e., no context switches are needed to perform system calls. We present more detailed system call analysis in section 6.4.

However, because of the limitation of using wireless network, measuring the latencies of Redis requests becomes less helpful. Instead, we perform a OS jitter analysis to better understand the behavior of applications on different platforms in section 6.3.

6.3 OS Jitter

OS jitter refers to the interference experienced by the user application due to scheduling of background daemon processes or handling of interrupts.

In our experiment, we use Sysjitter v1.4 [33] to measure the OS jitter on both OSv and Linux. It does so by running a thread on each assigned core, in which the Time Stamp Counter was polled in a tight loop. The time difference between consecutive reads, if larger than a predefined threshold, is recorded as one instance of interruption. All polling threads start and end at the same time to make sure there were no CPU power throttling during the test.

6.3.1 Settings and Results

The test program runs in three different settings: OSv, Linux on Xen, Linux on bare metal. It runs for 60 seconds on each platform, and ignoring any interruption that takes less than 200 nanoseconds. On both Linux platforms, the test program runs with highest priority (nice value of -20) and there were no other competing tasks running. To ensure fairness, we run benchmark on same number of cores across platforms. However, on one hand, since there are 2 cores and 4 threads on bare metal (4 cores from the perspective of Linux Bare metal), giving OSv VM 4 VCPUS would be unfair against the OSv because Dom0 has to run on enough cores (4 cores as recommended by Xen documentation [34]) to support other running VMs, thus resulting in frequent VM scheduling (and higher jitter) between Dom0 and OSv VM.

On the other hand, running benchmarks on all available cores usually results in higher jitter

compared to running benchmarks on only a subset of the cores. Table 6.2 illustrate the performance of running benchmark on 2 out of 4 cores, and 4 out of 4 cores on Linux bare metal, with all other conditions identical.

Platform and cores	N Samples	Mean	Median	Sum	STD Deviation
Linux Baremetal (2/4) core 0	18316	2389	2364	43759255	4639.40
Linux Baremetal (2/4) core 1	17436	4331	2015	75518879	16610.20
Linux Baremetal (4/4) core 0	19549	5125	2746	100193188	84691.97
Linux Baremetal (4/4) core 1	21033	12159	2621	255736168	278826.02
Linux Baremetal (4/4) core 2	27088	3531	2164	95636065	96686.58
Linux Baremetal (4/4) core 3	18958	4327	2645	82037344	13254.61

Table 6.2: Samples Summary of 2/4 and 4/4 benchmark

In this case, currently the best choice we have is to give all VMs two cores only and run benchmark on core 0 and 1 on Linux bare metal, which may give slight advantage over Linux Baremetal.

Figure 6.3 shows a distribution of the samples collected. Table 6.3 shows a summary of collected samples. The X-axis denotes the Sample ID, and the Y-axis denotes the interrupt duration in log scale.

Platform and cores	N Samples	Mean	Median	Sum	STD Deviation
OSv core 0	13022	2247	1359	29265417	12440.02
OSv core 1	12258	2002	1369	24540510	8163.58
Linux Baremetal core 0	18316	2389	2364	43759255	4639.40
Linux Baremetal core 1	17436	4331	2015	75518879	16610.20
Linux Xen core 0	28333	6455	6419	182896448	152263.61
Linux Xen core 1	29636	5420	6652	160640749	80364.89

Table 6.3: OS Jitter Samples Summary

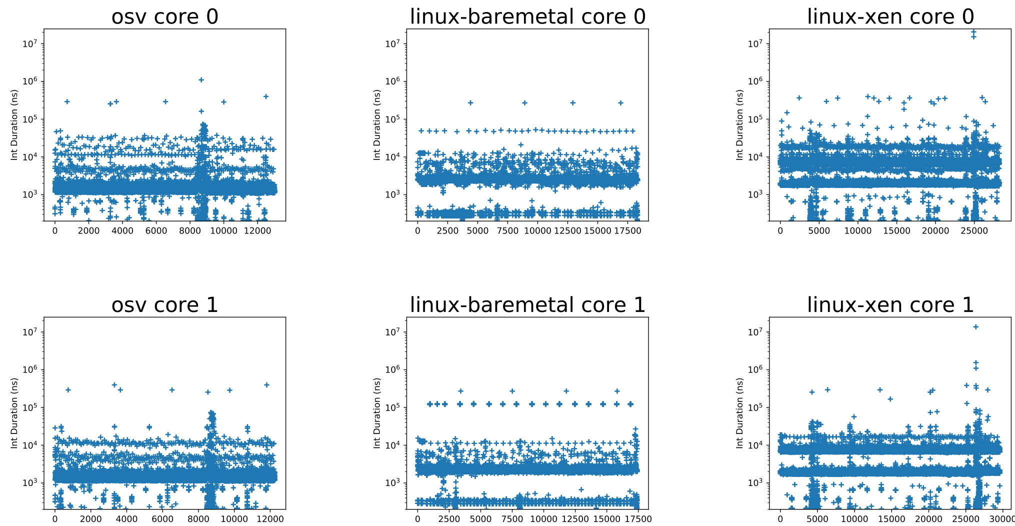

Figure 6.3: OS Jitter

From the figures and table we can see that although the variation is slightly higher, the number of jitters in OSv is about 40% less than that in Linux bare metal, with lower mean and median interrupt time than Linux bare metal.

In general, OSv is able to achieve a good jitter performance compared to Linux bare metal.

6.4 Remote System Call Performance

As shown in Fig 5.1, access to Linux system calls are forwarded through a shared buffer. In this section we measure the latency of such calls.

6.4.1 NULL System Call Performance

NULL system call refers to a system call attempt with an invalid system call number. On x86/x86_64 Linux, this would cause the Linux system call dispatcher to return immediately with an error of `ENOSYS` (i.e., no such system call). In this test, we invoke the NULL system call using `SYSCALL` instruction. Table 6.4 shows the NULL system call performance. OSv Poll refers to the polling-based notification implementation, while the OSv Int refers to the standard interrupt-

based notification implementation. Linux Xen refers to the Domain 0 on Xen. All the following experimentation are results averaged from 64 runs, with 10000 calls for each run.

Platform	Time (μs)
OSv Poll	2.26
OSv Int	$17 \sim 30$
Linux Xen	1.92
Linux Bare Metal	0.54

Table 6.4: NULL System Call Performance

Note that the performance of OSv Int was largely affected by the virtual interrupt delivery on x86. According to Intel Software Developer Manual [35] Chapter 33.3.3.4, the delivery of a virtual interrupt event can be affected by the priority of the virtual interrupt event, or the interruptibility of a guest VM (i.e., RFLAGS.IF being 0 indicates a non-interruptable state). If the guest is non-interruptible, the hypervisor may queue the virtual interrupt until it becomes interruptible. In addition, if the guest VM is not scheduled at the time of generation of the virtual interrupt event, the delivery of the virtual interrupt may happen at next `VMENTRY`, i.e., the transition of a logical processor from host mode (VMX root mode) to guest mode (VMX non-root mode). The scheduling within the Linux kernel may also affect the performance of the interrupt delivery.

In general, the interrupt-based implementation can achieve relatively low overhead while having good scalability, and poll-based implementation can achieve latency comparable to native system call at the cost of CPU cycles and memory bandwidth.

6.4.2 Read/Write System Call Performance

Shadow processes rely on Xen shared memory to gain access to the memory pages of the Unikernel. The shared memory is made available to shadow process by `mmap`ing the shared memory driver (i.e., `gntdev`). On Linux, the driver allocates empty shared memory from ballon memory driver, and Xen modifies the Extended Page Table to map the source and target memory pages to the same physical page frame.

In this case, one would naturally question how the use of shared memory would affect the performance of the remote system calls. To answer this question, we conduct `read`/`write` system calls on shared memory.

We `read` from / `write` to an in-memory file with random content of size 1 to 32768, to/from a buffer of 32768 bytes. The file was randomly modified and re-opened between calls. The results were averaged from 64 runs for each read/write size. Figure 6.4 and 6.5 shows the read and write system call latency. All calls in OSv cases, except for OSv local, were forwarded to Linux. OSv local refers to a call at local Unikernel (i.e., the call was served inside Unikernel).

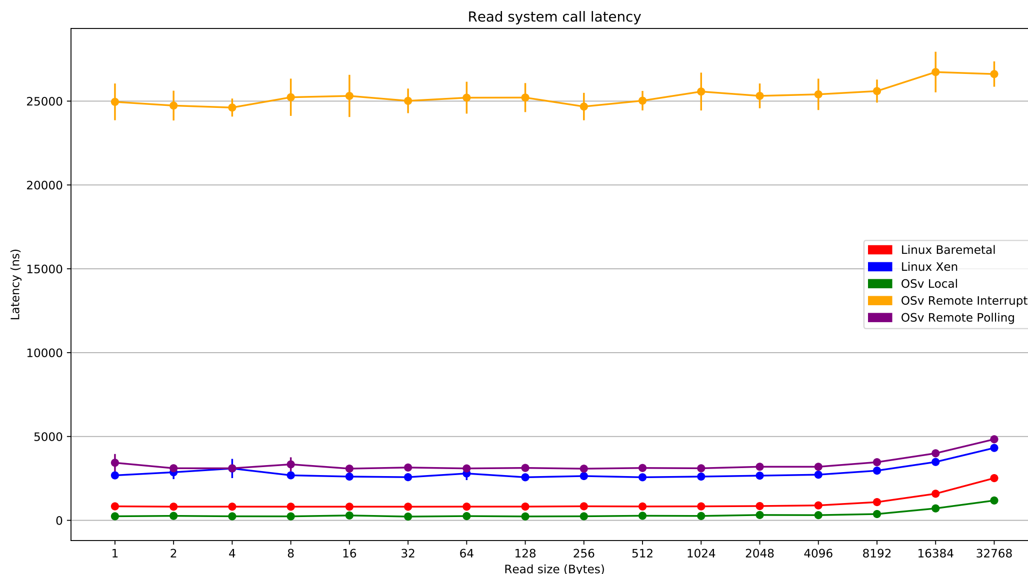

Figure 6.4: Read system call latency

As shown in the figures, the performance of OSv Remote Polling was very close to that of the Linux Xen, where the former consisted of a forwarding routine and a call to shared memory from Linux. From the figures we can conclude that the use of shared memory, in non-NUMA machines, will not incur additional performance penalties.

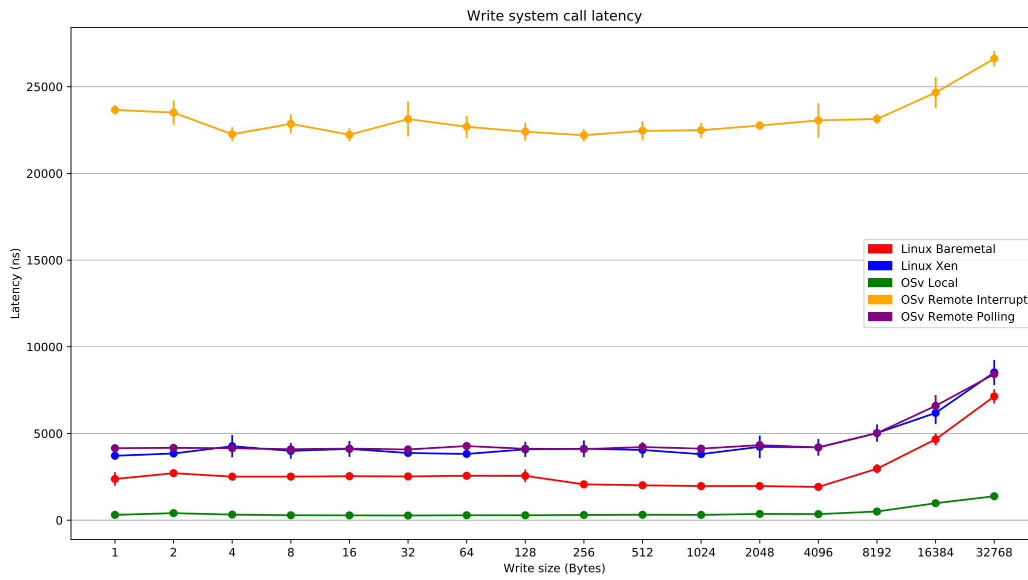

Figure 6.5: Write system call latency

7. CONCLUSION AND FUTURE WORK

This thesis presents Service Extrusion architecture, where services can be "extruded" from general-purpose kernel and run alongside the kernel. The application will be given access to both the general-purpose services and the extruded services. The performance of two types of services were isolated, and porting an application onto the extruded services requires no to minimal amount of code change. We also present a possible implementation of such architecture, namely Remote System Call Framework, that is able to access Linux system calls without privilege layer crossing. Remote System Call Framework satisfy all the design goals we have laid out for Service Extrusion.

The potential of our architecture is yet to be fully explored. Vertically, we can achieve even higher single-Unikernel networking performance by implementing a driver that supports NIC passthrough (i.e., granting Unikernel exclusive access to the NIC hardware). Horizontally, we can implement SR-IOV to offload the packet multiplexing task to the NIC itself, to reduce the cost of doing multiplexing in hypervisor software.

Access to general-purpose services from multiple extruded services can be further isolated by utilizing container technologies, where each shadow process resides in a different container, and therefore different and isolated service context. Memory inspection to the Unikernel is also partly achievable by instrumenting the shadow process.

We plan to add support for remote memory map in the future, where the application is able to memory map a file on Linux, in particular a driver file, into the memory. This would enable seamless access to Linux drivers without porting them to the Unikernel. A supporting library that was Remote System Call-aware can also be developed to fully utilize the shadow process. Some example usages includes creating a thread on, or `fork` to a remote machine.

Performance of our architecture on large NUMA machines still remains to be studied, and it is likely that additional shared memory or cache coherence optimizations are needed to maintain good performance.

REFERENCES

[1] S. Peter, J. Li, I. Zhang, D. R. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 4, pp. 1–30, 2015.

[2] D. Intel, "Data plane development kit," 2014.

[3] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft, "Unikernels: Library operating systems for the cloud," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, pp. 461–472, 2013.

[4] A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, *et al.*, "Jitsu: Just-in-time summoning of unikernels," in *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pp. 559–573, 2015.

[5] D. Williams, R. Koller, M. Lucina, and N. Prakash, "Unikernels as processes," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 199–211, 2018.

[6] C. Clark, "Xen grant table documentation."

[7] I. Habib, "Virtualization with kvm," *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.

[8] "IntelÂő virtualization technology for directed i/o (vt-d): Enhancing intel platforms for efficient virtualization of i/o devices."

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 164–177, 2003.

[10] "Xen event channel documentation."

[11] "Xenstore."

[12] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd, "The virtual interface architecture," *IEEE micro*, vol. 18, no. 2, pp. 66–76, 1998.

[13] J. C. Mogul, "Tcp offload is a dumb idea whose time has come.," in *HotOS*, pp. 25–30, 2003.

[14] V. Yodaiken and M. Barabanov, "Real-time linux applications and design," *Slides from Usenix presentation*, vol. 14, pp. 16–17, 1997.

[15] P. Mantegazza, E. Bianchi, L. Dozio, S. Papacharalambous, S. Hughes, and D. Beal, "Rtai: Real-time application interface," 2000.

[16] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "{IX}: A protected dataplane operating system for high throughput and low latency," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pp. 49–65, 2014.

[17] A. Schüpbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs, "Embracing diversity in the barrelfish manycore operating system," in *Proceedings of the Workshop on Managed Many-Core Systems*, vol. 27, 2008.

[18] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged {CPU} features," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pp. 335–348, 2012.

[19] W. Li, Y. Xia, H. Chen, B. Zang, and H. Guan, "Reducing world switches in virtualized environment with flexible cross-world calls," *ACM SIGARCH Computer Architecture News*, vol. 43, no. 3S, pp. 375–387, 2015.

[20] "Intel virtualization technology list."

[21] "Arm product list."

[22] H. Li, X. Xu, J. Ren, and Y. Dong, "Acrn: a big little hypervisor for iot development," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pp. 31–44, 2019.

[23] A. Lackorzynski *et al.*, "L4linux porting optimizations," *Master's thesis, TU Dresden*, 2004.

[24] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, *et al.*, "sel4: Formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 207–220, 2009.

[25] J. Che, C. Shi, Y. Yu, and W. Lin, "A synthetical performance evaluation of openvz, xen and kvm," in *2010 IEEE Asia-Pacific Services Computing Conference*, pp. 587–594, IEEE, 2010.

[26] A. Wick, "The halvm: A simple platform for simple platforms," *Xen Summit*, 2012.

[27] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library os from the top down," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pp. 291–304, 2011.

[28] A. Kantee, "Rumprun," 2016.

[29] A. Kivity, D. Laor, G. Costa, P. Enberg, N. HarâĂŹEl, D. Marti, and V. Zolotarov, "OsvâĂŤoptimizing the operating system for virtual machines," in *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pp. 61–72, 2014.

[30] D. Chisnall, *The definitive guide to the xen hypervisor*. Pearson Education, 2008.

[31] S. Sanfilippo and P. Noordhuis, "Redis," 2009.

[32] D. Magenheimer, "Xen tsc_mode documentation."

[33] D. Riddoch, "Sysjitter," 2017.

[34] "Tuning xen for performance."

[35] I. Intel, "Intel-64 and ia-32 architectures software developerâĂŹs manual," *Volume 3C: System Programming Guide, Part*, vol. 3, no. 64, 2020.