Unikernels for Serverless Architecture

Gunjan Patel
Santa Clara University
{gunjanpatel.v@gmail.com}

Abstract

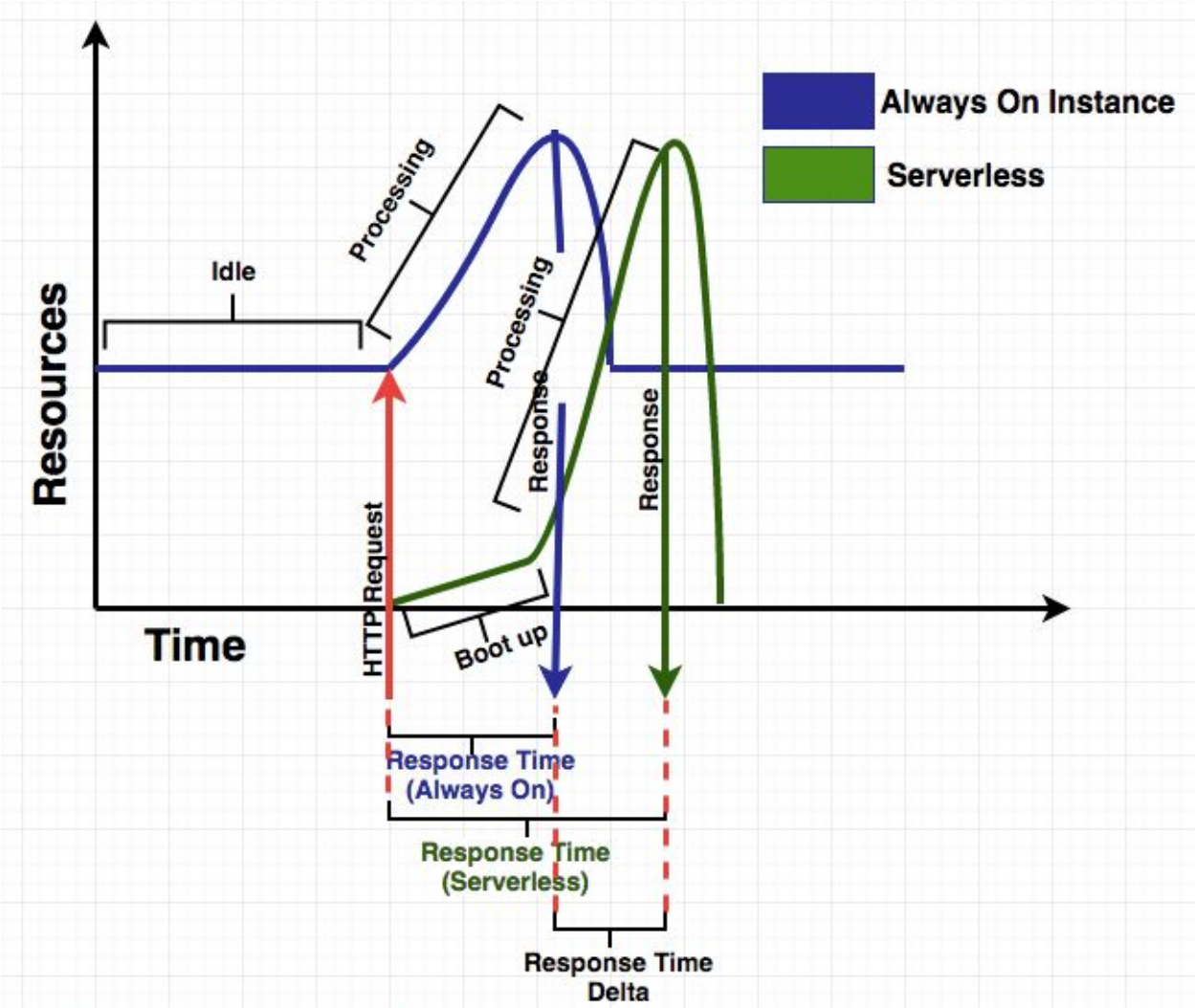
With the rise of the cloud computing and XaaS (Anything-as-a-Service), computing units are becoming more and more modular. Server side applications have evolved from baremetal systems to monolith virtual machine based applications to microservices and now to Serverless[1] architecture, also known as FaaS[2] (Function as a Service). Most new Serverless systems use containers as the base runtime system to serve a unit of service which lets the system to be more flexible and agile with the speed and performance of a container. In this research we are proposing the use of Unikernels as the service runtime for the serverless applications. Unikernels, when compared to similar container runtime, have faster boot-up time, which is very critical for a serverless system where instances are booted up when the need arises and then powered off after they are done serving the request. In this paper, we will compare and contrast boot-up performance of unikernels vs containers.

Motivation

Motivation for doing this research is to compare boot up time performance of containers vs. unikernels. Boot up times are very crucial for Serverless/FaaS (Function as a Service) architecture. Moving from VMs to containers enabled us to expand rapidly with Microservices architecture. Now, to take it one step farther Serverless/FaaS architecture breaks down services even granular and runs instances of them, those instances only boot up when they are needed, so boot up time is one of the most critical aspect for better performance.

Problem

The image below shows resource utilization vs time comparison for always running system compared with Serveless system.



[Figure-1: Resources vs Time to demonstrate the response time delta between always-on vs serverless system]

The problem with having an always running service, an application server for example, is that it racks up a huge compute, power, memory, network and storage bill from the cloud providers since they usually tend to charge for these resources per unit of time. Serverless architecture is designed to overcome this issue by bringing up the service only when it's needed

and then powering it down to save the resources. This solves part of the problem where it's not utilizing the resources when the service is not in use, but introduces another problem which is the service instance will need to be booted up whenever it's the service is needed, and this introduces extra latency in serving the requests.

Boot up times have increased radically since the introduction of containers, compared to Virtual Machines. However, containers still take some time to boot up, and request response time is still not as good as an always running service.

This request response time can be improved further if we use unikernels to host such service in Serverless
architecture.

Intent

Intent of this experiment is to see if unikernels can boot up faster compared to containers.
Instances boot up per request in Serverless/FaaS architecture.

The goals for this experiment are:

- Get a unikernel runtime working with a serverless/FaaS orchestrator (doesn't exist right now)
- Measure boot up time of unikernels vs containers (for the same code serving a simple HTTP service)
- Measure total size of unikernel instance vs container instance for a simple server (same code)

Experimental Setup

We have used the same runtime environment, OS and machine for running unikernels and containers. Both experiments have not been ran simultaneously, and no other applications were running during the experiment for isolation purpose with the exception of terminal to run the experiment script, and daemon processes for unikernel and container runtime during their individual runs.

Common Environment:

- Go[3] version: go1.8 darwin/amd64
- OS: OS X El Capitan version 10.11.6
- Hardware: MacbookPro (Retina, 15-inch, Mid-2015)
- Memory: 16GB 1600 MHz DDR3
- CPU: 2.5 GHz Intel Core i7

Specific systems and versions for each setup are listed in detail below.

Unikernel:

- Unikernel runtime used in this experiment is Project Unik [4], which is an open source project for unikernel compilation and deployment platform written in Go.
- Version of Unik used: Git hash ***`3698c2bbff7b85b58b6d53dae607fbf18a44055b`***
- Hypervisor:
    - VirtualBox version: 5.1.22 r115126 (Qt5.6.2)
- Docker version used to run the Unik daemon:
    - Server: 17.06.0-ce-rc2 (Git commit: 402dd4a)
    - Client: 1.30 (Git commit: 402dd4a)

Container:

- Container runtime used in this experiment is Moby[5] (previously known as Docker[6]), which is an open source container runtime implementation in Go.
- Docker version (Docker for Mac): 17.06.0-rc2-ce-mac14 (18280)
- Hypervisor:
    - macOS Hypervisor framework (Xhyve)
- Base image: golang:1.8[7] (official)

Experiment

In the experiment, we will measure boot up time for the unikernel and container instance of the same code, a simple Go server (see Appendix for the code).

Both unikernel and container images are pre built, so build time is not taken into the consideration.

There is two second wait time between each iteration of instance boot-up to allow the respective daemon processes to finish whatever processing they might need to perform after the instance is deregistered/deleted. One special case is for unik instance, we need to manually delete the instance since it will not automatically get deleted like with container which will die after the main thread is finished. To avoid any unnecessary race condition between registering the unikernel instance and deleting it before it might have been finished registering, we add one second sleep time between starting the instance and deleting it. This time is not taken into account for boot-up time calculation. This is necessary to have a fair comparison because we don't want to have more than one instance of unikernel running at the same time, just like we have for containers.

The docker image used to run the container instances is pushed to DockerHub[8]

Steps to to compile a unikernel are not included since they can change, especially since the projects are in their infancy and there is no stable API available.

Results

From the result below, we have the following observations:

- We can see that the average boot-up time for unikernels is 37.91% faster compared to containers for a hot run.
- We also see unikernel average boot-up time is lower than that of a container by 35.23% for a cold run.
- We also notice the image size of a unikernel is about 180% smaller than a comparable container image.

Boot-up time comparison

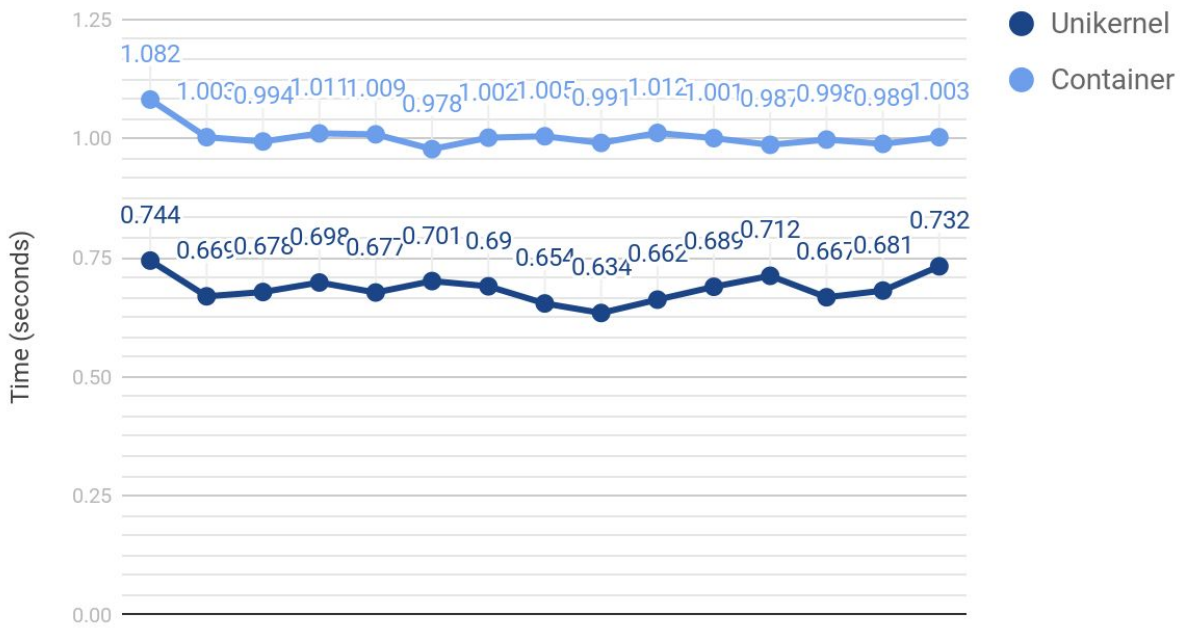
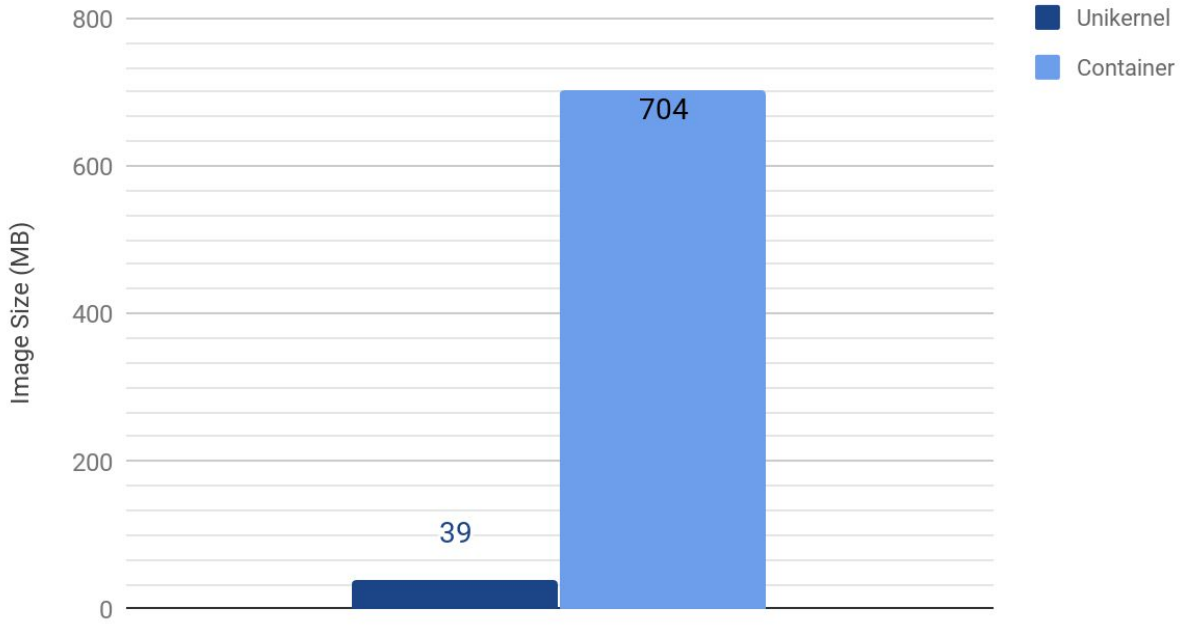


Image Size Comparison



	Hot Run		Cold Run		Image size
	Averag	Standard Deviation	Average	Standard Deviation	
Unikernel	0.684s	0.029	0.760s	0.034	39 MB
Container	1.004s	0.023	1.085s	0.025	704 MB

Conclusion

In conclusion, unikernels boot-up time seems significantly faster than that of containers, including cold and hot start. Serverless systems are on the rise, and boot-up time being one of the bottlenecks for serverless architecture adoption, unikernels as serverless runtime can really help by increasing boot-up time performance by as much as 30-40%. Anecdotal evidence also suggests that unikernels can better utilize the hardware and boot-up time performance increases when moved higher a memory/CPU/IO machine compared to containers. Unikernels also come with additional benefits of smaller image size and superior security by only including the kernel features needed by the application.

References

[1] https://en.wikipedia.org/wiki/Serverless_computing
[2] https://en.wikipedia.org/wiki/Function_as_a_Service
[3] https://golang.org/
[4] https://github.com/cf-unik/unik
[5] https://github.com/moby/moby
[6] https://blog.docker.com/2017/04/introducing-the-moby-project/
[7] https://hub.docker.com/r/library/golang/tags/1.8
[8] https://hub.docker.com/r/gunjan5/argo

Appendix- A

The simple Go server code used for the experiment:

```go
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/", handler)
    go http.ListenAndServe(":8080", nil)
}

func handler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serverless Unikernels!")
}
```