The University of New South Wales
School of Computer Science and Engineering

Zircon on seL4

by

Jack Suann

Thesis submitted as a requirement for the degree
Bachelor of Engineering (Computer Engineering)

Supervisor: Gernot Heiser

Student ID: z5017518
Submitted: May 29, 2018

Abstract

seL4 is a high-performance microkernel with a proof of functional correctness, providing isolation guarantees to properly designed applications. This makes it suitable for systems where strong security guarantees are required. Despite these benefits, seL4 lacks the support of other major operating systems, and manually porting the drivers and applications from such systems is often too expensive of a solution. A more feasible approach is to re-use components of these systems and port them on seL4, providing an environment for applications that matches the specifications of the original system.

This thesis presents Zircon on seL4, a port of Google's new Zircon microkernel to seL4. This is achieved using an API emulation approach, with a seL4 application managing Zircon kernel objects, and handling syscalls from Zircon applications that are delivered using seL4's IPC mechanisms.

This thesis details the kernel objects and syscalls that have been implemented in this port, and how they interact with the primitives provided by seL4. This implementation is evaluated in terms of performance and porting effort. Finally, an analysis of the the future work that remains to be done to fully support dynamic user-level Zircon-based systems on seL4 is given.

Contents

List of Figures

List of Tables

Abbreviations

ACL Access control list

API Application programming interface

CSpace Capability Space

ELF Executable and Linkable Format

FIFO First in, first out

IPC Inter-process communication

IRQ Interrupt

I/O Input/output

LK Littlekernel

MCS Mixed-criticality system

OS Operating system

POSIX Portable Operating System Interface

Rootfs Root file system

SC Scheduling context

TCB Thread Control Block

TLB Translation lookaside buffer

TSC Time Stamp Counter

vDSO Virtual Dynamic Shared Object

VKA Virtual Kernel Allocator

VMAR Virtual Memory Address Region

VMO Virtual Memory Object

VSpace Virtual Address Space

1 Introduction

The ability to run one system on another is a widely used concept, whether it be running another system in a virtual machine, or emulating its API so its applications can run seamlessly on another system. One major reason for doing this is to provide access to software and drivers not supported by the base system.

The seL4 microkernel presents major benefits compared to other platforms. It is the only microkernel with a proof of functional correctness [KEH$^+$09], and provides fine-grained access control of kernel resources, making it suitable for any system where strong security guarantees are desired. Even with these security benefits, seL4 is also a high-performance microkernel, providing communication between processes at a low cost. Despite these advantages, seL4 still lacks support for software and drivers compared to major platforms such as Linux. Part of this is due to the differences in seL4 and these traditional platforms, which makes it non-trivial to port device drivers and other application, many millions of lines of code in size, from these traditional platforms to seL4.

Zircon is a new microkernel being developed by Google [Goo17b], and provides the base platform for the Fuchsia operating system [Goo17a]. The project is under very active development, and with continued support there is a high possibility that Fuchsia will be adopted by mobile and other personal devices in the near future. Many devices running the Zircon microkernel would lead to increased software development and driver support on the platform, and having the ability to run such a system on seL4 would be very advantageous. Additionally, the security guarantees provided by seL4 can be leveraged to ensure misbehaving Zircon applications remain isolated from native seL4 applications.

This thesis presents Zircon on seL4, a project aimed at porting the syscall API of Zircon and its underlying kernel objects to seL4. This thesis contributes a port of the major kernel objects of Zircon, and a large proportion of the system calls it provides. We also perform an evaluation of the port, in terms of its performance relative to native Zircon and the effort required for its implementation. Additionally, this thesis details the future work that remains for a complete port. This includes the remaining kernel objects that need to be implemented, as well as the work required for other aspects such as multicore support, as well as potential approaches for forming communication channels with native seL4 applications.

2 Background

This section provides background information to help build an understanding of the concepts behind seL4 and Zircon. We start with a discussion of kernel design, as well as security models used in kernels. We then delve into each microkernel, detailing the features and primitives offered by seL4 and Zircon.

2.1 Kernel Design

A kernel contains the core functionality of an operating system, providing the interface for user applications to interact with hardware and other applications. They are thus a privileged component of the system, with full control over all resources. Traditional kernels such as Linux kernel are described as *monolithic*, in which all operating systems services, such as file systems and device drivers, are included in the kernel and are thus privileged components. As such, bugs in these components can compromise the entire system, and the large code base inherent to monolithic kernels makes eliminating all such bugs virtually impossible.

Microkernels take the opposite approach to monolithic kernels; they are designed to have a minimal trusted computing base, and avoid including operating system services in the kernel where possible. Microkernels provide a minimal set of kernel primitives to user applications, providing the necessary abstractions so that applications can still interact with hardware and other processes. Common abstractions include message passing for inter-process communication (IPC), execution primitives such as threads, and objects for managing virtual memory and interrupts.

2.2 Capability-based Security

As the kernel is a privileged component, it must only allow access to kernel objects to user applications that require them. A security model must be implemented in the kernel to protect these objects from malicious use by untrusted applications. In systems where security is paramount, it is desirable that *fine-grained security* is enforced, in which applications are supplied with the minimal set of permissions required for full functionality. This is based on the principle of least privilege: that users should only be granted privileges essential to their operation. This can prevent malicious or faulting applications from compromising other users by performing privileged operations that they do not require in the first place.

The traditional security model established by Unix and Unix-like systems are *access control lists* (ACLs). In this model, each object maintains a list of permissions that dictate which users can access the object and what operations these users can perform (such as read or write operations). The primary limitation of ACLs is that they can become unwieldy if they contain entries for each individual user. This can be addressed by limiting the scope of permissions maintained; this is done in the Unix permission model, in which the ACL only has entries for a user, a group and others. Such an approach can only enforce coarse-grained security, and lacks the flexibility required for high-assurance systems.

Capability-based security is an alternative approach, in which users hold keys, or capabilities, to kernel objects. The capabilities themselves dictate the access rights the user has; fine-grained security can be enforced by only supplying users with the capabilities they require. To share access to objects, these capabilities need to be copyable, and the permissions granted to these copies must be equal to less than those of the parent capability to prevent any escalation of privilege.

3 seL4

seL4 is the only microkernel that is formally verified; it has a proof of functional correctness demonstrating that the kernel follows its specifications. This is made possible due to seL4 being a microkernel; verification is made feasible through the small code base of the kernel. Systems that utilise seL4 correctly can enforce strong isolation properties between applications, ensuring that trusted programs can not be compromised by malicious code from untrusted applications. In addition to these security properties, seL4 is a high-performance microkernel, providing fast inter-process communication.

seL4 provides a small set of kernel objects for implementing larger user-level systems. One of the design aspects of seL4 is a lack of policy enforcement; it avoids restricting the user in their design decisions, providing objects that are flexible and can be applied to a variety of systems [HE16]. The following subsections will detail these kernel objects, particularly aspects of them relevant for porting Zircon to seL4.

3.1 Capabilities and Objects

seL4 uses capabilities (caps) for managing access to kernel objects. Every thread on seL4 has an associated *capability space* (CSpace) which stores the caps that a thread owns. A CSpace is composed of kernel objects known as *CNodes*; each CNode has a number of slots in which caps can be stored. These caps can include additional CNodes, which allows a graph of CNodes to be formed, starting from the root CNode of a CSpace. It is this graph of CNodes which forms the complete CSpace; this graph is constructed as a guarded page table to allow for the efficient lookup of caps in a CSpace. This allows for the exact design design of a CSpace to be flexible; a CSpace can be simple and consist of a single root CNode, or can have multiple CNode levels, with CNodes being allocated to the CSpace as required, allowing for more dynamic CSpace management.

Operations on seL4 kernel objects are referred to as kernel invocations. Almost all kernel invocations require a capability to be provided. Invocations common to seL4 objects are cap operations, including invocations to copy, move or delete capabilities from a CSpace. Kernel invocations send arguments to the kernel through message registers, and in many cases an IPC buffer. These are explained in more detail in the following section.

3.2 Endpoints

Endpoints provide the primary method of inter-process communication (IPC) between threads on seL4. They allow for the transfer of a small amount of data and caps through the use of message registers and an IPC buffer. The IPC buffer is a fixed region of memory assigned to a thread which is used to store the data of a message. A set of message registers is used to transfer the most important components of a message, such that if a message is small enough, it will only use the message registers. By avoiding the use of memory in such cases, IPC with small messages can be very efficient.

Operations on endpoints are synchronous and blocking; a thread will wait on an endpoint until they have sent or received a message. There are two primary invocations sending a message with an endpoint: send and call. Both of these will block until another thread is ready to receive before sending a message. A call invocation additionally sends a reply cap to the receiver. This reply cap is stored in the Thread Control Block of the receiving thread. This reply cap can later be used in a reply invocation, which sends back a message to the relevant caller. As such, a receiving thread can easily respond to a calling thread without any additional bookkeeping.

Another important aspect of endpoints is the ability to *badge* an endpoint capability. An endpoint cap can be minted, which creates a copy of the cap with a chosen badge value. This value is dictated by the invoking thread and can be up to 28-bits in size. Badges are used for the identification of a sending or calling thread; the badge of the endpoint cap used to send a message will be transferred to a receiving thread's badge register, where it can easily be read. Another important aspect of these caps is the inability to further copy or modify a capability with a non-zero badge. This ensures that badged caps can be reliably used to identify a sender, as there is no way for the sending thread to modify a badged cap that is assigned to it.

3.3 Virtual Memory Management

A virtual address space in seL4 is referred to as a *VSpace*. A VSpace is composed of various kernel objects that are used to manage virtual memory. These objects generally correspond to the underlying hardware page table structure. On 64-bit x86, this is a 5 level page table structure, with a single top level page directory, all the way down to 4096 byte pages. seL4 has equivalent kernel objects to match each level. Each level of the page table must be mapped in a VSpace before page frames can be mapped in.

As with all other kernel objects, page frame and page table have capabilities to them, which are used in their mapping invocations. Page table objects cannot be shared between VSpaces, but pages can. To share a page, a copy of its capability must first be made, then this new cap is mapped into the other VSpace. This allows for shared memory regions to be mapped into many VSpaces.

3.4 Threads

As mentioned earlier, a seL4 thread is associated with a *Thread Control Block* (TCB). A TCB is always associated with a CSpace and a VSpace; each of these can either be exclusive to the thread, or shared with other threads. The combination of these three components is largely equivalent to processes in other systems, and are often referred to as such. Threads can also be associated with an IPC buffer. Although not compulsory to have one, an IPC buffer is required for the transfer of larger messages over IPC and for most kernel invocations.

The regular variant of seL4 uses static priorities for threads, with round-robin scheduling. The priority of a thread can be changed manually through a kernel invocation. A mixed-criticality systems (MCS) branch of seL4 also exists [LMAH18], which adds scheduling context (SC) objects; these are object that represent CPU time. Threads must be bound to a scheduling context object in order to run. Scheduling contexts can be configured to change the upper bound on the period of a thread's execution, allowing for more flexible scheduling.

A thread can also be assigned a fault endpoint, which allows for the handling of a faulting thread by another. This fault endpoint exists in the CSpace of the thread; when a thread faults, the kernel delivers an IPC message to the endpoint which describes the fault. Another thread can then receive this message and handle the fault appropriately. If a fault can be resolved, the receiving thread can then resume the faulting thread.

3.5 Notifications

Notifications provide a simple primitive for signalling and waiting. They consist of single data word, which represents a set of binary semaphores, and can be signalled and waited upon. Caps to a notification object can be badged in the same way as endpoint caps can. Signalling a notification object with one of its capabilities causes the notification object's data word to be bitwise OR-ed with the capability's badge. Waiting on a notification object causes the thread to block on it if its data word is zero, adding the thread to the notification's wait queue. If the word is non-zero, or if the notification is signalled when threads is are waiting on it, the first thread is woken up and the value of the notification word is returned to it, after which the notification word is cleared.

One very useful aspect of notifications is their ability to be bound 1-to-1 with a TCB object. Any signals set on a bound notification are immediately delivered to a thread, even if it is waiting on an endpoint to receive a message. Notification signals can be distinguished from an IPC message by checking the badge value returned; it is the responsibility of the user to use different badges for every notification caps to ensure the source can be determined. The only limitation of binding a notification is that it can only be waited upon by the thread it is bound to.

Another aspect of notifications is their relationship with interrupts. seL4 provides IRQHandler capabilities for gaining access to specific interrupt vectors; these can be created using IRQControl capabilities. IRQHandlers can then be set to deliver interrupts by signalling a specific notification. Threads can then wait on or be bound to this notification in order to receive any interrupts that arrive. Once an interrupt has been handled, a thread can then use the IRQHandler to acknowledge the interrupt.

3.6 Libraries

Due to the minimality of many seL4 objects, various support libraries exist to assist in using them. One commonly used library provides a *Virtual Kernel Allocator* (VKA), which assists in allocating each kind of object from untyped memory, as well as providing generic interface functions for many seL4 invocations, abstracting away the underlying CSpace layout. These features allow for easier manipulation and management of seL4 objects. Other libraries include those assisting with VSpace management library, which help in the mapping of many pages and their backing page table objects, as well as platform support libraries to assist in starting timers and serial drivers.

4 Zircon

Zircon is a new microkernel currently under active development by Google. Zircon provides the base platform for the Fuchsia operating system; the many user-level libraries and applications of Fuchsia build upon Zircon to provide a fully-featured OS targeting modern phones and personal computers. Zircon specifically targets 64-bit x86 and ARM platforms.

Internally, the Zircon kernel builds upon constructs from littlekernel (LK), a real-time operating system targeting embedded devices with limited resources, and forms the basis for Android's bootloader, amongst other applications. LK provides resources such as timers and threads; these form the basis for the larger Zircon objects exposed by the syscall API. Additionally, the original LK constructs have morphed over time to meet the needs of Zircon objects; since Zircon targets platforms where resources such as RAM are far more plentiful compared to embedded devices, some constraints in LK may not apply to Zircon.

Zircon also adds features not found in LK, most notably memory protection features; while all code in LK is trusted and runs in privileged mode, Zircon implements the virtual memory protection mechanisms required for proper user-mode support, a feature essential for an operating system running potentially untrusted user applications.

Supplied with the Zircon codebase is a small set of user-level drivers, libraries and applications, which allows for a system to boot into user-level, initialise devices, and run user programs on the Zircon kernel. This allows for Zircon to be used and tested without having to run a large Fuchsia instance with many applications running.

4.1 Handles

Zircon's version of capabilities are known as *handles*. Like their seL4 counterparts, a handle denotes the ownership of a kernel object, and has certain rights associated with it. The first handle to an object is crafted at object creation time, with the maximum allowable rights for the given object. From here, the handle can be replaced with another handle with equal or lesser rights. Handles can also be duplicated in a similar fashion, providing that the handle has the right to do so.

Once a process no longer requires a handle, it can close the handle, which results in the handle's destruction. Once all handles to an object have been closed, the object is destroyed, although there are several exceptions to this rule. For example, a running thread will continue to execute even if all handles to it are closed; the underlying thread object will be destroyed once it has finished executing.

Handles are usually associated with an owning process. Process can only manipulate objects which they own handles to, and these handles must confer certain rights in order for the operation to succeed. The value of a handle can change depending on ownership, and the handle values do not translate in any meaningful way between processes; the handle value observed by one process would likely be invalid or map to a completely different object in another.

Handles can also be transferred between processes, primarily through channel objects (detailed in the following section). During transfer, the handle is stored in the kernel, and cannot be used by the process that previously owned it, nor any other process. Once the transfer is complete, the handle will once again have a single owning process.

4.2 IPC

Zircon provides three kinds of IPC abstractions: *Channels*, *Sockets*, and *FIFOs*. All three of these objects are bidirectional and two-ended; in the kernel, they are created as a pair that represents each end of the object. Each end of the object is referred to by different handles, and is associated with its peer for the lifetime of the object. A write to one end of the pair results in the delivery of the message to the other's queue or buffer. Thus if one end of the pair is destroyed, the other cannot perform any more write operation, although its remaining messages can still be read.

Like most other syscalls, operations on these IPC objects are generally non-blocking; read and write operations return instantly, and do not wait for the objects to be in any particular state. If a read or write operation cannot be performed at a certain time, the syscall will simply respond with a relevant error code.

Channels are the primary IPC construct used to share handles to objects between processes, providing the sole method of transferring an arbitrary number of handles to another process. Channels consist of a queue of message packets at each end of the channel, each containing a number of handles as well as a number of bytes of data. A write to the channel appends a new message packet to the queue, containing the data and handles supplied by the caller. Handles in a message do not have an owning process; if the channel is destroyed or the message discarded, the handles are closed by the kernel. A read to the channel takes the first message from the queue, writing its contents to buffers supplied by the caller and transferring ownership of the handles to the calling process. Channel operations are atomic in that there is no partial transfer of data or handles; in particular, if there is a handle which is not transferable, or if channel is supplied a handle to itself, the entire operation will fail, and no new message will be written. In addition to the regular read/write operations, channels also have a call operation. Unlike other IPC operations, this is a blocking syscall; the caller writes to the channel, waits for a response message, then reads the response before returning. Written messages are matched with replies through a transaction ID, which takes the place of the first four bytes of the message. This transaction ID is written by the kernel; the reader of these messages is responsible for adding the transaction ID to the response before sending it.

Sockets in Zircon are somewhat similar to their POSIX namesake in that they support transferring streams of data between processes. Sockets consist of a pair of variable sized buffers of data. Unlike channels, a socket's buffer does not support the transfer of handles, only data. However, they grant more flexibility than channels with reading and writing, allowing for short writes when the maximum buffer size is reached, and short reads if more data is requested than what remains in the buffer. Sockets can also be created to have various additional properties. They can be created as datagram sockets, in which data in written and read in packets as with channels. Sockets can also be created to have a control buffer, which is separate from the regular buffer and can hold a single message; processes can then exchange special messages without affecting the regular data in the socket. Another option for sockets is an accept queue, which can hold one socket handle at time. This allows for the sharing of new sockets over pre-existing socket connections. A socket cannot share itself or its peer in this way however.

FIFOs are far simpler than Channels and Sockets, and just consist of a pair of fixed-sized circular buffers. Unlike the other two, FIFOs are quite limited in size[1], and this size cannot be changed after the FIFOs creation. As with sockets, FIFOs do not support the transfer of handle, only data. However, these limitations allow FIFOs to be quite simplistic; they require no additional allocation of memory for reads and writes, making them suitable for quickly transferring small portions of data.

4.3 Memory Management

Zircon provides two abstractions for managing a virtual address space: *Virtual Memory Objects* and *Virtual Memory Address Regions.*

Virtual Memory Objects (VMOs) represent a contiguous region of virtual memory. VMOs can be read or written to directly with syscalls, but are more commonly mapped into an address space of a process. Pages of physical memory are allocated to a VMO as required, whether it be via a system call or an access to the VMO through an address space mapping (i.e. after a virtual memory fault within the bounds of the mapping). Pages can also be decommitted from a VMO by request, allowing for paging mechanisms to be implemented.

Virtual Memory Address Regions (VMARs) represent a contiguous region of an address space. Each process starts with a root VMAR which spans the entire user address space. A subregion of the VMAR can be then allocated to a child VMAR, potentially with different mapping rights. These child VMARs can further be divided into subregions as required. VMARs also accept the mapping of VMOs, allowing for sections of an address space to be backed by virtual memory. Child VMARs and mapped VMOs remain mapped within a VMAR until they are manually unmapped, or the parent VMAR itself

[1]The maximum size of a FIFO is 4096 bytes, while Channels and Sockets support messages several pages in size.

is destroyed. When a VMAR is destroyed, handles to it still remain valid; the VMAR will no longer accept mapping or other operations on it, and will be fully cleaned up once all handles to it are closed.

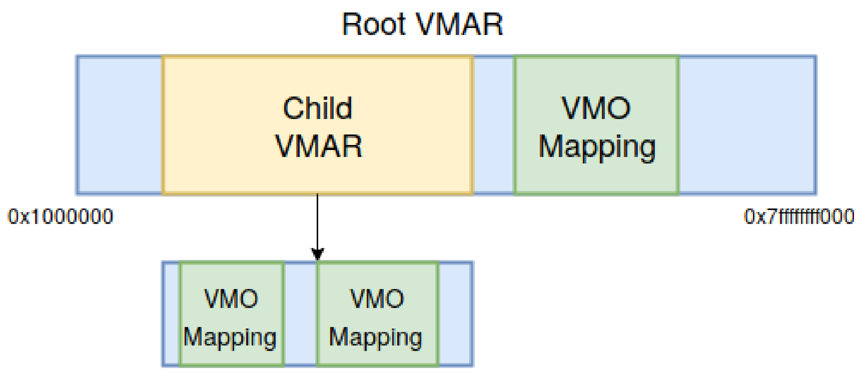

Figure 1: An example VMAR layout. The root VMAR can have child VMARs that hold their own VMO mappings.

The mapping of VMOs within a VMAR facilitates the use of shared memory in Zircon. Handles to a VMO can be shared between processes, which in turn can map the VMO in their address space. Thus the virtual memory represented by the VMO can be readily accessed by all processes.

4.4 Task Objects

Objects associated with the execution of a program in Zircon are referred to as Tasks. There are three kinds of task objects: *threads*, *processes* and *jobs*.

As with seL4 threads, a thread in Zircon represents execution context. At its core, a Zircon thread wraps an LK thread construct. However, Zircon threads differ in several ways. Native LK threads normally have static priorities and are scheduled round robin, but the Zircon kernel instead uses dynamic scheduling for threads, adjusting a thread's priority depending on its needs.

A Zircon process represents the instance of a program, and contains one or more threads, a root VMAR spanning the available virtual address space, and a set of handles owned by the process that dictate the kernel objects available to it. This provides the minimum set of resources required for running an application: binary code can be loaded into VMOs mapped in the VMARs of a process, and with at least one thread the process can begin executing code.

Jobs are an abstraction above processes, and represents a group of processes and possibly other child jobs. Just as a thread is owned by process, every process must have a parent job, which is dictated at process creation time. Jobs allow for many processes to be grouped as a single entity, such as instances of a multi-process application. Zircon allows for policies to be applied to jobs. For example, jobs can be denied the ability to

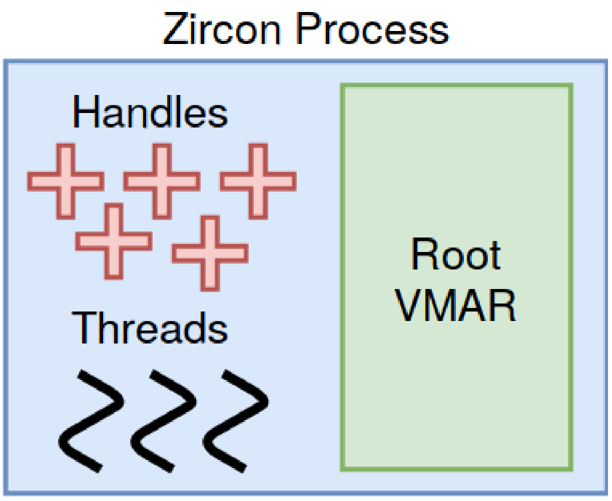

Figure 2: A Zircon Process contains a set of handles, a set of threads, and a root VMAR to represent the address space.

create certain kernel objects.

Since every process is required to have a parent job, the kernel creates a root job at boot time; the root job owns the first process to run in userspace. The boot process is supplied a handle to the root job by the kernel, allowing for new processes and jobs to be created.

4.5 Signalling and Waiting

In Zircon, the majority of kernel objects have a set of signals. Some of these signals are asserted by the kernel, while others can be set by the user. The kernel asserts and deasserts these signals when the state of an objects changes. For example, when an IPC object is written to, and it is no longer empty, the kernel will assert the signal representing that the channel is now readable. The user signals of an object can be set manually by processes, providing another form of communication between them. Paired objects, such as Channel and Sockets, can have each end of the pair signalled by the other.

Signals allow the many asynchronous operations in Zircon to be synchronous through the use of wait syscalls. A thread can wait on one or many objects for certain signals to become asserted. When a signal is asserted on an object, whether by the kernel or user, the list of waiters will be checked for a signal match. If an asserted signal matches a desired signal of a waiter, the waiting thread will be woken up. A deadline must also be specified for these syscalls; a waiting thread will wake up once the deadline has passed if no desired signal assertions occurred. Threads can also specify to wait forever if required.

There are many objects whose functionality is primarily associated with signalling. *Events* are objects purely intended for signalling by the user. *Event Pairs* are paired versions of regular event objects, and like other paired objects they allows for each end of the pair to be signalled by the other. Zircon also provides *Timer* objects, which allow for a deadline to be set; once this deadline has been reached a timeout signal will be asserted.

Ports are objects which have their own individual waiting mechanism, and have several uses. These objects contain a queue of port packets; these are small, fixed-size packets of data. When a packet is enqueued on a port, the first thread in the wait queue on the port is woken up, and the packet is delivered to the thread. Alternatively, if a port already has packets in its queue, and a thread attempts to wait on the port, it will instead be delivered the packet at the front of the queue.

There are several methods for enqueuing packets on a port. Packets can be manually enqueued by a process, but packets can also be delivered through certain kernel operations. One such operation is the asynchronous object wait. When a desired signal is asserted on an object, rather than having a thread block on the object, a packet is instead delivered to the specified port. This allows threads to do other work instead of waiting, while ensuring no signal assertions are missed.

4.6 Device Support

The syscall API for I/O and device-specific operations is still unstable, and some features currently included in the kernel are likely to change, or possibly be removed altogether (with functionality being moved to user level). This section will focus on the more static aspects of Zircon's device support at the kernel level.

For dealing with interrupts, Zircon provides interrupt objects. These objects can represent either a physical interrupt from the interrupt controller, or a virtual interrupt triggered by a user-level application. An interrupt object has a separate syscall for waiting, as well as methods for acknowledging received interrupts, and triggering virtual interrupts. Additionally, they can be bound to a port, which results in a packet being delivered to a port when an interrupt is received.

Other device-specific objects that have stayed relatively constant include variants of VMOs for managing contiguous sections of memory, or memory at a specific physical address. The cache attributes of this memory can also be changed so a device driver can ensure it has the correct variant of device memory.

Since obtaining access to hardware is a privileged operation, the Zircon kernel provides an additional object for managing the creation of device-specific objects. Resource objects are used to confer the ability to perform privileged syscalls and grant the ability to create such objects. A handle to a resource is supplied for validation of these privileged syscalls, which check that the resource is of the correct type and scope. For example, the creation of an interrupt object requires an interrupt resource which covers the desired interrupt vector. The user boot process is supplied a root resource by the kernel; this root resource can be used to validate any privileged operation. Finer-grained resources can be created from this root resource, ensuring processes only have access to the privileged resources they require.

4.7 System Calls

Zircon exposes its syscall API to processes through a *virtual dynamic shared object* (vDSO). This vDSO is a special ELF shared library which is loaded into the address of a processes. This is the only way in which processes have access to system calls; the Zircon kernel performs checks on syscall entry to ensure that the vDSO was used to perform the syscall. It does this by checking that the calling thread's program counter was within the bounds of the vDSO mapping when it performed the syscall.

A handle to the vDSO is supplied to the first user process by the kernel, where it is embedded at compile time. It is supplied to userspace in the form of a read-only VMO; this allows for it to be handled the same as other memory regions. When loading subsequent processes, the vDSO must be mapped into the address space of the process by its parent for it to have the ability to perform syscalls.

5 Related Work

5.1 Unix on Mach

Golub et al. provided an early example of an operating system being ported to run on a microkernel, as a demonstration that implementing Unix as an application was both possible and rational [GDFR90]. This was not the first time Unix had been implemented as an application program; previous examples include the virtualisation of Unix on IBM's CP/67 [PPTH72]. However, it was the first port to leverage constructs provided by a microkernel in its implementation; in this case, the Mach microkernel [ABB+86] was used.

The port was comprised of two main components: a Unix server and a Transparent Emulator Library. The Unix server was implemented as a Mach task, and provided the majority of Unix services. The server contained many user-level threads which handled user requests and performed internal routines that dealt with I/O. Mach IPC was used for communication with user applications as well as hardware. The Emulator Library contained implemented a Unix system call handler that would transform system calls into remote procedure calls to the Unix server. This library was loaded into each address space of each process, and was maintained during fork and execve operations.

This implementation of the operating system as a user-level server is a potential approach that can be taken for porting Zircon to seL4. Additionally, the Emulator Library is also relevant, as the native Zircon kernel provides access to system calls through a library loaded into the address space of a process. The approach of having the library use the host microkernel's IPC for communicating with the server would be beneficial for system calls with a Zircon server.

5.2 L4Linux

L4Linux is a port of the Linux operating system to the L4 microkernel that demonstrates performance close to that of native Linux [HHL+97]. Prior to L4Linux, Unix-like operating systems had only been ported to first-generation microkernels such as Mach, where they had suffered from poor performance [CBM+94]. This port was focused on minimising changes to the Linux kernel for running on L4, yet greatly outperformed other ports in which Linux had been tuned to the microkernel they had been ported to.

L4Linux uses a similar approach to Unix on Mach, employing a Linux server to provide the majority of the functionality of the operating system, while Linux user processes were contained in individual L4 tasks; they have a main thread for regular program execution, an additional thread for signal handling, and an emulation library for capturing page faults and sending them to the server for handling. Modified versions of the standard C library (static and dynamic) are supplied, which make use of L4 IPC for system calls; a user-level exception handler also exists to capture system calls from pre-compiled programs using a regular C library. The top and bottom-half interrupt handlers of Linux are implemented as threads that run at a higher priority to the main server thread, which matches the behaviour of the Linux kernel.

Compared to native Linux, L4Linux suffered an average performance penalty of 5–10%, while ports of Linux to Mach perform far worse. This demonstrates that a high performance port is possible, provided that the underlying microkernel is fast and utilised correctly.

5.3 Mungi

Mungi [HEV+98] is an implementation of a single-address-space operating system (SASOS) on the L4 microkernel. These operating systems forgo the traditional approach of processes having separate address spaces, instead using a single large address space for all processes, made possible with the advent of 64-bit architectures. The SASOS model provides the benefit of easy data sharing between processes, a task that is often difficult and expensive for traditional multi-address-space systems. Mungi is a pure SASOS; it relies on shared memory of inter-process communication, and devices are mapped into virtual memory and dealt with by user-level processes. Security between processes is enforced though protection domains, which refers to a set of objects a process has access to. Objects refer to a contiguous range of pages, somewhat similar to Zircon's concept of a virtual memory object. A protection domain is defined by a set of capabilities owned by each process. Processes can access objects in other protection domains through the use of protected procedure calls, where the callee extends a subset of its protection domain to the caller for the duration of the call.

Mungi demonstrates the versatility of microkernel such as L4; the flexible address space model provided by the microkernel allows for SASOS and other non-traditional operating system models to be implemented. Aside from the similarity to many Zircon objects, the single-address-space approach of Mungi has useful implications for implementing Zircon's IPC; VMOs of user processes can all be mapped into a single address space and thus be easily copied to the "in-kernel" buffers in the same address space.

5.4 Xen

Xen [BDF+03] is a virtual machine monitor (VMM) for the x86 architecture. Traditional VMMs aim for full virtualisation, in which unmodified operating systems can be run. However, the x86 architecture was not originally designed with virtualisation in mind, and thus certain aspects are difficult to virtualise, and can incur large performance costs. Xen instead takes a *paravirtualisation* approach, in which a virtual machine abstraction is presented to guest systems, to which existing operating systems can be ported to with minimal effort. Operations that normally require certain privileged instructions are replaced with hypercalls to the underlying hypervisor. Xen's approach is distinct from other ports, in that it focused on providing the platform for running a variety of systems, such that the porting effort required is minimal.

Paravirtualisation provides another approach for porting Zircon to seL4, in which the existing Zircon kernel is modified to interface with seL4 rather than directly with hardware.

5.5 Wombat

Wombat [LVSH05] is another port of Linux to L4 that specifically targets embedded systems. Wombat is designed to be readily portable between architecture, with the ability to run on x86, ARM and MIPS architectures, and is intended to be run in tandem with trusted L4 applications and drivers with real-time requirements. Wombat runs as a user-level Linux server above the Iguana resource manager, which builds upon the underlying L4 construct to enforce isolation between trusted application and the Wombat server (and other Linux applications). Linux processes that use the Wombat server can also be modified to run in "native mode", which allows them to also use Iguana services.

Wombat demonstrates the possibilities available to a port of Zircon to seL4. Zircon processes that leverage the security guarantees of seL4 can be isolated from other applications, and there is the potential for some Zircon applications to be modified so they can interact with non-Zircon applications running alongside them. This approach could also be used to re-implement Zircon device drivers (and avoid having to implement an unstable system call API).

5.6 Exokernels

Traditional kernels provide various abstractions over hardware that are designed to hide the underlying details of the hardware and provide a simpler interface for developers. However, these abstractions can be viewed as enforcing restrictions on hardware for user applications, and many applications suffer in performance as they are forced to use hardware in a sub-optimal manner. Exokernels [EK$^+$95] provide an alternative approach to the management of hardware resources, in the which kernel only enforces protection on hardware resources, and applications are in charge of how they manage the resources they are given. Applications running on exokernels are referred to as *library operating systems* because of this; the various operating systems abstractions required for managing hardware are included as part of the user applications. As these library OSes have full control over the management of hardware resources, they can choose abstractions tailored to their application and thus greatly improve their performance.

While seL4 is not an exokernel, its abstractions over hardware resources are minimal enough for a library operating system approach to be effective. Useful Zircon applications could thus be potentially modified to include kernel functionality within the application, and run directly on seL4.

5.7 Unikernels

Unikernels [MMR$^+$13] build upon the library operating system concept used by exokernels by instead developing them for standard hypervisor platforms such as Xen. They are highly specialised library OSes designed for specific purpose applications, and are compiled into a standalone kernel that can be deployed on a hypervisor. They make use of a single address space, and code that would traditionally be part of the kernel now runs at user-level, largely eliminating the overhead that would be contributed by context-switching and kernel entry. Additionally, unikernels only need to be built with the libraries that they require; this allows for far smaller binaries in comparison to a full-featured kernel image. Unikernels provide further evidence of the viability of deploying Zircon as a library operating system, and demonstrate the flexibility that could be provided by porting Zircon with this approach.

5.8 API Emulation

API emulation provides another method of running non-native applications on a host operating system. Popular examples include Wine [Aut17], which allows Windows binaries to be run on Unix-like systems, as well as Cygwin [S$^+$17], which provides a POSIX API for Windows applications. Both these examples use shared libraries to provide a compatibility layer to applications; these libraries translate foreign systems calls from the applications into calls understood by the host system. Additionally, functionally that may not be easily translated to host system calls and constructs can be provided by server applications that foreign applications can communicate with.

The use of a shared library to provide Zircon's API would integrate well with existing Zircon applications, due to the pre-existing use of a shared library to provide access to system calls. This library could simply be modified to translate Zircon system calls to those provided by seL4.

5.9 QNX

QNX [Hil92] is an operating system that provides the functionality of the POSIX API atop of a real-time microkernel. The microkernel itself only provides four main services: inter-process communication, process-scheduling, interrupt handling and low-level network communications (allowing for a distributed system of microkernels). The POSIX functionality is provided through resource manager processes running on the system. The process manager is the primary resource manager, and provides necessary services such as process creation and memory management. From here, additional resource managers can be run; these include the filesystem manager and the device manager, which provide support important POSIX routines on files (i.e. opening a file). POSIX library calls from applications are implemented as messages delivered to resource managers using the microkernel's IPC.

The use of user-level processes to provide the functionality of an API is an appealing option for porting Zircon. Abstractions that map poorly to existing seL4 constructs could instead call a user-level server, which would handle functionality such as IPC with channels or sockets, enforcing the properties of these objects that Zircon applications expect.

6 Approach

The aim of this thesis is to provide the functionality of the Zircon kernel while running atop of seL4. This requires the Zircon kernel to be virtualised or re-implemented to run at user-level, while making use of seL4 primitives where necessary. From here, we wish to be able to run various Zircon user-level systems, and potentially have them interact with native applications.

6.1 Design Requirements

Our design requirements for the implementation are as follows:

- **Low overhead**: We want to be able to run a user-level system with minimal impact on performance. If low overhead cannot be achieved, then there is little reason to use Zircon on seL4 over existing approaches.
- **Low porting effort**: porting the Zircon kernel should not require an extreme amount of effort, and we want to avoid having to extensively change a user-level system so it can run on seL4.
- **A dynamic system**: A Zircon user-level system has the potential to vary in its workloads and demands from the kernel, and our design should be able to handle such systems, and be able to run arbitrary Zircon applications.

6.2 Issues

There are several issues that may hinder development of a Zircon port:

- **Instability of API**: Zircon is still in development, and is a very immature project in itself. As such, we need to deal with the possibility that the API may change, and the implementation must be flexible enough to deal with this.
- **Differences in primitives**: while some Zircon primitives align nicely with their seL4 counterparts, others do not; we must find appropriate methods of handling these objects. For example, seL4 IPC is synchronous and based around small messages, while Zircon IPC is asynchronous and supports large messages.
- **Lack of documentation**: the documentation for Zircon is very limited, and figuring out its inner working primarily involves looking at the code itself.

6.3 Potential Approaches

From the related work provided in Section 5, we can narrow down the potential approaches to porting the Zircon kernel to seL4. These include paravirtualisation, API emulation, and a library OS or unikernel approach.

Paravirtualisation of the Zircon kernel is an attractive option. Porting the kernel to run on seL4 this way allows for the most code re-use of all the approaches, as much of the work is focused at the architectural level. For Zircon this could be particularly useful, as while its API is still somewhat unstable, much of the code that interacts with hardware has stayed reasonably similar to that of littlekernel. However, a paravirtualisation approach lacks integration with seL4; higher level operations in the Zircon kernel could instead be handled by seL4 itself, and without incurring the overhead of virtualisation.

API emulation requires more work than paravirtualisation, since all objects and syscalls of Zircon would need to be re-implemented to handle seL4 primitives. A user-level server would be required for handling many Zircon syscalls, due to Zircon's IPC and other operations requiring the storage of messages or other data in the kernel. However, this approach allows for greater integration with seL4; the internals of Zircon objects can be redesigned to better fit the primitives of seL4. This allows for greater leveraging of the security properties of seL4 objects, and allows for syscall operations to be fine-tuned to better match these objects, which can improve performance.

A library OS approach would result in an extensive redesign of Zircon objects, with all processes sharing a single address space. Various other objects could be modified for better performance, such as the use of user-level threads for implementing Zircon threads. As such, a library OS can provide the best overall performance of all three approaches. However, the changes required for a library OS would break the specifications of the API, and thus many user applications would need to be modified to handle this. As such, a library OS approach has the highest porting effort, but also cannot support dynamic systems as effectively as the other approaches.

6.4 Selected Design

For this thesis, an API emulation approached was selected to be used. This involves the implementation of a seL4 application to act as a Zircon API server. This server is responsible for handling syscalls from Zircon applications, and contains its own representations of Zircon kernel objects, which make use of seL4 objects where necessary.

This approach was selected as it provides a good balance for the many design requirements for the projects. With an API server, we have flexibility with our implementation; the server can implement Zircon objects in whatever way is desired, as long as the same API as native Zircon is provided. We can thus find an optimal tradeoff between porting effort and performance. Objects and syscall handlers implemented on the API server can be based upon their counterparts on native Zircon, but can be fine-tuned where appropriate to best match the underlying seL4 objects. We can potentially reduce the overall overhead of the API server as a result. The API emulation approaches also allows for tighter coupling of Zircon and seL4 objects. While an approach like paravirtualisation requires an adaption to code in the Zircon kernel, we can build the API server around the security properties and fast IPC provided by seL4.

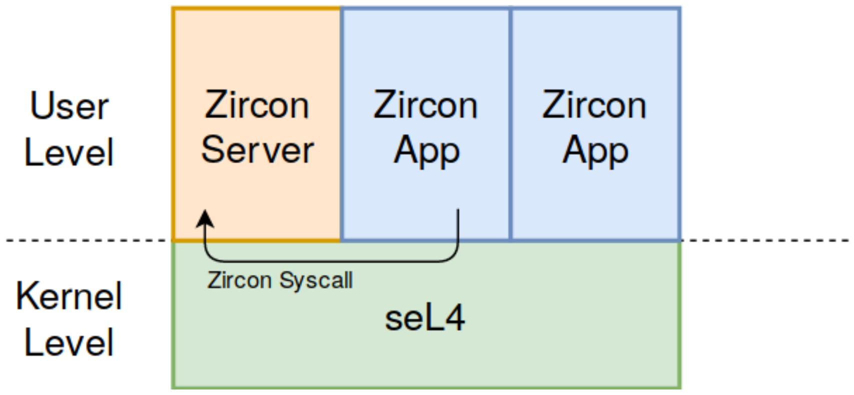

Figure 3: The Zircon server runs as an application at user-level, and receives syscalls with seL4 IPC.

7 Implementation

This section introduces the implementation of Zircon on seL4. This implementation is in the form of an API server, which is implemented as a standard seL4 application capable of managing its own Zircon kernel objects and handling Zircon syscalls. We also implement a modified syscall library which uses seL4 IPC to perform calls to the API server. We break down this implementation, detailing the setup of the API server, the method in which it handles syscalls, as well as how the many Zircon objects build upon seL4 primitives.

The Zircon API server is currently implemented for 64-bit x86, and is based on the version of Zircon described in the background section (circa March 2018). To assist in the development of the Zircon server, we relax some aspects of a Zircon system:

- **Static syscall library**: building a shared library to meet the requirements of Zircon's vDSO is not trivial, so the syscall library is instead built as a static library and linked with Zircon applications at compile-time.
- **No device objects**: due to the unstable nature of device support in the Zircon kernel during development, it was decided that this would need to be delayed until the API becomes more stable.
- **Single-threaded and single-core**: while having many threads run on many cores is beneficial for performance, concurrency issues are difficult to deal with, and multithreading is not explicitly required for the server to function.

Details on how these can be implemented in the future is discussed in the Future Work section.

7.1 Zircon API Server

The Zircon API server is based upon the standard model of a process in seL4, consisting of a thread and its associated CSpace and VSpace. The address space of the server is split into various sections:

- The server's code section and stack.
- Various object allocators. These are pool allocators initialised at boot time used for objects that require fast allocation and lookup,
- A page allocator. At startup, the server allocates and maps a number of page frames on the server. These page regions are allocated to certain objects with high and variadic memory requirements, specifically VMOs, Channels and Sockets.

The rest of the server's address space is used for mapping VMOs into the server. The reason these mappings are required is discussed in Section 7.4.

The server uses a VKA for allocating the many seL4 objects used by itself and other Zircon objects, including endpoints and page frames. Capabilities on the server are stored in a two-level CSpace; this allows the CSpace to grow dynamically as more objects and caps are created.

The server runs at a higher priority than other Zircon threads. This ensures that the server will always be scheduled before them, allowing for the timely handling of syscalls, faults and timer interrupts.

7.2 System Call Interface

As the Zircon kernel objects are implemented in a process at user-level, seL4's IPC mechanisms must be used for performing syscalls. The Zircon API server makes use of an endpoint, on which it waits to receive incoming syscalls from Zircon applications. Every thread has access to a capability which provides the right to communicate with the server. The API which the threads use for performing the syscalls remains the same, but the underlying implementations uses a seL4 Call routine for delivering syscalls to the server. Once the server has handled the syscall, it can then reply to a calling thread with the appropriate return values.

The majority of these API calls are performed in the same way; a message is created containing the relevant syscall number and syscall arguments before the call is performed. As with native Zircon, these syscall definitions are auto-generated; Zircon on seL4 makes use of a syscall generator script which parses a syscall definition file in the Zircon source code and generates the functions in the syscall library. The script also generates output for the server, most notably the syscall table, which is used to lookup syscall handlers by their syscall number; the server checks that the provided syscall number is valid before doing so. The syscall table is set up in such a way that all defined syscalls can be handled; syscalls that are not implemented by the server use a generic unimplemented handler which responds to the caller with the relevant error code. This ensures the syscall API provided by Zircon on seL4 matches that of native Zircon, even if a syscall is not actually implemented.

An endpoint cap provided to a thread is always badged. The value of the badge is unique per thread, and can thus be used by the server to identify a calling thread. This is needed as the majority of syscalls need to identify a thread and its owning process, not just for calls specific to these objects but also confirming any supplied handle value maps to a handle owned by the process. This design is backed by the property that a badged endpoint cap cannot be modified or copied, which ensures a thread cannot modify an endpoint cap and pose as another thread; this ensures badges can be used to reliably identify a thread.

The server performs other security checks to ensure a syscall has been performed correctly. Aside from checking the syscall number and badge, the server also checks that the number of arguments in a syscall message has been set correctly for the selected syscall; the syscall will be rejected by the server if this is not the case.

7.3 Zircon Objects and Handles

Objects on native Zircon are defined as C++ classes, and Zircon on seL4 implements its objects similarly. All objects make use of a base object class, which implements methods common to all objects, and helps facilitate the generic handling of objects. Zircon objects then inherit from this base class with their specific implementations, overriding certain methods from the base object class if their specific implementation requires it.

Handles are also implemented as a C++ class, and all handles are stored in a handle table on the server. The handle table is a pool allocator, and provides the allocation and freeing of handles in constant time. This is required, as handles are manipulated frequently, and their representation at user-level must quickly be translated to the underlying handle structure stored on the server.

As with native Zircon, every handle is given a base value, which can be used to lookup the handle's location in the handle table. This base value differs from the handle value seen by processes; this user-level value is calculated by XOR-ing the base value with a handle mask unique to each process. To ensure a handle value supplied by a process matches a handle on the server, the server performs the reverse computation on the handle value to get a base value, and uses the base value to get a handle from the handle table. Handles keep a reference to their owning process, which allows the server to check this owner matches the calling process.

Objects maintain a reference count of all handles that refer to them. Every time a handle is closed, the object is checked if the handle count has reached zero, and thus can be destroyed, in which case a generic object destroyer function is called. This calls the object's destructor before freeing the memory back to the appropriate allocator, whether this be the heap or an object-specific allocator (i.e. the thread table). Objects with additional requirements that must be satisfied have additional checks for destruction in the appropriate locations.

7.4 Memory Management

Zircon on seL4's VMO implementation consists of two constructs: the base VMO object and a representation of each VMO mapping.

VMO objects are patterned the same way as other objects, having a set of handles which refer to it. Each VMO is allocated a portion of the server address space, where all the page frames of the VMO can be mapped. These server mappings are required for facilitating the direct read and write syscalls to VMOs and for translating pointer arguments supplied in syscalls. Internally, the VMO object consists of two main parts: a two-level page table structure for storing seL4 frame objects allocated by the VKA, and a list of VMO mapping objects. We use a page-table for the storage of frame objects, as VMO can potentially be large and sparse, in which case a simple array is unacceptable. The second level of this page table uses pages from the server's page allocator so that large heap allocations can be avoided.

VMOs that are mapped into a VMAR are represented by a VMO mapping structure. VMO mappings contain information about the mapping, a reference to the VMAR in which they are mapped in as well as a page table structure for storing page frame capabilities, which are copied from pages stored in the base VMO. The page table operates similarly to that of the base VMO, with the second level also using pages from the page allocator. VMO mappings also keep track of the rights that are applied to its underlying page mappings.

Page frame objects are allocated to VMOs and their mappings on-demand. When a page is to be committed to a VMO, a page frame object is first allocated from the VKA, and stored in the page table of the VMO; the page is then mapped into the server mapping region at the appropriate address. When a page needs to be committed to a mapping of the VMO, a copy of the original page frame cap is created, which is stored in the page table of the mapping, and the page is mapped into the address space wherein the mapping is located. This is only done for the requested VMO; further cap copies will be made as other mappings request them.

VMARs primarily consist of an ordered vector of VM Regions, which are the base class for all VMARs and VMO mappings. This design allows for these objects to be stored in within a parent VMAR in a generic fashion. VM Regions are ordered by their base virtual addresses, allowing for a region to be quickly located using binary search with a given virtual address that lies within the parent VMAR.

VMARs are frequently used by the server for performing translations of user virtual addresses to server addresses. This is required for many Zircon syscalls, whether it be for reading or writing data to a buffer, or returning newly created handles to objects. Translation starts at the root VMAR of the process whose address space we wish to access. A binary search is performed on the vector of VM Regions that lie in the VMAR for the subregion that contains the user address. If a subregion is a VMO mapping, we can fetch the base VMO and calculate the server address. If the subregion is a child VMAR, we repeat the search on this VMAR, and continue to search child VMARs until a VMO mapping is found. If no VMO mapping is found, then we know that the supplied user address is invalid.

No seL4 page table or page directory objects are stored in VMARs; instead, they are stored within process objects. This is because VMARs represent the logical partitioning of an address space, and are sized at a page granularity; there is no alignment with page table objects, nor any alignment of the depth of a VMAR tree with the levels of page tables. The manner in which page table objects are stored in a process is detailed in the following section.

7.5 Threads, Processes and Jobs

As native Zircon threads are kernel threads, we take the same approach on seL4; every Zircon thread is mapped one-to-one with a seL4 thread, and thus a Zircon thread object contains a seL4 TCB object for managing this thread. Each thread is also associated with its own CSpace for storing capabilities. This CSpace is kept small[2], and contains two caps to the server endpoint: one for performing syscalls, and the other to use as the thread's fault endpoint cap. These caps are badged so that the thread can be identified by the server when it calls or faults. Endpoint caps are also created without read rights; this prevents a thread from receiving messages on an endpoint (but not replies from the server), which prevents a thread intercepting syscalls directed at the server.

As mentioned earlier, thread objects are allocated from a thread table. The primary reason for this is for identification by badge values; the index of the thread in the thread table forms the base value of the badge for each endpoint cap. The badges for each cap also have different bits set to distinguish the syscall cap from the fault cap; this allows the server to easily recognise what is being requested by a thread, and is the reason why threads do not just use the one cap for both syscalls and faults.

Separate CSpaces are required per thread rather than per process for several reasons. If we had a CSpace shared between processes, and separate caps for each thread, there is no way of preventing a thread from using the incorrect endpoint cap, and thus disguise itself as another thread. This could be prevented by having all threads share endpoint caps, but then it becomes difficult to distinguish calling threads from one another. Having a CSpace per thread allows for straightforward identification of threads; we can also use the same cap slots for each each thread, which makes CSpace manipulation far simpler.

As syscalls can have up to eight arguments, an IPC buffer is required for threads to perform syscalls. Additionally, many threads can perform syscalls at the same time, thus each thread requires a separate IPC buffer. When a thread is assigned to its parent process, the process allocates an address to each thread, which is used as the location of the thread's IPC buffer. These addresses are located in a region of memory below that of the first valid address in a root VMAR, preventing any risk of a VMO mapping clobbering any IPC buffer.

Threads are also allocated a cap slot in the server's CSpace, which is used by the server for saving reply caps. This is needed for blocking syscalls, such as wait or sleep calls. In this case, the calling thread will wait for a response; once the syscall is complete,

[2]Each thread's CSpace consists of a single root CNode with 8 slots.

the server finally replies to the thread by sending a response with the saved reply cap.

Similarly to native Zircon, process objects contain a list of thread objects and handles, as well as a root VMAR for address space management. Zircon processes on seL4 also store seL4 page table objects. The top level page directory is required for seL4 mapping invocations, and is contained in the process for easy access, while the lower level page tables are only allocated as required for page mappings, and just need to be stored in a linked list owned by the process.

Each thread and handle in their respective lists maintain a reference back to the owning process. For handles, this is required for confirming the ownership of the handle, while for threads, we frequently use this for getting a process after a calling thread has been identified, which is needed by many syscalls for querying the handles owned by a process, and for getting access to its root VMAR for address translations.

Each process also maintains a bitfield which is used for the allocation of IPC buffer addresses for threads. A new thread added to a process is supplied the first free bit in the bitfield; the location of this bit in the bitfield is used to calculate the IPC buffer address.

Jobs in Zircon on seL4 simply contain a list of child processes and a list of child jobs. As job policies are not essential to the functionality of a Zircon system, these were not considered a priority and thus are not implemented at this point in time. As with native Zircon, the zircon server creates a root job which contains the first user process; a handle to the root job is passed to the process through an initial boot channel.

7.6 IPC

At user-level, the two ends of an IPC object are often referred to as a single entity, but in both native Zircon and Zircon on seL4, these IPC objects are implemented internally as a pair of objects. The following section will thus refer to them as separate objects, with the object's other end known as its peer. This is consistent with the terminology used in the Zircon kernel. All IPC objects maintain a reference to their peer, allowing them to perform writes to their peer's buffer or queue.

Channel objects maintain a list of messages, with describe the number of handles and bytes contained in each message in the queue. These messages do not actually store the contents of the messages; these are instead stored in a handle list and a variable size buffer for the storage of data, known as a message buffer. Since channel messages are stored in a queue, we simply need to operate these constructs in a FIFO fashion to ensure messages are delivered in the correct order. When a message is written to the channel, we first check that the supplied handles are valid and permitted to be added to the channel. Then the handles are taken from the process and pushed to the back of the handle list, before writing the data to the message buffer and appending a new message to the message list. Reading involves removing the required handles from the front of the list, and reading from the front of the message buffer, as described by the first message taken from the front of the message list.

Like channels, sockets also make use of a message buffer for storing data; this is the only structure required for a basic stream socket. Datagram sockets operate similarly to channel in that they maintain a list of datagrams that describe the contents of the message buffer. For sockets with a control plane, a small fixed size buffer is allocated from the heap, while sockets capable of sharing can store a single socket handle.

The message buffer used by channels and sockets consists of a linked list of pages. These pages, like with VMO page tables, are allocated from the server's page allocator. This allows for the fast allocation of memory for large reads and writes, a desirable trait for these IPC objects. The pages themselves consist of a header maintaining the linked list; the rest of the available memory in the page holds data. As a message buffer is written to, pages are appended to the end of the linked list; for reads, once all data has been read from the page at the head of the list, it is taken from the list and freed.

As FIFOs have a small, fixed size data buffer, they do not need to use a message buffer. They instead simply allocate their buffer from the heap, which is done at FIFO creation time.

7.7 Signalling and Waiting

The Zircon server implements a timer for handling syscalls with deadline. This is achieved by using the HPET on x86. The server timer operates in a tickless manner, with the timer only being configured to fire on the next upcoming deadline. Timer interrupts are delivered to a notification object, which is bound to the server's TCB. This allows the server thread to be notified of timer interrupts while it is waiting on the server endpoint for syscalls.

The timer implements a timer queue, with nodes sorted by deadline. Each thread object has a timer node which can be added to this queue when the thread must block for a period of time. Once a deadline has passed, the timer node will be taken from the front of the queue, and the callback stored in the timer node is called. All callbacks result in the thread being replied to with the appropriate return arguments.

For handling the object wait syscalls, a thread object maintains a reference to a StateWaiter object (in the case of waiting on many objects, an array of StateWaiters is used). StateWaiters maintain the set of signals that are being observed on an objects. Objects that can be waited on maintain a list of these StateWaiters; StateWaiters are appended to this list when a thread starts to wait on the object. If an object's set of signals change, whether this be from the kernel or a user process, the set of StateWaiters is checked for a signal match. If a desired signal has become asserted on the object, the StateWaiter is removed from the list, and the thread is woken up.

There are other instances where a StateWaiter may need to be removed from an object. These include an expiry of the wait deadline, and handle invalidation. An object wait syscall requires a handle to the object to be given; if this handle is closed, the wait must be cancelled. StateWaiters thus also track the handle used to initiate the wait. When a handle is closed, the list of StateWaiters on the object is checked for a handle match. If a StateWaiter has a matching handle, it is removed from the list and its associated thread is woken up.

7.8 Fault Handling

Currently, the Zircon server implements basic fault handling. The only kind of recoverable fault is limited to a virtual memory fault. In this case, the address responsible for the VM fault is checked for validity; that is, the address lies within a VMO mapping contained in the address space of the faulting thread. If this is the case, a page is committed to the VMO mapping at the relevant location, and the thread is restarted. Any other fault triggered by a thread results in the thread being terminated, and destroyed if necessary.

Native Zircon has additional aspects to its fault handling. This is discussed in Section 9.

7.9 seL4 IPC on Zircon: Endpoints

As one of the aims of this thesis is to provide methods for Zircon application to interact with native seL4 ones, Zircon on seL4 progresses towards this by providing the ability for Zircon processes that are aware they are running on seL4 to use endpoints for communication. To achieve this, we define an additional set of syscalls that are used to manage a Zircon object wrapper for seL4 endpoints.

Creation of an endpoint object on the Zircon server requires access to a relevant resource, which is just the root resource for now. This is because endpoints are considered to be a privileged resource, and only trusted processes should be able to create them. As with other Zircon objects, the creation of an endpoint returns a handle to the endpoint, which can be duplicated, closed or passed in a channel as with other handles.

Once an endpoint object has been created on the server, processes with a handle to the endpoint can then request for caps to the be minted into one of the free slots of a thread's CSpace. Caps can also be deleted from a thread's CSpace once they are not required anymore. The management of the free slots in a thread's CSpace is left to the user; aside from checks to ensure that the server endpoint caps and other reserved slots are not modified, a process is free to mint caps in whichever free slot they choose. Any failure to mint or delete caps, such as attempting to mint a cap to an occupied slot, will be returned as an error by the server. Once a cap has been minted to a thread's CSpace, the thread is free to use the endpoint for IPC as it pleases. When all handles to an endpoint object are closed, all caps to the endpoint are revoked and the endpoint object is destroyed.

Providing endpoints to Zircon applications is one of the first steps towards providing communication methods between Zircon and native seL4 applications. Further work that can build upon this is discussed in Section 9.

7.10 Other Zircon Objects

There are various other Zircon objects that have been implemented on the server. Timer objects use a timer node for setting deadline, just as threads do for blocking syscalls. When the deadline expires, the callback given to the timer node sets the timeout signal. Event objects are even simpler, as they are largely just a wrapper class around the base constructs used in waitable objects, which provide all the mechanisms necessary for the signalling and waiting performed on Event objects. Event pairs are very similar to event objects, but as with IPC objects they consist of two linked objects. Each object in the event pair keeps a reference to the other peer so it can be signalled.

Other objects not previously discussed in this section were not implemented due to time constraints. The future approach for the implementation of these objects is discussed in Section 9.

8 Evaluation

In Section 6, we defined the design requirements of Zircon on seL4: low overhead, low porting effort and the ability to run dynamic systems. In this Section, we evaluate the server's ability to uphold these requirements. This includes various benchmarks performed on both Zircon on seL4 and native Zircon, as well as an analysis of the porting effort required.

8.1 Experimental Setup

In this evaluation of Zircon on seL4, we compare the performance of the Zircon server to that of the native Zircon kernel. There are several requirements that need to be adhered to in order to ensure a fair evaluation of both systems.

The first of these is the hardware used. Both systems were tested on the same machine; this was a x86-64 i7-6700 Skylake processor clocked at 3.4GHz, with 16 GB of RAM. Both the seL4 and Zircon kernel were configured to run on a single core, as the Zircon server only supports running threads on a single core.

To avoid any overhead attributed to other processes running in the background during benchmarking, we ran the benchmarks as early as possible in the boot process. For Zircon on seL4, we use the very first process to run as the process running the benchmarks, while on native Zircon, the benchmarking process is the next to be run after the initial boot process. This is acceptable and does not impact performance, as this initial boot process wait forever on the benchmarking process until it terminates, which only occurs once benchmarking has completed. Thus the boot process will never be scheduled to run during benchmarking and will have no impact on performance.

For measuring the time taken required to perform each test, the x86 Time Stamp Counter (TSC) is used on both platforms. This is facilitated by the syscall API of Zircon, which implements a user-level function for fetching the value of the TSC. The Zircon server implements this call in the same way as native Zircon does, ensuring fairness in checking the TSC. The overhead of checking the TSC is also accounted for in all benchmarks.

As many of the operations being tested only require hundreds of cycles to complete, the time measurements for each operation is taken over multiple iterations to minimise errors and improve reliability. All operations are performed 20 times between reads of the TSC. A number of warmup iterations are also performed prior to the actual measurements being taken, and in total, 10000 measurements are taken for each test.

One other constraint applied to the benchmarking of each system is ensuring that only the performance of Zircon syscalls is being compared. This means that if user-level functions are to be used, they must rely on the same implementation, otherwise misleading results may be produced. This places restrictions on what can be benchmarked, as even the C libraries of each system is different. While both Zircon on seL4 and native Zircon

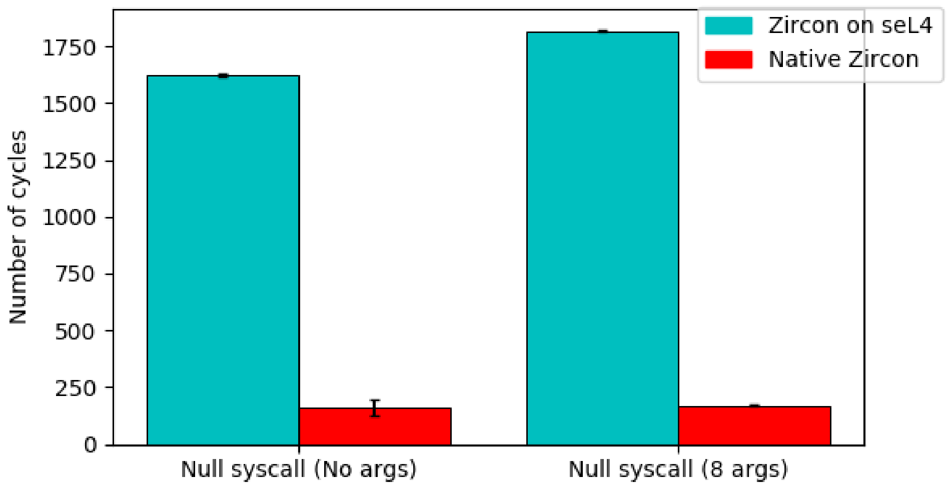

Figure 4: Null syscall cost for Zircon on seL4 and Native Zircon.

use a musl-based C library, both have distinct differences in certain areas of each library. As such, we must avoid C and other library calls while measurements are being taken, which limits the ability to compare larger workloads on each system.

8.2 Performance Evaluation

Various tests have been conducted to gauge the performance of Zircon on seL4 relative to that of the native Zircon kernel. We analyse the results of each test, explaining the differences in implementation of the two platforms, and how these differences may effect the performance of each. Potential areas in which the Zircon server can be improved are also discussed where better performance is required. All figures used to show the results of each test have error bars indicating standard deviation.

8.2.1 Syscall Entry and Exit

The first test conducted on Zircon on seL4 compares its syscall entry and exit times with that of Native Zircon. This gives an idea of the overhead contributed by performing any syscall, or in the case of the Zircon server, the overhead of performing IPC.

To perform this benchmark, two test syscalls provided in the Zircon API are used; one which accepts no arguments and returns zero, the other accepting eight arguments and returning the sum of all arguments. The second test syscall was selected to demonstrate any additional overhead incurred by Zircon on seL4 when accessing the IPC buffer to set or get syscall arguments.

Figure 4 demonstrates that syscalls handled by the Zircon server incur a far greater overhead than with the native Zircon kernel. This is understandable, as the native kernel does very little work aside from entering and exiting the kernel, while two syscalls are required for IPC with the server (call and reply), both of which require actual work to be done in the kernel, as well as a context switch to and from the server. Considering the amount of additional work being done, the overhead seen here is very reasonable, and is possible due to the low cost IPC provided by seL4.

An investigation was performed to see if the cost of a null syscall could be optimised further. In order to keep the cost of seL4 IPC low, we aim to hit the *fastpath* which is an optimised section of the kernel aimed at speeding up common operations. Certain requirements must be met in order for the fastpath to be used, and it was discovered that the Zircon server was not using the fastpath when replying to a calling thread.

The reason for this is due to the version of seL4 used[3]. The kernel used by the server only hits the fastpath if it replying to a thread with equal or higher priority. This is not the case for the Zircon server, as Zircon threads are assigned a lower priority. This was confirmed when the threads were given an equal priority to the server, which resulted in a reduction in the null syscall cost by approximately 250 cycles. However, such a change cannot be made permanent, as we want the Zircon server to preempt other threads to ensure the timely handling of syscalls, faults and timer interrupts. Instead, an upgrade to a newer kernel is all that is required; in these newer versions a reply to a lower priority thread hits the fastpath as long as there is no other thread with a higher priority (excluding the replying thread). As all Zircon threads share the same priority, the fastpath would be used in this case.

In many of the following benchmarks, we account for the cost of a null syscall with zero arguments in the overall cost. This is done to give a better idea of the additional work being done on the server to handle each syscall. We do the same with native Zircon so a fair comparison can be made.

8.2.2 Handle and Object Operations

Figure 5 shows a performance comparison of common object and handle operations. For all these operations, the Zircon server has roughly comparable performance to native Zircon when the overhead of performing a syscall is ignored. However, there is a clear additional overhead for the duplicate and close operations on handles.

For both these tests, we duplicate and close the same number of handles as the iteration count for each run, while for the replace test the old handle value is replaced with a new one; there is no increase in the number of handles allocated. Decreasing the iteration count reduced the overhead of the duplicate and close operations (this was not retained, as the overall results had greater variance and were less reliable). As such, this overhead appears to be contributed by allocation and freeing of handles.

The reason for this overhead is likely due to differences in the memory arrangements

[3]Version 7.0.0 of seL4 is used for the current implementation of Zircon on seL4.

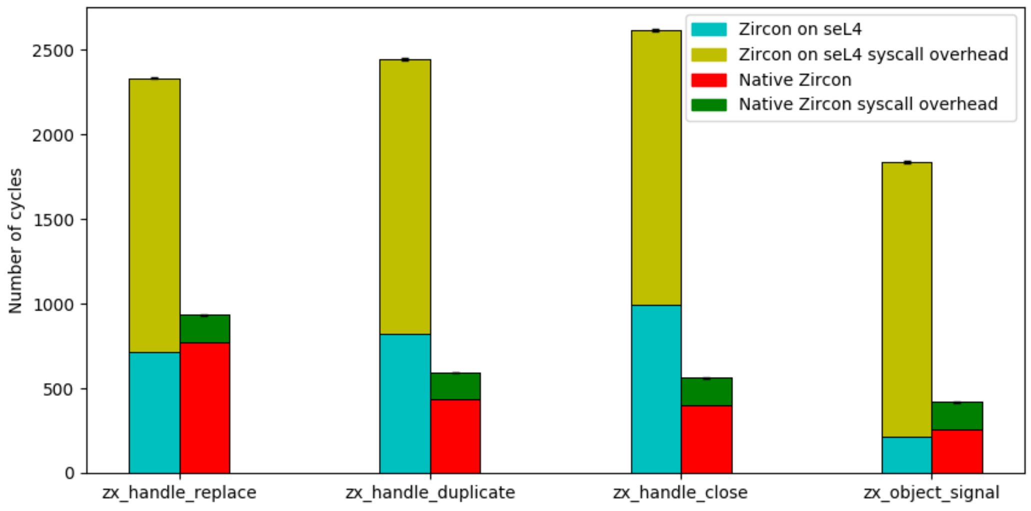

Figure 5: Cost of commonly used handle and object syscalls.

of the allocators used. Both allocators have sections of memory reserved for the control data structures, as well as the memory pool itself. The native Zircon kernel uses its internal VMO and VMAR representations to map these data structures close together, with a single guard page separating the two. However, the Zircon server allocates the control structure from the heap, which is located much further away from the memory region reserved for the memory pool. It is this poorer locality on the Zircon server which may be affecting the performance of the handle table; further work should be performed to optimise the design of the pool allocator used by the Zircon server with regards to the cache.

8.2.3 VMO Read/Write Performance

In this test, we compare the performance of the direct read and write syscalls. The reason for performing this benchmark is to observe what effect the use of the server mappings has on VMO read and write speeds. The test simply involves a thread writing a buffer of random data to the VMO, then reading the data back; each operation is benchmarked separately. We also perform sanity checks to ensure the data read back matches the data that was written.

Figure 6 demonstrates the cost of writing to a VMO with an increasing number of bytes, while Figure 7 shows a similar test, but with reading from the VMO. Initially, the Zircon server performs comparably to the native kernel, but as the number of bytes written or read increases, the performance of the Zircon kernel improves relative to native. This is likely due to the differences in the VMO implementations. The native Zircon maps all physical memory as a single chunk in its kernel address space, and allocates pages of memory from this chunk; VMOs on native Zircon are allocated pages this way. When performing a read or write to the VMO, the kernel goes through its list of pages, writing to each page individually. For the Zircon server, we have a contiguous mapping of all pages in the VMOs, thus direct reads and writes are sequential and have better performance.

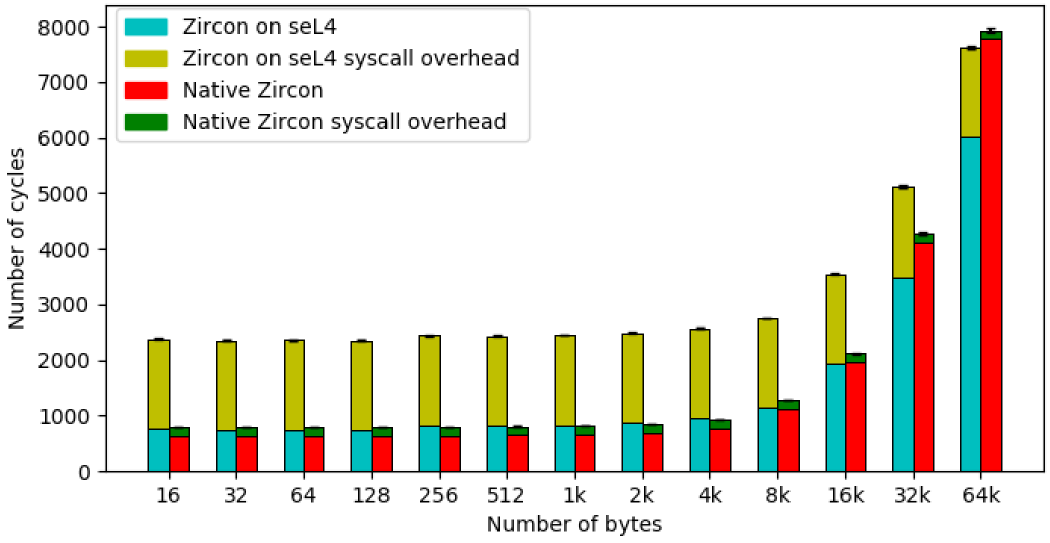

Figure 6: Performance of a direct write to a VMO.

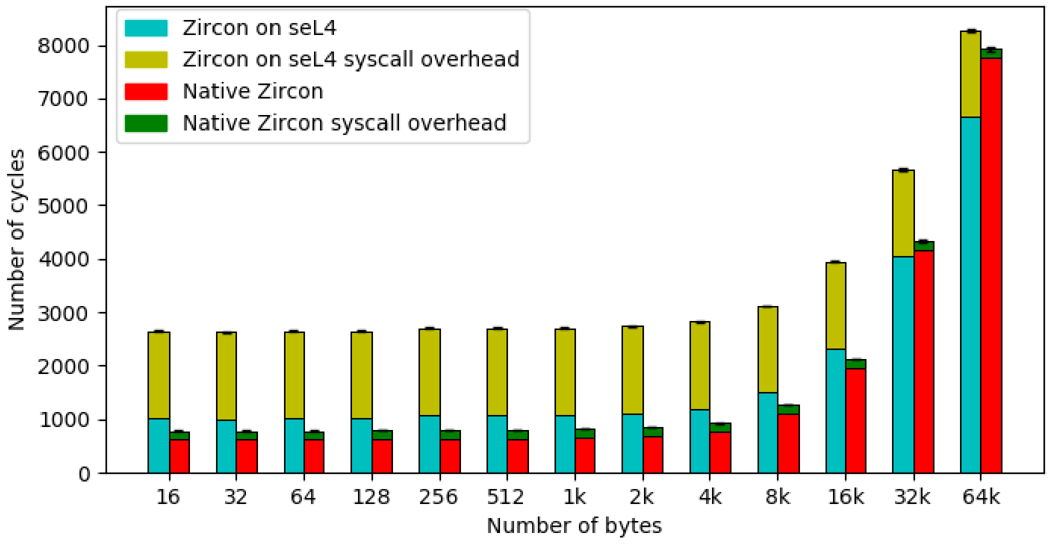

Figure 7: Performance of a direct read from a VMO.

8.2.4 Channel Performance

In this experiment, we analyse the performance of performing data writes and read on channel objects. This involves a single thread writing or reading randomised messages to and from each end of a channel, with the size of messages gradually increasing. The primary purpose of this test is to gauge the effectiveness of the implementation of channels; it is not a reliable indicator of IPC cost, as there is only a single thread operating on a channel, with no context switches being performed between processes.

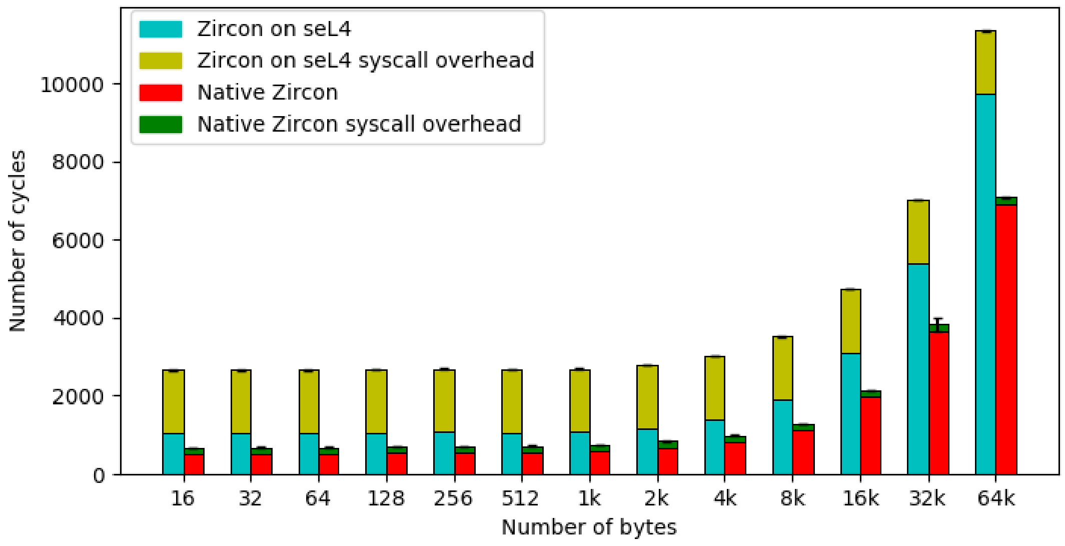

Figure 8: Performance of writing messages to a channel.

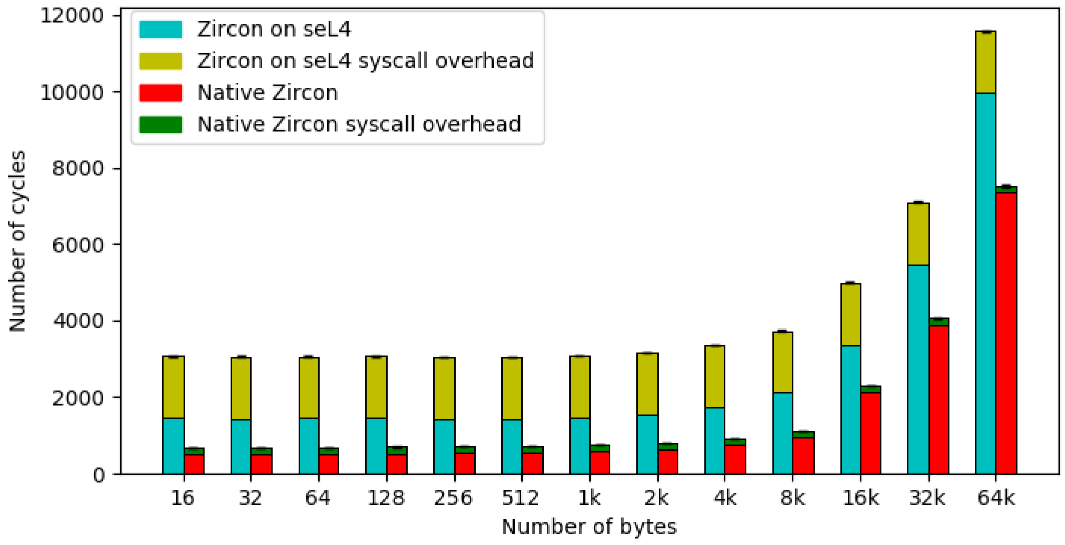

Figure 9: Performance of reading messages from a channel.

Figures 8 and 9 demonstrate the performance for the respective channel operations. It is clear that the Zircon server lags behind the performance of native Zircon, which worsens as the size of the message increases. The reason for this is due to the differences in storing each message.

The Zircon server allocates a separate message packet from the heap to describe the dimensions of the message, the actual data is written to the channel's message buffer. This is done as the server uses a static heap, and we wish to avoid large allocations in order to avoid running out of heap space. Native Zircon instead allocates the entire message packet from the heap, where it stores all parts of the message. These differences in design have two implications on performance:

- There is overhead on the Zircon server contributed by page allocations to the message buffer in addition to a heap allocation, and
- native Zircon has a contiguous region of memory for sequentially writing and reading the data of a message, thus benefiting from cache prefetching, while on the Zircon server, data is dispersed over many pages in message buffer, making reads and writes less predictable.

These are two factors that are likely resulting in the performance discrepancy of the Zircon server and native Zircon, and demonstrate areas where the implementation of channels on the server can be improved. One option is to remove the message packet list; instead, the message buffer would store a combination of a message header, describing the message contents, followed by the message data. Another option is to replace the static heap used by the Zircon server with a dynamic allocator used in other kernels, such as a slab allocator [B+94]. This would allow for the allocation of messages in the same way as native Zircon, allowing performance to be matched.

8.2.5 Socket Performance

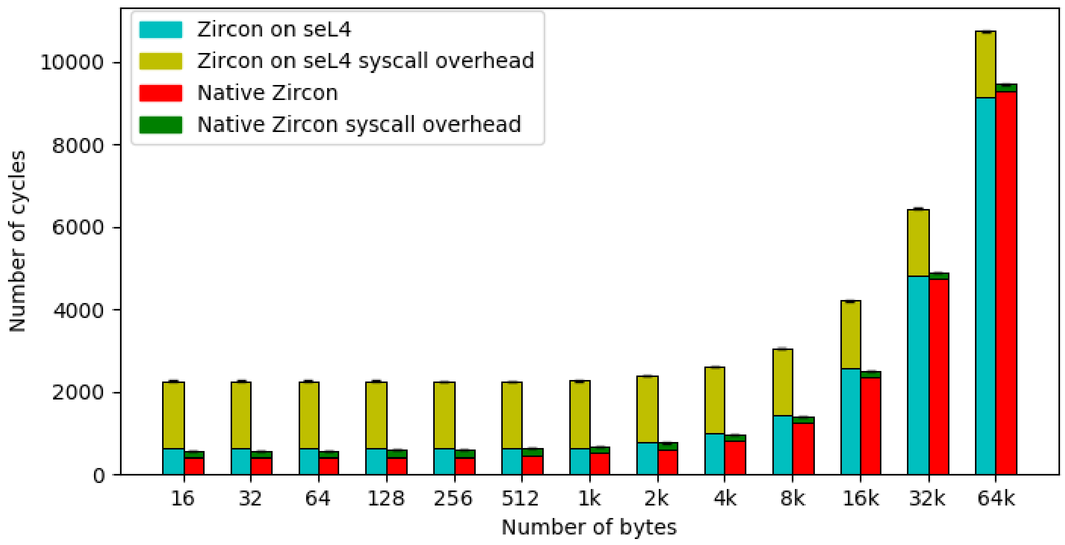

Figure 10: Performance of writing messages to a socket.

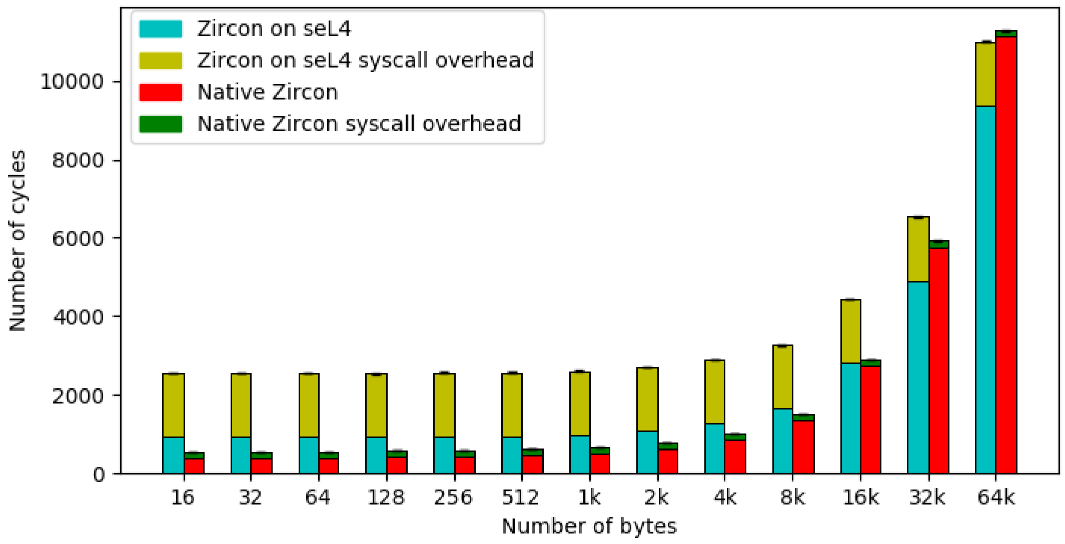

Figure 11: Performance of reading messages from a socket.

The performance of sockets is tested in the same way as channels, with the socket being written to and read from with increasing message sizes. The results of these tests are displayed in Figures 10 and 11. Sockets on the Zircon server fare much better than channels, with performance coming close to or even beating that of native Zircon for larger messages. Native Zircon uses a variable sized buffer structure very similar in design to the message buffers, with two key differences. One is that native Zircon does not use a page allocator to back message data. Instead, each memory segment is allocated from the heap, which would be a more expensive operation than an allocation from the constant time page allocator used on the Zircon server. The other difference is that these memory regions are half the size of a page, thus for larger message sizes more memory segments need to be allocated. These two factors would both contribute to the decline in performance relative to the Zircon server; it also demonstrates that the use of the page allocator provides high scalability for socket operations.

8.2.6 IPC Call and Reply

While the previous tests are focused on the performance of single operations, it is also necessary to evaluate the performance of more general operations comprised of multiple syscalls. This test focuses on performing synchronous IPC between processes through a channel. Such a test also allows us to gauge the cost of performing context switches between Zircon processes.

Before starting the experiment, the main benchmarking process (Process 0) creates a second process (Process 1), and provides the new process a handle to one end of a channel object. We cannot benchmark this step due to the differences in initialising new processes on native Zircon and Zircon on seL4. Once Process 1 is ready to receive messages, we start the experiment. Process 0 writes a message to one end of a channel, and waits on the channel for a response. Process 1 waits for the other end of the channel to become readable, then reads the message and writes back the same message as a response. Process 0 then reads this response. After timing has stopped, the message is checked for validity. A total of six syscalls is performed in the experiment, with each process performing three syscalls.

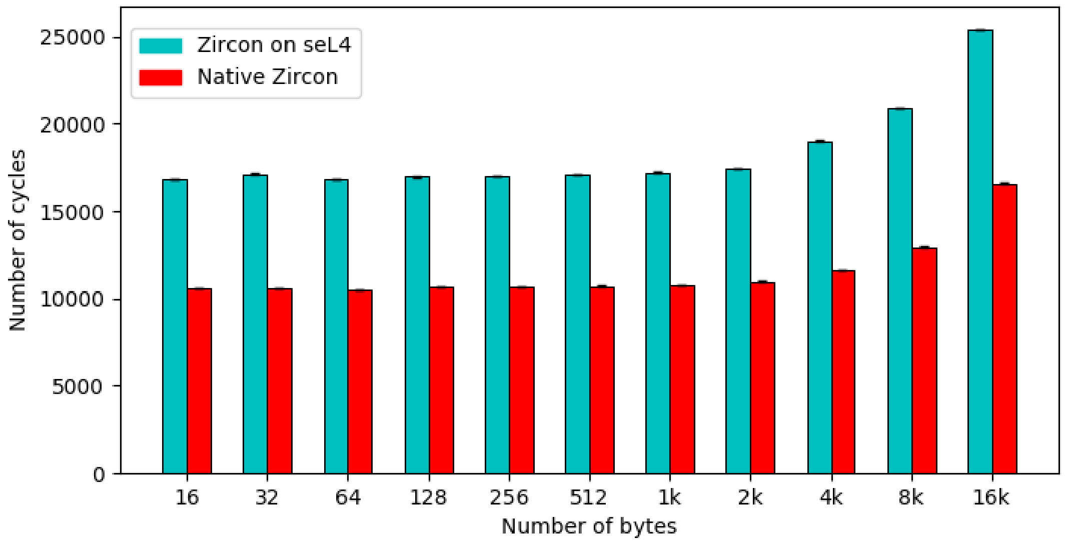

Figure 12: Cost of performing a synchronous call and reply with Zircon kernel objects.

Figure 12 shows the results of this test. Although Zircon on seL4 is outperformed by native Zircon, its performance is quite reasonable given the number of syscalls being performed in this test, each incurring the overhead of IPC with the server. The slight deterioration in performance with larger message sizes is likely attributed to the poorer performance of channels discussed previously. With improvements in these two areas, it is likely that the performance of the Zircon server can become close to native.

It is important to note that in this test each process waits forever on the channel. The Zircon server has an optimisation in which a thread will not register a timer if it is to wait forever. As such, it completely avoids any overhead from reprogramming the timer. If a finite deadline is given, the cost of waiting blows out significantly. If each process in this test uses a finite deadline when waiting, the timer is reprogrammed for both processes, and the cost of this operation exceeds 60000 cycles.

An experiment was performed by modifying the timer from tickless to ticking, to see the impact on waiting, as well as general performance. The mean cycle counts for all tests increased by roughly 2000 cycles, including the mean cycle count for this test; the use of a finite deadline did not significantly affect this value. If the use of finite deadlines is common in user-level Zircon systems, it may be necessary to switch to this ticking timer to avoid this high overhead for waiting.

8.3 Port Evaluation

8.3.1 Implementation Effort

As discussed in Section 7, the implementation of Zircon on seL4 is comprised of two components: the library used for performing Zircon syscalls with seL4 IPC, and the Zircon API server, which is a seL4 application that handles these syscalls. Table 1 shows a breakdown of the source lines of code required to implement each component.[4] This includes all code used to implement all objects defined on the Zircon server, as well as all the syscall handlers that are defined. The values presented in the table also include code that is generated by the syscall generator script.

Component	**Source lines of code**
Zircon Server	6893
Syscall Library (libzircon)	1476
Total	8369

Table 1: Source lines of code in the Zircon on seL4 implementation.

While the server still requires certain syscalls and objects to be added, it is likely that the total source lines of code will remain less than 10000. Furthermore, much of the syscall handling code follows a similar order of operations to that of native Zircon, which eased the implementation effort during development. In comparison, the lines of code in the native Zircon kernel alone is over 65000, and this value does not include several user-level libraries that are used by the kernel. As such, the effort required to implement the Zircon server is not too high, and the smaller codebase facilitates the debugging and auditing of the server.

8.3.2 Porting Zircon Applications

Although the porting of native Zircon applications is limited due to the use of different C libraries, it is still possible to port native Zircon code to Zircon on seL4 with little effort. This is demonstrated through a port of the mini-process user-level Zircon library.

This library allows for the creation of very simple, single function processes. While these processes are very limited in use, they can still be used to verify that Zircon processes are being created and started correctly; it was for this purpose that this library was ported to Zircon on seL4. Table 2 shows the lines added and removed from the library in order for the library code to compile and be used. It also includes a small ELF loading library that the mini-process library depends on. Most of the changes to the code involved

[4]Generated using David A. Wheeler's 'SLOCCount'.

disabling vDSO related function calls that could not be used on seL4. Although lacking these particular functions, the library is still mostly usable, and is capable of creating mini-processes on seL4.

File	**Lines Added**	**Lines Removed**
mini-process.c	4	9
subprocess.c	4	6
elf-load.c	0	1
elfload.h	0	1
Total	8	17

Table 2: Lines added and removed for porting the mini-process library.

The porting of Zircon applications is also assisted by an additional ELF loading function that has been added to the syscall library. This function is capable of loading and starting any Zircon application that have been ported to the seL4 build system, while exclusively using Zircon objects.

The root application on seL4 is capable of storing a CPIO archive of all applications built for a system; it is from this archive that the Zircon server fetches the application that will run as the first process of the system. The Zircon server also defines an additional syscall which allows for an ELF file to be taken from the CPIO archive and dumped into a VMO. This is used by the ELF loading function to fetch the application that is to be loaded. From here, a new process is created, and each ELF segment is loaded into a VMO mapped into the root VMAR. A thread is also created, and a VMO that acts as the thread's stack is allocated. The new process can then be started. This is the method used to start the Process 1 of the IPC Call and Reply test, and can be used for many other Zircon applications. This provides support for a multi-process Zircon system to be brought up on seL4, providing the Zircon applications can be built with seL4's build system. Further work for dynamic system support is discussed in the next section.

9 Future Work

While the current implementation of Zircon of seL4 provides the core foundation for running user-level Zircon applications, there are still many aspects of the implementation which remain unfinished, or need to be updated to match those of native Zircon. Additionally, there are many improvements that can be made to the performance and scalability of the Zircon server. This section details the future work required in these areas.

9.1 Missing Objects and Features

9.1.1 Ports

Port objects are one of the primary objects of the Zircon kernel that require completion. There are many aspects to the implementation of Ports; these include the allocation and storing of port packets, asynchronous waiting, and threads waiting on them.

For their implementation on seL4, Ports would contain a list of port packets; these would likely be allocated and freed with a pool allocator, as this is the approach taken by native Zircon. As with how StateWaiters are used to represent threads waiting on an object for a state change, Ports would have a similar PortWaiter class for threads waiting for packets to be delivered.

Asynchronous waiting can be supported with a modified StateWaiter class. When updating the list of waiters after an object state change, each waiter just needs to be checked if it is owned by a Thread or a Port to determine the appropriate action to take. In the case of a Port, each StateWaiter has an associated port packet which is delivered to the port.

One other use of ports is the ability to bind them to a Thread, Process or Job as an exception port. When a thread faults, and it or one of its parent processes or jobs has an exception port set, a packet describing the fault is delivered to the port. Another thread waiting on the port can then receive the message and handle the fault appropriately. For Zircon on seL4, we would need to modify the fault handler to check a thread for an exception port, rather than just kill it. The fault handler can then allocate a port packet and deliver it to the relevant port.

9.1.2 Futexes

Zircon also offers Futexes, which are a common primitive found in many kernel used to implement mutexes and other locking mechanisms [FRK02]. All futex operations are based upon an integer value from a user address space, which is accessed atomically by threads. There are three operations:

- **Wait**: if the value provided by a thread is equal to the atomic integer, the thread is put to sleep, and is considered to be waiting on this integer.
- **Wake**: wakes up N threads waiting on the integer.
- **Requeue**: wakes up N threads waiting on the integer, and requeues M threads to wait on a different atomic integer.

The implementation on seL4 would likely be similar to native Zircon, with a hash table used to lookup a waiting thread based on the user address of the integer. No seL4 objects would be required for futexes.

9.1.3 Miscellaneous Features

While most channel operations are defined, the call operation for channel objects still needs to be implemented. To support calling, Channels would have a list of channel waiters, representing the waiting threads. Each call or write to a channel would perform a check to see if a message's transaction ID matches the ID of a thread waiting on the channel for a response. If a waiter is found, the message is immediately delivered to the waiting thread.

Another feature is VMO cloning, which allows for the duplication of VMOs with a single syscall. VMOs are expected to track the VMOs cloned from them as part of their state; this can be achieved by each VMO maintaining a list of child VMOs. Additionally, VMOs can be copy-on-write clones. VMOs cloned this way would need to create copies of its parent's frame caps for mapping pages that have not been written to.

Job policies are another feature missing from the current implementation. Policies dictate what actions are permitted to be performed by a job and its child jobs and processes. Policies can be augmented to additionally generate an exception or kill a process when it performs particular operations. Examples of policies include restrictions on the creation of various kernel objects, and limitations on what can be done with certain syscalls. Implementing policy enforcement on the Zircon server primarily requires addtional policy checks to be added to the various syscall implementations.

One other area that Zircon on seL4 lacks support in is the 64-bit ARM architecture, as development was restricted to the x86 platform to simplify the project. However, the Zircon server has been implemented in a fairly generic fashion, and there a few areas in the codebase which use architecture-specific code directly. As such, adding support for the ARM platform would not be too difficult.

9.2 Device Support

Although support for hardware I/O is very important, the unstable API for device support in Zircon means that this support must wait. Over the course of this thesis, many significant changes to the API have been witnessed, and it would be wise to delay development until stability is reached. However, potential approaches for implementing the more consistent abstractions for Zircon on seL4 can still be discussed.

Interrupt objects, like any other Zircon object, are accessed by handles; this includes waiting on the object for an interrupt to be delivered. As such, the Zircon server would be responsible for receiving interrupts delivered to a notification object, which it would then forward to a waiting thread. If this added latency is undesirable for some device drivers, a wrapper object for seL4 notifications could be introduced. These would work similarly to endpoint objects, with the server handling creation of notifications and the copying of caps to the CSpaces of Zircon threads. The server could then be instructed to deliver the interrupt to the notification object, allowing device drivers to directly use the seL4 API to handle interrupts.

To support VMOs with contiguous or physical memory we can use the existing VKA library, which supports the allocation of contiguous chunks of memory, or memory at a certain physical address. The existing VMO implementation would need to be extended so it can be made aware it is handling these varieties of memory. Mapping functions would also requires several changes for these cases, as well as performing the mappings with the correct cache attributes.

9.3 Syscall Library as a vDSO

As mentioned in the Background section, Zircon system calls on seL4 are currently compiled as a static library, which is linked to applications at compile time. As such, the system call interface is included as part of the code section in each application. However, for proper compatibility with native Zircon, transitioning to a shared library that matches the specifications of Zircon's syscall vDSO is a necessity. Zircon's libc, although musl-based like seL4's, is built around the existence of the vDSO, and thus cannot be used for building any Zircon applications that are to run on seL4. This is also the case for many other Zircon libraries, which makes porting to a static library difficult. As such, it is imperative that Zircon on seL4 transitions to a vDSO for full support of Zircon applications.

Building a vDSO also allows for the leveraging of the Zircon build system. Zircon's build system creates a root filesystem (rootfs) that contains all compiled user applications. There is no reason why applications from this rootfs cannot be linked to a vDSO made for seL4; if a vDSO can be built to the correct specifications, it should be possible to dynamically link it with other code at runtime, regardless of its internals. As the vDSO is the only piece of "glue code" that performs seL4 invocations, and is the only gateway for Zircon applications to perform syscalls, there should be no other user-level code requiring modification to run on seL4.

Due to this property, it should then be possible to create a boot image composed of the Zircon server, the syscall vDSO and the rootfs. Then, when the Zircon server is starting up, it can get the first boot process from the rootfs, load the vDSO into its address space, and start it. With access to the vDSO, the boot process would have the same environment as native Zircon, and can start the rest of the system without needing any modifications. This allows for the Zircon build system to be re-used for building all Zircon applications, rather than having to manually port them to seL4.

9.4 Multicore Support and Scalability

The Zircon server is currently single threaded, and only supports running threads on a single core. This allowed for concurrency issues to be avoided during development. However, Zircon is targeted at modern systems, which virtually always have multiple cores. As such, having a multithreaded server would be necessary for providing multicore support with reasonable performance; a system with heavy contention of the server, where many threads are performing syscalls frequently, would become bottle-necked by a single-threaded server.

The approach for concurrency control on the Zircon server would need to be carefully considered. Native Zircon implements fine-grained locking of kernel resources to allow for many threads to enter the kernel at once and perform operations simultaneously. A multithreaded server is a very different model, with syscalls from N many Zircon threads being serviced by M seL4 threads running on the server. As such, simply taking the concurrency protocols directly from Zircon may not be appropriate. An analysis of the most contended resources on the server may be required before locking structures are finalised.

To assist the development of Zircon on seL4, the current Zircon server was designed with relatively small systems in mind. This can been seen through the static data structures, such as the allocators, which have a fixed limit before they run out of memory; this limit is set at compile time. This would not be suitable for large and very dynamic systems, where resource requirements may fluctuate in different areas over time. As such, many improvements could be made to the server that would allow the server to scale itself appropriately.

The static allocators used by the server, such as the thread and handle tables, pose limitations on the maximum amount of objects that can be created on the server. Simply increasing the size of the allocators would be an inefficient solution, as memory is wasted on objects not allocated. Thus it is more suitable to make these allocators dynamic in nature, growing and shrinking them as required by the user-level system. However, fragmentation becomes an issue in such allocators; objects can become spread around the allocator, which can impede the ability to reduce the memory used by an allocator. The ability to relocate objects within the allocator may need to be introduced to deal with this issue.

Another limitation is the server mappings. With many VMOs mapped on the server, the amount of address space used increases, which leads to excess memory usage with all the page tables that need to be allocated, as well as poor locality, with the sheer number of pages mapped potentially leading to frequent TLB misses, and thus performance degradation. As such, it will likely be necessary to avoid always mapping a VMO on the server in the future.

Instead, server mappings could be performed on-demand, only being allocated when required for direct VMO reads or writes, or for the translation of user addresses to server addresses for syscalls. Reallocating server mappings too frequently would harm performance, so a mapping would need to remain mapped on the server for a period of time before removal. This could be achieved through the use of a page-replacement algorithm for the eviction of old server mappings, and would allow for VMO mappings that are referred to frequently to remain mapped on the server, while overall reducing the total number of mappings. To avoid performing mapping replacement when syscalls are being handled, the server could use an additional low priority thread that runs in idle time to perform unmappings when the number of server mappings reaches a certain threshold.

One further optimisation for server mappings would be to paginate them; that is, the VMO is partitioned into equally sized chunks, and each chunk is allocated mapping space, just as how virtual memory is paginated. This has several advantages over the current design of allocating a large, fixed-size address space region for the entire VMO. While this design is very simple and has low overhead for the management of mappings, it has poor locality; mappings are very spread out, requiring more page tables to be allocated, and likely increasing cache and TLB misses. It also limits the maximum size of a VMO. Paginating server mappings into chunks allows for mappings to be kept in a more compact region of the address space, improving locality. Combined with on-demand mapping, it also removes any size restriction on VMOs, as we can allocate and free VMO chunk mappings as required. To implement this, each VMO would require a mapping table to track the address regions it has been allocated by the server for mappings. The exact size for the chunks would need to be investigated; they should still be large enough to both fit most user-level buffers, and minimise the additional memory required for tracking server mappings.

One major issue with this pagination approach is the handling of non-contiguous buffers. This issue arises from a Zircon process with region of memory that is contiguous in its address space, but is spread over two or more mapping chunks for the server. This complicates address translations, and the copying to and from these regions of memory, as the server currently expects the region to also be contiguous in its server mapping. The handling of user memory regions would need to be modified so that these non-contiguous regions can be handled. One option would be to have a wrapper class for user memory regions that can transparently read and write across many server mappings.

Another potential feature to improve performance and increase scalability would be the deferral of work that can instead be performed when Zircon threads are idle. Such work includes the destruction of objects, which does not fail, and syscalls do not require this to happen before they return. Thus it would be more appropriate to perform this work at a time where both the server and Zircon threads are doing little or no work, such as sleeping or waiting for I/O. One or more low-priority threads could be added to the server to run during this time, and perform work that has been deferred. Overall, this would allow for faster response times for many syscalls, as their handlers have less work to do immediate and can reply to threads sooner.

9.5 Dynamic Scheduling

Currently, Zircon on seL4 uses static scheduling for threads, which all share the same priority. Thus execution time is shared equally between threads in a round-robin fashion, and the server provides no way for the priority of threads to be changed in a system. Native Zircon uses dynamic scheduling, with the priorities of threads being adjusted on-the-fly as their CPU time requirements change, and Zircon on seL4 should be upgraded to use a similar form of scheduling at some point in the future.

For the implementation of an efficient dynamic scheduler on the Zircon server, a migration to the mixed-criticality systems (MCS) branch of the seL4 kernel may be desired, as it can be used to implement efficient user-level schedulers. Each thread would need to be allocated a scheduling context; the server can have a thread responsible for adjusting the allocated period and budget of each thread's scheduling context according to their needs.

An extension to this is to make the Zircon handle syscalls as a passive server. This provides a type of migrating thread model, where syscall-servicing threads have no SC of their own, and instead make use of a SC borrowed from the calling thread. The advantage of this model is that it provides stricter control of the execution time of threads. If the server threads use their own SCs, a thread can potentially run "for-free" as they are using the server's allocated time rather than their own. By having threads donate their SCs, usage of the Zircon server is accounted for in a thread's allocated timeslice.

9.6 The seL4 Root Task

To make development of the Zircon server more straightforward, the server currently runs as the root user task on seL4. However, a common paradigm for systems running on seL4 is to use the root task as an application driver. It is normally responsible for starting all other applications, granting them the resources they require, and acting as a watchdog if applications crash. This design offers several advantages.

With a separate root task, other seL4 applications can be started completely independent from the Zircon server. Currently, additional seL4 applications can only be launched through modifications to the server, which can create other seL4 processes at startup.

The root task also allows for greater restrictions to be applied on a Zircon server instance. We can easily limited the kernel resources that are supplied to it, since we use a VKA for managing these resources; the VKA allows for this to be handled transparently.

Having the root task act as a watchdog allows for any faults on the server to be recoverable. The root task can wait on the Zircon server's fault endpoint; if the server crashes it can then be restarted by the root task.

The root task can also be capable of creating multiple Zircon server instances. These instances would be independent and isolated from each other; if one crashes, the other remains unaffected. This would be useful for situations where redundancy is required, or where individual Zircon subsystems are required, with some subsystems being more critical than others and thus needing to be isolated from them.

9.7 Communication with seL4 Applications

Support for communication with native seL4 applications through endpoints is partially implemented. Currently, the creation of an endpoint requires an ID to be set. This ID is intended to provide a method of discovery for endpoints for applications that do not deal with handles. The server implements a lookup table so that endpoint can be located by their ID. This lookup table is kept quite small, as we do not expect many endpoints to be created; this is enforced in Zircon applications by requiring a resource be supplied for creation of endpoints.

The next step is to expose an API to native seL4 applications. These would be supplied caps to the server's endpoint at boot time, with badge values distinct from those used by Zircon threads. Thus the server can recognise that it is communicating with a native application. We allow for these applications to create an endpoint object on the server; these endpoints are marked as being "externally created". The syscall used to create endpoints for Zircon processes already has an option to check for an externally created endpoint; if one is found, the process is returned a handle to the existing endpoint. Closing these handles to the endpoint does not destroy the endpoint.

Once an endpoint is created in this way, native apps can then request for a cap to the endpoint. This would be performed using seL4 IPC, which allows for a cap to be shared over an endpoint; the cap is placed in the receiver's CSpace in the requested slot. Zircon processes would get their caps with the existing API. Thus both seL4 and Zircon applications have access to the same endpoint, and can then perform IPC as they wish.

The above outlines one method of providing a communication channel between Zircon applications and their native seL4 counterparts. However, it requires these Zircon applications to be seL4-aware; that is, these apps must be built with an extended syscall library containing these additional endpoint functions to allow for this communication channel to exist. As the IPC model provided by endpoints is quite different to Zircon's IPC, existing apps may need to be extensively modified in order to support endpoint objects. It may be more beneficial to instead implement the reverse: provide seL4 applications a way of obtaining Zircon objects to form a communication channel.

This could be done by allowing seL4 apps to create "dummy" Zircon process and thread objects. A dummy process would be required to allow seL4 apps to store and manipulate handles, while a dummy thread is needed so the app can be given an endpoint cap for performing syscalls as if it were a Zircon thread. With access to the Zircon syscall library, the app would then be free to create and manipulate Zircon objects using the standard syscall API. An application driver could then be added to a Zircon user-level system to expose seL4 application nodes in a Zircon user-level system. Zircon applications can then begin to interface with seL4 applications, and create communication channels with existing Zircon IPC objects.

10 Conclusion

This thesis provides the core implementation of a port of the Zircon microkernel to seL4. This was achieved through an API emulation approach, with a seL4 application running as an API server for managing the kernel objects and syscalls of Zircon.

An evaluation of the performance of the Zircon API server against that of the native Zircon kernel has also been provided. Despite the additional overhead of performing syscalls, the server has performance that is reasonable, and many avenues for improvement of performance have been identified. An analysis of the porting effort was also performed, which demonstrated that the API emulation approach that this thesis has taken is reasonable to maintain.

Additionally, the many areas of this project which require further work have been discussed. This not only includes the implementation of the remaining kernel objects of Zircon, but also areas for improving that performance and scalability of the Zircon server. By providing seL4 endpoints as Zircon objects, we have a platform for forming communication channels between Zircon processes and native seL4 applications, and can further this concept to provide Zircon objects to seL4 programs.

With continued development of Zircon on seL4, true support for a dynamic user-level Zircon system can be established. This thesis lays the groundwork for achieving this, and further work will allow for seL4 applications to fully leverage the software and drivers offered by the Zircon platform.

References

[ABB$^+$86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. 1986.

[Aut17] Wine Authors. WineHQ - run Windows applications on Linux, BSD, Solaris and macOS. `https://www.winehq.org/`, 2017. Visited 17 Oct 2017.

[B$^+$94] Jeff Bonwick et al. The slab allocator: An object-caching kernel memory allocator. In *USENIX summer*, volume 16. Boston, MA, USA, 1994.

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS operating systems review*, volume 37, pages 164–177. ACM, 2003.

[CBM$^+$94] Michael Condict, Don Bolinger, Eamonn McManus, Dave Mitchell, and Steve Lewontin. Microkernel modularity with integrated kernel performance. Technical report, Technical report, OSF Research Institute, Cambridge, MA, 1994.

[EK$^+$95] Dawson R Engler, M Frans Kaashoek, et al. *Exokernel: An operating system architecture for application-level resource management*, volume 29. ACM, 1995.

[FRK02] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in linux. In *AUUG Conference Proceedings*, volume 85. AUUG, Inc., 2002.

[GDFR90] David B Golub, Randall W Dean, Alessandro Forin, and Richard F Rashid. Unix as an application program. In *UsENIX summer*, pages 87–95, 1990.

[Goo17a] Google. Fuchsia - Fuchsia is not Linux. `https://fuchsia.googlesource.com/docs/+/master/book.md`, 2017. Visited 9 Oct 2017.

[Goo17b] Google. zircon - Git at Google. `https://fuchsia.googlesource.com/zircon`, 2017. Visited 11 Oct 2017.

[HE16] Gernot Heiser and Kevin Elphinstone. L4 microkernels: The lessons from 20 years of research and deployment. *ACM Transactions on Computer Systems (TOCS)*, 34(1):1, 2016.

[HEV$^+$98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The mungi single-address-space operating system. *Software: Practice and Experience*, 28(9):901–928, 1998.

[HHL$^+$97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. The performance of μ-kernel-based systems. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 66–77. ACM, 1997.

[Hil92] Dan Hildebrand. An architectural overview of qnx. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

[KEH$^+$09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[LMAH18] Anna Lyons, Kent McLeod, Hesham Almatary, and Gernot Heiser. Scheduling-context capabilities. 2018.

[LVSH05] Ben Leslie, Carl Van Schaik, and Gernot Heiser. Wombat: A portable user-mode Linux for embedded systems. In *Proceedings of the 6th Linux. Conf. Au, Canberra*, volume 20, 2005.

[MMR$^+$13] 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. In *ACM SIGPLAN Notices*, volume 48, pages 461–472. ACM, 2013.

[PPTH72] Richard P. Parmelee, Theodore I. Peterson, C Ci Tillman, and Donald J Hatfield. Virtual storage and virtual machine concepts. *IBM Systems Journal*, 11(2):99–130, 1972.

[S$^+$17] Cygnus Solutions et al. Cygwin. `https://www.cygwin.com/`, 2017. Visited 17 Oct 2017.