A Performance Evaluation of Rump Kernels as a Multi-server OS Building Block on seL4

Kevin Elphinstone
UNSW and Data61
k.elphinstone@unsw.edu.au

Amirreza Zarrabi
UNSW and Data61
a.zarrabi@unsw.edu.au

Kent Mcleod
Data61
Kent.Mcleod@data61.csiro.au

Gernot Heiser
UNSW and Data61
gernot@unsw.edu.au

ABSTRACT

In the paper, we argue that it is worthwhile to revisit building microkernel-based multiserver operating systems, and introduce a multiserver OS architecture. We argue that recent formal verification of microkernels provides a compelling platform for constructing general purpose systems, and that existing systems are not appropriate to take advantage of a formally verified microkernel.

Our vision is of mostly-POSIX multiserver systems based on *rump kernels*, with a small set of fundamental services and frameworks. We expect the approach to provide a balance between componentisation, development effort, and legacy system compatibility.

We present our initial efforts with a promising performance evaluation of a rump kernel running on seL4.

ACM Reference format:
Kevin Elphinstone, Amirreza Zarrabi, Kent Mcleod, and Gernot Heiser.2017. A Performance Evaluation of Rump Kernels as a Multiserver OS Building Block on seL4. In *Proceedings of APSys '17, Mumbai, India, September 2, 2017,* 8 pages.
DOI: 10.1145/3124680.3124727

1 INTRODUCTION

Dating back to the 1970s, the vision of microkernel-based operating systems has been of a more modular, extensible, reliable, and secure operating system (OS) compared monolithic OSes [Brinch Hansen 1970].

The microkernel-based multiserver is one embodiment of the microkernel vision. In such a system, OS-services are just applications that provide services to other applications. The services-as-an-application model intrinsically provides strong modularity via enforced application boundaries, extensibility via application replacement or addition, improved reliability by fault containment and service restart [Herder et al. 2006], and enhanced security. While the last remains challenging, it becomes more tractable via a reduction in the trusted computing base (TCB) and the establishment of clear trust dependencies. This has been advocated at least since the Orange Book [DoD], and explored in the context of separation kernels [Rushby 1984], and microkernels [Hohmuth et al. 2004].

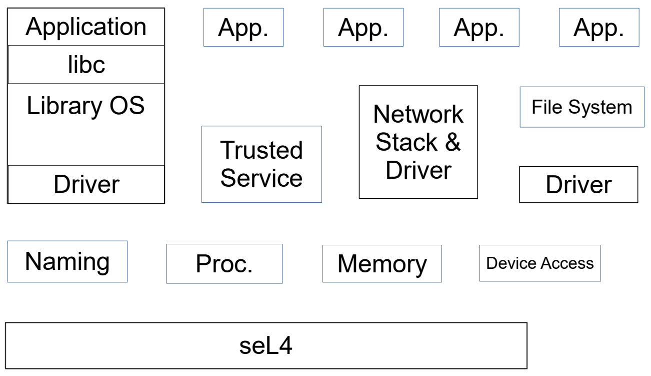

Figure 1: An example multi-server architecture.

Our vision is to enable the construction of multiserver systems on seL4 that can leverage the formal guarantees of the kernel with only modest effort. To this end we envisage a distributed (but local) framework of co-operative protocols and a small number of fundamental services that enables a wide variety of multiservers to be constructed without being tied to a specific instance or configuration of a system, as shown in Figure 1.

APSys '17, Mumbai, India

DOI: 10.1145/3124680.3124727

The fundamental multiserver services shown in Figure 1 consist of simple low-level services to locate high-level services (naming), coordinate and manage application processes (proc), arbitrate access to physical memory (memory), and manage access to devices for device drivers (device access). These fundamental services define a framework and protocols that facilitate the creation of services to raise the level of abstraction up to applications.

Beyond the low-level services, the system is free to meet domain-specific architectural needs. In general, systems are composed of only the services required and the *principle of least authority* (POLA) is applied pervasively to ensure a robust and secure system in the presence of a fault or exploit in a untrustworthy component of the system. Four scenarios are shown in Figure 1.

(1) A unikernel-like application, vertically integrating the application, libraries, a library OS, and driver, thus forming an isolated software stack spanning driver to application for efficiency.
(2) A trusted service with only a dependence on the lower-level fundamental services which form the trusted computing base (TCB). The trusted service may be developed using rigorous application of formal methods or more traditional methods, though either way, it relies on the isolation guarantees provided by seL4 to preserve security.
(3) A combined network stack and driver that provides network services to applications.
(4) A software stack of applications, a shared file service, and potentially shared storage driver. This stack features "horizontal" protection boundaries to improve robustness and security.

Our long-term focus is on the frameworks and services that enable flexible system construction, together with future examination of raising the formal reasoning foundation beyond seL4. However, taking a practical perspective, a sufficiently complex proof-of-concept multiserver is required to validate the system with real-world use cases. To this end, we are using *rump kernels* to provide (potentially shared) OS services and legacy POSIX support to applications without having to re-engineer or implement large code bases [Kantee 2012]. Rump kernels are subsets of the NetBSD kernel running in alternative environments (see Section 3 for details). This paper presents our port of the Rumprun unikernel (a specific instance of a rump kernel) to seL4 and evaluates its performance. This work provides a baseline characterisation of a rump kernel on seL4, which we hope will form a flexible building block for future multiserver systems.

We argue it is time to revisit a microkernel-based multiserver OS. Microkernels can now provide a *guarantee* of correctness and security [Klein et al. 2014]; together with library OSes they can provide legacy POSIX compatibility with low engineering effort. The next section details our argument in the context of related work.

2 MOTIVATION AND RELATED WORK

In this section we review related work and propose practical challenges that need addressing in addition to any longer-term goal of formal verification or reasoning about low-level services.

2.1 Verification

The successful application of formal verification to the *seL4* microkernel [Klein et al. 2009] has established a clear value proposition for basing a system on a microkernel. seL4 provides mathematical guarantees of functional correctness, and of some security properties (integrity, access-based confidentiality, and a formulation of the absence of information flow [Murray et al. 2013; Sewell et al. 2011]). Unlike previous microkernels, a multiserver OS on seL4 can rely on correctness and security properties of the kernel that rule out a broad class of reliability and security problems.

In contrast to previous microkernel-based systems (for example, Mach or Windows NT [Solomon 1998], hybridisation, i.e. the performance-motivated addition of functionality to the kernel instead of providing it through user-level servers, has a clear detrimental consequence: adding code to the kernel invalidates any proof guarantees, and may be sufficiently invasive to make proof re-establishment impractical.

In theory, one could retain microkernel verification guarantees by architecting a system around a large OS server (e.g. the UNIX server in Mach or even a guest OS in a virtual machine). However, this places a large software component in the TCB of all dependent applications (if not the entire system), as it creates false dependencies for applications only dependent on a small subset of the services. Thus the benefits of verified microkernel are undermined by an unnecessarily bloated TCB.

Challenge: OS services must be functionally decomposed, to avoid expanding the TCB via false dependencies.

Challenge: Global OS services should be avoided where possible, and any global OS service must aim for verifiability so as to not undermine verification guarantees of the underlying kernel.

2.2 POSIX

The desire to retain API compatibility in order to provide a gradual evolution path for legacy software is a recurring theme, both in industry and research. Drawbridge examines the problem by proposing a simplified API that supports library OSes, and consequently, legacy applications [Porter et al. 2011]. However, the services required by the library

OS are provided by a centralised, rich API of the underlying kernel, resulting in a large TCB. Howell et al. [2013] build upon Drawbridge to support POSIX applications on Microsoft Windows. This API diversity is aligned with our goals, but still relies on a monolithic, feature-rich OS to provide core services.

(Re)building operating systems with low effort has also been approached by many. The SawMill project attempted to evolve Linux into a multiserver OS by extracting parts of Linux into services [Gefflaut et al. 2000]. The project succeeded in extracting a file system, and identified distribution issues as one of the main challenges in building a multiserver system.

The narrower (but practically very important) problem of re-using existing device drivers has been addressed by hosting them either in individual (minimally-configured) guest OSes, each running their own virtual machine (VM) [LeVasseur et al. 2004], or a common administrative VM [Fraser et al. 2004]. Again, these approaches suffer from a large TCB, in this case a whole Linux kernel.

Challenge: Existing approaches to simplifying the provision of legacy OS services and code are still dependent on large monolithic OS components.

2.3 Implicit Authority

Graphene has similarities to our vision of constructing a multiserver OS from a library OS [Tsai et al. 2014]. It builds a distributed POSIX abstraction between co-operating unikernel instances within a sandbox. However, Graphene assumes an underlying implementation of OS services, accessible via a Drawbridge-like platform adaption layer (PAL), and consequently assumes that the underlying (monolithic) OS services have complete access to the sandbox and Graphene picoprocesses in order to provide the services. This assumption is not true in a small microkernel environment where the OS services are unprivileged user-level processes. In a multiserver environment, OS services only have access to explicitly-shared memory between services and clients. Implementing a system call, such as `fork()`, that assumes access to the client's complete memory range and "in-kernel" book-keeping state becomes challenging, even in scenarios where servers violate the principle of least authority (POLA) and have complete authority to manipulate their clients. In a system designed with POLA, `fork()` is problematic as it relies on access to all state required to create a child of the parent. In a multiserver system, this state is distributed across multiple servers of varying trustworthiness, and thus `fork()` becomes a distributed coordination and snapshot problem involving potentially uncooperative participants.

Challenge: Legacy APIs that assume a central implementation and global authority should be avoided..

Graphene also does not explicitly address memory sharing, as would be required for efficient implementation of POSIX `read()/write()` semantics to avoid extra copying between clients and servers. Instead, it falls back on the drawbridge ABI's implicit authority to access the client's memory.

Challenge: OS services must not rely on total authority over their clients.

Graphene's security model is also unsuitable for implementing a trusted service shared between mutually distrusting clients, such as an encryption or key storage service. This is a result of Graphene's strict sandboxing model, where a service is either inside the sandbox (and at the mercy of its clients), or outside and inaccessible by clients.

VirtuOS [Nikolaev and Back 2013] decomposes an operating system into multiple service domains for a single process, and thus encounters the baked-in assumption in POSIX of system calls having implicit authority over a process. VirtuOS also avoids tackling the distributed shared-memory coordination problem required for efficient POSIX `read()/write()`, and therefore has to introduce an extra memory copy to an intermediate buffer in the system call path. It also does not support `mmap()` of the storage service, due to lack of distributed memory management.

Our goal is build the distributed framework required to enable efficient implementation of POSIX I/O without resorting to either a monolithic VM and I/O service in the case of Graphene, or the overhead of extra copying of VirtuOS. We expect to revisit the work of the SawMill VM framework, which tackled distributed VM within a multiserver environment, but without the benefit of an underlying capability model for fine-grained authorisation [Aron et al. 2001].

Challenge: Virtual memory must be a distributed framework capable of securely sharing memory where and only where required.

2.4 Library OSes

The concept of a Library OS was originally introduced in 1990's as a per-application customisation method [Anderson 1992] but it lacked of legacy compatibility. Library OSes have recently been examined as a way to provide improved isolation between subsystems [Porter et al. 2011], completely decoupling the application's OS personality from the host OS. A small abstraction layer connects the library OS to the underlying platform.

Unikernels [Madhavapeddy et al. 2013], designed for Cloud services, are an extreme specialisation of a Library OS, treating each service as a single-purpose application, compiled into a standalone, bootable image. This image includes the entire software stack (system libraries, language runtime, and applications) while stripping away unused functionality at compile-time. This potentially improves system efficiency, by eliminating duplication between the guest OS and the hypervisor and avoiding user-kernel boundary crossing. With the exception of Graphene described above, the design only targets single-process applications and omits some of the commonly used multiprocess abstractions (e.g. `fork()`, signals, or System-V IPC).

Graphene-SGX removes the need to trust the underlying OS by utilising Intel's SGX extensions to run library OSes in SGX enclaves [Tsai et al. 2017]. Graphene-SGX uses techniques that are complementary to our vision, such as shielding applications that are interacting with an untrusted service using runtime checks. We envisage utilising shielding where appropriate, however, given trusted fundamental services, we also envisage the equivalent of software-implemented formally-guaranteed enclaves, with consequent improved performance by not needing shielding when appropriate, and not requiring encrypted communication between co-operating enclaves.

The *rump kernel* project [Kantee 2012] aims to make the NetBSD OS a portable software stack, i.e. a library OS targeting a variety of execution environments. It provides support for running unmodified kernel code in user processes or on bare metal, and grouping kernel code into components according to their functionality and dependencies to tailor the library OS to specific applications. We hope to leverage rump kernels to validate the projects claim of enabling "you to build the software stack you need without forcing you to reinvent the wheels".

Challenge: Leverage existing library OSes to provide POSIX OS services and compatibility.

2.5 Summary

We have outlined recent work that tackles similar problems to what we expect to tackle in building a multiserver on seL4. In our analysis we have identified challenges to either applying recent work, or overcoming its limitations. Addressing the challenges in their entirety is an ambitious vision. As a first step towards building a multiserver OS using existing components, we ported the Rumprun bare-metal kernel (BMK) (a unikernel) [Kantee 2012] to seL4, including the underlying drivers (in our case we examined networking). This first step serves several purposes: (1) It shows that OS services can be quickly developed from unikernels with modest effort, (2) it serves as a performance baseline for evaluating the performance of unikernels as services, (3) it provides a realistic environment for developing the distributed protocols required to coordinate multiserver systems.

3 RUMPRUN UNIKERNEL

In this section, we provide an introduction to the Rumprun Unikernel and surrounding concepts as described by Kantee [2012]. Figure 2 provides an architectural overview of Rump kernels in general.

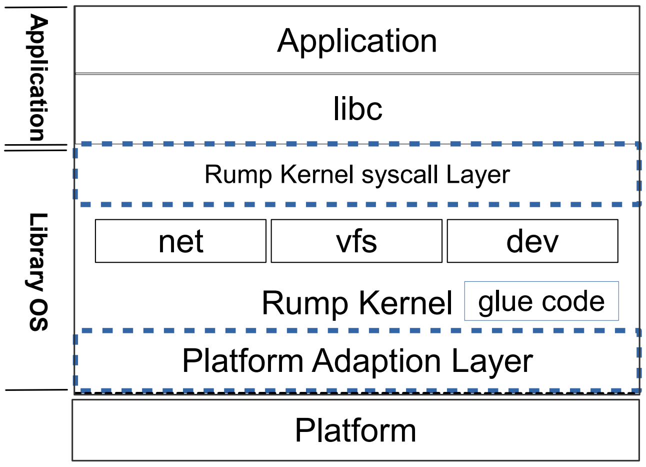

Figure 2: Rump kernel overview.

Anykernel is defined as kernel code that "allows the kernel's unmodified drivers to run in various configurations such as application libraries and microkernel style servers, and also part of a monolithic kernel". In the anykernel context, *drivers* also include file systems and network stacks.

With reference to the Figure 2, the anykernel software components consist the virtual file system layer (VFS), the BSD networking stack (net), and the associated device driver (driver).

Rump kernels are an instance of anykernel code combined with glue code and a platform adaption layer that provides an execution environment for anykernel code that is compatible with the what would normally be the surrounding NetBSD kernel. The rump kernel can be tailored to a specific application by including or excluding kernel functionality from the anykernel subset of NetBSD.

The glue code and platform adaption layer provide an interface to threads, synchronisation, scheduling, memory allocation, interrupt delivery and device access management. Depending on the adaption layer, a rump kernel can run as a normal process hosted by an OS. For example, the threads interface would be implemented by the POSIX thread support of the host OS. Alternatively, a rump kernel my run as a native software stack where the platform adaption layer implements thread support natively.

The rump system call layer provides access to the rump kernel from a client. Access can be via direct function calls in the can the client is linked with the rump kernel to use it as a library OS, or access can be via remote procedure calls in the case of a rump kernel being used as a server on a microkernel.

Rumprun BMK (bare metal kernel) is a specific configuration of an instance of a rump kernel with a platform adaption layer that supports running the anykernel natively, i.e. on bare metal or within a virtual machine. Rumprun BMK unikernel provides services to a client by acting as a library OS, thus forming a unikernel.

4 SEL4 AS RUMPRUN PLATFORM LAYER

As an initial step towards our vision, we modified the Rumprun x86 platform layer to execute on seL4, thus allowing us to run Rumprun POSIX applications, a rump kernel instance, and the shared Rumprun runtime layer. Our goal was to evaluate performance of the library OS and the effort required to run it on seL4. It also provides a performance baseline implementation of a library OS on seL4 prior to introducing any future frameworks and services.

We started with the Rumprun BMK platform and modified it to use seL4 abstractions rather than low-level hardware mechanisms. These changes can be grouped into the following categories:

- **Execution contexts** Rumprun provides a low-level non-preemptive threading implementation that manages thread resources and scheduling assuming a single memory address space. A rump kernel binds thread contexts to virtual CPUs. On seL4 we bind this virtual CPU to a single seL4 thread resulting in a co-operative threading strategy where Rumprun retains its own scheduling policy and implementation.
- **Interrupt delivery** On the Rumprun BMK, a hardware interrupt will preempt the CPU and run an interrupt handler on the current thread's context that defers actual interrupt processing by recording the interrupt number and marking the Rumprun interrupt handling thread as runnable. The handler then resumes the previously running Rumprun thread. The interrupt handling thread will eventually be scheduled and then handle the interrupt.

  seL4 delivers interrupts by signalling a Notification object that is associated with the interrupt. To receive an interrupt, a seL4 thread waits for a Notification by waiting (blocking) using an inter-process communication primitive (IPC). Upon an interrupt, the seL4 thread unblocks, handles the interrupt, acknowledges the device, and waits for the next interrupt.

  On the Rumprun seL4 platform, we use a dedicated seL4 thread running at a higher priority than the Rumprun thread for receiving interrupts. This thread then performs the same behaviour as the Rumprun bare metal interrupt handler, marking the Rumprun interrupt thread as runnable and storing the source interrupt number for later inspection by the rump kernel.
- **Synchronisation** Synchronisation between Rumprun threads (recall they are multiplexed on a single seL4 thread) are managed by Rumprun-implemented sync primitives. We handle synchronisation between the two seL4 threads via binary-semaphore-based signalling using seL4 Notification objects (effectively in-kernel blocked queues) provided by seL4 thread synchronisation libraries.
- **Memory management** On Rumprun BMK, all virtual memory is directly mapped to physical memory to enable native execution. On the seL4 platform, Rumprun has a virtual address space and we use the seL4 user-mode memory allocator libraries to map in a contiguous range of memory which is passed on to Rumprun to manage, this giving the illusion of running natively.

  Memory for DMA is managed by a seL4 user-level DMA library and MMIO memory is mapped in on-demand to available virtual addresses.
- **Low level device access** The primary hardware devices that Rumprun requires is a clock source, timeouts and console output. The BMK platform provides this through its own device drivers. We mostly do the same on the seL4 platform, using the x86 *time stamp counter* (TSC) for a monotonically increasing timestamp, COM serial ports for console output, and a programmable timer for timeouts. We experimented with both PIT and HPET timer devices. Other hardware, such as PCI-based devices are accessed through hardware management support libraries that provide interrupt management, MMIO access, and DMA.

5 EVALUATION

Our evaluation examines two areas of interest: the performance of the rump kernel on seL4 and the amount of the code changes required to support rump on seL4. To evaluate performance, we chose an I/O intensive workload where the system consists of a rump unikernel focused on networking that supports an `Iperf 3.1.3` server, including a device driver directly managing the Ethernet card as illustrated in Figure 3, running on the seL4 microkernel. The seL4 specific code changes are limited to the platform adaption layer, with the above layers being a relatively standard Rumprun unikernel configured for networking.

This initial evaluation is an important step toward our vision as it sets performance expectations of our approach on "native" seL4 before the addition of any more general framework and services. It also provides the opportunity to tune our implementation on seL4 compared to the native BMK Rumprun unikernel.

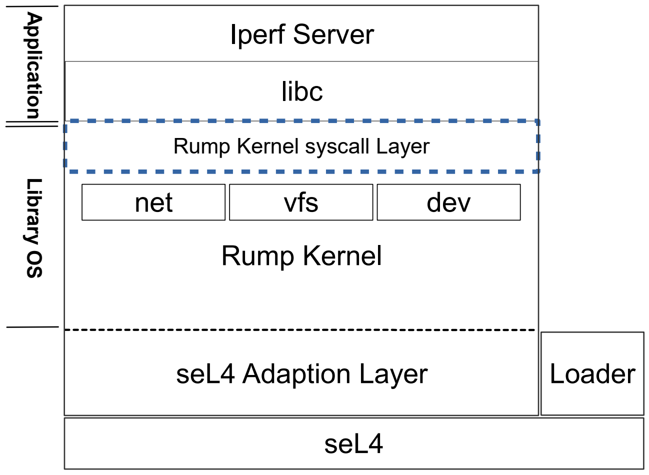

Figure 3: Block diagram of system architecture under test.

Iperf 3.1.3 [Dugan et al.] is a networking benchmark for measuring maximum achievable bandwidth on IP networks. We use Iperf to measure throughput of the rump kernel networking stack as well as CPU utilization at the same network throughput. We measure CPU utilisation to compare performance as maximum networking card throughput is reached before the maximum CPU utilisation.

We benchmark our implementation to measure the overhead introduced by seL4 and the performance difference between the rump kernel drivers on seL4 compared to other systems performing natively. Additionally, we evaluate the amount of required work to use rump kernels on seL4 based on degree of changes in source code.

5.1 Experimental Setup

We run benchmarks across three different systems/software stack along with our Rumprun unikernel on mainline seL4:

- **Rumprun on bare-metal kernel** A standard Rumprun unikernel built for bare-metal x86 32-bit architecture with same configuration as for the seL4 version.
- **Native NetBSD** A NetBSD release kernel built from the same sources as the unikernel in single user mode.
- **Linux** An up to date Debian distribution configured to boot in single user mode.

An Iperf server is started on the target machine, and a client is used on load generation machine to configure, start and stop the benchmark and to collect results. The machines are connected to each other using a HP ProCurve 2708 Switch and 1Gb/s network card interfaces:

- **Target machine** 3.4 GHz Sandy Bridge i7-2600 processor in 32-bit mode with Intel Corporation 82574L Gigabit Network Connection Ethernet.
- **Load generator machine** 3.1 GHz Haswell E3-1220 processor in 64-bit mode with PCIe Broadcom Limited NetXtreme BCM5720 Gigabit Ethernet.

Measurements were taken in steps of 180Mb/s from 180Mb/s up to 900Mb/s. Each test ran for 20 seconds where the first 10 seconds was ignored to allow for the TCP sockets to reach a stable state. Each sample was run 5 times, and error bars indicate standard deviation. Where error bars are not shown it is because the standard deviation is too low. The socket size is set to 87380 (85KiB). For receiving and sending, the TCP congestion algorithm is set to cubic and a 128KiB buffer is used for each read and write request. In all experiments, hyper-threading support was disabled and number of enabled cores set to one in the BIOS settings (as the verified seL4 is a uniprocessor variant).

Measuring CPU Utilisation. In all experiments, CPU utilisation was calculated by measuring system idle time. For Linux, this was achieved by reading the /proc/uptime file while in NetBSD, sysctl syscall was called. The instrumentation for Rumprun on BMK and seL4 runs in the idle thread meaning that it runs when the processor would otherwise be idle and adds no overhead.

5.2 Performance Results

We used Iperf to generate TCP load by sending packets to the target machine and measured the CPU utilisation for each system as shown in Fig. 4. Based in initial results which showed a sensitivity to the hardware interrupt delivery mechanism, we also measured different hardware interrupt mechanisms implemented in seL4.

The Rumprun BMK has lower utilisation compared to NetBSD and seL4 as it represents the bare minimum implementation of the Rumprun platform layer with no preemption or protection boundaries. The Linux has lowest utilisation but it utilises a different network card driver compared to NetBSD which makes it difficult to draw direct comparisons. Additionally, Linux also uses dynamic interrupt throttling while NetBSD has a static throttle rate. This could explain the Linux data points which have a high variance in samples due to the switching between different interrupt throttle rates. We see a slight reduction in CPU utilisation for the seL4 variant that does not use the PIC.

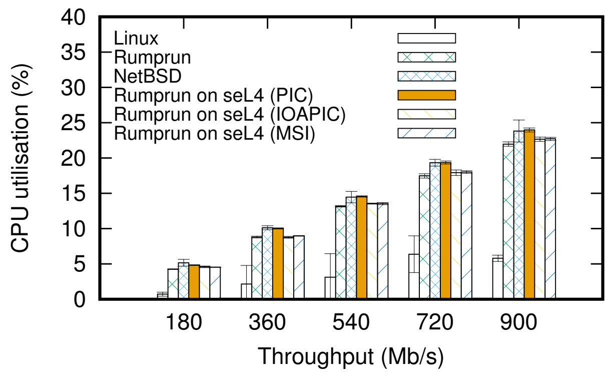

Figure 4: Thread utilisation under applied TCP load.

Examining seL4 overhead more closely, Fig. 5 shows the breakdown of the overhead for each method inside the seL4 kernel. For the PIC, the longest task is handling IO Port operations which are privileged instructions and thus the kernel has to be entered to access ports. In this case we use the IO ports to program an interval timer. Receiving interrupts and delivering them to the user process and then handling the acknowledgement of the interrupt make up about 50% of kernel overhead.

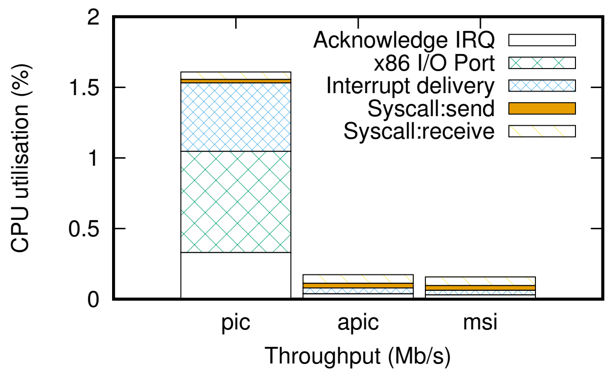

Figure 5: seL4 in-kernel CPU utilisation overhead breakdown.

Additionally, we used HPET for programming timeouts which uses MMIO for both IOAPIC and MSI interrupt delivery methods to restrict access to IO ports. In both cases the CPU utilisation reduced to less than 0.3% of the overall CPU available. The remainder of the overhead is attributed to send and receive syscalls which are used to provide synchronisation primitives.

5.3 Code changes

To evaluate the engineering effort of porting the Rumprun unikernel to seL4, we examine the lines of code removed and added to the Rumprun unikernel. We excluded the seL4 *roottask* from the analysis as it is orthogonal to Rumprun's seL4 support. The roottask acts as loader for the rump kernel and will eventually be replaced by our fundamental services.

Code change	Lines
Code removed	1402
Code added	677

Table 1: Rumprun changes to support seL4

Given we re-used and modified code from Rumprun BMK, we used a white-space indifferent `diff` to measure code removed and added after stripping comments and blank lines. We then counted lines removed and added as summarised in Table 1.

The Rumprun unikernel changes involving removing code that manages low-level hardware that is now managed by seL4. Specifically, code removed included GDT management, low-level trap and boot code, multiboot header parsing, and clock utilities. This was replaced by initialisation code that sets up seL4 threads, notifications, together with a timer and console driver for our platform. Code changes included modifying interrupt management, PCI and DMA management and address translation used for I/O.

We see a net negative change in the number of lines of code in Rumprun when targeting the seL4 platform. Even ignoring code removed, only 677 lines of code were added to support seL4.

6 CONCLUSIONS

We have argued for revisiting microkernel-based multiserver OSes. We described a vision of a flexible multiserver environment supported by a small number of fundamental services and frameworks. We identified a number of issues with related work that require addressing to succeed in our vision. We have taken then first step toward a multiserver by evaluating the Rumprun unikernel as a building block for multiserver systems. We showed that:

- A Rumprun unikernel was supported on seL4 with little performance impact compared to a bare metal version. We even show improved performance in configurations where seL4 used more efficient hardware mechanisms. Overall, the performance exceeded that of native NetBSD.
- The effort required to port the Rumprun unikernel to seL4 was relatively small and it is now possible to reuse many of the existing rump kernel modules in the context of a unikernel running on seL4.

Given this promising first step, we can now focus on the core issues of providing a secure distributed framework for supporting OS services together with a small number of correct fundamental services.

REFERENCES

Thomas E Anderson. The case for application-specific operating systems. In *Workshop on Workstation Operating Systems*, 1992.

Mohit Aron, Yoonho Park, Trent Jaeger, Jochen Liedtke, Kevin Elphinstone, and Luke Deller. The SawMill framework for VM diversity. In *Proceedings of the 6th Asia-Pacific Computer Systems Architecture Conference*, pages 3–10, Gold Coast, Australia, January 2001.

Per Brinch Hansen. The nucleus of a multiprogramming operating system. *Communications of the ACM*, 13:238–250, 1970.

DoD. *Trusted Computer System Evaluation Criteria*. Department of Defence, 1986. DoD 5200.28-STD.

Jon Dugan, Seth Elliott, Bruce A. Mah, Jeff Poskanzer, and Kaustubh Prabhu. Iperf3: The tcp/udp bandwidth measurement tool. URL https://iperf.fr/.

Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe hardware access with the Xen virtual machine monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS)*, 2004.

Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin J. Elphinstone, Volkmar Uhlig, Jonathon E. Tidswell, Luke Deller, and Lars Reuther. The Sawmill multiserver approach. In *Proceedings of the 9th SIGOPS European Workshop*, pages 109–114, Kolding, Denmark, 2000.

Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM Operating Systems Review*, 40(3):80–89, July 2006.

Michael Hohmuth, Michael Peter, Hermann Härtig, and Jonathan S. Shapiro. Reducing TCB size by using untrusted components — small kernels versus virtual-machine monitors. In *Proceedings of the 11th SIGOPS European Workshop*, Leuven, BE, September 2004.

Jon Howell, Bryan Parno, and John R. Douceur. How to run POSIX apps in a minimal picoprocess. In *Proceedings of the 2013 USENIX Annual Technical Conference*, June 2013.

Antti Kantee. *Flexible operating system internals: the design and implementation of the anykernel and rump kernels*. PhD thesis, Department of Computer Science and Engineering, Aalto University, October 2012.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, et al. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, US, October 2009.

Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70, February 2014.

Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, San Francisco, CA, US, December 2004.

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 *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symposium on Security and Privacy*, pages 415–429, San Francisco, CA, May 2013.

Ruslan Nikolaev and Godmar Back. Virtuos: An operating system with kernel virtualization. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 116–132, 2013.

Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library OS from the top down. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 291–304, New York, NY, USA, 2011. URL http://doi.acm.org/10.1145/1950365.1950399.

John Rushby. A trusted computing base for embedded systems. In *Proceedings of 7th DoD/NBS Computer Security Conference*, pages 294–311, September 1984.

Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In *International Conference on Interactive Theorem Proving*, pages 325–340, Nijmegen, The Netherlands, August 2011.

David A. Solomon. *Inside Windows-NT*. 1998.

Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the 9th EuroSys Conference*, 2014.

Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference*, Santa Clara, CA, 2017.