Survey of Unikernels

Tyler Weekley

Advisor: Dr. Paul West

Department of Computer Science
Charleston Southern University

This thesis is submitted for the degree of
Masters of Computer Science

December 2020

Abstract

Unikernels are vastly different than a general purpose operating system. The image a unikernel creates is significantly smaller and only designed to run one process. Typically designed to run on the cloud, this research is going to evaluate the unikernels IncludeOS, OSv, and Rumprun by utilizing a variety of high computation benchmarks through SPLASH-2 and Linpack. Additionally, a dive into the design and implementation of IncludeOS, OSv, Rumprun, and other unikernels is provided. This thesis demonstrates that unikernels perform better than Linux when utilizing one to two processors, while providing significant security benefits.

Table of contents

1 Introduction **1**

1.1 Problem Definition . . . 1

2 Literature Review **2**

3 Performance and Security **5**

3.1 Performance . . . 5

3.2 Security . . . 6

4 Looking at the Unikernel Design **6**

4.1 IncludeOS . . . 6

4.2 OSv . . . 7

4.3 Rumprun . . . 8

4.4 Other Unikernels . . . 8

5 Performance Evaluation **9**

5.1 SPLASH-2 Kernels . . . 9

5.2 SPLASH-2 Applications . . . 10

6 Experiment and Results **10**

6.1 Experiment Design . . . 10

6.2 Results . . . 12

6.2.1 SPLASH-2 Kernels . . . 12

6.2.2 SPLASH-2 Applications . . . 13

6.2.3 Rumprun and Linpack . . . 14

6.2.4 Impact of Results . . . 14

7 Future Work **15**

8 Discussion **15**

9 Conclusion **16**

References **17**

1 Introduction

A Unikernel is a specialized executable image that does not rely on a general purpose Operating System. Given the large size of a general purpose operating system, certain services take longer to complete due to overhead. When looking to complete one task, it would be more efficient to have a system designed to specifically complete that one task. By eliminating features of a general purpose OS, a unikernel can provide services at higher performance and with fewer security vulnerabilities.Bratterud et al. (2015) Unfortunately, due to the specialized nature of Unikernels and the explosion of devices available, creating a configurable Unikernel has proven difficult. This thesis investigates the performance and security benefit of developing a unikernel that is designed specifically to solve these problems. Unikernels investigated were IncludeOS, OSv, HermitCore, RustyHermit, MirageOS, Rumprun, HalVM, RuntimeJS, and UKL.

1.1 Problem Definition

Prior to the cloud, companies had to have their own infrastructure in order to provide services. This infrastructure typically needed servers, network equipment, space, and man power, which is a lot of upfront costs. Today, we have Google, Amazon, and Microsoft, to name a few, offering cloud services to companies. This allows the companies to rent only what they need while no longer needing to maintain infrastructure. Instead, the company only needs to maintain what is being deployed in the cloud. This allows companies to save money and is proving to be a viable resource as the cloud continues to grow at a rapid rate Weins (2020).

Typically general purpose operating systems are being deployed in the cloud. The issue with deploying a general purpose OS in the cloud is that it was designed to handle multiple users as well as a wide variety of different hardwares and services. Whereas that might seem useful, typically applications in the cloud are providing one service, such as a web server or DNS server. Cloud providers are going to charge the consumer on networking, storage, and computation Palian (2015). General purpose OS are going to be large in size as they contain support for a large amount of drivers as well as being highly tuned to handle multitasking. It would be beneficial for the consumer to be able to deploy an image in the cloud that will remove unnecessary components. This will decrease the storage needed and will lower computation cost as there is going to be less processes running, which frees up resources.

A unikernel will solve the problem detailed above. The unikernel is going to remove all unnecessary components that are not needed to complete the desired task. Furthermore, the unikernel is only going to know how to do one thing, which is the one desired task. By doing this, a specialized image is going to be created and deployed. The specialized image is going to be greatly reduced in size and show better efficiency with resources. The outcome will result in lowering costs in the cloud while also providing better performing services.

The work done in this research will focus on comparing the raw performance on IncludeOS, OSv, and Rumprun to Linux through a variety of benchmarks that do not focus on networking. The benchmarks in this research are going to be single-threaded and multithreaded. In addition, this research is also going to explore the security enhancements unikernels provide through their design. Section 2 is going to look at relevant literature. Section 3 will focus on why performance and security should be improved in a unikernel over the Linux kernel. Section 4 will explore all the unikernels listed above designs. Following at section 5 we will evaluate the benchmarks chosen as our testing suite. Section 6 will be the setup of the benchmarks with an explanation of the results after in section 6. Sections 7 will be future work.

2 Literature Review

As unikernels gain traction, there are numerous project surrounding their design and performance. RustyHermit (Lankes et al., 2019) is an implementation of HermitCore (hermitcore, 2019), but in the Rust language. To do their evaluation of RustyHermit, they called getpid and schedyield 10 million times. RustyHermit and HermitCore both outperformed Linux in getpid, schedyield, and thread creation. They also used the matmul (matrix multiplication) benchmark from Rayon to compare Linux and RustyHermit when using a 8192x8192 matrix. Linux performed slightly better overall, but they believe the difference could be caused by using a Virutal Machine and wanted to further investigate the results. Another interesting work is (Raza et al., 2019), where the authors aim to use the Linux kernel and then strip it down into a unikernel. A fully functioning UKL unikernel would be novel for the unikernel scene because it would allow a unikernel to be built with the latest Linux kernel. Furthermore, it would allow the unikernel to have accesss to all the drivers that are available to the Linux kernel. The UKL unikernel would also gain additional benefits that typical unikernels suffer from, such as, a lack of community support and scheduling optimizations. Many of the unikernels used today are done so in some networking capability. Some examples are: DHCP server, DNS server, OpenVPN, TLS reverse proxy, and more. The amount of research on unikernels is limited mainly to an explanation of the unikernel's implementation with small benchmarks showing networking capability. As the unikernel community is rather small, the research showing the true power of a unikernel is lacking, and is what will be evaluated in this research.

IncludeOS was evaluated at (Bratterud et al., 2015). They compare IncludeOS memory usuage of Hello World, memory bandwidth reported by the STREAM Triad workload, and the CPU time required to execute 1,000 DNS queries. The memory comparison showed that "Hello World in Java without an operating system required more than 3 times the memory of IncludeOS with an operating system" (Bratterud et al., 2015). The STREAM Triad workload produced results detailing that IncludeOS performed better than Ubuntu on Intel, but on AMD, Ubuntu performed better than IncludeOS. However, "on all servers, the difference between IncludeOS and the Ubuntu VM's were lass than 0.5%" (Bratterud et al., 2015). When running the DNS query test on AMD, IncludeOS "uses 20% fewer CPU-ticks on average total, and 70% fewer ticks on average spent inside the guest CPU" (Bratterud et al., 2015). Evaluating the same test on Intel, "IncludeOS uses 5.5% fewer CPU-ticks on average total, and 66% fewer ticks inside the guest" (Bratterud et al., 2015). The DNS queries test displayed IncludeOS spent significantly less CPU time, which shows a more resource efficient unikernel over Ubuntu.

IncludeOS was also evaluated based on Firewall performance in comparison with Linux at (Tambs, 2018). The researcher used different size rule sets alongside different types of rules in order to compare how well both IncludeOS and Linux scale and "if some rule types aer heavier to process than others, ex. TCP destination port vs source IP address filtering" (Tambs, 2018). To verify the firewall, Hping3 was used, which is going to ensure that the firewall is blocking the intended traffic. When setting up the firewall filters, the filters are going to be applied to the FORWARD chain with a policy of accepting any packets that do not match any rule defined in the chain. Each packet is going to have to match every incoming packet against ever rule. "Larger rule sets should therefore lead to more overhead and probably lower throughput" (Tambs, 2018).

The paper (Tambs, 2018) ran 30 tests for 30 seconds each to evaluate IncludeOS and Linux. The first and second run were testing TCP throughput with "all the VMs acting as nothing more than forwarding routers, with no filter applied" (Tambs, 2018). In each run, IncludeOS had the highest throughput with IncludeOS performing at 9.29Gbps for the first run and 9.67Gbps for the second, while Linux was 8.52Gbps for the first run and 9.12Gbps for the second. When exclusively doing IP address filtering, when the source address rules reaches 5000, Linux's throughput was cut in half to 4.24Gbps and IncludeOS was still above 9Gbps. The following test is TCP throughput with destination port only rules, and both test runs show similar results with Linux throughput dropping down to 2.78Gbps with IncludeOS still sustaining approximately 9Gbps. Following all the tests and analyzing the results, the author concludes that IncludeOS not only used a fraction of the resources that Ubuntu used but also out performed Ubuntu in every test. Furthermore, when utilizing a unikernel in the cloud when serving large amounts of customers, "switching to a unikernel like the one tested here could help save enourmous amounts of resources and conseqeuently costs" (Tambs, 2018).

An evaluation on OSv at (Enberg, 2016) displays results of network performance using Netperf compared to Linux and Docker. Using netperf, the author utilizes TCP stream, TCP reverse stream (TCP-MAERTS), TCP Request/Response (TCP-RR), TCP Connect/Request/Response (TCP-CRR), UDP Stream, and UDP Request/Response (UDP-RR). The researcher used a 1 Gigabit NIC for the experiment For TCP Stream and UDP-Stream, Docker showed the least amount of overhead with OSv performing slightly worse, but the author concluded that OSv's networking stack is suboptimal with slower NICs for TCP receive. TCP-MAERTS results displayed that OSv had the least amount of overhead TCP-RR and UDP-RR analysis shows that Docker performed the best, but OSv was in a close second. TCP-CRR produced interesting results with Docker leading the way and then with Linux beating OSv. The author suggests that this indicates that there is an "inefficiency in OSv's TCP connection establishment because we do not see the same overhead in the TCP-RR test" (Enberg, 2016). While the author was unable to definitively identify the inefficiency, it is suggested to likely be in the TCP/IP stack, the network channel architecture, or the scheduler (Enberg, 2016). Analyzing all the Netperf tests, the author concludes that Docker has the least overhead, with the exception of TCP-MAERTS, and concludes that OSv shows, "a very promising optimization for hypervisor-based virtualization for raw networking performance but the technology needs to mature to deliver the performance potential" (Enberg, 2016).

In the research reviewed above, the benchmarks were mainly based on networking performance. The unikernel shows great signs of improvement in networking performance as their research proves, but their research lacks proof of raw computational power being better than Linux, which is what this research is going to focus on. The research conducted in this paper is novel as there has not been any research done on unikernels using applications that are computation heavy. This research is going test IncludeOS, OSv, and Rumprun (Kantee, 2012) with multiple different computation intensive benchmarks, multithreaded and singlethreaded, and will further provide this field with evidence of how well unikernels perform when a network is not involved. This research is also novel as it compares unikernels to each other and not solely to Linux as done in all the research prior to this.

3 Performance and Security

Unikernels should increase the performance and security of a single applications since, through design, Unikernels remove anything that is unnecessary for an application to run. After removing extraneous kernel features and drivers, a Unikernel packages the OS and application into a single image. The resulting Unikernel image is a substanially decrease. Typically, small is fast and big is slow and Linux and Windows have hundreds of drivers, whereas a unikernel might only have a couple. One thing unikernels do differently than general purpose OS is that a unikernel only includes necessary drivers, whereas a general purpose OS includes all available drivers regardless of it they are needed or not. As unneeded components are no longer included, security inherently improves due to a significantly smaller attack surface area (Bratterud et al., 2016).

3.1 Performance

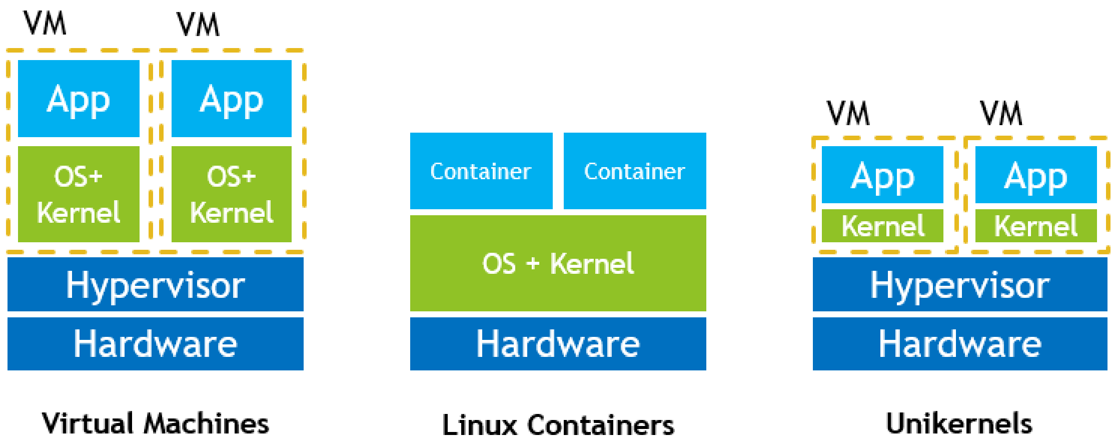


Fig. 3.1 Image showing how the unikernel compares to containers and VMs. Reprint from (Longree and Dupont, 2018)

Performance in a unikernel is enhanced in many different ways. Most unikernels are single threaded, so they will have no context switches. Unikernels are also single address space since they are only designed to handle one process. This single address space eliminates the use of virtual memory, which significantly improves performance. If concurrency is a desired capability, then it is recommended to spin up another unikernel instance (Bratterud et al., 2016). Building on that, since system calls are turned into regular function calls, the overhead that would have occurred during a context switch is removed. Memory footprint will be smaller as shown in (Bratterud et al., 2015), where a Java program running “Hello World” used roughly 20MB more than an IncludeOS booted instance of “Hello World” on Qemu. Figure 3.1 demonstrates the difference between a virutal machine, a linux container, and a unikernel. From Figure 3.1 you can see that a unikernel contains the significantly lighter load on infrastructure than VMs. Containers and unikernels are similar, but a unikernel still posses a hypervisor layer, which adds some additional security that the hypervisor can provide through seperation. Due to the lighter load on infrastrucutre, this leads to a substantial decrease in bootup times for a unikernel. OSv has boasted boot up times as fast as approximately 5ms (Cloudius-Systems, 2020).

3.2 Security

Due to the nature of a unikernel aiming to be as minimalistic as possible, there is a reduced amount of system calls as there are significantly fewer lines of code. For comparison, Linux has roughly 28 million lines of code (Bhartiya, 2020) and Windows has roughly 50 million (Microsoft, 2020). HermitCore has roughly 20 thousand lines of code (Lankes et al., 2019) and is going to allow a unikernel to have less “system calls”, which as we noted above is a regular function call. Unikernels also provide service isolation, so 0 bytes of code need to be shared between services at runtime. All data is read only, all executable code is write-protected, and all areas of memory that can be written to are not executable (Bratterud et al., 2016). The immutability aspect a unikernel provides is a main design principle. Futhermore, unikernels do not have a terminal that can be accessed, which is a security benefit because if an actor gains access to the unikernel, they are severly limited on what can be done from there. System calls in unikernels are turned into regular function calls (Bratterud et al., 2015) and unikernels, like IncludeOS, utilize ASLR (Lu et al., 2015).

4 Looking at the Unikernel Design

4.1 IncludeOS

IncludeOS aims to be a minimalistic unikernel (IncludeOS, 2020). Written in C++ for C++ services, IncludeOS is able to run in a virtual environment or bare metal. Adding “include <os>” at the beginning of the program will result in the IncludeOS operating system being added to the program at link-time. IncludeOS does not use virtual memory and a bootable includeOS running “Hello World” only needs 5MB of RAM when on x86-64. The boot time is generally around the 300ms range when booted on Qemu/KVM. IncludeOS provides modern C++ support and a standard C library using musl. There are drivers for virtio and vmxnet3 and the TCP/IP-stack is highly modular (IncludeOS, 2020). The hypervisors that are supported by IncludeOS are KVM, VirtualBox, and VMWare. IncludeOS uses conan profiles that have prebuilt packages to allow for easy setup of an IncludeOS environment. It also provides functionality to put the unikernel into editable mode and change the kernel as desired and then readily test it, all of which is well documented on their Github README.

4.2 OSv

OSv is a highly comapatible unikernel designed to run in the cloud that is capable of running: Java, Python 2 and 3, NodeJS, Ruby, Erlang, C, C++, Golang, and Rust (Cloudius-Systems, 2020). OSv's boot times can be as short as approximately 5ms on Firecracker, but is also able to be ran on: Qemu/KVM, Xen, VMware, VirtualBox, Hyperkit, AWS, GCE, and OpenStack. The documentation on OSv is extensive and setup and deployment scripts are provided (Cloudius-Systems, 2020). Beginning to look at the design, OSv chose to use virtual memory over physical memory mapping. The first reason OSv went with virutal memory is "the x86-64 architecture mandates virtual memory usage for long mode opertation" (Kivity et al., 2014). The second reason is "modern applications following traditional POSIX-like APIs tend to map and unmap memory and use page protection" (Kivity et al., 2014). OSv uses the mmap API to support demand paging and memory mapping. For large mappings, OSv is going to fill the entire mapping with 2MB pages and by doing so, this will improve performance of applications as it will redue the number of TLB misses (Kivity et al., 2014).Since unikernels are designed to run only a single application, OSv does not support page eviction.

OSv does not protect between userspace and kernel since they all share a single address space, so when OSv makes a system call it is as efficient as making a function call. Userspace and the kernel can also write to each other's memory, which "brings performance benefits, and a lot of flexibility" (Systems, 2020). The OSv unikernel uses two methods of scheduling: global fairness and cheap to compute, which may cause multithreading in OSv to be slower than Linux. The scheduler "uses per-cpu run queues, so that almost all scheduling operations do not require coordination among CPUs, and lock-free algorithms when a thread must be moved from on CPU to another" (Cloudius-Systems, 2020).

There is a lock-free algorithm in place by avoiding the use of any spin-locks to prevent lock holder preemption (LHP) (Shan et al., 2015), which is a key example of how OSv was designed for the cloud. The issue with spinlocks when dealing with virtualization comes from when the hypervisor pauses a VCPU. "Lock holder preemption describes the situation when a VCPU is preempted inside the guest kernel while holding a spinlock" (Cloudius-Systems, 2020). While this lock stays acquired other VCPUs requesting this lock are no longer doing any useful work and are wasting resources. The spinlock free design allows OSv to avoid a decrease on performance that LHP would cause. OSv went with a mutex implemenation that is based on a lock free design at (Gidnstam and Papatriantafilou, 2007).

4.3 Rumprun

The Rumprun unikernel was designed for C and C++ applications. Documentation provides users to build and deploy Rumprun images. Rumprun can be ran on top of Xen and KVM, and claims to be able to be ran on bare metal, but the author was unsuccessful with booting a bare metal Rumprun. Rumprun drivers require runtime context information in order to function properly. This information includes "process/thread context and a unique rump kernel CPU that each thread is associated with" (Kantee, 2012). Rumprun does not use virtual memory, which differs from OSv, and also "does not provide support for page faults or memory protection" (Kantee, 2012). Rumprun developers left virtual memory protection and page faults to be handled by the host of the Rumprun kernel if desired.

4.4 Other Unikernels

This thesis focuses on three unikernels: IncludeOS, OSv, and Rumprun. However, there are many more unikernels such as: HermitCore (hermitcore, 2019) and RustyHermit (Lankes et al., 2019), MirageOS (Mirageos, 2020), HalVM (GaloisInc, 2020), RuntimeJS (Runtimejs, 2019), and UKL (Raza et al., 2019). RustyHermit was developed in Rust with the intention of demonstrating that writing a unikernel in Rust would show the same performance benefit while also gaining the memory safety benefits that Rust provides. Their work showed that RustyHermit was able to perform at a level on par with HermitCore (which is written in C), which is a promising result in Unikernel development for ensuring memory safety (Rust, 2020).

MirageOS is written in OCaml and supports running directly on the Xen hypervisor. The developers chose OCaml because it is a "full-fledged systems programming lanuage with a flexible programming model that supports functional, imperative and object-oriented styles" (Mirageos, 2020). Similar to MirageOS is HaLVM (GaloisInc, 2020), Haskell Lightweight Virtual Machine. As the name suggests, HaLVM is written in Haskell to allows developers the opporunity to write high level, lightweigh VMs that runs directly on the Xen hypervisor. Runtime.js (Runtimejs, 2019) was developed in C++ and was designed to run Javascript. It was built on the V8 Javascript engine and currently supports KVM, however, it is no longer being maintained. UKL (Raza et al., 2019) is an emerging Unikenerl that aims to reduce Linux to a Unikernel. UKL aims to be a part of the kernel source tree, and with achieving that, UKL will gain many of the advantages of Linux, with the biggest being the community, and still retain the many advantages a unikernel provides.

5 Performance Evaluation

To test out the performance of the unikernels, SPLASH-2 and Linpack were used. SPLASH-2 consists of complete applications and computational kernels (Woo et al., 1995). SPLASH-2 is going to consist of many multithreaded benchmarks and Linpack is a single threaded benchmark. Using multithreaded benchmarks was necessary to evaluate the design of the unikernels scheduler in order to determine the viability of executing multithreading as a unikernel. The different applications and kernels are going to use a, “variety of computations in scientific, engineering, and graphics computing” (Woo et al., 1995). Radix, lu, and fft SPLASH-2 kernels were built for the Unikernels as well as fmm and ocean from the SPLASH-2 application group. The Linpack benchmark is used to, “rate the performance of a computer on a simple linear algebra problem” (Burkardt, 2019). Linpack can either be single precision or double precision. Linpack was configured to use a matrix A of size N filled with random values, a vector B, which is the product of matrix A and a vector X that is all 1’s. "The first task is to compute an LU factorization of A and the second task is to use the LU factorization to solve the linear system A * X = B" (Burkardt, 2019). The number of floating point operations is equal to $\frac{2*N^3}{3+2*N^2}$

5.1 SPLASH-2 Kernels

Radix is a sorting kernel that is iterative and performing one iteration for each radix digit of the keys. During each iteration, “a processor passes over its assigned keys and generates a local histogram, and the local histograms are then accumulated into a global histogram” (Woo et al., 1995). Once the global histogram is generated, each processor will permute its keys into a new array, which will then be used in the next iteration. Permutation requires an all-to-all communication and the keys are communication through writes.

Lu “factors a dense matrix into the product of a lower triangular and upper triangular matrix” (Woo et al., 1995). An *nxn* matrix will be divided into an *NxN* matrix with *BxB* blocks, which is going to exploit temporal locality. Communication is reduced by assigning block ownership using a 2-D scatter decomposition. An ideal block size is one that is large enough to minimize the cache miss rate, but small enough to have a good load balance. Lu contains a contiguous and a non contiguous implementation. The contiguous implementation is going to increase spatial locality benefits.

FFT is optimized to reduce the communication between processors, and the data is going to have n complex data points, that will be transformed, and another set of n data points that are referred to as the roots of unity. “Both sets are organized as $\sqrt{n}x\sqrt{n}$ matrices so that every processor is assigned a contiguous set of rows which are allocated in its local memory” (Woo et al., 1995). All to all interprocessor communication is required and it occurs in three matrix transpose steps. Transposes are blocked to exploit cache line reuse. In order to avoid memory hotspotting, submatrices are staggerly communicated meaning processor i is going to transpose a matrix from processor $i+1$ and then a matrix from processor $i+2$ after.

5.2 SPLASH-2 Applications

FMM is an application that simulates a system of bodies over a number of timesteps. FMM simulates interactions in a 2-D manner using the method known as Fast Multipole Method (FMM). The data structures is a body and tree cells and each cell has multiple particles. The tree is going to traversed in one upward and downward pass per timestep and will compute the interaction amongst the cells and send the effects down to the bodies. Communication patterns are “quite unstructured and no attempt is made at intelligent distribution of particle data in main memory” (Woo et al., 1995).

Ocean is another benchmark that contains a contiguous and noncontiguous implementation. Ocean is going to simulate large-scale ocean movements that are based on eddy and boundary currents. The grids are partitioned into square subgrids. Grids are going to be represented as a 4-D array with all subgrids being allocated either contiguously or noncontigously. This version of Ocean uses a red-black Gauss-Seidel multigrid equation solver.

6 Experiment and Results

6.1 Experiment Design

All the SPLASH-2 benchmarks used can be found at (West, 2020). All benchmarks were ran on a system with 4 XEON e7-8850 processors, 256GB DDR3-1066 RAM, and Ubuntu

18.04LTS The benchmarks were modified to not require any user input for parameters as well as using any files for input or output. All parameters used in the benchmark are the same unless noted below.

- Radix's number of keys to sort was changed from 262,144 to 1,677,216.
- FFT's total complex data points transformed was changed from 10 to 24.
- Lu's block size was changed from 16 to 32, and the matrix size was changed from 512 to 2,048 for both contiguous and non-contiguous.
- FMM's number of particles was changed to 392,352.
- Ocean's contiguous and non-contiguous grid size was changed from 258x258 to 2050x2050.
- Linpacks' matrix size was changed from 1000x1000 to 5000x5000.
- Ubuntu, OSv, and IncludeOS were all ran with 1, 2, 4, 8, 16, and 32 processors.
- Rumprun was evaluated with the same benchmarks but was only ran with 1 processor as using multiple processors would fail.

It was important to increase the problem size by orders of magnitudes for each since the benchmarks were establish in the 1990s and hardware has improved substantially since. Similar to the experiments done by relevant work, instead of stressing networking, these tests are going to stress raw computational power a unikernel can provide. The tests are going to be single-threaded and multithreaded. The tests did provide output, but not all unikernels had the functionality to utilize a timing mechanism. To get around this the time command provided by Linux, which outputs the time in seconds, was used to have a common timing mechanism for all benchmarks. The importance of showing single-threaded and multithreaded tests is that a unikernel is designed to typically be single-threaded, whereas a general purpose OS is likely to be better with multithreading. I hypothesize that a unikernel will show better performance than Linux when single-threaded given the nature of less overhead, but as more threads are added I think Linux's scheduler will start to take over and perform better than a unikernel.

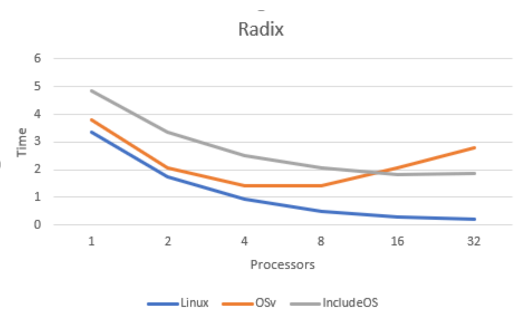


Fig. 6.1

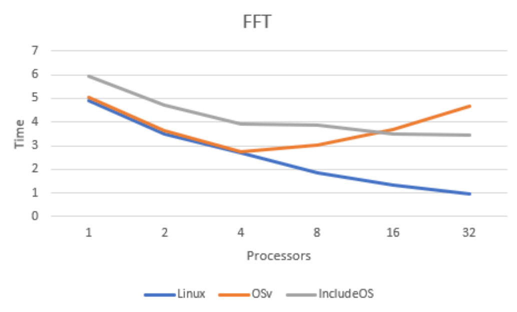


Fig. 6.2

6.2 Results

6.2.1 SPLASH-2 Kernels

For 1,2, and 4 threads, Ubuntu and OSv share comparable performance. However, as OSv's number of processors go up so does the time after the jump from 4 processors to 8. This trend indicates OSv's multithreaded scheduler is less performant that Linux's. The same results in 6.1 are seen in 6.2 with OSv and Ubuntu showing comparable performance.

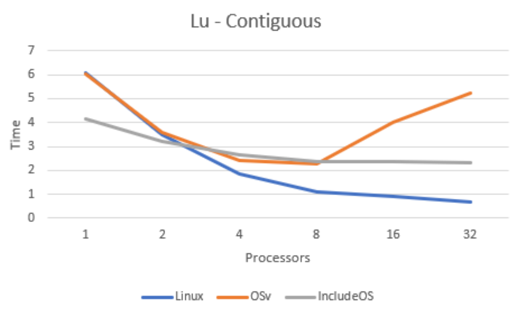


Fig. 6.3

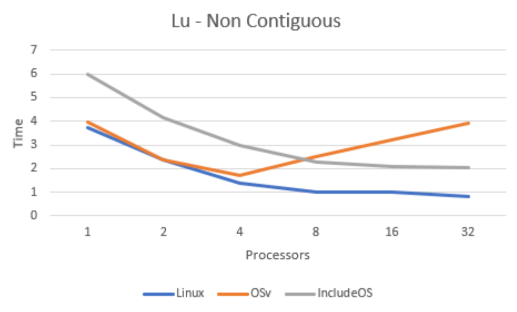


Fig. 6.4

Figure 6.3 shows that both unikernels outperformed Ubuntu with 1 processor and then also shows that IncludeOS outperformed Ubuntu at 2 processors. With Lu contiguous taking advantage of spatial locality, IncludeOS excelled with 1 processor outperforming both OSv and Ubuntu by 2 seconds. When looking at Lu non-contiguous, 6.4, the results are flipped. OSv and Ubuntu both outperformed IncludeOS by 2 seconds and Ubuntu outperformed both unikernels the entire time. This suggests that IncludeOS might excel better in applications that take advantage of spatial locality.

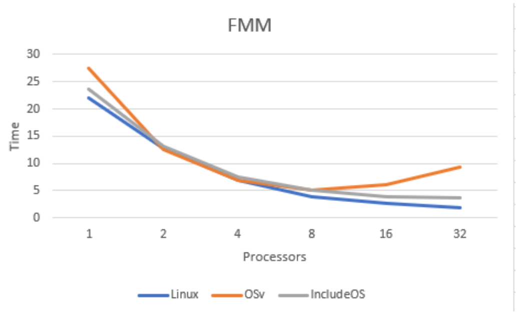


Fig. 6.5

6.2.2 SPLASH-2 Applications

FMM, displayed in 6.5, produced interesting results as prior benchmarks do not show the three operating systems converging after the test utilizes more than one processor. FMM's results shows Ubuntu performing the best with one processor, but for processors two and four, the times are almost equivalent with the exception of IncludeOS being slightly slower. IncludeOS did show continous improvement in time decreasing as processors increased, whereas OSv maintained the consistency of time increasing after four processor threshold. Thus far, IncludeOS has shown consistency in the number of processors continually decreasing the time taken to complete the task, which shows that IncludeOS' scheduler is more suitable for multiple threads versus OSv.

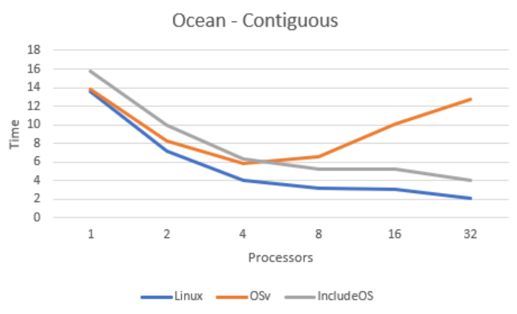


Fig. 6.6

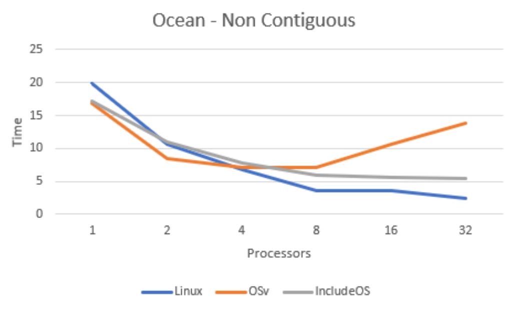
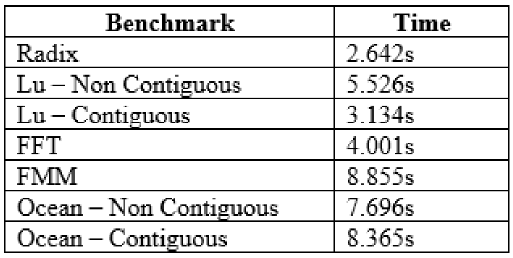


Fig. 6.7

Figure 6.7 shows the results for Ocean with the contiguous implementation. Interestingly, IncludeOS peformed the worst initially, whereas in Lu contiguous, IncludeOS performed the best. In Lu non-contiguous, Linux performed the best, slightly outperforming OSv at one processor, but in Ocean non-contiguous, OSv and IncludeOS both outperformed Ubuntu up until the test that utilized four processors. To this point IncludeOS and OSv have outperformed Ubuntu twice, performed comparable to Ubuntu three times, and have performed worse to Ubuntu two times.

6.2.3 Rumprun and Linpack



Benchmark	Time
Radix	2.642s
Lu – Non Contiguous	5.526s
Lu – Contiguous	3.134s
FFT	4.001s
FMM	8.855s
Ocean – Non Contiguous	7.696s
Ocean – Contiguous	8.365s

Fig. 6.8

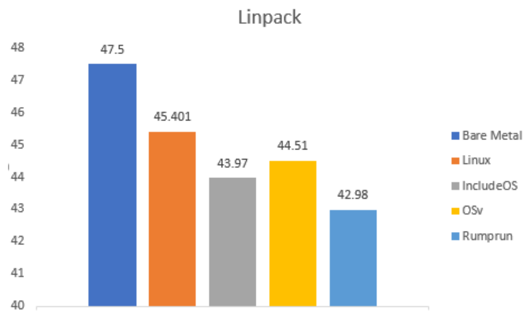


Fig. 6.9

Table in 6.8 displays the single processor results of the SPLASH-2 kernel and application benchmarks. Rumprun outperformed Ubuntu, IncludeOS, and OSv 85% of the time when the test was single threaded. With Rumprun's performance, unikernels have outperformed Ubuntu in every test with the exception of Lu non-contiguous, which OSv performed comparative to Ubuntu.

The linpack benchmark is a single threaded application and was evaluated on Ubuntu, OSv, IncludeOS, Rumprun, and bare metal using IncludeOS' ability to create a bare metal bootable image. The bare metal boot performed the worst, but it is worth noting the time was taken using a stopwatch, so there is a slight delay in the start and stop. All three unikernels outperformed Ubuntu with Rumprun leading the way again. Unikernels outperform Linux in seven of the eight benchmarks.

6.2.4 Impact of Results

With a unikernel demonstrating performance improvement 87.5% of the time when utilizing single-threading, the usefulness for unikernels is clear. As predicted, Linux outperformed unikernels when multithreading got involved, but that was anticipated. These unikernels were designed with the intention of solving issues of providing fast, secure services in the cloud. By taking out the networking aspect of a unikernel, this research was able to take a unikernel out of its intended scope and thoroughly test its overall capability. Through demonstrating that a unikernel does perform higher than a general purpose OS with a single thread, it further proves that a unikernel in the cloud will get performance and security enhancements while also reducing the cost of running a service in the cloud.

7 Future Work

Additional testing could reveal the network capability as these unikernels were design for the cloud. The UKL (Raza et al., 2019) unikernel would also be great to do some development on because a working UKL unikernel would be a great template unikernel for many applications since it is aiming to be compatible with any Linux driver and gain the benefits of the latest Linux kernel. Another idea worth investigating is running a unikernel as a process. As noted in (Williams et al., 2018), running a unikernel as a process can provide many benefits. This will allow the unikernel to inherit process-specific characteristics, such as high memory density. As this research displayed, unikernels currently show good promise in performance benefits, so if running a unikernel as a process can provide more benefits, then the use cases for unikernels might grow even further.

8 Discussion

The findings of this research further prove the effectiveness a unikernel can provide. Furthermore, the findings show that a unikernel can be used outside of pure networking capabilities. This research also shows that a program that would excel with multiple threads might not be best suitable for a unikernel due to the simple scheduling implementations. It is clear that unikernels would excel in the cloud due to their lightweight nature. Being lightweight in the cloud would allow for instances to be cheaper and also highly scalable since boot up times are quick. It is also clear unikernels need a bigger community to make them slightly more adaptable. With only a few individuals working on each unikernel, they unikernel tend to fall behind in current technology and capabilities. One reason unikernels might have a small community is because they are typically specific to a problem they are aiming to solve. For a specific problem to be solved by a unikernel, a new unikernel might need to be developed to be able to handle the work as efficiently as possible, which is a workload many likely would not want to take. It is in my opinion that the unikernels evaluated in this research would also be viable general purpose unikernels as none of the unikernels in this research were designed to run SPLASH-2 or Linpack and results were still promising.

9 Conclusion

Unikernels have started to emerge the last few years. There are still a few under active development, but with such a small community, many have already become no longer maintained. Little research into the raw computational power has been done, till now, and this research showed that a unikernel of some flavor will outperform Linux 87.5% of the time. Linux, currently, seems to perform better when multithreading gets involved, whereas a unikernel performs better when single-threaded. With strong performance benefits, such as computational power, memory usage, and physical size, paired with increased security, unikernels are a great fit for the cloud and possibly other solutions that run on top of some type of hypervisor. One big question remains of "Does the amount of work required to build a unikernel outweigh the performance and security benefits?". It is definitely a case by case situation, but as the community continues to grow for unikernels, it is likely that more unikernels appear in the cloud to increase productivity while decreasing cost.

References

Bhartiya, S. (2020). *Linux in 2020*. https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd Retrieved: September 30, 2020.

Bratterud, A., Engelstad, P., and Haugerud, H. (2015). Includeos: A minimal, resource efficient unikernel for cloud services. pages 1–9.

Bratterud, A., Happe, A., and Duncan, B. (2016). Enhancing cloud security and privacy: The unikernel solution. pages 1–8.

Burkardt, J. (2019). *The Linpack Benchmark*. https://people.sc.fsu.edu/ jburkardt/c_src/linpack_bench/linpack Retrieved: September 28, 2020.

Cloudius-Systems (2020). *OSv*. https://github.com/cloudius-systems/osv Retrieved: September 28, 2020.

Enberg, P. (2016). *A Performance Evaluation of Hypervisor, Unikernel, and Container Network I/O Virtualization [University of Helsinki]*. https://helda.helsinki.fi/handle/10138/165920.

GaloisInc (2020). *HaLVM*. https://github.com/GaloisInc/HaLVM Retrieved: September 28, 2020.

Gidnstam, A. and Papatriantafilou, M. (2007). Blocking without locking or lfthreads: A lock-free thread library. https://domino.mpi-inf.mpg.de/internet/reports.nsf/c125634c000710d0c12560400034f45a/77c097efde9fa63fc125736800444203/$FILE/MPI-I-2007-1-003.pdf.

hermitcore (2019). *HermitCore*. https://github.com/hermitcore/libhermit Retrieved: September 28, 2020.

IncludeOS (2020). *IncludeOS*. https://github.com/includeos/IncludeOS Retrieved: September 28, 2020.

Kantee, A. (2012). The design and implementation of the anykernel and rump kernels. http://www.fixup.fi/misc/rumpkernel-book/rumpkernel-bookv2-20160802.pdf.

Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D., and Zolotarov, V. (2014). Osv - optimizing the operating system for virtual machines. pages 60–72. https://www.usenix.org/system/files/conference/atc14/atc14-paper-kivity.pdf.

Lankes, S., Breitbart, J., and Pickartz, S. (2019). Exploring rust for unikernel development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, PLOS'19, page 8–15. Association for Computing Machinery.

Longree, G. and Dupont, Sebastien Ezequiel Vara Larsen, M. (2018). Unikernel and immutable infrastructures.

Lu, K., Song, C., Lee, B., Chung, S. P., Kim, T., and Lee, W. (2015). Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 280–291. Association for Computing Machinery.

Microsoft (2020). *Windows 10 Lines of Code*. https://answers.microsoft.com/en-us/windows/forum/all/windows-10-lines-of-code/a8f77f5c-0661-4895-9c77-2efd42429409 Retrieved: September 30, 2020.

Mirageos (2020). *Mirage OS*. https://mirage.io/ Retrieved: September 28, 2020.

Palian, J. (2015). *How the Cost of Cloud Computing is Calculated*. https://expedient.com/knowledgebase/blog/2015-05-01-how-the-cost-of-cloud-computing-is-calculated/.

Raza, A., Sohal, P., Cadden, J., Appavoo, J., Drepper, U., Jones, R., Krieger, O., Mancuso, R., and Woodman, L. (2019). Unikernels: The next stage of linux's dominance. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 7–13. Association for Computing Machinery.

Runtimejs (2019). *runtime.js*. https://github.com/runtimejs/runtime Retrieved: September 28, 2020.

Rust (2020). *Meet Safe and Unsafe*. https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html Retrieved: September 30, 2020.

Shan, J., Ding, X., and Gehani, N. (2015). Aple: Addressing lock holder preemption problem with high efficiency. page 1.

Systems, C. (2020). *OSv Designed for the Cloud*. http://osv.io.

Tambs, T. (2018). *Unikernel Firewall Performance: IncludeOS vs Linux*. https://www.duo.uio.no/handle/10852/63891.

Weins, K. (2020). *Cloud Computing Trends: 2020 State of the Cloud Report*. https://www.flexera.com/blog/industry-trends/trend-of-cloud-computing-2020/.

West, D. P. (2020). *SPLASH-2 Posix*. https://github.com/ProfessorWest/splash2-posix Retrieved: September 28, 2020.

Williams, D., Koller, R., Lucina, M., and Prakash, N. (2018). Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 199–211. Association for Computing Machinery.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. (1995). The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, page 24–36. Association for Computing Machinery.