OSv Research Report

2021-04-22

Graham Schock, Sarah Morin

Contents

1 OSv

Most cloud systems run a general-purpose version of a Linux operating system. In many cases, the traditional operating system has more features than what is required by what it runs and duplicates isolation features already provided by the cloud systems hypervisor. The excess provided by the system comes at a high cost.

OSv supports the execution of a single Linux application with less overhead than a traditional operating system. It runs as a virtual machine, thus relying on the host's hypervisor to provide isolation. In some sense, OSv aims at being the happy medium between containers and robust virtual machines by providing the isolation of a virtual machine in conjunction with the efficiency of a container.

2 Target Domain

In this section we will discuss where OSv is a good fit, and where OSv might not be a good fit. We will dive into two possible scenarios and see how different aspects of OSv might hinder or help a certain situation.

2.1 A Memcached application

Memcached is an in memory key value system. It allows us to not have to visit the “slow” path of acsessing the database every time, instead we can accsess a fast cache that has information that we use often. This is a *good* use case for OSv for the following reasons:

- It is a single application system
- It is memory intensive

2.1.1 Single Application Systems

Single applications like memcached are a good fit for OSv because they only rely on a single address space. When we fuse an application to OSv there is only one address space that an application can acsess. This means we can get added benifits that system calls are as fast as regular function calls.

2.1.2 Memory Intensive

Memcached is a memory intensive system as it focuses on caching. Memory intensive operations are fast in OSv because we only have one address space. This means we can do things like giving an application direct accsess to page tables which can tremendously speed up memory operations.

2.2 Running multiple applications on baremetal

However, there are cases where running OSv for an application may not be the best choice. An example of this would be running multiple applications on bare metal hardware. There are two main reasons why this might be a *bad* use case for OSv:

- There is no hypervisor
- There are multiple applications

2.2.1 Hypervisor

It is required that OSv is running on top of a hypervisor like Qemu or KVM. This is because OSv is focused on running applications on the cloud which have a hypervisor.

2.2.2 Multiple Applications

OSv will not provide memory abstractions such as seperation of address spaces into a kernel and userspace. If there are no memory abstractions all applications will have acsess to all other applications memory. Therefore running OSv for multiple applications is a bad idea.

2.3 Modules

In this section we will discuss what modules the OSv has and the relationship between those modules.

OSv modules include shared memory, networking channels, and thread scheduler. At first, we have processes that make a call down into the “kernel”. The “kernel” provides a classifier associated with a channel which is a single producer/single consumer queue. Then the channel transfers packets to the application thread. It saves much processes in the traditional network stack and removes the lock and cache-line contention. Also, during this progress OSv allows only one thread to access the data which is stored in a single address space. Compared to the traditional thread scheduler, OSv’s thread scheduler does not use spin-locks and sleeping mutex. Moreover, based on the fairness criteria of threads on the CPU’s run-queue, the scheduler chooses the most appropriate thread to do the next operation which keeps the queue of one CPU not much longer than others’, which is efficient.

Let’s dive into the modules a little bit now.

2.3.1 1. Virtual Memory

At first, OSv doesn’t have multiple spaces. OSv runs an application with the kernel and threads sharing a single space. It means the threads and kernel use the same tables, which make system calls as efficient as function calls and also make context switches quicker. Also, because OSv share a single address space, it doesn’t maintain different permissions for the kernel and applications. Therefore, the isolation is managed by the hypervisor. This way achieves simpler code, better performance and also reduces the frequency of TLB misses.

2.3.2 2. Networking channels

OSv provides a new network channel so that only one thread could access the data which simplifies the locking. Most of TCP/IP is moved from kernel to the application level, while a tiny packet classifier is running in an interrupt handling thread. Therefore, it could reduce the run time and context switches overhead. The code is implemented in net_channel.cc. It shows that after finding the ipv4 packet which has the same item in the hash table, it will use the pre-channel and wake it up.

```
bool classifier::post_packet(mbuf* m)
{
    WITH_LOCK(osv::rcu_read_lock) {
        if (auto nc = classify_ipv4_tcp(m)) {
            log_packet_in(m, NETISR_ETHER);
            if (!nc->push(m)) {
                return false;
            }
            // FIXME: find a way to batch wakes
            nc->wake();
            return true;
        }
    }
    return false;
}
```

2.3.3 3. Thread Scheduler

OSv is designed to use the scheduler which runs different queues on each CPU. Therefore, almost all scheduling operations do not need coordination among CPUs and the lock-free algorithms when a thread must be moved from one CPU to another.

3.1 Lock-free algorithm OSv assumed that only a single pop() will be called at the same time, while push()s could be run concurrently. The push() code is like the following. Because only the head of the pushlist is replaced, before changing the head, they will check whether the head is still what they used in item-next, which guarantee that changing is correct.

```
inline void push(LT* item)
    {
        // We set up item to be the new head of the pushlist, pointing
           to the
        // rest of the existing push list. But while we build this new
           head
        // node, another concurrent push might have pushed its own
           items.
```

```
6           // Therefore we can only replace the head of the pushlist with
                a CAS,
7           // atomically checking that the head is still what we set in
8           // item->next, and changing the head.
9           // Note that while we have a loop here, it is lock-free - if
                one
10          // competing pusher is paused, the other one can make progress.
11          LT *old = pushlist.load(std::memory_order_relaxed);
12          do {
13              item->next = old;
14          } while (!pushlist.compare_exchange_weak(old, item, std::
                memory_order_release));
15      }
```

3.2 Thread scheduling OSv shedule has global fairness and is easy to compute. Each task has a measure of runtime it receives from the scheduler. The scheduler picks the runnable thread with smallest R, and runs it until some other thread has a smaller R.

Beside of keeping each CPU has its own separate run-queue, which removes the lock mechanism, OSv uses a load balancer thread on each CPU so that it could keep the run queue in one CPU have similar tasks as others. If the queue of one CPU is longer than others', the load balancer thread will pick one thread from this CPU and wakes it up on the remote CPU. The more detailed implementation about schedule is here:

```
1  // This CPU is temporarily running one extra thread (this thread),
2           // so don't migrate a thread away if the difference is only 1.
3           if (min->load() >= (load() - 1)) {
4               continue;
5           }
6  WITH_LOCK(irq_lock) {
7      ... ...
8      ... ...
9      min->send_wakeup_ipi();
10 }
```

Ax explained above, when OSv calls down to the data, it shares a single address space and avoids context switches. Also, by using the "channel", OSv saves much processes in the traditional networking stack. OSv avoids the socket locks and TCP/IP locks. According to the performance result, the network stack's performance for TCP and UDP consistently outperforms Linux and latency is about 25% less than Linux. Also, the context switching (thread switching) is much faster in OSv than in Linux - between 3 and 10 times faster.

3 Core Abstractions

In this section we will discuss and provide examples of core abstractions. We will look at the basic Linux API that OSv provides as well as some additional APIs that are specific to OSv and might not be in Linux.

3.1 Linux API & ABI

One of the main goals of OSv is to be able to allow the user to add as little code as possible to an existing Linux application to run on OSv. On the OSv wikipedia the authors say the following:

> OSv mostly implements Linux's ABI. This means that most unmodified executable code compiled for Linux can be run in OSv.

cloudious-systems

An important thing to note when reading this quote is the difference between an ABI and an API. An API is something a user would write source code and interact with. For example a user interacting with a Linux API would look like the following:

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    void* a = (void*) malloc(10);
}
```

In this case malloc would be part of the Linux API that the user is interacting with. An ABI is when a Binary is able to compiled in one system is also able to be compiled and run in another system that is ABI compatible. When OSv says that it is ABI compatible, it means that binaries that can be compiled and run on Linux can also be compiled and run on OSv (most of the time).

We can also see that OSv provides a Linux API from its build process. We can see the API being built in multiple steps with some examples of what is being compiled in those sections:

- bsd (general kernel things like IPC)

```
bsd += bsd/sys/kern/kern_mbuf.o
bsd += bsd/sys/kern/uipc_mbuf.o
bsd += bsd/sys/kern/uipc_mbuf2.o
```

- zfs (file system)

```
zfs += bsd/sys/cddl/contrib/opensolaris/common/zfs/zfeature_common.o
zfs += bsd/sys/cddl/contrib/opensolaris/common/zfs/zfs_comutil.o
zfs += bsd/sys/cddl/contrib/opensolaris/common/zfs/zfs_deleg.o
```

- libtsm (terminal emulator)

```
libtsm += drivers/libtsm/tsm_render.o
libtsm += drivers/libtsm/tsm_screen.o
libtsm += drivers/libtsm/tsm_vte.o
libtsm += drivers/libtsm/tsm_vte_charsets.o
```

- musl + libc (provide standard library functions)

```
libc += internal/_chk_fail.o
libc += internal/floatscan.o
###################################
musl += ctype/__ctype_get_mb_cur_max.o
musl += ctype/__ctype_tolower_loc.o
```

- drivers (to interact with the screen and network)

```
drivers += drivers/vga.o drivers/kbd.o drivers/isa-serial.o
drivers += arch/$(arch)/pvclock-abi.o
drivers += drivers/virtio.o
drivers += drivers/virtio-pci-device.o
```

All of these elements compiled together enable OSv to give applications a Linux API.

3.2 Beyond the Linux API

In order to motivate this discussion let's look at the following graph OSv provides when they talk about performance of different types of `memcached` running on OSv.

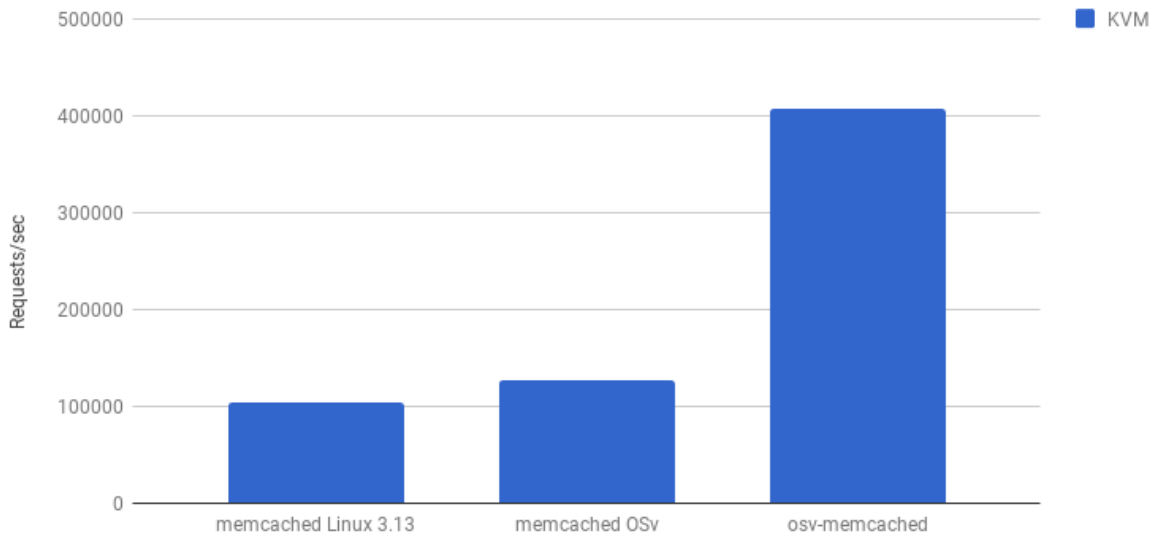

Figure 1: different levels of memcached performance

As we can see from figure above there is a dramatic improvement (around 4x) from the Linux version of memcached running on OSv and `osv-memcached`. One of the reasons we are able to get better speeds is because we use APIs outside of Linux that will be discussed below.

3.2.1 The Shrinker API

Shrinker is something that OSv implements beyond the Linux API. Essentially Shrinker takes advantage that we are allowed truly dynamic memory on a Uni-kernel. On a regular Operating System most systems like a dynamic cache must statically determine their size before hand. This can be limiting as when there are boosts of memory. Sometimes we do not have enough cache entries and sometimes we have allocated too much memory.

The Shrinker API fixes this issue by allowing us to dynamically increase or decrease our memory. We are allowed to increase our memory to the entire memory of the system or image.

3.2.2 Using the Shrinker API

In the OSv Memcached source we can see we use the API. The first thing that we do is grab the Shrinker lock:

```
1 WITH_LOCK(_locked_shrinker) {
```

Once we have the lock we can simply update our cache sizes:

```
1 it->second.lru_link->mem_size = memory_needed;
```

This way we can be way more efficient with our caches and dynamically size them based on our needs.

As we can see OSv provides a mixture of true Linux APIs as well as some extensions beyond that for performance reasons.

4 Performance

In this section we will discuss and provide examples of performance. We will also provide hypothesis for why certain performance measures are better than others. We will break down perforance based on the two types of tests performed in the paper: Macrobenchmarks and Microbenchmarks.

4.1 Macrobenchmarks

A macrobenchmark is a test that can run over multiple elements. It will usually test an entire application. We will examine two different macrobenchmarks:

- Memcached
- SPECjvm2008

4.1.1 Memcached

Here is the same figure that we presented in section 4 about memcached, but will focus on all different types of memcached.

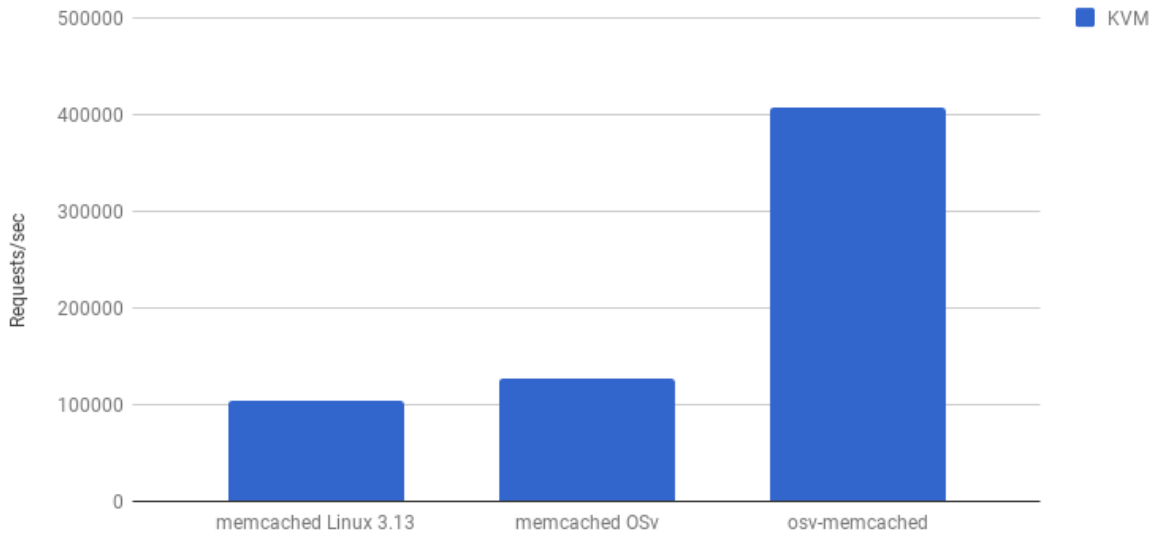

Figure 2: different levels of memcached performance

As we can see from the figure there are three different types of memcached:

- memcached Linux 3.13 This is a generic version of memcached running on the Linux kernel. This is what you would find in a memcached in the wild.
- memcached OSv This is a generic verion of memcached, but instead of running on Linux its source code is fused together with OSv.
- osv-memcached This is a special version of memcached using the APIs discussed in aprt 4.

As we can see, simply compiling the same memcached source code that runs on Linux into OSv we can an improvement of around 20%. We hyppthesize this is because memory accsess are faster in memcached because there is only one address space that has to be dealt with. This means we can give more memory to memcached then we normally could with a kernel which allows it to go faster. We can even get faster performance by using the API detailed in part 4.

4.1.2 SPECjvm2008

SPECjvm2008 is a suite of a variety of applications. It optimizes for testing the performance of the JVM, which is what OSv will be running.

Benchmark	OS^v	Linux	Benchmark	OS^v	Linux
Weighted average	**1.046**	**1.041**	sor.large	27.3	27.1
compiler.compiler	377	393	sparse.large	27.7	27.2
compiler.sunflow	140	149	fft.small	138	114
compress	111	109	lu.small	216	249
crypto.aes	57	56	sor.small	122	121
crypto.rsa	289	279	sparse.small	159	163
crypto.signverify	280	275	monte-carlo	159	150
derby	176	181	serial	107	107
mpegaudio	100	100	sunflow	56.6	55.4
fft.large	35.5	32.8	xml.transform	251	247
lu.large	12.2	12.2	xml.validation	480	485

Table 2: SPECjvm2008 — higher is better

Figure 3: different levels of SPECjvm2008 performance Linux vs OSv

Computation intensive tasks The first thing that we notice is that this performance is a farcry away from the perforance boost of memcached. This is because a lot of the tasks are Computation intensive tasks which OSv can not optimize too much for.

Filesystem intensive tasks There are even some aspects that are slower for example `compress` and `crypto.rsa`. These tasks focus on using the file system. The fileystem is just a ZFS bolted onto the kernel, which is not as optimized to the level as the Linux File system is.

4.2 Microbenchmarks

Instead of focusing on an application as a whole, Microbenchmarks focus on specific elements of applications i.e networking. We will focus on two specific benchmarks:

- Networking benchmarks
- Context Switching benchmarks

4.2.1 Networking

To measure networking benchmarks we use netperf which sends requests and packets between 2 hosts on the same network and measures the performance. The paper provides the following table:

Benchmark	OSv	Linux	Benchmark	OSv	Linux
Weighted average	**1.046**	**1.041**	sor.large	27.3	27.1
compiler.compiler	377	393	sparse.large	27.7	27.2
compiler.sunflow	140	149	fft.small	138	114
compress	111	109	lu.small	216	249
crypto.aes	57	56	sor.small	122	121
crypto.rsa	289	279	sparse.small	159	163
crypto.signverify	280	275	monte-carlo	159	150
derby	176	181	serial	107	107
mpegaudio	100	100	sunflow	56.6	55.4
fft.large	35.5	32.8	xml.transform	251	247
lu.large	12.2	12.2	xml.validation	480	485

Table 2: SPECjvm2008 — higher is better

Figure 4: different levels of networking performance Linux vs OSv

As we can see OSv consistenly outperforms Linux as far as networking is concerned. This is because OSv does not have to deal with complicated lock structures or expensive context switches when changing threads while networking.

4.2.2 Context Switching benchmarks

The contest switching benchmark was homemade. It simply measured the average time of a number of a predertmined number of context switches between threads.

Guest OS	Colocated	Apart
Linux	905 ns	13148 ns
OS^v	328 ns	1402 ns

Table 6: Context switch benchmark

Figure 5: Context Switching benchmarks

OSv is better at context switching. This is because of all the optimizations that the thread scheduler has. Using lock-free algorithms and idle-time polling.

As we can see there are variety of places where OSv performs better and some places where OSv may perform worse.

4.3 Technology

The core technologies include the single address space, thread scheduler with lock-free algorithms and networking channels, ZFS file system. It is the hypervisor that multiplexes the hardware into different applications and the application runs in a single address space, which does not need to copy and validate system call parameters. Therefore, the context switches are faster and more efficient. Also, when packets are transferred from hardware to application threads, threads use the lock-free algorithms to avoid the thread switches. When lock is necessary, OSv provides a lock-free mutex without a spin-lock. What's more, ordinary thread handles interrupts in a special interrupt context rather than in OSv. Unlike the traditional spin-locks when protecting data from multiple access at a same time, OSv chooses to wake up the interrupt-handling thread to do the interrupt processing tasks. If locks are needed, it chooses to use a mutex rather than a spin-lock. When threads receive packets from the classifier, OSv reduces much overhead by removing the socket locks, TCP/IP lock and packets are processed at the application level. Instead of using the VFS file system, in OSv ZFS has many more advantages including easy to create and grow automatically within the space.

5 Security

In this section we discuss the security properties of the system and how it adheres to the principles for secure design. In addition to the principles, we analyze a few unikernel-specific topics within the OSv implementation.

5.1 Security in Unikernels

Many unikernels claim to be more secure because their small nature provides a reduced attack surface while virtualization still provides isolation from the host system. In general, a minimal system implies a reduced attack surface; however, the optimizations provided by some unikernels, OSv inlcuded, actually expose vulnerabilities which might not exist in mroe robust systems. The following provides a general overview of the security advantages and concerns specific to unikernels and an analysis of OSv with respect to these topics.

There are two main types of unikernels: ***clean-slate*** and ***legacy***. OSv is a legacy unikernel, meaning it aims to support unmodified software by implementing a subset of POSIX and re-implementing some Linux system call interfaces. Unikernels provide the *medium* level of isolation from the host system as they are more isolated than a container but generally less isolated than a full virtual machine from the host system.

Common vulnerabilities specific to unikernels include:

- **Lack of Address Space Layout Randomization** TODO
- **Single Protection Ring** Large systems utilize protection rings to specify privilege classes of principals. Unikernels often provide a single protection ring, meaning the kernel and software code run in the same protection domain, i.e. with the same privileges.
- **Absence of Guard Pages** Guard pages are sections of unmapped memory placed betweeen allocations used to trigger segmentation faults. Without guard pages, the system is extremely vulnerable to memory related attacks.
- **Inability to Debug** It is often extremely difficult, if not impossible, to debug a unikernel while it is running. If the system is compromised or facing incorrect behavior, it likely needs to be destroyed and rebuilt.

Strategies to increase security in unikernels include:

- **Increase Entropy** Increasing the use of random values and ensuring randomly generated values are as random as possible by disallowing the use of repeated seeds increases the overall entropy of the system.
- **Hardening** Essentially reducing the attack surface with technique such as eliminating or disabling unecessary services, bound checking within libraries, ASLR, and encryption.

5.1.1 Secure Aspects of OSv

Guard Pages

Unlike many unikernels, OSv implements guard pages. The memory allocations in OSv place guard pages between application allocations. With these guard pages, the `mmu` can easily generate page faults for improper access, limiting the possibility of overflow attacks.

Library Hardening

A Security Perspective on Unikernels cites supporting the `_FORTIFY_SOURCE` macro as an option for library hardening in unikernels. The macro forces the use of functions with more extensive checking in the C library implementation. OSv's C library implementation is based off og `musl` and does, in fact, support the macro. When the macro is enabled, some standard functions are replaced by implementations in `__[function]_chk.c` files thourghout OSv's C library implementation. For example, if the macro is enabled, the standard `read()` function calls code from `libc/__read_chk.c`:

```
1 ssize_t __read_chk(int fd, void *buf, size_t count, size_t bufsize)
2 {
3     if (count > bufsize) {
4         _chk_fail(__FUNCTION__);
5     }
6     return read(fd, buf, count);
7 }
```

Although this is an extremely simple example, we can see that a bound check has been implemented which ensures the count to read is not larger than the buffer size. If the bound check fails, the `_chk__fail()` function is called:

```
1 extern "C" void _chk_fail(const char *func)
2 {
3     abort("%s: aborting on failed check\n", func);
4 }
```

The function simply aborts due to the failed check.

5.1.2 Vulnerabilities of OSv

Protection Ring 0

Many systems use protection rings to specify the privileges of principals. Limiting the privileges of a principal to the necessities improves system security. Unfortunately, OSv has a single protection ring, ring 0, which the kernel and application share. Clearly this presents a vulnerability.

Network Hardening

Disabling unecessary services and eliminating the shell altogether are efective options for network hardening in Unikernels; OSv supports a shell and does not limit networking. As stated in A Security Perspective on Unikernels "… in OSv the REST API used to control the Unikernl can replace the command line, read and write files and directories. Exposing it to attackers gives them command execution, Local and Remote File Inclusion.".

5.2 Principles of Secure Design

As we saw above, the small nature of OSv does not guarantee more security than a traditional system. Put simply, there are both advantages and disadvantages to being small and efficient. Now we move to a more general analysis of OSv with respect to the principles of secure design. Since OSv is intended to be a virtual machine, the thread it poses to the host system can be reduced to that of the hypervisor. Thus, the following focuses on the principle within the OSv system.

5.2.1 Complete Mediation

5.2.2 Least Privilege, Separation of Privilege, and Minimal TCB

Each of these principle is violated, somewhat by design, in OSv. Internally, OSv violates least privilege by design and has a single protection domain. Within the context of the larger host system, OSv's privilege is in the worst case (i.e. highest possible privilege) that of the hypervisor. As there is a single protection domain there is no separation of privilege. The TCB is essentially the entire system; if anything it is maximal.

5.2.3 Economy of Mechanism

In general, OSv seems to obey the economy of mechanism. It is small by nature and implements a subset of the features contained in a more robust system. Since it can execute only a single application it is fairly simple. OSv bases its C library on `musl` which is fairly common. Thus, the *new code* in OSv's C library is relatively minimal. The only point of contention is where OSv differs from common mechanism implementations, such as its avoidance of spin-locks, and introduces complexity as compared to system using "standard implementations".

5.2.4 Least Common Mechanism

The notion of least common mechanism is slightly strange within OSv. On one hand, the address space and resources shared between the kernel and application seem to violate this principle. On

the other hand, the application is the only *productive* software running on the system. That is, the kernel, scheduling threads, memory management, networking channels, etc. are only relevant if an application is running. If these principals are corrupted by the application they will simply affect the execution of the application itself.

5.2.5 Defense in Depth

OSv runs unmodified applications. In order to change the system, it must be destroyed and rebuilt. Although this is not a barrier within a running system, it proses significant difficulty to an attacker trying to modify a running system.

5.2.6 What is the Reference Monitor?

In unikernels, OSv included, the reference monitor is essentially the hypervisor. Within OSv, both the kernel and application run in the same protection domain, so there is no sensical method of validating privilege. Further, the application has direct access to the virtual memory it shares with the kernel and the system only has one virtual address space. Between OSv and the host system, the hypervisor takes on the responsibility of complete mediation, temperproof-ness, and trusthworthiness. Whether or not the goals are provided is dependent on the specific hypervisor.

6 Thoughts on OSv

Here the authors provide their personal thoughts on the system.

6.1 Sarah Morin

I thoroughly enjoyed learning about OSv. I enjoy the idea of systems attempting to compromise the pros and cons of virtual machines and containers; however, I'm not sure if it is anything more than an interesting idea. Throughout my research I found much more information specific to clean-slate unikernels than I found for OSv or legacy unikernels in general. It makes me beg the question: are the benefist of OSv and other legacy unikernels over containers and full VMs actually worth the effort? Are clean-slate unikernels the preferred unikernel? I don't have enough experience in production level cloud system to venture a guess the costs and benefits of clean-slate vs. legacy; however, the internet certainly seems more tantilized by the clean-slate option.

6.2 Graham Schock

I have always enjoyed looking at Unikernels. OSv was written from a different perspective than other Unikernels I have looked at. Before this research project I looked into MirageOS which had a hands on build and configuration process where the user had to write OCaml in order to get there applications fused into their Unikernel. It was difficult even getting the OCaml compiler and dependencies to work. It was even harder to learn and understand OCaml enough to configure MirageOS with a simple `hello world` application. However, OSv did not have this problem. I was easily able to create my own applications and run them immedietly with OSv. I think this is the major upside to OSv, but can also be a downside. I think we lost the true Unikernel approach and get a hybrid DOS kernel. Whether I compiled mySQL or hello_world.c into my Unikernel they were both almost the same size but had very different requirments. However, I think the ease of use in OSv takes Unikernels in the right direction in where they need to go.