OSv Research Report

2021-04-22

Graham Schock

Contents

1 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.

1.1 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

1.1.1 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.

1.1.2 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.

1.2 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

1.2.1 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.

1.2.2 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 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.

2.1 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.

2.2 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.

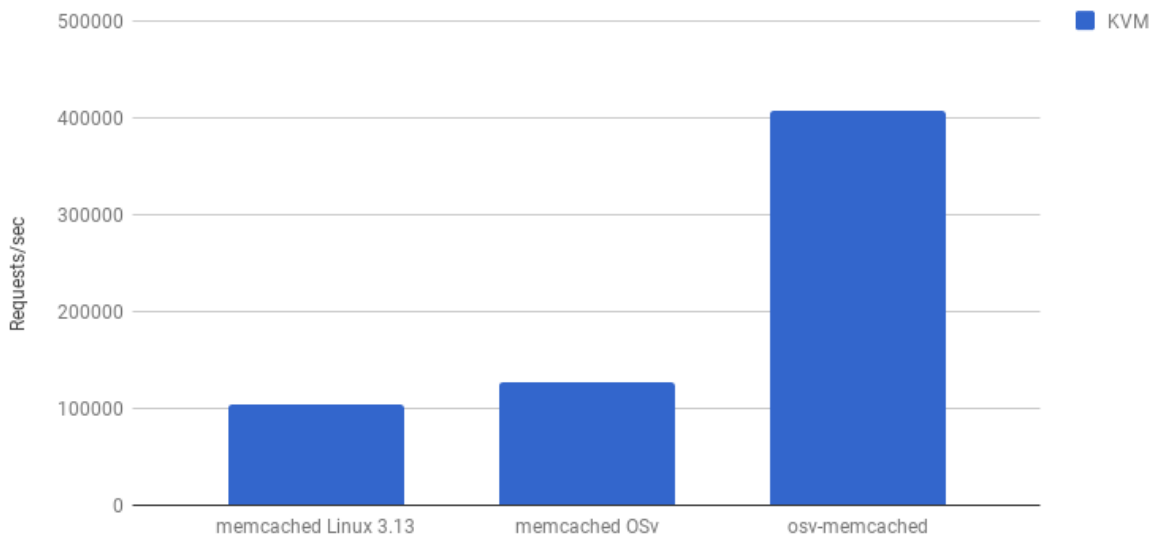

Figure 1: different levels of memcached performance

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.

2.2.1 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.

2.2.2 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.

3 Title