RICE UNIVERSITY

Exploring Intralingual Design in Operating Systems

By

Ramla Ijaz

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

APPROVED, THESIS COMMITTEE

Lin Zhong
Lin Zhong (Nov 30, 2020 14:43 EST)

Lin Zhong
Committee Chair,
Professor of Electrical and Computer
Engineering and Computer Science

Joseph Cavallaro

Joseph Cavallaro
Professor of Electrical and Computer
Engineering and Computer Science

ang chen
ang chen (Nov 28, 2020 14:26 CST)

Ang Chen
Assistant Professor of Computer Science and
Electrical and Computer Engineering

HOUSTON, TEXAS

November 2020

ABSTRACT

Exploring Intralingual Design in Operating Systems

by

Ramla Ijaz

Safe languages have been used to avoid memory safety violations in Operating Systems (OS). Software isolation techniques have been presented as a way to eschew the high cost of hardware isolation. Single Address Space/ Single Privilege Level (SAS/SPL) OSes use type and memory-safe languages to avoid system calls and changing address spaces. This thesis extends the use of safe languages in OSes to provide benefits beyond memory safety and avoiding the overhead of hardware isolation. We demonstrate that *intralingual OS design*, using language-level mechanisms to ensure OS correctness, allows us to build systems that can provide isolation without specialized hardware. Intralingual system design also allows us to transfer responsibilities that were previously the OS developer's to the compiler, such as memory and power management. We use Rust, a safe, systems programming language with a runtime similar to C, to apply intralingual design to four parts of the OS: the heap, Inter-Task Communication (ITC), networking and power management. We present a SAS heap shared by multiple processes that can decrease fragmentation by up to 30% over per-task heaps. We implement a shared-memory ITC channel that is 20% faster than the fastpath IPC of a leading microkernel. We use type safety and Rust's ownership feature to safely share a Network Interface Card's (NIC) resources among applications while ensuring direct access to hardware. Lastly, we demonstrate that a whole class of power management bugs can be eliminated when shifting reference counting to the compiler.

Acknowledgments

The first person I would like to thank is my advisor, Lin Zhong. His consistent encouragement, patience and insistence on "avoiding low hanging fruit" has motivated me to search for difficult and interesting problems. Over the past three years, I have learned how to ask the right questions and how to dig deeply to answer them.

I would also like to thank Dr. Joseph Cavallaro and Dr. Ang Chen for serving on my thesis committee. Their questions and comments have helped me to thoroughly understand aspects of my work and improved it overall.

I had the privilege to work with fellow PhD students at Rice for this project. Kevin Boos has been generous with his time and feedback since the semester I joined the group, and because of that I have been able to understand his work on Theseus and contribute to it. Namitha Liyanage is usually the first person I approach with technical problems or to sound off new ideas, and he has always given his help freely. I'd also like to acknowledge every member of the RECG. Their interest in my work and feedback has always been encouraging.

My family has been essential in this time. My husband has been a constant source of support and motivation. Weekly calls with my father and siblings in the past few years have been a way for me to take a step back when feeling overwhelmed. This road would have been much harder without all of them, and so, last but not least, I'm grateful to all of them.

Contents

Illustrations

Tables

Chapter 1

Introduction

Traditional OS designs use hardware mechanisms to enforce isolation. Separate address spaces managed by the Memory Management Unit (MMU) isolate processes. Hardware privilege levels allow the OS greater capabilities than user applications. Safe languages in OSes have mostly been used to eliminate memory errors and to avoid the overheads of hardware isolation. We believe there are more advantages to using a safe language in OS design than have been explored before. In this thesis we explore an alternative design philosophy: intralingual OS design. The major goal of our work is to design different parts of an intralingual OS to take full advantage of the features provided by a safe language, and shift as much work to the compiler as possible.

One principle of intralingual OS design is the removal of dependency on hardware. Previous work on SAS/SPL OSes explored this feature by shifting the responsibility of process isolation to the language [1–3]. We explore this even further and show that a device can be shared among applications without CPU virtualization support. It is important to note that high performance systems that run on top of Multiple Address Space/ Multiple Privilege Level (MAS/MPL) OSes consistently try to avoid the overhead of hardware isolation. Recent examples include systems which use kernel bypass networking and safe languages to run user code in a privileged domain [4–8]. These systems make design decisions that require a trade-off with other design goals: running one thread per core reduces CPU utilization [5], userspace networking can lead to memory safety bugs and only allows one process to safely access a NIC [9,10]. We do not claim that trade-offs are not present in an

intralingual design, but we posit that the penalty is less severe.

In recent years, there has been a trend for applications to bypass the kernel and use virtualization hardware to access devices directly [4,6,11–13]. A trusted piece of software, usually a hypervisor, runs at a higher privilege level than the guest OS and configures the devices for sharing among multiple Virtual Machines (VMs). The hypervisor has a smaller attack surface than the OS, making it more trustworthy. At the core of hardware isolation lies the ability to add higher levels of privilege at which smaller pieces of software run. These privilege levels will always come with a switching overhead, e.g. VM exit and entries, and system developers will design their systems to avoid them. In addition, the recent Meltdown [14] and Spectre [15] attacks have brought into question the isolation guarantees of hardware, which are difficult to independently verify. We suggest that, now that programming languages have become powerful enough to provide isolation with minimal overhead, the role of hardware in computer systems should not be to provide isolation, but to focus on performance and efficiency.

The second principle of intralingual OS design is minimize developer effort. The goal is to shift OS responsibilities to the compiler and avoid human error. Distributed resource management and converting OS runtime checks to static checks are areas that have been studied before [3]. In this thesis we explore how power management can be made easier by using compiler-tracked reference counts. We still rely on correct implementation logic by the OS developer, but if the OS developer encodes OS semantics into their implementation, future developers are less likely to make errors when using their code.

Among the many benefits of intralinguality, one that has been studied extensively is the reduced process isolation overhead. An in-depth study of the overhead of different isolation mechanisms was completed by the Singularity project [1]. They showed that the overhead of virtual address spaces and privilege levels can be significant for basic OS operations and

presented Software Isolated Processes (SIPs) as a lightweight alternative to the traditional hardware-based approach. The efficiency benefits of software isolation have been brought into question by results put forward by the L4 microkernel family. They have proven that, through careful system design, the overhead of hardware protection is acceptable. They have optimized their IPC fastpath so that, despite additional operations of kernel entry, kernel exit and address space switch, it is faster than IPC between Singularity SIPs. We believe, while exploring new intralingual designs, it is important to revisit the question of efficiency of different process isolation methods since much has changed in the systems landscape.

Since Singularity was published 12 years ago, both hardware and software isolation costs have been reduced. Hardware-isolation costs have decreased with MMU improvements including tagged Translation Lookaside Buffers (TLB), MMU caches and large page sizes. Software-isolation costs have also decreased with the advent of a safe, systems programming language that avoids a garbage collector. Rust provides type and memory safety with a runtime similar to C by relying on static analysis to track when an object goes out of scope. Rust's low runtime overhead and unique features gives us an opportunity to rethink OS design beyond just avoiding system calls and address space switches, the cost of which have already been mitigated by hardware advances.

There are certain features of intralingual design that ensure maximum power is given to the language, and we highlight them in our designs. They include: (*i*) Applications can share resources without going through the kernel. (*ii*) There is a 1-to-1 mapping between hardware resources and software variables which allows us to use Rust's ownership semantics to manage hardware as well. (*iii*) Applications are given direct access to hardware devices; type safety ensures they cannot misuse the device. (*iv*) All reference counting is done by the compiler, removing a responsibility from the OS developer which has been

shown to be error prone.

Initial work on intralingual OS design was done by the developers of the Theseus OS, a SAS/SPL OS written in Rust [3]. They present memory and task management subsystems that rely heavily on the compiler to ensure correctness. We develop and evaluate intralingual designs of four additional parts of the Theseus OS. Our heap design provides a system-wide heap that can be used by any entity, differing from other SAS/SPL designs that prefer to provide private heaps. We show that sharing a heap across processes can lead to fragmentation improvements as more allocation requests can be satisfied by the same heap. Under the hood, the heap is split into multiple per-core heaps for scalability, so the fragmentation benefits are limited to each per-core heap. Our per-core heap is shown to decrease fragmentation by up to 30% as compared to per-task heaps as the number of tasks running on a core increases.

Our ITC channel is an asynchronous shared-memory mechanism to pass messages between processes. We optimize this channel so that the compiled code is only 10 instructions on both the send and receive paths, the most expensive of which is the atomic compare and exchange. This simple ITC channel is 1.2x - 3.3x faster than the fastpath IPC of seL4, a leading microkernel.

Our networking subsystem design uses type safety and ownership to virtualize a NIC so that multiple tasks can access the NIC hardware concurrently. We implement an ixgbe driver for an Intel 82599 NIC in Rust which allows safe and mutually exclusive access to hardware. Currently, other networking solutions use virtualization hardware to grant multiple entities access to the same NIC.

Lastly, we implement power management within the ixgbe driver using compiler-tracked reference counting. We ensure that every entity that uses the NIC acquires a reference counted wakelock object along with access to NIC hardware. The compiler is responsible

for tracking the reference count of the wakelock and switching the device to a low-power state when there are no more outstanding references. We show that a class of power management bugs can be eliminated when using intralingual power management.

Our work shows that recent advancements in safe languages have given system developers a chance to rethink OS design. Using Rust, we can design systems that have multiple benefits that result from ease of sharing resources between processes. These benefits go beyond improved cycle counts for OS operations. Intralingual system design allows us to avoid depending on hardware for isolation and shifts developer responsibilities to the compiler, increasing the chance of correctness.

Chapter 2

Background

2.1 Cost of Hardware-Based Isolation

Privilege Levels: MAS/MPL OSes written in unsafe languages require privilege levels to ensure protection. The mechanism to switch from user to kernel privilege level is a system call. In Linux, the kernel entry path usually involves the following steps: (*i*) a libc wrapper function which arranges arguments in registers as required by the SYSCALL instruction, (*ii*) the SYSCALL instruction which changes the privilege level, stores the current instruction pointer and loads the address of the syscall handler, (*iii*) the syscall handler which swaps the stack and sets up for the system call function by saving registers and checking validity of arguments. The kernel exit path is similar and in reverse: restore registers, stack swap, SYSRET instruction which restores the user privilege level and instruction pointer. The actual time these steps take is known as the direct cost of the system call. We have measured that a null system call on an Intel skylake machine can take up to 100 cycles.

In addition there is an indirect cost to a system call, the pollution of the processor state by kernel information. During a system call, kernel specific state is loaded into the instruction and data cache, the branch prediction buffers outcomes are skewed by kernel state, and translations for the kernel portion of memory are added to the TLB. During this process, user space state can be evicted from these processor structures leading to lower performance once the userspace application resumes after the system call. One study showed that it took 14,000 cycles for userspace IPC to return to its pre-system call performance for

a SPEC CPU 2006 benchmark [16]. Unlike the direct cost of a system call, the indirect cost is harder to quantify and can depend upon the machine specs, the userspace application and the portion of the kernel code that is run.

Virtual Address Spaces: Virtual memory is an abstraction that provides a user application with the illusion that it alone has access to the entire memory space. This abstraction makes programming easier and applications portable since the developer does not have to consider the underlying memory hardware when writing code. The virtual memory system also allows for applications to run in an address space that can be much larger then the memory available to the CPU by swapping pages between memory and storage. One of the most important uses of virtual memory in modern OSes is to provide protection and isolation. Each user application is provided its own virtual address space which ensures that it cannot write or read to another application's memory. It is this last function of virtual memory, isolation, that is removed from a SAS/SPL OS.

Virtual memory, while providing many essential functions, also has many overheads that can range from 5-20% of the runtime [17]. A constant overhead that is present even in a SAS/SPL OS is the translation from virtual to physical address on every memory access. This translation requires a page table walk which can take 5 memory accesses in the x86_64 virtual memory subsystem, which uses a four-level page table. TLBs are an essential part of the MMU since they reduce the overhead of translation by storing recently accessed translations. The overhead of virtual memory can be found in a study conducted by the authors of Singularity; there is a significant increase in the cost of basic OS functions such as page allocation, process creation and IPC when comparing a physically addressed system with a SAS virtually addressed one [1].

The virtual memory overheads that a SAS/SPL OS avoids are two: (*i*) The CR3 register is updated with the page table base address for the next process in every context switch.

Updating this register can take hundreds of cycles [18]. (*ii*) A new page table has to be created every time a new process is created, which can increase process creation times by 14% [1].

2.2 Advances in Hardware MMUs

The effort to reduce the cost of using virtual memory has usually taken one of two directions: reducing the number of page table walks or making them faster. In the former category, tagged TLBs have helped by storing the translations of processes even after they have been switched out. This is done by assigning each process an Address Space Identifier (ASID) and storing that in each TLB entry alongside the translation. Previously, flushing a TLB after every context switch was one of the largest contributing factors to the indirect costs of multiple address spaces. Other advancements in this area are two-level TLBs that work similar to multi-level caches, and larger page sizes [17]. Increasing the TLB size would increase the access time for a translation, so having a hierarchy of TLBs is preferred. The base page size in most OSes is 4KiB, and now the option for 2MiB and 1GiB pages is supported as well so that one TLB translation can cover more memory. Page table walks are made faster through MMU caches that store PTEs from each level of the page table [19]. Architectures also support more page table walkers per CPU so that multiple translation walks can take place in parallel [17]. All of these advancements have reduced the performance gap between SAS and MAS systems.

2.3 Overhead of Software-Isolation

Traditionally, overheads of using a safe language have been of two types: garbage collection and runtime safety checks. Applications written in a garbage-collected language

can experience latency spikes and reduced throughput due to garbage collection pauses. Network device drivers written in garbage-collected language can have throughput ranging from 10% - 80% of a driver written in C [9]. Go is one of the fastest garbage-collected languages, and even it can lead to garbage collection delays of hundreds of microseconds [20]. Runtime safety checks have been shown to have a lower, more acceptable overhead and usually range from 2% - 6% of the total runtime [9, 20, 21]. It is the elimination of the first overhead that makes Rust an acceptable replacement for C when building computer systems.

2.4 The Rust Programming Language

The Rust programming language provides type and memory safety without the overhead of garbage collection, giving Rust a runtime similar to C and making it a practical choice for systems-level programming. Rust has a linear type system which enables Rust's most distinctive feature, *ownership*. Through ownership, the *lifetime* of an object can be tracked statically and its resources can be released when it falls out of scope.

Take for example the short Rust program given below. In L3, space is allocated on the heap for one byte, which is then owned by the variable `heap_object`. Ownership of the heap value is transferred to `transferred_object` in L11, after which `heap_object` can no longer be used (L12). In L15, both the variables `heap_object` and `transferred_object` go out of scope. When a variable falls out of scope, the Rust compiler inserts its drop handler at that point. Drop handlers are Rust's version of destructors and can be specialized to execute multiple operations beyond freeing the memory used to store a variable. Since `transferred_object` is the owner of the `Box<u8>`, when it is dropped the drop handler for `Box` is called as well which initiates the heap deallocation routine.

Rust also allows sharing of variables. Its borrow checker ensures memory safety by

```
1   fn main() {
2
3       let mut heap_object: Box<u8> = Box::new(0);
4
5       {
6           let borrowed_object = heap_object.as_mut();
7           // compute(&mut *heap_object,5); Error: heap_object already borrowed as mutable
8           compute(&mut *borrowed_object,5);
9       } /* borrowed_object is dropped */
10
11      let mut transferred_object = heap_object;
12      // compute(&mut *heap_object,5); Error: borrow of moved value: heap_object
13      compute(&mut *transferred_object,5);
14
15  } /* heap_object and transferred_object are dropped */
16
17  fn compute<T>(x: &mut T, y: T)
18  where T: Mul + Mul<Output = T> + Copy
19  {...}
20
```

Listing 2.1: A sample of Rust code which highlights its important features

statically checking that a variable only has one mutable reference to it at a time. It also ensures that the lifetime of a borrowed variable does not exceed that of the original object to prevent dangling references. In L6, the value in `heap_object` is mutably borrowed, and can only be accessed through the `borrowed_object` variable until it falls out of scope (L9). If we try to access the value through `heap_object` during that time, the borrow checker will return an error (L7).

Another important feature of Rust is *traits*. Traits define an interface, a set of functions that must be implemented by a type. We use *trait bounds* to define a set of shared behavior that generic types must implement. In L18, the `compute()` function requires that its input parameter type (`T`) must implement at least three specified traits. Traits in Rust have zero overhead due to monomorphization; multiple versions of a generic function are compiled, one for each type.

Rust also provides race-free concurrency. Types that can be shared safely among threads have to implement the `Send` trait. Types that can be safely mutated by multiple

threads have to implement the `Sync` trait. If any type doesn't implement those traits, it must be wrapped in types that do, e.g. `Mutex` or `Arc` (atomic reference counted pointer). Otherwise, the compiler will return an error warning of a potential data race.

Rust, for all its benefits, comes with certain drawbacks. The first is the steep learning curve experienced by most new users [22]. Unlike C, which allows a programmer to write low-level code in almost any way they desire, the Rust compiler will generate errors wherever the rules for safe Rust are not followed. Dealing with lifetimes, references, trait bounds and error handling can be daunting at first and requires the programmer to learn as they code.

The Rust compilation process is know to be slower than C and its executable to be larger [23]. One reason is due to Rust being a language still under development, and we expect to see improvement in both areas. Some slowdown is essential for checking that submitted code follows the Rust type system rules. Compilation time can also increase due to monomorphization because more code has to be generated. Monomorphization also leads to increased executable size, along with compiler-inserted code like bounds checks and drop handlers.

2.5 The Theseus Operating System and Intralingual Design

Theseus is a SAS/SPL OS written in Rust which aims to ease state management by clearly defining bounds between OS components [3]. It does so by reducing state spill, the amount of state that different components hold for each other. Theseus uses Rust to realize an intralingual design, which contributes to the reduction in state spill.

Intralingual design aims to: (*i*) Match the compiler's understanding of the OS runtime with the actual execution environment. This is accomplished through the SAS design and avoiding operations where language level information has to be reconstructed extralin-

gually, e.g. in system calls, type and lifetime information is lost at the user/kernel boundary. (*ii*) Move resource management to the compiler. This allows for distributed resource management where entities store their own state and the compiler can track when resources need to be returned to the OS. An example of this in Theseus is the memory management subsystem; when a mapping goes out of scope, its drop handler returns pages to the allocator and removes the corresponding Page Table Entries (PTE). An explicit call to the OS to unmap a virtual memory area is not required as it is in Linux. (*iii*) Enable the compiler to check OS safety and correctness invariants. When implementing an OS intralingually, many OS check can be shifted to compile time rather than runtime. For example, the memory subsystem in Theseus removes runtime checks for executable and writable pages by wrapping mappings created with those flags in special types (`MappedPagesMut` and `MappedPagesExec`).

There are limitations to the intralingual approach. The most prominent is that, except for a few crates that use unsafe Rust code, all other parts of the OS and applications have to be written in safe Rust. Otherwise the safety guarantees do not hold. In addition we have to trust the Rust compiler and low-level Rust crates that we use like `core` and `alloc`, though there is ongoing work to formally verify the Rust compiler and core crates [24].

2.6 Modern NIC Hardware

A NIC manages packets through the use of receive and transmit queues that can be accessed by both the driver and the hardware, as shown in Figure 2.1. For the rest of the paper they'll be referred to as Rx/Tx hardware queues. On initialization, a driver sets up the the queues for the NIC by writing the physical starting address, length, and the head and tail values to the NIC registers. The queues are DMA ring buffers, where each element of the queue is a packet descriptor. Packet descriptors have multiple fields, one of which is the physical address of the packet buffer where data can be read/written to by the NIC and

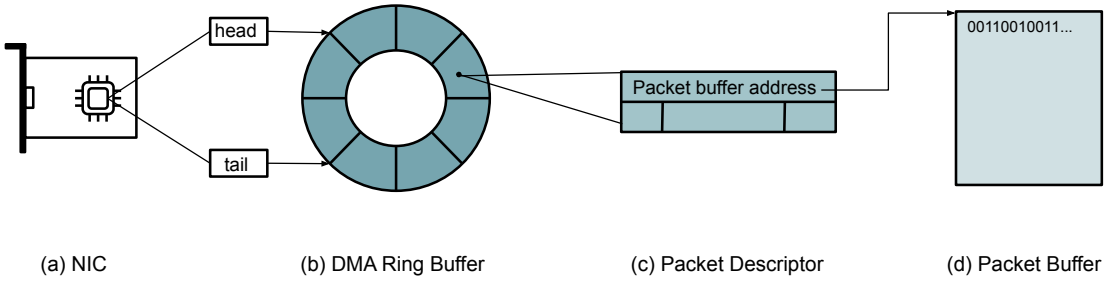

Figure 2.1 : NICs have registers to store the placement of multiple receive and transmit queues in memory. The queues are DMA ring buffers, and each element in the buffer is a packet descriptor. Packet descriptors contain the physical address of a packet buffer.

driver. Communication between the NIC and driver takes place through these queues and the registers that control them. Modern NICs have multiple Rx/Tx queues; the Intel 82599 has 128 of each [25]. NICs also have filtering mechanisms that can send incoming packets for certain MAC or IP addresses to different queues.

NICs are virtualized according to the SR-IOV specification [26]. The purpose is to remove the hypervisor from the packet path, and for multiple VMs to interact with the NIC as if they were running on bare-metal with sole access to the device. SR-IOV enables a single physical PCI device (Physical Function: PF) to appear as multiple virtual PCI devices (Virtual Function: VF) when scanning the PCI bus. The PF has access to all registers of the device. The VFs have access to a subset of registers for operations that can be controlled by the VM. Usually the Rx/Tx hardware queues of a NIC are divided between the VFs. Intel VT-d enables the IOMMU for DMA remapping of memory accesses by VFs; it prevents a VF from accessing memory outside its assigned domain [27]. Usually the driver in the VM is a para-virtualized driver which is aware that it cannot do everything a normal physical driver can. For operations that will affect the whole device, the VM driver will need to message the hypervisor (or whatever other entity is responsible for the PF)

to configure the device. Essentially, the VM, through the VF, has access to the subset of registers needed for manipulating its assigned queues.

Chapter 3

Features of an Intralingual Design

Below we present a set of features that are found consistently in intralingual OS design. The first two features are a result of our first design principle; removing dependency on hardware allows us to have a SAS/SPL system. The second two features result from our second design principle. They allow a developer to shift resource management responsibilities to the compiler.

3.1 Resource Sharing Without Kernel Mediation

Our first feature is the ability of applications to share resources without any interference from the OS. Traditionally, an OS's job is to ensure that all applications can access the system's resources safely and without corrupting the state of any other process. In an intralingual OS we can wrap access to system resources in types that ensure an application cannot misuse them. We can share a heap among multiple processes because Rust prevents any illegal memory accesses. It prevents information being leaked between processes because all variables are initialized before they can be read. In a MAS/MPL OS, sharing memory between two processes would require the kernel to map the underlying frame to both process's virtual address spaces. Safely sharing memory smaller than a page size, like a heap allocation, would be impossible since the granularity of hardware protection is a page.

Our ITC channel also relies on shared memory. The `Sender` and `Receiver` types that

wrap around a shared variable ensure that the memory can only be accessed througn the corresponding `send()` and `receive()` functions. These functions ensure correctness and synchronization between the two participating tasks. In contrast, the seL4 fastpath IPC channel requires a trap into the kernel which then checks if the sending and receiving processes have the capabilities to complete the IPC operation, after which the receiving process's message registers are updated by the kernel.

Lastly, our network driver is designed to allow multiple applications to use the NIC at the same time without any packet passing through the kernel. A NIC device with `N` receive and transmit queues can be safely shared among `N` processes. This is in stark contrast to Linux where every packet passes through the kernel network stack before being passed to an application. Even DPDK, a preferred choice for kernel bypass networking, only allows one process to safely use a NIC port at a time. The entire memory-mapped registers of a NIC are mapped to one user process's address space.

3.2 Direct Access to Hardware by Applications

Through type and memory safety we can provide applications with direct access to hardware. OSes based on the exokernel design philosophy also keep this as an goal but use different techniques. Our network driver allocates receive and transmit queues to requesting applications. These applications have direct access to memory-mapped registers of their assigned queues, and are thus able to manage their own networking. The memory where the NIC registers are located is split into distinct memory regions, each region contains the registers for one queue. The memory regions are wrapped in a type which prevents any illegal accesses to the registers of another queue.

3.3 Bijective Mapping of Software and Hardware Resources

Our third feature is the bijective mapping of each hardware resource to a software variable. The granularity of the hardware resource depends on the granularity of sharing. For example, in the memory management system of Theseus, there is only one `Frame` and `Page` object created for each physical frame and virtual page. The Performance Monitoring Unit (PMU) of Theseus also ensures there is only as many software `Counter` objects as there are actual PMU counters. In our ixgbe driver code, we create one `RxQueue` object for each Rx hardware queue. As the maximum number of queues in the 82599 NIC is 128, the maximum number of `RxQueue` variables in the driver is also 128. Such bijective mapping eases resource management; within the drop handlers of these types, the OS is alerted that they are no longer in use by an application. It also ensures isolation, since rather than copying the software objects, we transfer ownership of them to the application. While the application owns the software object, it is as if it owns the hardware resource as well. Even the kernel cannot access the resource until it revokes access from the application and reclaims it.

3.4 Reference Counting is the Compiler's Responsibility

Our last feature allows us to remove a burden from the shoulder of developers and move the responsibility of reference counting to the compiler. Reference counting in Rust is closely related to drop handlers. When an object wrapped in a reference counted smart pointer, `Rc<>` or `Arc<>`, falls out of scope, the drop handler of the reference counted pointer is automatically called. This drop handler contains the code to check if the pointer is the last outstanding reference to an object, and calls the object's drop handler if it is.

Reference counted smart pointers are a standard Rust type. They are used in many

places within Theseus whenever a resource is shared by threads. Most prominently, they are used within our power management system to implement a generic wakelock. Any device can create a wakelock and register a power down function with it. As long as the OS ensures that applications acquire a copy of the wakelock when they gain access to a device, the device will be powered down when all applications with device access exit.

Chapter 4

A Safe Heap

The heap design of a SAS/SPL OS differs from a typical OS in three major ways. The first is that a heap written in a safe language can automatically avoid many heap errors [28]. We rely on Rust's `alloc` crate as an interface for applications to allocate memory from the heap. Heap-allocated objects are managed by the compiler and have a statically-determined lifetime that prevents use-after-free, double-free, and invalid-free errors. Rust's type system will prevent errors like read or write overflows.

The second is that all kernel and user tasks share the same allocator. There is no distinction in the address spaces of processes, and no portion of memory that is specifically reserved for the kernel. Thus there is no need for separate heaps. This differs from the MAS/MPL approach where the number of heaps is `n+1`, `n` being the number of live processes. We use Linux as an example of extralingual heap design. In that OS, kernel memory is requested from the buddy allocator or the slab allocator. The buddy allocator returns memory in the order of pages. The slab allocator builds on top of the buddy allocator and stores memory for oft-used kernel objects in caches. Userspace memory requests are satisfied by a separate heap which is backed by a virtual memory area in the process's address space.

The third major difference is in the design of the heap itself. In previous work [3], we presented the design and overhead of a safe heap. In such a design the bookkeeping of the heap, which is usually managed through unsafe pointer arithmetic, is replaced with type safe operations. For example, when a pointer is returned by the `dealloc()` function, the

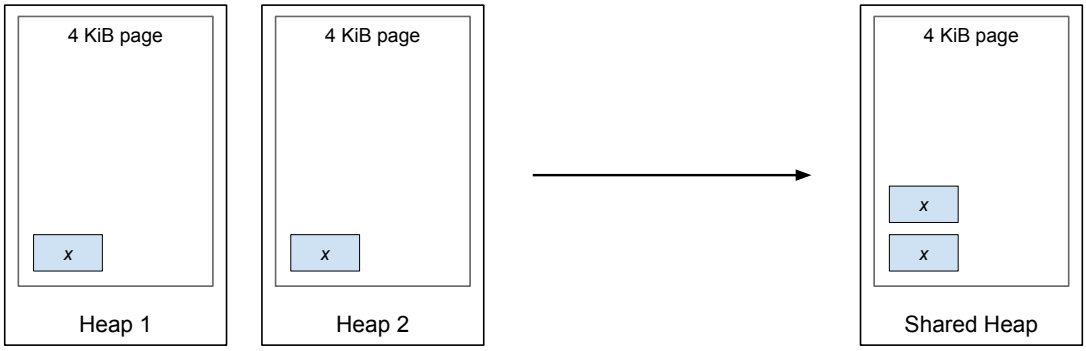

Figure 4.1 : Motivation for studying fragmentation: Sharing a heap can lead to more requests being satisfied from the same memory page.

heap does not take an offset to the pointer to retrieve the object metadata. Instead, the heap searches for the backing mapping that corresponds to that pointer, and uses the mapping type's member functions to access the metadata. Thus ensuring that the mapping is still valid and memory outside of the page limits isn't accessed.

The limitations of such a heap are twofold: (*i*) there is an added overhead to search for the backing mapping and access memory through it, (*ii*) the size of the heap is predetermined since the memory for its bookkeeping data structures has to be allocated statically. Such a heap design can be useful in embedded systems that already have constrained memory requirements.

Here we extend the discussion beyond the internal mechanics of the heap to how sharing a system wide heap between processes can affect fragmentation. Fragmentation is defined as the ratio of the maximum amount of memory in the heap relative to the maximum amount of memory in use by an application [29, 30]. These two events can occur at different points in the application's lifetime. In a SAS/SPL OS, we claim that all processes can share the same heap since no process will be able to access another process's memory illegally or read uninitialized memory. For performance reasons the heap may consist of many sub-

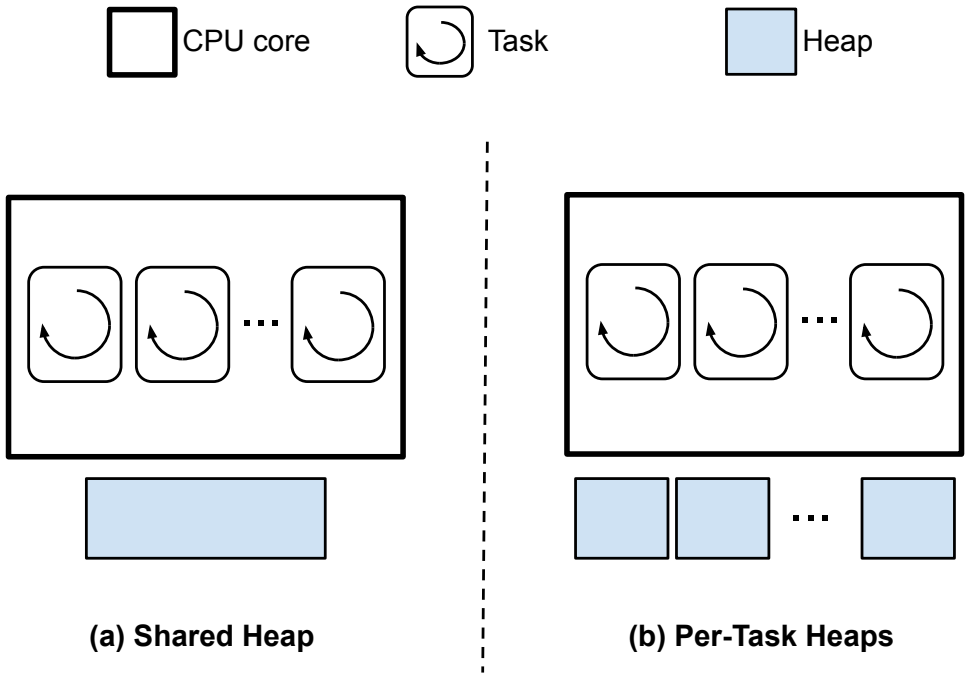

Figure 4.2 : A SAS/SPL OS gives us the option to share a heap among multiple tasks, reducing the memory in the heap and lowering fragmentation.

heaps, but their backing memory is all allocated from the same address space.

The motivating example for considering fragmentation improvements is simple, as shown in Figure 4.1. Consider two tasks starting off with empty heaps. As soon as a task requests x bytes from its heap, the heap will request, at minimum, 1 4KiB page from the OS. With per-task heaps, the total fragmentation would be 4KiB/x. With a shared heap, the total fragmentation would be 2KiB/x, half the previous value. Different heap designs will have different fragmentation values, but the common feature among all heaps is that they must grow at intervals of page sizes, which is almost always greater than the request size.

In Theseus, we implemented per-core heaps for scalability; allocation requests from tasks running on the same core are satisfied by the same heap. Note that in Theseus we do not differentiate between processes and threads; all execution units are tasks. This is in contrast to Linux where each process is given its own heap (Figure 4.2). Threads running on different cores may be assigned to various sub-heaps in the parent process's address space, but threads of different processes will never share a heap.

Each per-core heap in Theseus contains multiple slab allocators; each slab allocator contains multiple slabs from which objects of a certain size class are allocated. Rather than caching kernel objects, we return memory in blocks that are a size of a power of 2. The size of each slab is 8KiB, so whenever memory is added or removed from the heap it is in units of 8 KiB. Since we are not measuring performance, only fragmentation, we remove a slab from the heap as soon as it's not in use, so it is not counted when calculating fragmentation. This allows us to find the minimum fragmentation values. We hypothesize that in a SAS/SPL system, we will see reduced fragmentation since more allocation requests can be satisfied by one heap.

Chapter 5

Shared-Memory Inter-Task Communication

The greatest argument in favor of existing OS designs based on the process abstraction are microkernels. They show that frequent interaction with a higher-privileged kernel does not have to lead to poor performance, and neither do separate address spaces. Microkernel architects achieve this level of performance through careful design of a minimal kernel. The basic design principle of microkernel is the *minimality principle* [31]: only the most essential services are kept within the kernel. Most OS services can be implemented in userspace, e.g. memory management, where there can be many competing implementations. Keeping the kernel small leads to increased communication between services and user applications, and so IPC RTT has become the main way to measure microkernel performance.

Most microkernels implement a fastpath IPC which allows for small messages to be sent very quickly provided a certain set of conditions are met. In our discussion on IPC we consider seL4 as our representative microkernel since it currently has one of the fastest IPC fastpath implementations [32]. The seL4 fastpath is a synchronous IPC mechanism which can be broadly divided into 5 steps: kernel entry, checking of capabilities till the point of no return, updating message registers, direct context switch, and kernel exit. If any of the capability checks fail, the thread switches to the slowpath IPC. The fastpath sends messages through message registers that can either be physical registers, a per-thread memory area known as virtual registers, or both. Direct context switch is a technique by which a thread, once it is ready to send, donates its remaining time slice to the receiving thread without involving the scheduler. The seL4 fastpath has been optimized in C to use a

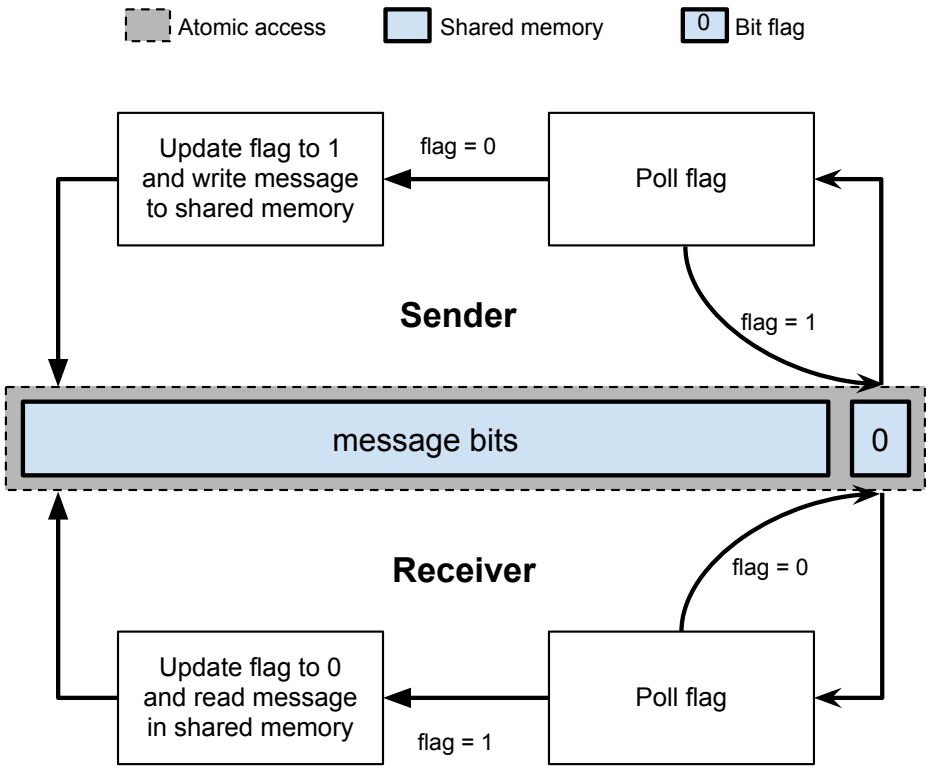

Figure 5.1 : Lightweight shared-memory ITC in a SAS/SPL OS.

minimum number of cycles [33].

Our goal when exploring ITC designs in a SAS/SPL OS was to find one with the fastest RTT, which is why we worked on shared memory. In previous work done jointly, we presented a simple shared-memory ITC mechanism that uses a buffer accessed through atomic operations [3, 34]. This ITC channel polls a 1-bit flag to detect when the buffer can be read and written to. Both the sending and receiving threads are given access to the memory through a special struct, `Sender` or `Receiver`. The methods of the structs ensure

correctness; the sender can only write when the flag is set to 0 and the receiver can only read when the flag is set to 1, as shown in Figure 5.1.

In this work we optimized the shared-memory ITC by studying the output assembly for further improvements. The code was small to begin with but we were able to make changes that led to about 100 cycles reduction in the RTT. One major improvement was changing the return value of the receive function to a simple `Option<u8>` type rather than a `Result<u8, &'static str>`. This helped us to avoid a map operation that converted between the two. In Rust, error handling has been made easy through the interchangeability of these two types, but when measuring cycle counts it is important to carefully select the one that suits the situation. `Option` was preferred because it can encode the fact that no message has been received. We also condensed the code to have only two branches and inlined the functions called within the send and receive path. The final send and receive operations were only about 10 lines of assembly each; the most expensive instruction was the atomic compare and exchange. We believe that this design represents the fastest RTT we can achieve for shared-memory ITC while maintaining synchronization between threads.

Chapter 6

Networking

Improving the performance of the networking subsystem has been a prime area of research for many years. Though the Linux kernel networking stack has gone through improvements, it still has too high a overhead for applications with high incoming packet rates and low tail latency requirements, as found in data center applications. The solution for high performance networking has usually been to remove the kernel from the packet's path (Figure 6.1). One way to implement kernel-bypass networking has been by adding a layer between the driver and the host network stack to reroute packets to the application [35–37]. Another is to shift management of the device completely to the user level using userspace device drivers. DPDK, one of the most popular softwares for high performance networking currently, uses this approach [10]. The last way is for the kernel to grant user applications direct access to the hardware device, where the kernel's only role is to ensure no application can affect the networking behavior of another. This is the exokernel approach and has been accomplished through virtualization hardware [13]. In our design we adopt the last approach but ensure protection using language-level constructs. We use features of modern NICs to allow applications control of their own networking, while also ensuring that the kernel is not involved in the receive and transmission paths of a packet.

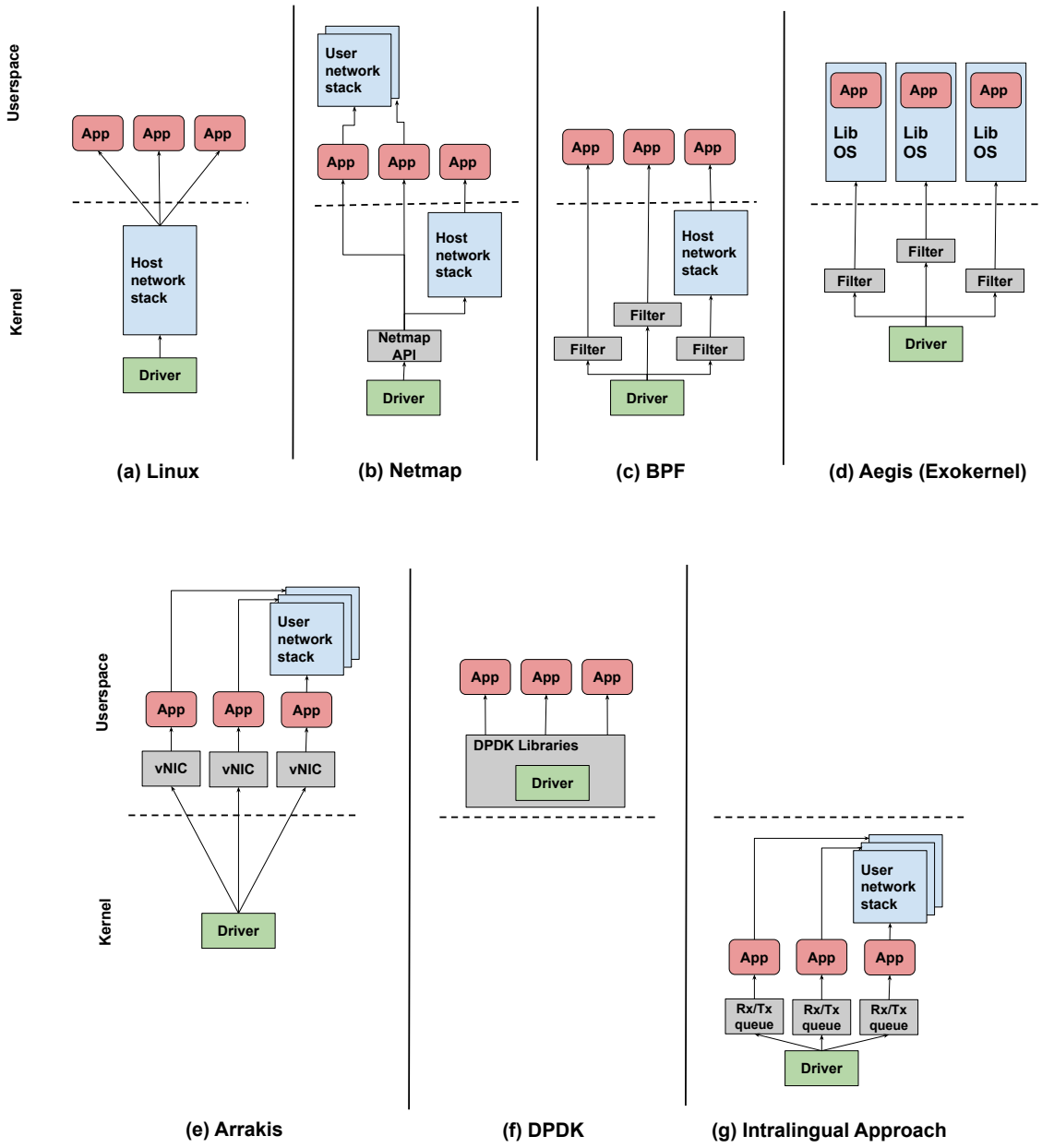

Figure 6.1 : Networking comparison of different systems: (a) Linux's host network stack adds overhead to each packet. (b) Netmap provides an API which allows packets to be redirected from the driver to the application, but it still requires batched system calls to transfer the packets. (c)The Berkeley Packet Filter (BPF) uses filter code injected into the kernel to forward only relevant packets to the user. (d) The exokernel approach multiplexes the NIC between the application level library OSes using software packet filters. (e) Arrakis uses the kernel only for initialization of virtual NICs using SR-IOV, after which user applications have direct access to the memory of their virtual pci device. (f) DPDK shifts all network functionality to userspace by memory mapping the device registers to a process's address space. (g) Our intralingual approach uses type safety and ownership to give users exclusive access to Rx and Tx queues.

```
1       /// Set of registers associated with one receive descriptor queue
2       pub struct RegistersRx {
3           pub rdbal:                          Volatile<u32>,
4           pub rdbah:                          Volatile<u32>,
5           pub rdlen:                          Volatile<u32>,
6           pub dca_rxctrl:                     Volatile<u32>,
7           pub rdh:                            Volatile<u32>,
8           pub srrctl:                         Volatile<u32>,
9           pub rdt:                            Volatile<u32>,
10          _padding1:                          [u8;12],
11          pub rxdctl:                         Volatile<u32>,
12          _padding2:                          [u8;20],
13      } // 64B
14
```

Listing 6.1: A struct representing a set of registers for a receive queue

Our proposed design rests on this fact: each Rx/Tx hardware queue is an independent area in memory with its separate set of memory-mapped registers. Once a NIC has been initialized, send and receive operations on a queue only need access to the queue registers. There are 8 registers for each queue in the 82599 as shown in Listing 6.1. During the initialization procedure, we retrieve the base address of the memory-mapped registers of the device. Then we overlay the register struct at the address where the queues's registers begin, so that we create a `RegistersRx` variable for each hardware Rx queue. The same procedure is repeated with the Tx queues.

In our system we create a software queue object for every hardware queue, thus ensuring a bijective mapping between them (section 3.3). The software queue object is a struct as shown in Listing 6.2,L3, one member of which is a registers struct which gives access to the queue's underlying control registers (Listing 6.2,L6). When an application wants control over its packet processing, it requests a virtual NIC (vNIC) from the OS (Listing 6.2,L21). The OS assigns an IP address and a set of queues to the application. It sets the hardware filter so that all packets for that application are sent to the assigned queues. Then it hands over the vNIC to the application, which now has ownership over both the software queues and the hardware queues they represent. The drop handler ensures that when the vNIC is

```
/// A struct that holds all information for one receive queue.
/// There should be one such object per hardware queue.
pub struct RxQueue {
    pub id: u8,
    /// Registers for this receive queue
    pub regs: RegistersRx,
    /// Receive descriptors
    pub rx_descs: BoxRefMut<MappedPages, [AdvancedReceiveDescriptor]>,
    .
    .
    .
    /// The list of rx buffers in use by the descriptors
    pub rx_bufs_in_use: Vec<ReceiveBuffer>,
    /// Frames that have been received and are ready to be processed
    pub received_frames: VecDeque<ReceivedFrame>,
    pub filter_num: Option<u8,
}

/// A struct that holds a set of queues that have been assigned to an application.
/// When this is dropped, they will be returned to the original NIC.
pub struct VirtualNic {
    rx_queues: Vec<RxQueue>,
    tx_queues: Vec<TxQueue>,
    .
    .
    .
    mac_address: [u8; 6],
    wakelock: Wakelock,
}

```

Listing 6.2: Structs representing a receive queue and a virtual NIC. The code has been shortened and simplified for clarity

dropped, the queues are returned to the OS and the filters are disabled. With type safety we ensure that an application can only access the registers of the vNIC it owns. With ownership we ensure that a Rx/Tx queue can only be used by one task at a time; when the task completes, the queue is returned to the OS.

It is important to note here, the basic idea for how to share a NIC is the same whether we're using language mechanisms or hardware (Figure 6.2); an application/VM needs to be given isolated and direct access to the hardware queue registers. This ensures safety from the CPU side; no process can interfere with the networking of another process because process memory is protected. From the device side, illegal memory accesses are prevented in different ways. With Intel VT-d, the IOMMU makes sure the VF can only access the memory in its protection domain, despite the value a malicious process may have written as

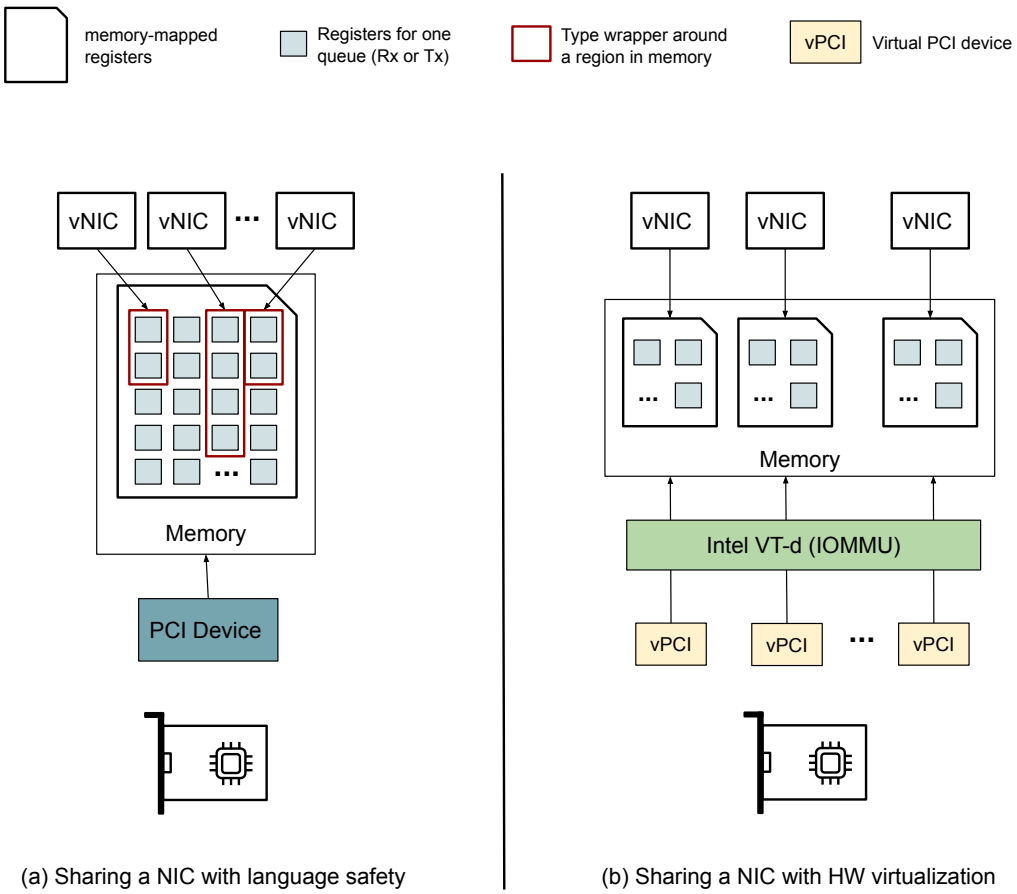

Figure 6.2 : Using type safety, a SAS/SPL OS can give applications direct access to NIC registers, similar to what virtualization hardware does.

the packet buffer address. In our intralingual subsystem, we ensure protection from illegal device DMA accesses by preventing an application from writing directly to the packet descriptors. An application requests a `ReceiveBuffer` or `TransmitBuffer` and passes that on to the function which updates the descriptor with the packet buffer address.

There are overheads of using hardware virtualization: VM exit and entry, an additional address translation for every device memory access through the IOMMU. These overheads can often be minimized; the IOMMU has TLBs as well as caches that store the protection domain of a pci device. So the performance of virtualized drivers is similar to userspace

drivers [2, 13]. The intralingual approach has other advantages as well. It provides a lightweight method for sharing an I/O device among processes, which is useful in systems that don't have hardware virtualization support. It also allows for more fine-grained sharing. In the 82599 NIC, VMs can be assigned either 2, 4 or 8 Rx/Tx queue pairs; these values are predetermined by the hardware [25]. With the intralingual approach, queues can be divided up arbitrarily.

Chapter 7

Compiler-Tracked Power Management

The basis of device runtime Power Management (PM) in systems like Linux and Android is proper reference counting by developers. These systems provide APIs wherein a developer increments a reference count to indicate that she is using the device, and then decrements the reference count when done. The device can be powered off or switched to a low-power state when the reference count is equal to zero. In Android, a *wakelock* is acquired as a handle to the reference count for a device; the wakelock is released to decrement the reference count. Improper reference counting can lead to a device never being switched off, even when there are no users, resulting in a class of insidious energy bugs known as *no-sleep bugs* [38]. In a System-on-Chip (SoC) the negative effects of improper reference counting can go beyond the device itself. In such a system, multiple devices can share clock and power sources to form domains [39]. All the devices in a domain have to be disabled to shut down the domain's clock or power source. A device with a no-sleep bug can prevent other disabled devices in the same domain from being switched to a low-power state.

Previous work has divided no-sleep bugs into 3 main categories [38]:

1. *No-sleep code path:* A wakelock is acquired but then not released in either a normal or exception code path.

2. *No-sleep race condition:* The PM of a device is divided between threads sharing the same wakelock. So, the thread that is supposed to release the wakelock runs before the thread that is supposed to acquire it.

3. *No-sleep dilation:* A long time passes before a wakelock is released due to a process going to sleep, or a large amount of code is placed between acquire and release unnecessarily.

Since no-sleep bugs can lead to extremely high energy consumption as well as crash the system [40], it is time to remove the burden of PM from the developer. We propose an intralingual approach. That is, shift reference counting for devices to the compiler, and any system which relies on reference counting for proper functionality will benefit. Rust provides two types of reference counted pointers, `Rc<>` and `Arc<>`. `Rc<>` cannot be shared among threads and `Arc<>` provides inter-thread safety using atomics. Reference counting in Rust requires the programmer to be aware of the increased reference count, in the form of a reference counted pointer. The default case is that, unless the programmer explicitly stores the `Arc<>` object, the reference count will always be decremented when it is dropped. In other languages, the default case is the increased reference count will always be there, unless explicitly decremented. This is similar to the malloc/free case in C vs Rust's static analysis for the heap. PM can be implemented in drop handlers which can check when a reference counter has dropped to zero, and then transition the device state.

As a proof-of-concept we implemented a `Wakelock` that should be created by any device that wants to implement runtime PM, as seen in Listing 7.1. When creating the `Wakelock`, the device should register a function that will be called when the reference count falls below a specified threshold. The device driver is responsible for passing a clone of the `Wakelock` to any task that wants to use the device. For example, in the ixgbe driver, we ensure that when the NIC is initialized, a permanent `Wakelock` is created that is associated with the NIC. Whenever an application acquires a vNIC, the `Wakelock` is cloned to increase the reference count and stored in the vNIC (Listing 6.2, L28). The application cannot access the `Wakelock` since it is not made public. When a vNIC is dropped, so is the `Wakelock`;

```rust
/// A Wakelock is a wrapper around an atomic reference counted pointer.
/// When a Wakelock is dropped, the reference count is decremented automatically.
pub struct Wakelock {
    refcount: Arc<()>,
    refcount_threshold: usize,
    power_down_fn: PowerDownFn,
}

impl Wakelock {
    /// One Wakelock should be created per device.
    pub fn create_wakelock(refcount_threshold: usize, power_down_fn: PowerDownFn)
        -> Wakelock
    {
        Wakelock {
            refcount: Arc::new(()),
            refcount_threshold: refcount_threshold,
            power_down_fn: power_down_fn
        }
    }

    /// This should be called whenever another task requests access to a device.
    /// The task will own the newly created Wakelock.
    pub fn clone_wakelock(&mut self) -> Wakelock {
        Wakelock {
            refcount: self.refcount.clone(),
            refcount_threshold: self.refcount_threshold,
            power_down_fn: self.power_down_fn
        }
    }

    pub fn refcount(&self) -> usize {
        Arc::strong_count(&self.refcount)
    }
}

impl Drop for Wakelock {
    /// Dropping the Wakelock will automatically call the drop handler for the
    /// Arc<> and decrement the reference count.
    /// The power down function is only called when the reference count falls
    /// below a threshold.
    fn drop(&mut self) {
        if Arc::strong_count(&self.refcount) <= self.refcount_threshold {
            (self.power_down_fn)();
        }
    }
}

```

Listing 7.1: A wakelock can be implemented with automatic reference counting

when the last vNIC is dropped, the power management function is called.

Through intralingual PM we can eliminate no-sleep code paths as well as no-sleep race conditions. In the former case, there is no possibility of a reference count not being decremented, since a wakelock's drop handler will be called when it goes out of scope. At the very latest, when a thread exits, any wakelock it had acquired will be dropped. If an exception occurs, the wakelock will be dropped in the unwinding process. The latter case is prevented by Rust concurrency rules. First, no thread would be able to share a wakelock without a synchronization mechanism; each thread would have to acquire its own wakelock for the same device. Second, a thread could pass ownership of a wakelock to another thread. If it is dropped by the second thread then the first thread could not access that wakelock again and would have to acquire a new one.

We have shifted the responsibility of ensuring thread-safe access to a wakelock and the responsibility of ensuring all code paths in a function release a wakelock to the compiler, thus removing a programming burden from the developer. The developer's remaining responsibility is to minimize the time for which a wakelock is held to eliminate no-sleep dilation bugs.

Chapter 8

Evaluation

Our motivation for exploring intralingual OS design was the many shortcomings of modern OSes including memory access violations, dependency on hardware for isolation and encumbered programming for the OS developer. When designing different parts of the OS, we tried to remove those shortcomings by applying intralingual design principles. Here we discuss how our designs improve upon traditional extralingual OSes in two parts: a qualitative analysis and a quantitative one. The former is necessary because an intralingual design allows us to rethink systems starting from the lowest level of hardware. It becomes difficult to quantify the benefits of changing the hardware MMU in a microbenchmark. Similarly, an intralingual OS presents a completely different way of dealing with certain developer tasks like reference counting, and no equivalent microbenchmark can be designed for another OS as a comparison. It is our goal to present a thorough discussion of the potential benefits of our designs and to quantitatively analyze a subset of those designs for performance benefits.

8.1 Qualitative Analysis of Intralinguality

Within our qualitative analysis, we classify benefits as the result of one of our two design principles: (*i*) removing dependency on hardware for isolation and virtualization. (*ii*) minimizing OS developer effort by moving as many tasks to the compiler as possible.

8.1.1 Removing Dependency on Hardware

Lowered MMU Cost: In a SAS/SPL OS, the MMU can be simplified. ASIDs were added to TLBs to lower the cost of switching address spaces by avoiding a TLB flush. In a SAS/SPL design, we can remove the ASID portion of TLB entries. The number of bits in a TLB that are saved is given by:

$$\textbf{\textit{TLB ASID bits}} = TLB\_entries * bits\_per\_ASID$$

For standard 4KiB page TLBs, the Intel skylake architecture has a 64-entry L1 data TLB, 128-entry L1 instruction TLB and 1,536-entry L2 shared TLB. The ASID for Intel architectures is 12 bits. So by removing ASIDs we could save 20,736 bits, which is equivalent to 124,416 transistors for SRAM(6-transistor cell) TLBs.

MMU cost is also reduced by the removal of per-process page tables. Page tables are created in memory, so the MMU uses a CPU resource. Intel x86_64 architecture uses 4-level page tables [41]. There is only one top level table (PML4) per address space. It has 512 entries; each entry covers 512 GiB allowing for 256 TiB in addressable memory. Each second level table (PDPT), third level table (PD), and fourth level table (PT) covers 512 GiB, 1 GiB, and 2 MiB of contiguous memory respectively. There can be multiple of these three tables depending on the amount of memory mapped in a process. The total number of bytes used by a 4-level page table is given below:

$$\textbf{\textit{page table bytes}} = page\ size * (1 + \#PDPT + \#PD + \#PT)$$

Each table uses 1 page (4 KiB) to store its entries. Many times, when calculating the memory taken by a page table, we consider entries used. That underestimates the space

required since a whole page has to be allocated for a table even if only one entry is filled.

Another area where there is potential improvement in the MMU is the L1 cache type. L1 caches tend to be Virtually-Indexed and Physically-Tagged (VIPT) which means that the TLB is accessed in parallel with the L1 cache, and the physical address is used as part of the cache lookup [17]. This is required due to effects of using multiple address spaces, which include homonyms (the same virtual address points to different physical addresses) and synonyms (more than one virtual address points to the same physical address).

In a SAS/SPL OS, where there is a bijective mapping between frames and virtual pages, homonyms and synonyms would not exist. Thus, we could consider the use of Virtually-Indexed and Virtually-Tagged (VIVT) L1 caches. These caches can potentially speedup cache lookups and increase energy efficiency by referencing the TLB only on an L1 cache miss [17, 42].

Fine-Grained Isolation: The increased isolation granularity of language-based protection mechanisms has been presented in previous works [21, 43]. They have explored how language mechanisms allow the OS to grant access to only the minimum data required, as compared to hardware mechanisms where the protection granularity is a page, usually 4 KiB. It is this fine-grained isolation that enables a SAS/SPL design. We extend the discussion of fine-grained isolation to virtualization of devices. With language-based virtualization, we do not need virtualization hardware and can create virtual devices with our chosen set of resources, bypassing hardware limitations. For example, in the Intel NICs, the number of vNICs that can be created and the number of queues assigned to them is limited. The 82599 10GbE NIC only allows a configuration of 64, 32 or 16 vNICs, each with 2,4 or 8 queue pairs respectively [25]. The X710 40GbE NIC only allows up to 128 vNICs, each with a maximum of 16 queue pairs [44]. With language mechanisms, the

Function name	Number of files where function is called	%driver files	Total references in the Linux kernel
`kref_get`	384	66.7	646
`kref_get_unless_zero`	112	60.7	193
`kref_put`	415	73.2	1007
`kref_put_mutex`	6	66.7	8
`kref_put_lock`	3	33.3	3

Table 8.1 : In the Linux kernel, reference counting is done explicitly by the developer using `kref`. It is used throughout the linux kernel, though majority of the reference counting is done within device drivers. With an intralingual design, we can avoid having an OS developer explicitly increment (`kref_get`) and decrement (`kref_put`) a reference count every time they use a kernel object.

number of vNICs is only limited by the number of queues in the NIC. Each vNIC can have any number of queues, provided they are available.

8.1.2 Minimizing Developer Effort

In our PM subsystem design, we posit that all reference counting should be done by the compiler using the reference counted pointers provided by Rust. To show how we can unburden a developer from explicit reference counting, we count the number of times that a kernel object reference counter (`kref`) is incremented and decremented in the Linux kernel (v5.9.11). The results are shown in Table 8.1. Reference counts are explicitly incremented 839 times and explicitly decremented 1018 times. In an intralingual design, decrementing a wakelock can be completely avoided by the developer since the compiler will insert drop handlers for that. Incrementing a wakelock by individual users can be avoided if the system designer combines incrementing a reference count with device access, as we presented in the NIC driver.

8.2 Microbenchmarks

Here we present the results of microbenchmarks that quantify the benefits of sharing resources without kernel mediation, a feature of intralingual designs. All measurements were taken on an Intel NUC with an i7-6770HQ CPU @ 2.60GHz, unless otherwise specified. The CPU is based off the Intel skylake microarchitecture and has 4 cores (8 hyper-threads). Each CPU core has a private L1 and L2 cache and shares the L3 cache.

8.2.1 Heap Fragmentation Improvements

We measure total fragmentation with increasing number of threads per application in two well known heap benchmarks: threadtest and shbench. In threadtest, tasks allocate and deallocate a total of a hundred million 8-byte objects [29]. In shbench, tasks randomly allocate and deallocate a total of 19 million objects ranging from 1 to 1000 bytes [45]. We modified the benchmark so that all tasks are spawned on the same core and compare the fragmentation in a shared heap with per-task heaps. On every allocation and deallocation, the total allocated and total used bytes across all heaps in use by the application are measured. Note that while the number of tasks that can use a heap differs, the underlying heap implementation is the same.

This experimental setup may seem contrived and counter-intuitive to the usual setup of running one thread on a core, but many recent works have found that CPUs are underutilized in these conditions and have presented ways to share cores among workloads [5].

In Figure 8.1 we can see that in both benchmarks a single shared per-core heap has the same or better fragmentation values than per-task heaps. This improvement is more apparent in the shbench benchmark since the allocation sizes are random. There may be many 8KiB slabs in the heap in which there are only small portions that are actually allocated. The remaining portions would be unused since allocations of that size have not been

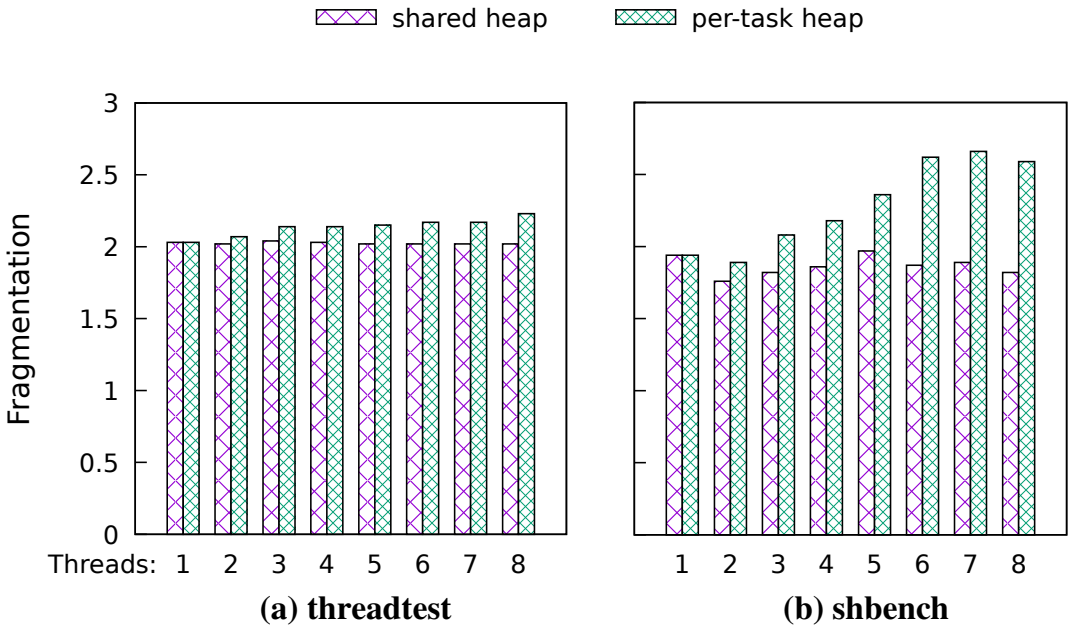

Figure 8.1 : The total fragmentation for per-task heaps increases at a greater rate than for a shared heap as the number of threads running on a core increases.

requested again, and this effect would be exacerbated in per-task heaps.

We would most likely see even better fragmentation results if we had a true single heap for the whole system as compared to per-core heaps, but such a heap design would limit scalability. So any fragmentation improvements are restricted to tasks running on a single core.

8.2.2 Shared-Memory ITC Times

seL4 provides a set of benchmarks that include measurements of one-way IPC times for inter-thread and inter-process communication [46]. In our results we provide the server to client IPC times, since those are slightly lower than the client to server times, and disable meltdown mitigations. For Theseus we ran a benchmark that ping-pongs 1 byte between two tasks using shared-memory ITC. We ran this benchmark for 10,000 iterations and noted

OS	ITC type	Thread core placement	One-way time (cycles)
Theseus	shared-memory	different	320 ± 12
Theseus	shared-memory	hyperthreads	120 ± 19
seL4	fastpath (inter-process)	same	401 ± 4
seL4	fastpath (inter-thread)	same	161 ± 2

Table 8.2 : Shared-memory ITC in a SAS/SPL OS can be faster than a microkernel IPC fastpath. Results are presented as mean $\pm$ standard deviation.

the RTT. We report half the RTT as the one-way ITC time.

We present the mean and standard deviation of different configurations of Theseus ITC and seL4 IPC in Table 8.2. The one-way time for shared-memory ITC is 320 cycles, faster than the seL4 fastpath, on this hardware. The difference between the seL4 inter-thread and inter-process communication times is a write to the CR3 register to update the address space. That can take up to hundreds of cycles [18].

The Theseus ITC times are for tasks running on different cores or on hyperthreads of the same core. When running on different cores, there is an overhead for cache coherency as only the L3 cache is shared. In addition we use atomic operations with sequentially consistent memory orderings which ensure that the store buffers are always flushed on a read or a write. ITC between hyperthreads is much faster since they share the L1 cache.

For a SAS/SPL OS, explicit communication between threads on the same core can be avoided, since a single thread can directly call functions exposed by a server. This approach is taken by the RedLeaf microkernel; they show a cross-domain invocation call can take just 124 cycles [2].

Chapter 9

Related Work

Our work is based off the initial study of intralingual design done by the Theseus OS developers [3]. Though they coined the term, other OS designers have also realized the potential of Rust for OS design beyond type and memory safety. RedLeaf is a recent OS that uses Rust to implement domains that provide access control and fault isolation [2]. We consider their work orthogonal to our own and the techniques they've set out for access control can be used here.

Software Isolation Using Safe Languages: Many experimental OSes have explored SAS/SPL designs [43, 47, 48]. The most prominent of them is Singularity [21], which inspired us to question how OSes would look if we leveraged the power of a modern systems programming language like Rust. Many other systems have used safe languages for lightweight isolation. Examples include kernel extensions [49], Network Function Virtualization [8], database extensions [7] and microservices [50].

Heap Security and Fragmentation: A large number of heap errors such as over-reads, use-after-frees, and double-frees [28] can be eliminated through the use of safe Rust. The Rust `alloc` library contains a collection of heap-allocated types that ensure type and memory safety, and the safe heap design in Theseus extends safety further into the heap bookkeeping mechanisms [34]. In addition to security, reducing fragmentation is one of the main goals when designing a heap. The Hoard allocator [29] presents a per-core heap design that bounds memory consumption. Much of Theseus's heap design is based on Hoard, but we believe that we are the first to explore the fragmentation benefits of a SAS system.

Microkernel Performance: The L4 microkernel family implements a fastpath IPC channel which allows for processes to pass messages through registers when certain conditions are met. They have implemented techniques such as lazy process scheduling and direct process switch that remove the scheduling overhead in their fastpath [31]. Limitations of this approach are that fastpath RTTs are only seen when the two threads run on a single core and that indirect costs of system calls are not avoided [16, 18]. We took inspiration from seL4's fastpath optimization [33] to similarly analyze our shared-memory ITC for any unnecessary instructions.

Kernel Bypass Networking: Bypassing the Linux kernel network stack has become a staple of high-performance networked systems. Netmap [35] showed we can achieve near line-rate throughput when packets can be passed directly to the user application. Similar results are presented by DPDK [10] which provides userspace drivers and libraries to applications for fast packet I/O. Userspace drivers map the memory region of a device to a process's address space, prohibiting sharing a NIC device among processes. Arrakis [13] gave multiple applications direct access to the NIC using hardware virtualization. There have also been previous works that implement a network driver in Rust for increased safety with high performance [2, 9].

Power Management in the OS: There have been multiple solutions proposed to deal with no-sleep bugs. Some solutions propose tools that can analyze source code and alert the developer to code paths that have improper reference counting [38, 40]. Another tool removes PM from the driver to another PM software entity that monitors device register access [39]. Lastly, hardware changes have been proposed that keep track of whether the device is in use or not [39].

Chapter 10

Conclusion

In this work we present the intralingual design of four parts of the OS and the resulting benefits. We show that we can use language features to provide functionality that was previously defaulted to hardware. We also show that OS development can be made easier and less error-prone if certain tasks are left to the compiler. Our work extends the understanding of how an OS would look if we took full advantage of a programming language that provides high-level features with minimal overhead. Future work would explore such a system holistically to prove correctness and safety at the lowest levels where unsafe code is used, as well as to measure the benefits of an intralingual OS for real world applications.

Bibliography

[1] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus, "Deconstructing process isolation," in *Proceedings of the 2006 workshop on Memory system performance and correctness*, pp. 1–10, ACM, 2006.

[2] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev, "Redleaf: Isolation and communication in a safe operating system," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[3] K. Boos, N. Liyanage, R. Ijaz, and L. Zhong, "Theseus: an experiment in operating system structure and state management," in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pp. 1–19, 2020.

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

[5] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pp. 361–378, 2019.

[6] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive scheduling for μsecond-scale tail latency," in *16th {USENIX}*

Symposium on Networked Systems Design and Implementation ({NSDI} 19), pp. 345–360, 2019.

[7] C. Kulkarni, S. Moore, M. Naqvi, T. Zhang, R. Ricci, and R. Stutsman, "Splinter: Bare-metal extensions for multi-tenant low-latency storage," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 627–643, 2018.

[8] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of {NFV}," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 203–216, 2016.

[9] P. Emmerich, S. Ellmann, F. Bonk, A. Egger, E. G. Sánchez-Torija, T. Günzel, S. Di Luzio, A. Obada, M. Stadlmeier, S. Voit, *et al.*, "The case for writing network drivers in high-level programming languages," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 1–13, IEEE, 2019.

[10] L. Foundation, "Data plane development kit (DPDK)," 2015.

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

[12] G. Prekas, M. Kogias, and E. Bugnion, "Zygos: Achieving low tail latency for microsecond-scale networked tasks," in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 325–341, 2017.

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

[14] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[15] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[16] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exception-less system calls.," in *Osdi*, vol. 10, pp. 1–8, 2010.

[17] A. Bhattacharjee and D. Lustig, "Architectural and operating system support for virtual memory," *Synthesis Lectures on Computer Architecture*, vol. 12, no. 5, pp. 1–175, 2017.

[18] Z. Mi, D. Li, Z. Yang, X. Wang, and H. Chen, "Skybridge: Fast and secure inter-process communication for microkernels," in *Proceedings of the Fourteenth EuroSys Conference 2019*, pp. 1–15, 2019.

[19] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 48–59, 2010.

[20] C. Cutler, M. F. Kaashoek, and R. T. Morris, "The benefits and costs of writing a {POSIX} kernel in a high-level language," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 89–105, 2018.

[21] G. C. Hunt and J. R. Larus, "Singularity: rethinking the software stack," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 2, pp. 37–49, 2007.

[22] L. Tung, "Programming language rust's adoption problem: Developers reveal why more aren't using it," Apr 2020. Accessed: 2020-11-30.

[23] T. R. L. Developers, "Frequently asked questions." Accessed: 2020-11-30.

[24] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Rustbelt: Securing the foundations of the rust programming language," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–34, 2017.

[25] Intel, *Intel 82599 10 GbE Controller Datasheet*, 3 2016. Rev. 3.3.

[26] Intel, *Intel 82599 SR-IOV Driver Companion Guide*, 5 2010. Rev. 1.0.

[27] Intel, *Intel Virtualization Technology for Directed I/O*, 6 2019. Rev. 3.1.

[28] S. Silvestro, H. Liu, C. Crosser, Z. Lin, and T. Liu, "Freeguard: A faster secure heap allocator," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2389–2403, 2017.

[29] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *ACM Sigplan Notices*, vol. 35, no. 11, pp. 117–128, 2000.

[30] M. S. Johnstone and P. R. Wilson, "The memory fragmentation problem: Solved?," *ACM Sigplan Notices*, vol. 34, no. 3, pp. 26–36, 1998.

[31] K. Elphinstone and G. Heiser, "From l3 to sel4 what have we learnt in 20 years of l4 microkernels?," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 133–150, 2013.

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

[33] B. Blackham and G. Heiser, "Correct, fast, maintainable: choose any three!," in *Proceedings of the asia-pacific workshop on systems*, pp. 1–7, 2012.

[34] K. Boos, *Theseus: Rethinking Operating Systems Structure and State Management*. PhD thesis, Rice University, 2020.

[35] L. Rizzo, "Netmap: a novel framework for fast packet i/o," in *21st USENIX Security Symposium (USENIX Security 12)*, pp. 101–112, 2012.

[36] S. McCanne and V. Jacobson, "The bsd packet filter: A new architecture for user-level packet capture.," in *USENIX winter*, vol. 46, 1993.

[37] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr, "Exokernel: An operating system architecture for application-level resource management," *ACM SIGOPS Operating Systems Review*, vol. 29, no. 5, pp. 251–266, 1995.

[38] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pp. 267–280, 2012.

[39] C. Xu, F. X. Lin, Y. Wang, and L. Zhong, "Automated os-level device runtime power management," *ACM SIGPLAN Notices*, vol. 50, no. 4, pp. 239–252, 2015.

[40] J. Mao, Y. Chen, Q. Xiao, and Y. Shi, "Rid: finding reference count bugs with inconsistent path pair checking," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 531–544, 2016.

[41] Intel, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, 05 2018.

[42] H. Yoon and G. S. Sohi, "Revisiting virtual l1 caches: A practical design using dynamic synonym remapping," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 212–224, IEEE, 2016.

[43] C. Hawblitzel and T. Von Eicken, "A case for language-based protection," tech. rep., Cornell University, 1998.

[44] Intel, *Intel Ethernet Controller X710/XXV710/XL710 Datasheet*, 10 2020. Rev. 3.7.

[45] M. Inc., "Microquill smartheap 4.0 benchmark." `http://microquill.com/`.

[46] N. I. Australia, "sel4bench." `https://github.com/seL4/sel4bench`.

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

[48] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska, "Sharing and protection in a single-address-space operating system," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 4, pp. 271–307, 1994.

[49] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, “Extensibility safety and performance in the spin operating system,” in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pp. 267–283, 1995.

[50] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, “Putting the" micro" back in microservice,” in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pp. 645–650, 2018.