EXPLORING UNIKERNELS FOR SERVERLESS COMPUTING

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2023

Nathan Edward Cliffe Jones

Department of Computer Science

Contents

Abstract **5**

Declaration **6**

Copyright **7**

Acknowledgements **8**

1 Introduction **9**
1.1 Aims and Objectives 10

2 Background **11**
2.1 Serverless Computing and FaaS 11
2.1.1 FaaS Under the Hood 12
2.2 Unikernels and their Application to FaaS 18
2.3 Related Works 19

3 Design **21**
3.1 Needed Features of the Extension 21
3.2 Open-Source FaaS Platform Selection 22
3.3 Extension to OpenFaaS: OpenFaaS-Hypervisor 23

4 Implementation **27**
4.1 OpenFaaS-Hypervisor 27
4.2 Function Instances 30
4.2.1 Unikernel 30
4.2.2 MicroVM 31
4.2.3 Container 32
4.3 Deployment of the OpenFaaS-Hypervisor 32

5 Evaluation **34**
5.1 Method 34
5.2 Results 37
5.2.1 Function Memory Usage 37
5.2.2 Function Execution Time 38
5.2.3 Artefact Size 40

6 Conclusion **42**
6.1 Summary of Results 42
6.2 Project Aims 43
6.3 Critical Analysis 44
6.4 Further Work 45

Acronyms **47**

Glossary **48**

Bibliography **49**

Word Count: 11284

List of Figures

2.1 Diagram depicting a gVisor container. Source: [24] 16

3.1 GitHub star history of candidate open-source FaaS platforms 22
3.2 Diagram of OpenFaaS deployed to Kubernetes using faas-netes 23
3.3 Diagram of the unikernel extension to OpenFaaS 24

4.1 Diagram illustrating the sequence of events when function instances register as ready 28

5.1 Total RSS memory usage of function instances 37
5.2 Function execution times 39
5.3 Function artefact size. 41

Abstract

EXPLORING UNIKERNELS FOR SERVERLESS COMPUTING
Nathan Edward Cliffe Jones
A report submitted to The University of Manchester
for the degree of Bachelor of Science, 2023

Serverless computing is a cloud computing paradigm where the cloud service provider abstracts away the underlying servers and physical infrastructure from the user. Function as a Service (FaaS) is one example of this that allows developers to write and deploy application code without having to provision servers or manage any infrastructure. When building a FaaS platform for developers to use, many requirements need to be met. Most of these requirements are easy to meet individually, but hard to meet collectively. Current FaaS platforms use either container or microVM technologies to meet these requirements, but each technology has its drawbacks.

Unikernels are a new technology that allows users to compile their application code alongside the kernel code into a single purpose binary image that runs directly on the hardware or hypervisor. They provide many advantages over traditional methods of packaging and deploying application code.

This report proposes an extension to an existing serverless platform to make use of unikernels with the hope that this new extension meets the requirements of a FaaS platform better than the traditional solutions of containers and microVMs. It explains the design of this unikernel extension and how it was built. This unikernel extension is then evaluated by measuring how well it solves the requirements of a FaaS platform as compared to containers and microVMs.

This paper shows that unikernel based FaaS platforms are not only able to meet the FaaS platform requirements but actually meet them more closely than microVM or container based FaaS platforms.

Declaration

No portion of the work referred to in this report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420`), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see `http://www.library.manchester.ac.uk/about/regulations/`) and in The University’s policy on presentation of Theses

Acknowledgements

I would first like to thank my project supervisor, Pierre Olivier, for providing me with support, guidance and excellent domain knowledge throughout my project. I would also like to thank Syed Samar Yazdani and Alan John, who attended my weekly project meetings, providing input and suggestions on my work.

Chapter 1

Introduction

Serverless computing is a cloud computing paradigm that is used to label cloud services that can be used out of the box with little to no configuration of the underlying servers and infrastructure [25]. Serverless technology has become prevalent because it allows developers to focus on providing functionality and value through the code they write without spending time building the underlying infrastructure that the code relies on.

This project focuses on one specific serverless service all major cloud providers offer: Function as a Service (FaaS). FaaS allows users to write application code as a set of functions and then configure a set of events that trigger the functions to execute. The cloud service provider handles everything else, including provisioning the servers on which the function executes, routing triggering events to the function, and scaling the function up and down in response to demand.

Building a FaaS platform is complex, and many requirements must be met. Some of these requirements oppose each other, meaning it is hard to meet them all effectively. FaaS platforms currently employ one of two technologies to meet these requirements as closely as possible: containers and microVMs.

This report first introduces the opposing requirements faced by FaaS platforms and the technologies currently used to try and meet them. It then introduces a new technology called unikernels that allows developers to compile their application and kernel code together into a single executable. The conceptual idea of how a unikernel based FaaS platform can theoretically meet its requirements better than a microVM or container based FaaS platform is then explained. This conceptual idea is then realised as an extension to an existing open-source FaaS platform. Finally, an evaluation is made by measuring how well the unikernel based extension meets the requirements

compared to traditional technologies.

1.1 Aims and Objectives

The aims and objectives of this project are to:

- Research and gain an understanding of the requirements that need to be met when building a FaaS platform and the challenges faced when trying to meet them. Research how existing FaaS platforms try to meet these requirements and how they overcome the challenges.
- Research and gain an understanding of unikernels and how they have already been used to improve cloud applications. Conceptualise how unikernels can be used by a FaaS platform to meet its requirements more closely.
- Propose and implement an extension to an existing FaaS platform to support the use of unikernels.
- Evaluate how well a unikernel based FaaS platform meets its requirements as compared to FaaS platforms based on more traditional solutions already in use.

The success criteria for the first two aims are to describe in this report what FaaS and unikernels are and how unikernels can be used by a FaaS platform to meet its requirements.

The success criteria of the third and fourth aims are to have a working FaaS platform that can be deployed to a server to handle real function requests. A valid method must be developed to evaluate how well a unikernel based FaaS platform meets its requirements. Finally, this evaluation method needs to be executed, and data needs to be gathered and interpreted to decide if unikernels are a sensible addition to a FaaS platform.

Chapter 2

Background

2.1 Serverless Computing and FaaS

Computer scientists and software engineers are always looking for ways to abstract away low level details to allow them to build more extensive and complex systems. For example, when developers moved from writing machine code to assembly and then assembly to high-level programming languages. This trend can be seen throughout computer science and shows itself in cloud computing.

In the past, when a developer needed a database, the developer had to rent or buy a server, install and maintain an operating system, set up networking, install database software and more. Nowadays, all of this work can be abstracted away, and the developer can use a service such as DynamoDB [2] to provision a database without having to perform any of the abovementioned tasks.

This type of abstraction, where "virtually all the system administration operations" are handled by the cloud service provider, is called serverless computing [25]. This trend towards serverless computing allows developers to spend more time writing code and less time managing infrastructure, meaning they can provide more value to their customers.

FaaS is a serverless service that allows users to write their application logic in a set of functions that can be triggered in response to events such as web requests. All the user has to do is write function code and select a set of triggers. The cloud service provider handles everything else such as provisioning the servers that the function executes on, routing triggering events to the function, scaling the function up and down in response to demand and more.

So why is it important to study and improve serverless and FaaS? As stated by Jonas

et al. in Cloud Programming Simplified: A Berkeley View on Serverless Computing [25], FaaS has many benefits to its users, such as increased programming productivity and cost savings. It also has many benefits to the cloud service provider, such as drawing in new customers and increasing the utilisation of their resources. According to one survey, in 2020, 30% of respondents used ‘serverless architecture/FaaS’; in 2022 this jumped to 53% [13]. There are also many documented serverless success stories from major enterprise companies such as Bosch, Taco Bell, AstraZeneca, Sky and Waitrose [9, 6, 4, 5, 7]. Because serverless computing and FaaS have significant benefits over traditional methods of deploying applications and their usage is growing at a fast rate, it is essential to study them and try to improve how they operate to increase the value to both the user and the provider.

2.1.1 FaaS Under the Hood

From the user’s point of view, FaaS is designed to be simple by abstracting away all the complex management of the underlying servers; however, from the FaaS provider’s point of view, this is far from simple. Many requirements need to be met when building a FaaS platform, such as:

1. FaaS providers need to ensure that all running functions are isolated. If one function is performing a heavy calculation, this should not impact the performance of another function. More importantly, no function should be able to access the internal state or alter the intended execution of another function. If this were to happen, in the best case, a function may exhibit undefined behaviour, but in the worst case, another user may maliciously access private and sensitive data from another user’s function. Throughout this paper, the first type of isolation is referred to as ‘performance isolation’ and the second as ‘operational isolation’.

2. FaaS providers want to minimise their operational costs. To do this, they must maximise resource utilisation to reduce the computing resources required to run the FaaS platform.

3. The users want their functions to execute with fast and consistent performance. Users require consistency so they can model their application and predict how it scales.

These requirements are referred to throughout this report and are often referred to generally as ‘the requirements’.

These requirements are not hard to meet individually, but it is challenging to meet them all simultaneously. If a cloud service provider needs to run 100 functions that each require 1 CPU core, then it is cheaper and easier to buy a single 100-core machine than to buy 100 single-core machines. Because of this, to maximise resource utilisation, FaaS platforms run multiple functions on the same machine. This then causes isolation concerns, so the FaaS platforms have to provide some form of software based isolation. However, achieving isolation through software causes the function's performance to drop.

Many FaaS platforms already exist and have had to deal with the above requirements. Each one provides its own solution, striking a balance between the three requirements:

- **AWS Lambda** was the first public cloud offering FaaS.

  AWS Lambda, like all other FaaS platforms, allows developers to have many function definitions, each with its own unique name, code and triggers. Throughout this report, these are referred to as 'functions'. The term 'function instance' is used to refer to the place where a specific function's code can execute. So each *function* has a unique set of *function instances* that can execute that specific *function's* code, and each *function instance* can only execute the code of a specific *function*.

  Functions are executed in response to events. An example of an event may be someone updating a file or making a web request. These events can be considered as requests to invoke the function. These are referred to as 'function invoke requests' or just 'invoke requests'.

  On AWS, each function instance is a virtual machine (VM), and each VM only serves one invoke request at a time. So if 100 simultaneous requests are made to a function, 100 VMs are required to execute them in parallel. If a new request is made while all the 100 functions are still executing, AWS creates a new VM to serve this 101st request. On the flip side, when all 101 functions have finished executing, these VMs are no longer needed, so they can be removed to give space for new function instances. The issue with this approach is that each invoke request has a high latency because a VM has to boot up before the function code can execute. To combat this, AWS does not instantly remove the VM after function execution. Instead, it keeps it running and reuses it if that same function needs to be invoked again. These VMs are considered 'warm'. A warm VM sits

idle, waiting for an invoke request. As soon as it gets invoked, it is no longer considered warm. Instead, it is considered 'hot' and again becomes 'warm' after the function has finished executing. If a warm VM is not invoked for a long time, it is removed to free up resources and give space for new VMs [3, 8].

When a function invoke request needs to wait for a new VM to be created before the function can be executed, like with the 101st request mentioned above, this is called a 'cold-start'. When the function invoke request can be routed to a pre-booted warm VM, this is called a 'warm-start'. Warm-start invokes are quicker than cold-start because there is no delay caused by the VM booting.

VMs take a long time to boot. This causes cold-start times to be much longer than warm-start times, decreasing the average function performance and consistency. This violates the performance and consistency requirement of FaaS platforms (requirement number 3). To combat this, AWS made some developments to improve their boot times:

- A virtual machine monitor (VMM) is a piece of software that creates and manages VMs. Traditional VMMs such as QEMU can be slow to boot VMs, so AWS created their own VMM called Firecracker, which is designed to be lightweight and fast, and can boot VMs $\sim$1.3x faster than QEMU [1, 8].
- AWS also employ microVMs to help decrease boot times. A microVM is a minimal OS that runs on top of the VMM and provides only the features required to run the application [1]. MicroVMs usually use a custom compiled minimal Linux kernel and a minimal set of userspace software. MicroVMs speed up boot times by reducing how much needs to be loaded into memory and reducing the number of features that need to be initialised.

Traditional VMs also use lots of system resources, reducing resource utilisation and working against FaaS platform requirement number 2. Firecracker and microVMs also help with resource utilisation:

- Firecracker has a much smaller memory overhead per VM than traditional VMMs. QEMU has a memory overhead of around 131 MB, while Firecracker has an overhead of around 4MB [1]. Also, microVMs require much less memory than traditional OSs due to their limited feature set. This reduction in memory overhead from Firecracker and microVMs means more

function instances can be placed on a single machine, directly increasing resource utilisation.

  – Firecracker and microVMs can indirectly increase resource utilisation through fast boot times. The faster the boot times, the lower the cost of cold-starts. Consequently, the FaaS platform can be more aggressive in removing warm function instances. Fewer warm function instances mean less memory is consumed by idle VMs, allowing more hot function instances to exist on a single machine, increasing resource utilisation.

FaaS providers need to ensure that all function instances are isolated. The fact that each instance runs within a VM ensures this. Each VM is restricted to using only a portion of the host's resources. So as long as the total resources required by the provisioned VMs does not exceed the resources of the machine they are running on, each VM has performance isolation. The hypervisor creates a barrier between the VM and the host machine; this means that multiple VMs running on the same host cannot interact with each other. This provides operational isolation.

- **Google Cloud Functions** is Google's FaaS offering. Unlike AWS Lambda, which uses VMs for each function instance, Google Cloud Functions use containers. Each function instance is a container and each container can only serve one invoke at a time [21].

  Traditional container technology uses Linux kernel features, namely cgroups and namespaces, to provide both performance isolation and operational isolation. The issue with traditional Linux-kernel-based containers is that they expose Google to Linux kernel bugs, allowing malicious users to escape from the container and potentially access other users' containers [22]. To mitigate against this, Google developed a new, more secure containerisation method. Google developed an application kernel called gVisor. gVisor emulates the Linux kernel and implements "a substantial portion of the Linux system call interface" [24]. gVisor consists of two user space applications: the sentry and the gofer. A process running within a gVisor-based container has all of its system calls intercepted by the sentry. If possible, the sentry processes these system calls internally, bypassing the host kernel entirely. This is depicted in figure 2.1. The sentry, gofer and application code all run within a traditional Linux-kernel-based container. So to break out of a gVisor container, a user must exploit a bug in

gVisor to gain control over the sentry/gofer and then exploit a bug in the Linux kernel to escape the Linux-kernel-based container. This extra level of isolation between the application code and the host OS's kernel provides the operational isolation needed.

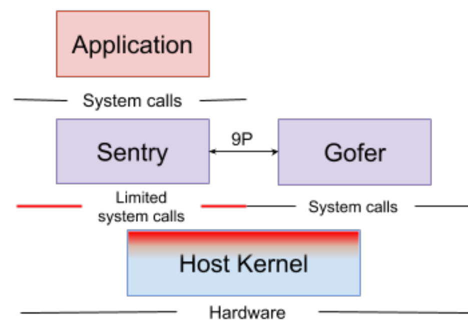


Figure 2.1: Diagram depicting a gVisor container. Source: [24]

Google Cloud Functions work similarly to AWS Lambda, scaling with demand and performing warm and cold-starts.

According to Dong Du et al. in Catalyzer: Sub-millisecond Startup for Serverless Computing with Initialization-less Booting [15], across multiple different applications, gVisor and Firecracker have comparable startup latencies, with gVisor sometimes booting faster and Firecracker booting faster at other times. This means that gVisor and Firecracker may have similar performance concerning the speed and consistency requirements of a FaaS platform.

gVisor uses more memory than traditional Linux-kernel-based containers [23], reducing resource utilisation. However, since Google is using gVisor for their function instances instead of Linux-kernel-based containers, Google must consider this cost beneficial for the extra security provided.

- **OpenFaaS** is a popular open-source FaaS platform. It has 22.9k stars on GitHub, making it the most popular FaaS platform on GitHub [28]. OpenFaaS runs on top of Kubernetes, which is an industry standard container orchestration tool for deploying production applications that are highly available and scalable. Each function instance is, by default, a traditional Linux-kernel-based container; however, gVisor can be used if required. Unlike AWS Lambda and Google Cloud Functions, each function instance can run multiple invocations concurrently. Each invocation runs as a separate process within the container.

Compared to AWS Lambda and Google Cloud Functions, OpenFaaS has a much looser level of isolation. Two invocations of a function running simultaneously in the same container may compete for resources, meaning there is no performance isolation. There is also less operational isolation. Multiple processes within the same container share the same file system, meaning a poorly designed function could exhibit unintended behaviour if using temporary files. From a security point of view, it is also less isolated because if a malicious user exploits the function code to gain control of a process, they could potentially access the state of another process, for example, by reading another process's temporary files. This differs from AWS Lambda and Google Cloud Functions, where a malicious caller must exploit both the function code and Firecracker/gVisor to access another executing function. However, there is a resource utilisation benefit to this model. Linux-kernel-based, and gVisor-based containers have overheads. By sharing the overhead between multiple function invocations, resource utilisation is increased.

OpenFaaS scales differently to AWS Lambda and Google Cloud Functions. OpenFaaS provisions a number of containers per function to serve the invoke requests, and it sends an incoming request to one of the containers at random, regardless of how busy the container already is. When the number of requests per second to a particular function reaches a threshold, OpenFaaS scales up the number of function instances by a 'factor'. While the new containers are being created, new incoming requests do not wait for the new instances but instead are routed to the already busy containers.

There is a balance that needs to be met when setting up OpenFaaS scaling. This balance is between function performance and resource utilisation. If the 'requests per second' threshold is set low and the scaling 'factor' high, all invocations will have more than enough resources to run, meaning there would be high performance but low resource utilisation. On the other hand, if the 'requests per second' threshold is high and the scaling 'factor' is low, each container will be fully utilised, meaning a high resource utilisation. However, each running process will be fighting for resources within the containers meaning a low performance.

2.2 Unikernels and their Application to FaaS

A unikernel is an application packaged into a "single bootable VM image that runs directly on a standard hypervisor" [27]. Traditionally, application code and kernel code execute separately. They both have a separate memory address space and execute at different privilege levels. The application cannot directly access system resources and must make system calls to the kernel to do so. Unlike traditional applications, unikernels combine the application and kernel code into a single-purpose standalone image where application code and kernel code execute as one. Unikernels execute with a single memory address space and only one privilege level. Kernel features, such as memory allocators or networking support, are implemented as libraries included with the unikernel at compile time. Any features/libraries that would exist in a traditional kernel but are not required by the application are not included in the unikernel, making the unikernel image small and reducing the attack surface, making the unikernel more secure.

Because a unikernel only includes the necessary features, a running unikernel takes up much less space in memory. For example, when Kuenzer et al. ported Nginx, SQLite, and Redis to unikernels, each application used less than 10MB of memory compared to 29-30MB when running as a microVM [26]. Including only the necessary features also improves boot times because less code is loaded into memory, and less has to be initialised.

Unikernels also allow for performance improvements. Firstly, because unikernels are single-purpose and immutable after compile time, they only require a single address space, and everything can run at the highest privilege level. This removes the need for costly system calls and domain switches because system calls can be replaced with simple function calls [26].

Unikernels further allow for performance improvements by allowing users "to hook the application at the right level of abstraction to extract best performance" [26]. This has allowed Kuenzer et al. to achieve 1.7x performance improvement for particular applications when run on unikernels compared to the same application running on a Linux VM [26].

If a FaaS platform used unikernels instead of containers or microVMs, it might better meet the FaaS platform requirements mentioned in 2.1.1:

- First is the function isolation requirement. Unikernels run as VMs on top of a hypervisor, similar to how AWS's microVMs run on top of a hypervisor. Each

VM can be limited to a portion of the host's resources, providing performance isolation. The hypervisor acts as a barrier between the application code and the host OS, meaning that multiple unikernels running on the same host cannot interact with each other, providing operational isolation.

- Second is the requirement to maximise resource utilisation. Unikernels take up less space in memory than traditional Linux VMs, meaning more function instances can be fit onto a given machine, increasing resource utilisation.

  In addition, unikernels can boot in 10s of milliseconds [26], meaning the cost of cold-starts is low. Consequently, the FaaS platform can remove warm function instances more aggressively, freeing up space, and allowing more hot function instances to exist on a single machine, further increasing resource utilisation.

  Finally, if properly optimised, unikernels can have improved performance, reducing execution time. This increases the number of invokes per second that can be handled by a single machine, increasing resource utilisation.

- Finally is the requirement of fast and consistent performance. Unikernels have been shown to have higher performance on particular applications as compared to the same application run on a Linux host. Unikernels also have faster boot times than microVMs reducing the performance disparity between cold and warm starts, increasing performance consistency.

2.3 Related Works

There are few papers looking into how unikernels can be used for serverless computing. Two notable papers are:

Fingler, Akshintala and Rossbach looked into how unikernels can improve serverless extract transform and load (ETL) [19]. They state how "all serverless workloads can be transformed to ETL", then focus on the design of the unikernel itself and how the unikernel can be optimised to perform ETL on a serverless platform.

Cadden et al. looked into how unikernels can be deployed and initialised faster by caching VM snapshots of pre-initialised unikernels. This reduces the function start time by "skipping the following paths: booting the unikernel, initializing the language runtime, and importing and compiling the function code and dependencies" [11].

The work in this third year project differs from the above two papers because it focuses more on orchestrating unikernel on a FaaS platform and not on the design of

the unikernels themselves. Also, described in this paper is a unikernel based extension to a production-grade FaaS platform. This allows for a more real-world evaluation of how a unikernel based FaaS platform performs.

Only two papers are mentioned in this section because little research has been done into this area. This could be because unikernels and their build systems are still very immature and can be hard and finicky to use. There are not many documented cases of people using unikernels in production systems showing that they are not currently accepted as robust and dependable by the industry. But all of this is just speculation.

Chapter 3

Design

One of the aims of this project is to extend an existing FaaS platform to execute functions as unikernels. First, an appropriate and existing open-source FaaS platform was selected, and then the extension was designed.

3.1 Needed Features of the Extension

- The FaaS platform should be able to execute functions inside unikernels, microVMs and containers. This is so a fair analysis of the three technologies can be performed.

- Ideally, this extension should be as compatible with the original FaaS platform as possible. Users should require little to no extra work to use the extension. Once set up, the function callers should be blind to which underlying technology is used to run their functions (other than potential performance differences).

- The FaaS platform should be able to scale up unikernel, microVM and container function instances as demand grows by performing cold-starts.

- The FaaS platform should be able to perform warm-starts by reusing unikernels, microVMs and containers. This will reduce the number of cold starts required, increasing the average performance of the platform.

3.2 Open-Source FaaS Platform Selection

Ideally, the FaaS platform chosen to extend should be modular in design, making it easy to extend and making the extension compatible with the original platform. It is also important that the FaaS platform is widely used. The most popular FaaS platforms will be well-built, have a good set of features, have a practical design and be reliable. The higher the quality of the original FaaS platform, the higher the quality of the final solution.

By searching GitHub, reading many online articles and watching recordings of conferences, a list of candidate, open-source FaaS platforms was compiled.

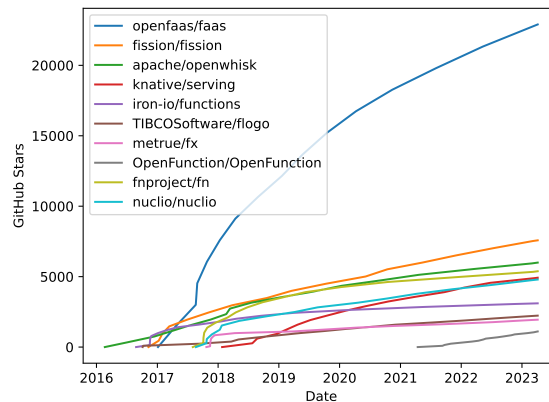


Figure 3.1: GitHub star history of candidate open-source FaaS platforms

As shown by figure 3.1, OpenFaaS is by far the most popular open-source FaaS platform according to GitHub stars. OpenFaaS has also been presented many times at the well-respected Cloud Native Computing Foundation conference [16, 17, 18, 10]. This, alongside their high quality and extensive documentation supplied on their website, gives a high level of trust that this project is of high quality and is a good candidate for extension.

OpenFaaS also meets another requirement that the platform be modular in design. OpenFaaS has a frontend called the gateway and a modular backend called the provider. The unikernel extension to OpenFaaS is implemented as this OpenFaaS backend provider. This provider can be selected when deploying the OpenFaaS platform, allowing unikernel function instances to be supported natively in OpenFaaS.

OpenFaaS already has a recommended, production-ready provider called `faas-netes`.

`faas-netes` deploys its containers using Kubernetes, a container orchestration tool designed to host highly scalable and highly available production applications. `faas-netes` was used as the base implementation, extended to support unikernels.

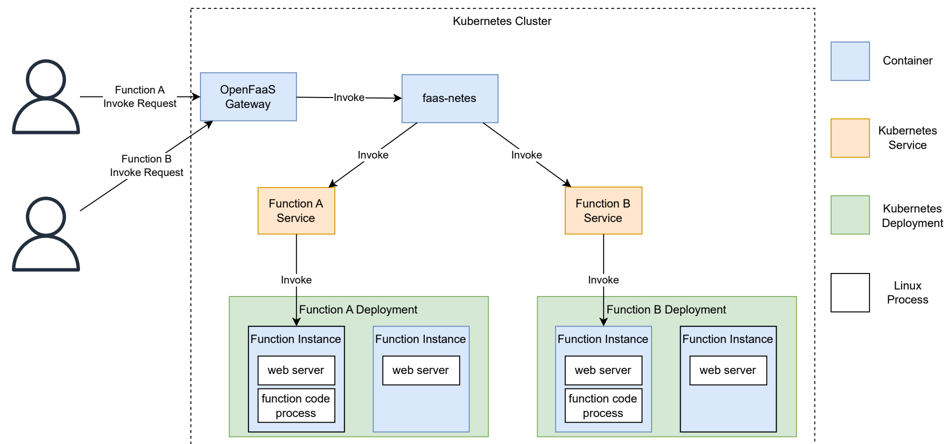


Figure 3.2: Diagram of OpenFaaS deployed to Kubernetes using faas-netes

The `faas-netes` provider comprises many components deployed as containers on Kubernetes as depicted in Figure 3.2. The OpenFaaS gateway is the frontend of the OpenFaaS deployment. It provides endpoints to manage and invoke functions. Most requests to the OpenFaaS gateway are forwarded to the `faas-netes` provider. The `faas-netes` provider uses a Kubernetes feature called deployments to manage and scale functions and their running instances. When `faas-netes` gets an invoke request, it routes this request to an appropriate function instance using another Kubernetes feature called services.

3.3 Extension to OpenFaaS: OpenFaaS-Hypervisor

First, the unikernel function instance scaling model was decided. `faas-netes` function instances can execute multiple invocations simultaneously in the same container, and `faas-netes` scales up the number of containers by a factor if the number of requests per second reaches a threshold. There are downsides to this approach:

- Functions are not adequately isolated from each other. There is no performance isolation because processes within the same container will compete for resources. There is weak operational isolation because processes share resources like the file system.

- The scaling model is complex. Users must decide at what threshold they should scale up and by what factor. When the threshold is set low and the factor high, all invocations will have more than enough resources to run, meaning there would be high performance but low resource utilisation. On the other hand, if the threshold is high and the factor is low, each container will be fully utilised, meaning a high resource utilisation. However, each running process will be fighting for resources within the containers meaning a low performance.

A much simpler solution adopted by both AWS and Google allows only one function invocation at any time per function instance. This fixes the isolation issue and simplifies the scaling model. This allows functions to scale up directly in response to demand, creating new function instances as they are required. As long as the cold-start times are low enough, this scaling model provides much more predictable performance. Consequently, this extension to OpenFaaS was designed to use AWS's and Google's scaling model for both unikernels and microVMs. To obtain a valid evaluation at the end of this project, OpenFaaS had to scale containers similarly. Because of this, the container scaling model was changed to match.

As described by Google [22], Linux-kernel-based containers do not provide enough isolation and Google uses gVisor to solve this. This same method is used in this extension to OpenFaaS to provide the same level of isolation as provided by Google.

When creating VMs, short boot times and low memory overheads are desired. For this reason, it would be best to use the Firecracker VMM over other available VMMs because it is a new, state-of-the-art VMM designed for speed and scalability.

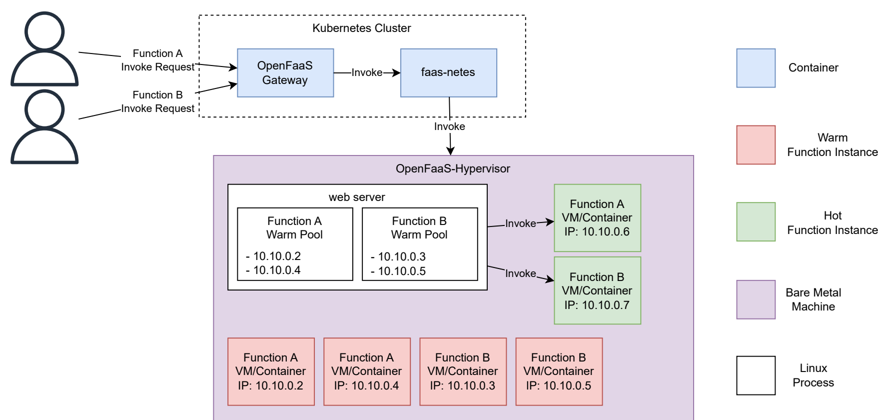


Figure 3.3: Diagram of the unikernel extension to OpenFaaS

To implement unikernels, microVMs and the new container model, a new component was added to `faas-netes`. This component is called the 'OpenFaaS-Hypervisor'

and is depicted in figure 3.3. The `faas-netes` provider was modified to no longer create containers but instead route function invocations to the OpenFaaS-Hypervisor. All function instances exist on the OpenFaaS-Hypervisor and run as either unikernels, microVMs or gVisor containers. When the OpenFaaS-Hypervisor receives a function invocation request, it routes this request to a function instance to execute it, creating a new one first, if required.

The OpenFaaS-Hypervisor is comprised of three main components:

- The **web server** acts as the interface into the OpenFaaS-Hypervisor. It implements all the control logic required to manage function instances and route function invocation requests to function instances.

- The **function instances** run as VMs or containers. These VMs/containers execute the function code. Each VM/container is configured to run a particular function and can only serve one invoke at a time.

- The **VMM** is used to create VMs and the **gVisor container runtime** is used to create gVisor containers.

The function instances and the OpenFaaS-Hypervisor need to be able to communicate. The OpenFaaS-Hypervisor has to be able to pass invoke requests to the instance, and the instance has to be able to respond with the function output. To perform this communication, the function instances and the OpenFaaS-Hypervisor pass HTTP requests through a virtual network. HTTP was used because it creates a straightforward and easy to understand communication model. Each function instance has a unique IP address, and when the OpenFaaS-Hypervisor wants to invoke a particular instance, it sends an HTTP post request, and the function output is sent in the HTTP response.

The OpenFaaS-Hypervisor must be able to scale up using cold-starts and reuse warm function instances for warm-starts. To be able to do this, the OpenFaaS-Hypervisor must keep track of all function instances and know which ones are currently being invoked and which ones are idle/warm and are ready to be invoked. To do this, a pool of warm function instances is maintained. This pool is initially empty. When an invocation request occurs, the OpenFaaS-Hypervisor checks the pool for any warm instances. If there are none, it creates a new function instance and sends that new instance the invoke request. But, if the pool has a warm instance available, the instance is removed from the pool and is sent the invoke request. After the instance has served the invoke request, it is placed back into the warm pool to be invoked again.

Creating and initialising a new function instance takes time, and this time depends on which technology is used and what machine the OpenFaaS-Hypervisor is run on. If an invoke request is sent to a function instance while it is still booting, the function instance will not be able to process the request. So, to prevent invoke requests from being sent before a new function instance is ready, a mechanism is needed to know when the instance has been initialised. To handle this, once the function instance has been initialised, it sends an HTTP request to the OpenFaaS-Hypervisor, registering itself as ready. Once this has happened, the OpenFaaS-Hypervisor can send the new instance an invoke request.

Chapter 4

Implementation

4.1 OpenFaaS-Hypervisor

As mentioned in the design section, one of the OpenFaaS-Hypervisor components is a web server. The web server was written in Golang. The first reason for this is that the whole of OpenFaaS is already written in Golang, and keeping everything consistent across the project is desirable. The second reason is that the language is fast. This is important because one of the requirements of FaaS platforms is to provide fast function execution. The third reason is that Golang is designed for concurrent programming, which helps with the implementation of the OpenFaaS-Hypervisor.

To keep track of the function instances, the web server stores an `InstanceMetadata` struct for each instance. These structs are all stored in a map which uses the function instance IP as the key since each instance has a unique IP.

As mentioned in the design section, the OpenFaaS-Hypervisor needs to keep a pool of warm function instances. Multiple pools are actually required, one for each function. These pools must be thread-safe because multiple invocation requests may simultaneously be looking for a warm instance. These pools were implemented as thread-safe lock-free queues. To add a function instance to the pool, the instance's `InstanceMetadata` is added to the end of the queue. To take an instance from the pool, the `InstanceMetadata` at the front of the queue is removed and returned to the caller. If the pool is empty when trying to remove an instance, the pool creates a new instance, returning it to the caller.

Golang web servers use separate threads to handle each incoming HTTP request. As depicted in figure 4.1, when a cold-start happens, a request enters the web server and is processed on thread A. This thread proceeds to create a new function instance.

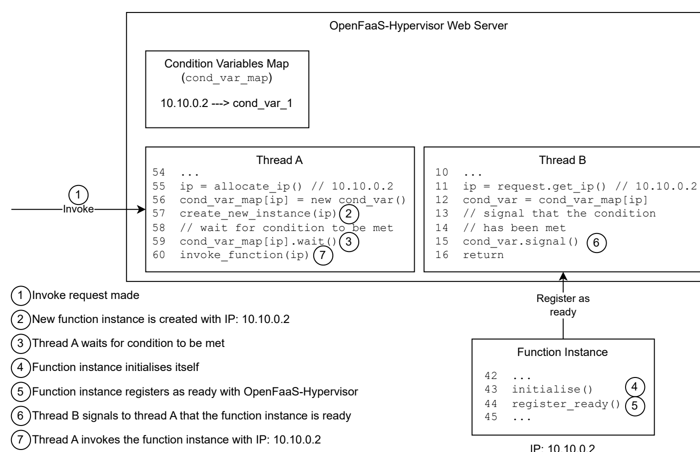


Figure 4.1: Diagram illustrating the sequence of events when function instances register as ready

When this new function instance has been created and initialised, the new instance registers itself as ready with the OpenFaaS-Hypervisor by sending an HTTP post request. This request is processed on thread B. Now thread B has to communicate to thread A that the function instance is ready to be invoked. So, a mechanism is needed to communicate from one thread to another. A condition variable is used to perform this communication. Before creating a new function instance, a new condition variable is created and stored in a map using the instance IP as the key. After creating the condition variable and requesting a new function instance, thread A blocks and goes to sleep, waiting for the condition to be met. Once the new instance is ready, it sends a request to the OpenFaaS-Hypervisor indicating so. This request is handled on thread B. Thread B reads the instance IP from the request, and uses this to get the condition variable that thread A is waiting on. Thread B then signals that the condition has been met. Thread A is then woken from sleep and invokes the new instance.

Configuring networking between the OpenFaaS-Hypervisor and the function instances was more challenging than expected. When the OpenFaaS-Hypervisor starts up, one of the first things it does is set up a network bridge. Network bridges are a feature of Linux that act like network switches, routing packets between network interfaces. Linux has another feature called TAP devices. TAP devices are virtual network devices which emulate physical network cards that carry Ethernet frames. Every time the OpenFaaS-Hypervisor creates a new virtual machine, the OpenFaaS-Hypervisor creates a new TAP device, attaches it to the network bridge, assigns it an IP address and gives it to the VMM. The VMM attaches this TAP device to the VM allowing the

VM to communicate with everything else attached to the bridge, including the host machine and, by extension, the OpenFaaS-Hypervisor.

When running containers on the OpenFaaS-Hypervisor, the networking is configured slightly differently. There exists a project called the Container Network Interface (CNI) project. This project believes that "many container runtimes and orchestrators will seek to solve the same problem of making the network layer pluggable" [14]. Hence they created a project to reduce code duplication and simplify the process. This directly addresses the needs of this project because the OpenFaaS-Hypervisor is a container orchestration tool and needs a pluggable network layer. Linux-kernel-based containers and gVisor-based containers both use a Linux kernel feature called namespaces. Namespaces are used to reduce the kernel resources visible to each process. Network namespaces are used to reduce the scope of network resources available to a process. Before creating each container, the OpenFaaS-Hypervisor creates an empty network namespace with no network interfaces. The OpenFaaS-Hypervisor then uses the CNI to create a virtual network interface in this namespace attached to a network bridge on the host. When the OpenFaaS-Hypervisor creates the container, it sets the container's network namespace to the one just created. Consequently, the container has access to one network interface that can communicate with the OpenFaaS-Hypervisor.

Throughout the OpenFaaS-Hypervisor's web server, thread safety needed to be ensured. Many maps are used throughout the web server, which need to be accessed concurrently. Luckily Golang has a `sync.Map` type, which implements a thread-safe map. Each VM needs a unique IP address and TAP device name. To ensure this, these IPs and names have to be allocated centrally. This allocator also had to be thread safe. An atomic counter was used to implement these allocators. This allowed the concurrent creation of unique names and IP addresses for each VM instance.

To evaluate a unikernel based FaaS platform against microVM and container based FaaS platforms, statistics have to be gathered. The web server has a statistics variable that stores a list of function instance initialisation times and a list of function execution times.

Recording the function execution times is simple. When a function invocation request comes in, the OpenFaaS-Hypervisor stores the start time, and once the function has been executed, it records the end time. The OpenFaaS-Hypervisor finds the difference between the start and end times and stores this value in the above statistics variable. To record the function instance initialisation time, the OpenFaaS-Hypervisor first records the start time when the function instance is created. This is stored in the

function instance's `InstanceMetadata` struct. Then, when the function instance registers as ready, the OpenFaaS-Hypervisor records the end time and subtracts the start time stored in the function instance's `InstanceMetadata` struct. This value is then stored in the above statistics variable.

The structure that stores the statistics is accessed concurrently by many threads. To make the structure thread-safe, locks ensure only one thread has access at any time.

Once all the statistics have been gathered internally, they must be exported from the OpenFaaS-Hypervisor web server. An endpoint was added to the web server, allowing easy access to the data. Not all the raw data gathered was output, instead, the data was pre-processed on the web server before sending it out. This reduced the amount of processing required later. The web server outputs the number of running VMs and the average, standard deviation, 95th percentile and maximum for both the function instance initialisation times and the function execution times.

When deploying a function instance as a unikernel, microVM or container, different artefacts are required. For microVMs, a kernel binary and a file system is needed. For unikernels, a kernel binary is needed. For containers, a file system is needed. To simplify the design, all the required artefacts are precompiled and included with the OpenFaaS-Hypervisor before deployment. These artefacts are stored on the file system so the OpenFaaS-Hypervisor can access them.

4.2 Function Instances

Each function instance is implemented as either a unikernel, microVM or container. Regardless of which technology is used, each function instance has to make an HTTP post request to the `/ready` endpoint on the OpenFaaS-Hypervisor when it is ready to be invoked, and each function instance has to implement the `/invoke` HTTP endpoint that can be called to invoke the function.

4.2.1 Unikernel

Many build systems exist that allow developers to build unikernel binary images. For this project, a build system called Unikraft was used. Unikraft has a core set of APIs that are implemented by micro-libraries. These micro-libraries implement core features of Unikraft. The Unikraft build system allows developers to choose which micro-library implementation of a specific API they want to include in their unikernel and it

also allows developers to exclude whole APIs altogether, removing redundant code from their unikernel. This allows developers to create highly specialised unikernels that are minimal and use specific micro-libraries that best optimise what they are doing [26].

Unikraft was chosen for the build system because it produces small image sizes, with low memory requirements and high execution performance compared to other unikerel build systems [26]. Another benefit is that Unikraft is actively being developed, meaning it will likely improve and continue to be supported. In addition, an active Discord server exists, providing support when needed. This is sometimes required due to the immaturity of unikernel and Unikraft technology.

When evaluating the unikernel based FaaS platform, the focus was on function orchestration. Because of this, only simple functions were used. This means that the unikernel used in testing was optimised to be very lightweight. It uses musl as the c standard library and uses lwip, a very lightweight networking stack.

As mentioned in the design section of this report, Firecracker is the desired VMM. Unfortunately, Unikraft unikernels cannot run on Firecracker. Instead, QEMU was used to create unikernels because it is supported by Unikraft and is widely used.

4.2.2 MicroVM

For the microVM a minimal Linux guest OS is required. The Linux guest comprises the kernel and the user space software.

A standard Linux kernel shipped with a traditional OS, such as Ubuntu, contains many features that FaaS function instances do not require. Because of this, a custom Linux kernel was compiled, which only contains the necessary features required by the function instances. To do this, a `.config` file is provided at compile time to select which Linux kernel features are required.

Like the Linux kernel, the user space software provided by a traditional OS is too extensive and contains many applications, libraries and services not required by the function instance. A minimal file system containing only the required user space applications, libraries and services for the function instances to run was created. Alpine Linux is a minimal Linux distribution whose file system was used as a starting point. First, the Alpine Linux file system was extracted from the Alpine Linux container image. When booting into a Linux system, after the kernel has been initialised, the kernel looks for an executable at `/sbin/init`, which is run as the first process on the system.

This process's job is to initialise the machine to operate correctly. Containers work differently and do not use this `/sbin/init` executable, meaning the Alpine file system extracted from the container does not contain this executable. Typically, `systemd` or `init` are used as initialisations systems, but `openrc` was used when creating this file system because it is very lightweight. `openrc` was configured to start a web server, which first registers itself as ready with the OpenFaaS-Hypervisor and can then be called to invoke the function code. This file system is packaged as an ext4 file system passed to the VMM when creating the VM.

Firecracker VMM was used to create and manage the microVMs because it is designed for speed and scalability.

4.2.3 Container

As mentioned before, gVisor-based containers are used in the OpenFaaS-Hypervisor because they provide the necessary isolation level.

gVisor containers can be created using an "Open Container Initiative (OCI) runtime called `runsc`" [24]. OCI runtimes are applications that follow the OCI runtime specification, allowing users to create and manage containers on a system. Because gVisor implements this standard interface, it is relatively easy to set up and use. OCI runtimes can create a container from a `config.json` file and a `rootfs` directory that contains the file system. So for each function, a `config.json` file and a file system containing the function code are needed. For each function instance, a copy of `rootfs` is made, and an edited copy of the `config.json` is made, which references the instance's network namespace configured by the CNI.

As referenced above, the CNI is used to aid container networking setup.

The artefacts required to create a container, `config.json` file and `rootfs` directory, are very small. The function used to evaluate the system only has a single binary executable in the `rootfs` directory.

4.3 Deployment of the OpenFaaS-Hypervisor

Once the OpenFaaS-Hypervisor had been built and `faas-netes` was modified to use the OpenFaaS-Hypervisor, the whole system needed to be deployed to a Kubernetes cluster.

Setting up a Kubernetes cluster manually is a complex task, but luckily there exist many Kubernetes distributions that make the installation process a lot easier. The Microk8s Kubernetes distribution was used to create the Kubernetes cluster for evaluating the unikernel based FaaS platform. Microk8s is an officially certified Kubernetes distribution [12], meaning it is of high quality, allowing for a valid evaluation of the FaaS platform.

Unfortunately, the OpenFaaS-Hypervisor needs to run on a bare metal machine and not within a container. This is because, from my testing, the function instances run slower when inside a container. Because of this, the OpenFaaS-Hypervisor has to be deployed manually on a bare metal machine and not within a Kubernetes container.

The original `faas-netes` is typically deployed using a tool called Helm. Helm is a tool that allows developers to define, deploy and manage a set of Kubernetes resources. Helm uses yaml files, called charts, to define the Kubernetes resources required. These charts can then be deployed, creating the required resources on the Kubernetes cluster. OpenFaaS has a Helm chart that defines all the necessary components to run OpenFaaS on Kubernetes. So to deploy the unikernel extension of OpenFaaS, the Helm charts were updated to use the modified version of `faas-netes` and the new Helm charts were then deployed.

Chapter 5

Evaluation

5.1 Method

To evaluate the unikernel based FaaS platform, measures were made to see how well it meets the requirements of a FaaS platform, as compared to a microVM or container based FaaS platform.

FaaS platforms want to minimise their operational costs by maximising resource utilisation. Four measures were made to evaluate how well each technology meets this requirement:

- The system memory required to run a single function instance using each technology was measured. The lower the memory requirements, the more function instances can fit on a given machine, increasing resource utilisation.

- The function instance initialisation time for each technology was measured. The lower the function instance initialisation time, the lower the cost of cold-starts. Consequently, the FaaS platform can be more aggressive in removing warm instances, allowing more hot instances to exist on the server, increasing resource utilisation.

- The total cold and warm start function execution times for each technology was measured.

  This provides a measure of the disparity between cold and warm start times, giving another data point for the cost of cold-starts. This measure differs from the initialisation time because it incorporates the time to execute specific function

code. Initialisation times give a more general result regarding the cost of cold-starts.

Also, the faster a function executes, the more invokes per second can be handled by a single machine, increasing resource utilisation. It may also allow FaaS providers to run functions on older, cheaper hardware without affecting customer experience.

- The static disk space required to store the compiled function artefacts for each technology was measured. Smaller artefacts allow for more functions to be stored on a given drive and more functions to be transmitted over a given network, increasing resource utilisation. Modern storage technologies are cheap, and modern networks have high bandwidths. But this does not mean that these measures are pointless. In a large-scale, high-demand system such as AWS Lambda, the amount of storage and the total bandwidth used to transmit functions adds up. Also, a FaaS platform may wish to use a caching mechanism to reduce the network delay in fetching artefacts. In this scenario, storage size has a significant impact.

Users want their functions to execute with fast and consistent performance. Two measures were made to evaluate how well each technology meets this requirement:

- The function instance initialisation time for each technology was measured. The lower the function instance initialisation time, the smaller the disparity between warm and cold starts, improving performance consistency.

- The total cold and warm start function execution times for each technology was measured.

  This provides a measure of the disparity between cold and warm starts, giving another data point for performance consistency. This measure differs from the initialisation time because it incorporates the time to execute specific function code. Initialisation times give a more general result regarding performance consistency.

  The cold and warm start times also directly measure function performance.

- As mentioned in the above points, the function instance initialisation times, cold-start times and warm-start times were measured. Looking at these as single data points tell us something, but it is also important to look at how these measures

change when the system is under load. So, the above measures were also measured at many different levels of system load. The above measures were first gathered by sending a single function invocation request, then gathered a second time by sending two simultaneous invocation requests, then a third time by sending three simultaneous invocation requests and so on.

This allows the FaaS platform's performance to be evaluated under different loads, giving another measure for consistency of performance.

FaaS platforms have a requirement to isolate each function instance. Performance isolation was evaluated by looking at the functions' cold and warm start times when the system is at varying loads. If the functions execute slower when the system is under a high load, then there is low performance isolation. Unikernels have as much operational isolation as microVMs because they use the same mechanism to isolate between the application code and the host OS. Because unikernels provide the same amount of operational isolation as microVMs used by AWS Lambda, a major cloud service provider, it can be said that unikernels provide enough operational isolation between functions with no further testing.

The amount of memory required by each function instance was measured as the resident set size (RSS) of the function instance. The RSS measures how much physical memory is allocated to a process. This measure includes both the memory consumed by the function code and the underlying runtime. For the VMs, this is the combination of the memory used by the VMM plus the memory used by the guest OS and function code. For the containers, this is the memory used by gVisor, plus the memory of each process running within the container. This RSS measure is good because it measures exactly how much physical memory is used, meaning it scales up linearly with the number of running instances.

When recording cold-start times, the OpenFaaaS-Hypervisor initially had zero function instances on it. A Bash script and curl was used to send a number of invoke requests in parallel. The statistics were then gathered using the statistics endpoint. The OpenFaaaS-Hypervisor was then reset so it again had zero function instances and the above was repeated except a different number of simultaneous invoke requests were sent.

When recording warm-start times, a number of function instances were pre-instantiated using a /pre-boot/<number> endpoint on the OpenFaaS-Hypervisor's web server. Then, like with cold-starts, a number of invoke requests were sent in parallel, and statistics gathered.

To keep everything consistent between each test, the exact same function code was used for each function instance. Of all the most widely used programming languages for FaaS, Python was the only one supported by Unikraft. But Python has a significant performance penalty when running in a unikernel versus a container. This is likely to be because of the immaturity of unikernels rather than an inherent problem with them. So to make a comparison between unikernels, VMs and containers that represents a real-world, mature, production system, the functions were written in C, which has comparable performance across unikernels, microVMs and containers.

Economically, it is cheaper for a FaaS platform to run a few machines with many cores and lots of memory than many machines with few cores and small amounts of memory. For this reason, most production FaaS platforms are running on high-specification machines. For example, AWS Lambda runs its functions on `m5.metal` EC2 instances with 48 cores, 96 threads and 384 GB RAM [8]. So, these tests require a comparable machine to get realistic performance measures when under a high load. The project supervisor, Pierre Olivier, kindly gave access to a server with two 24-core, 48-thread CPUs and 64GB of RAM. The CPU's hyperthreading had been disabled for performance consistency reasons.

5.2 Results

5.2.1 Function Memory Usage

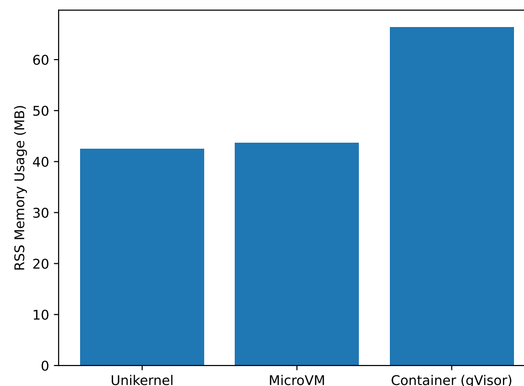


Figure 5.1: Total RSS memory usage of function instances

Firuge 5.1 shows the total RSS of each function instance. It shows that microVMs

and unikernels have similar physical memory footprints, and gVisor containers are much larger. Something important to note is that in these tests, the unikernels were run on the QEMU VMM, whereas the microVMs were run on the Firecracker VMM, which is known to be much more lightweight. Unikraft is currently working on support for Firecracker, but it will not be ready until late April (just past the third-year project deadline).

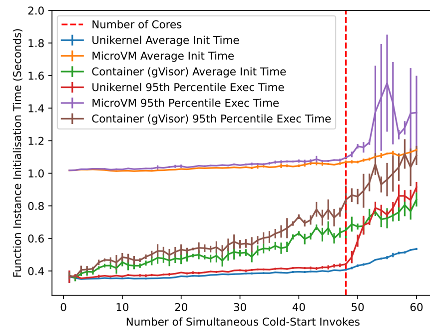
An estimate of the RSS of a unikernel running on Firecracker can be made if the Firecracker overhead is known and the memory usage of the unikernel guest OS can be measured. Firecracker has an advertised overhead per VM of less than 5MB [20]. To measure the unikernel guest OS memory consumption, the unkennel VM was repeatedly booted with decreasing amounts of memory until it failed to boot. The unikernel successfully booted with 4MB of memory but not 3MB, meaning that the unikernel guest uses at most 4MB of memory. Adding this to Firecracker's advertised 5MB overhead gives an RSS of 9MB, multiples less than the $\sim$ 43MB (see figure 5.1) of the microVM. So even if this estimate is off by a significant margin, it can be said with confidence that unikernels running on Firecracker will have a much lower RSS than microVMs or gVisor containers.

At first sight, it was slightly surprising how much worse gVisor performed. To validate that the gVisor results were correct, the results were compared against data published on gVisor's website [23]. Here they compare the memory usage of containers run using `runc` (traditional Linux-kernel-based containerisation) and `runsc` (gVisor-based containerisation). gVisor found that `runc` uses about 20MB-50MB less memory than `runsc`. The function instance used in the OpenFaaS test used 11MB when ran with `runc`, meaning that the results are comparable to what gVisor found.

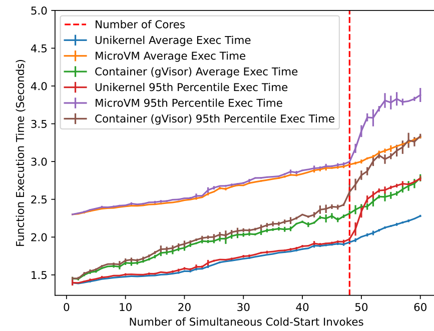
To conclude, when trying to maximise memory utilisation, unikernels today provide a minimal advantage over microVMs but a significant advantage over gVisor containers. But, in the near future, when unikernels can be run on Firecracker, unikernels may provide a significant memory utilisation advantage over both microVMs and gVisor containers.

5.2.2 Function Execution Time

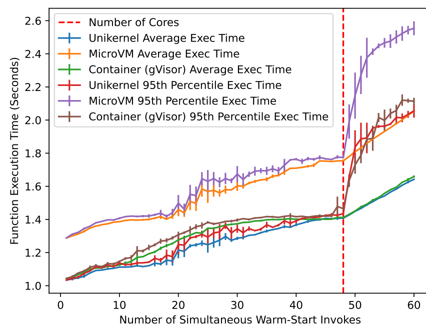
The graphs in figure 5.2 show timing data up to 60 simultaneous invoke requests. The machine on which the tests were run has 48 CPU cores. Each function instance is allocated a single CPU core because the function code is not optimised for multicore. This means that when there are more than above 48 function instances, the resources



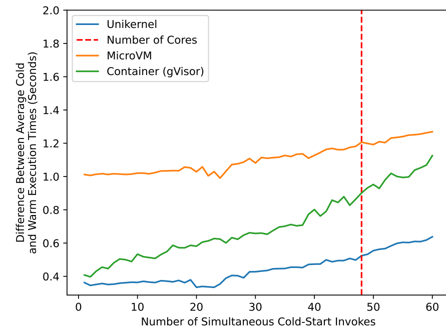
(a) Function instance initialisation times.



(b) Cold-start function execution times.



(c) Warm-start function execution times.



(d) Difference between the average cold-start and warm-start execution time.

Figure 5.2: Function execution times

required are greater than what is available. As can be seen from the graphs, this causes significant performance degradation and performance isolation significantly decreases. In a production system, whatever technology is being used, the FaaS platform has to ensure that the resources required are less than what is available. For the rest of the evaluation, only the graphs data between 1 and 48 invoke requests is considered.

Firstly, looking at figure 5.2a, when creating only a couple of new function instances, unikernels and containers have similar initialisation times, but as the number of invokes exceeds three, unikernels outperform gVisor containers by a significant margin. MicroVMs have the worst initialisation times by an even larger margin at all scales.

Secondly, looking at figure 5.2d, unikernels at all scales have the smallest disparity between average cold and warm start times.

To summarise, figures 5.2a and 5.2d firstly show that unikernels have a smaller

cold-start penalty compared to microVMs and containers. This allows the FaaS platform to more aggressively remove warm instances, allowing more hot function instances to exist on a single machine, increasing resource utilisation. Secondly, they show that unikernels provide more consistent performance. This is because a cold-start performs similarly to a warm-start.

Looking at figure 5.2b, unikernel function instances execute the function faster than containers and microVMs at all scales when performing a cold-start. Looking at figure 5.2c, microVMs are slowest when performing warm-starts and unikernels and microVMs have similar performance. Considering the average performance across warm and cold starts, unikernels execute fastest.

To summarise, figures 5.2b and 5.2c show that on average, unikernels have the fastest execution time as compared to microVMs and containers, meaning more invokes per second can be handled by a given machine, increasing resource utilisation. The figures also show that unikernels better meet the function speed requirement than microVMs and containers.

Finally, looking at figures 5.2a, 5.2b, 5.2c, as the number of simultaneous invoke requests increases, the unikernel lines increase at the same or a slower rate than the microVM or container lines. This shows that the average timings for unikernels scale as well or better than both microVMs or containers. The unikernel 95th percentile lines are all close to the average lines. This shows that each individual function execution has a consistent performance at any given scale. Both these values show that unikernels performance consistency matches or exceeds microVMs and containers. It also shows that function performance isolation matches or exceeds microVMs and containers.

5.2.3 Artefact Size

When measuring the artefact size, what needs to be considered is what will actually be stored for each function. The FaaS platform will not want to store and transmit files that are the same for each function. For unikernels, the code is compiled into a single binary kernel image, so the whole unikernel needs to be stored. For microVMs, only the executable file needs to be stored and the underlying Linux kernel and file system will be consistent between functions. For containers, only the executable file needs to be stored because the container runtime provides everything else required.

As shown by figure 5.3, both MicroVMs and Containers have identical artefact sizes because they are just storing the executable, but unikernel artefacts are 30% larger because they also have to include all the underlying kernel functionality.

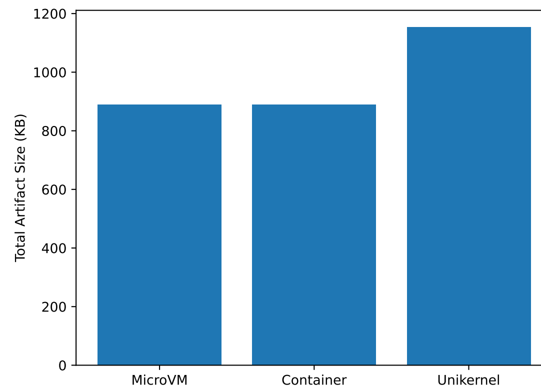


Figure 5.3: Function artefact size.

Something to note is that this function code is written in C, a compiled language. The most widely used FaaS languages are interpreted. So if an interpreted language was being used, like Python or JavaScript, all three technologies would have the same artefact size, the size of the text file containing the code. The unikernel would be compiled as the language runtime, and the code file would be provided through a file system attached to the VM at runtime.

Chapter 6

Conclusion

6.1 Summary of Results

Three of the key requirements of a FaaS platform are, providing isolation between function instances, minimising operational costs by maximising resource utilisation and providing fast and consistent performance. FaaS platforms of today typically use either microVMs or containers to meet these requirements. Unikernels are a new technology that can be used in place of microVMs and containers.

This report concluded that unikernels provide a high level of isolation between function instances. The VMM and hypervisor allocate a portion of the host machine's resources to each function instance, providing performance isolation. The VMM and hypervisor provide a barrier between the executing function code and the host's OS, which provides operational isolation.

The results section of this report shows that unikernels also help to maximise resource utilisation. The total physical memory used by each function instance is less than microVMs and gVisor containers meaning more can fit onto a single machine. When a new function instance is required, new unikernels can be initialised faster than microVMs and gVisor containers. This means that the cost of cold-starts is lower. Consequently, the FaaS provider can remove warm function instances more aggressively, allowing more hot function instances to exist on a single machine, increasing resource utilisation. Functions also execute faster on unikernels compared to microVMs and faster compared to gVisor containers when cold-starting. This increases the number of invokes per second that can be handled by a single machine, increasing resource utilisation.

This report also found that unikernels help to improve function speed and consistency. When cold-starting, unikernel based functions execute faster than both containers and microVMs, and when warm-starting, they perform better than microVMs and similar to gVisor containers. This means that unikernels performance is at least as good or better than the alternatives. Also, as the number of simultaneous function executions increases, unikernels scale at a very similar rate to microVMs and containers, and their 95th-percentile execution times compared to the averages are similar to both microVMs and containers. The disparity between cold and warm start times of unikernels is less than microVMs and containers. This all means that the predictability of the performance of unikernels matches or exceeds the alternatives.

All of the data gathered throughout this project shows that unikernels are an excellent candidate for function instances in a FaaS platform. As compared to microVMs and containers, they match the level of isolation, improve resource utilisation, improve average execution time, and match or improve the predictability of the performance.

6.2 Project Aims

The first two aims of this project were to research FaaS platforms and unikernels. An understanding of how FaaS platforms work and the requirements they face was successfully gained and this knowledge was explained within this report. An understanding of unikernels was also gained and explained in this report. A conceptual idea of how unikernels could theoretically be used in a FaaS platform was proposed and the reasons why they might better meet the requirements of a FaaS platform as compared to microVMs and containers were explained.

The third aim was to extend an existing FaaS platform to support unikernel function instances. After researching existing open-source FaaS platforms, a suitable platform was picked to extend. This platform was successfully extended. This platform extension supported unikernels, microVMs and containers all using the same scaling model so that a valid evaluation between the three could be made.

The final aim was to evaluate how well unikernels met the requirements of a FaaS platform compared to microVMs and containers. An evaluation strategy was developed and then the necessary data points were gathered. This data was then used to evaluate how unikernels compare to microVMs and containers. This report finds that for each requirement, unikernels either outperformed or matched the performance of the other technologies meaning that unikernels are an excellent candidate for function

instances.

6.3 Critical Analysis

- The unikernel extension to OpenFaaS described in this report is only able to scale the number of function instances up but not down. This is fine for the analysis carried out in this report but would not be acceptable in a real world production system. To implement scaling down, a developer would have to define some time period after which warm instances are removed.

- The unikernel extension to OpenFaaS was meant to be a fully compatible extension that, from the user's point of view, works identically to the original version. This, unfortunately, was not achieved. The original OpenFaaS uses a different scaling model from the one required by the unikernel extension. This means the user would have to define a different set of scaling parameters.

- As described in the implementation section of this report, the functions were all supplied to the OpenFaaS-Hypervisor at the time of deployment, and no new functions could be easily added afterwards. This is not acceptable for a production system. To implement this feature, a fast and scalable function store would be required to store the function artefacts. A new function will be stored here and the OpenFaaS-Hypervisor will fetch the function artefacts from here. To make this natively compatible with OpenFaaS, the artefacts must be packed as an OCI-compatible container image. Given more time, this would be achievable.

- As described in the implementation part of this report, the OpenFaaS-Hypervisor had to be deployed manually. On top of this, the OpenFaaS-Hypervisor itself cannot be scaled up, so if the OpenFaaS-Hypervisor is full of running function instances, then no new function instances will be able to be created. This is not acceptable for a production system. It would have been good if Kubernetes Custom Resources or maybe a tool like Ansible were used to deploy the OpenFaaS-Hypervisor and scale it automatically. A complete scaling solution would have been outside this project's scope due to the extra time required to implement it. It would require the function store described above. It would also require an extra routing service to send invoke requests to an instance of the OpenFaaS-Hypervisor with a warm function instance of the specific function being invoked.

This whole system would have a large amount of distributed state management to deal with.

6.4 Further Work

- As mentioned in the critical analysis, it would be good if this FaaS platform could scale down the number of function instances. Having low cold-start times allows the FaaS platform to be more aggressive when scaling down. This will improve the potential resource utilisation. It would be interesting to measure how much cold-start times affect resource utilisation. To do this, the amount of 'utilised resources' could be measured as the total amount of CPU time spent executing the function code and all the other CPU time (when initialising or when sitting warm but not executing) can be measured as 'non-utilised resources'. A realistic production load could be simulated and sent to the FaaS platform. The rate at which warm instances are removed can then be optimised to maximise the resources utilised. The maximum obtained resources utilised for microVMs, unikernels and containers could then be compared to see how much of an effect the faster boot times and cold-starts have on resource utilisation.

- As mentioned in the critical analysis, it would be good for functions to be added to the platform after deployment. It would be interesting to see how this new model would affect the measures made in this report's analysis.

  In this report, C was used to implement the function code, which meant that each function had its own unikernel image because the C code was compiled into the unikernel. But most FaaS functions are written in interpreted languages. This means the FaaS platform could have one generic unikernel binary per language runtime and then dynamically load the function code into the generic unikernel at runtime. Because there would only be one unikernel binary per language runtime, these could be cached/stored locally on the OpenFaaS-Hypervisor. Only the small text file containing the code would have to be transmitted. It would be interesting to see how this model would affect the performance measures made in this report's analysis.

  This then leads to the idea of having a function build system. Users of a FaaS platform should only have to write code, select triggers and upload that to the platform. This simple process is not supported by the FaaS platform developed

for this project. This would require a build system that would take the user's code, compile it into a unikernel, and then store that in an object store.

- Once Unikraft supports the Firecracker VMM, it would be interesting to get a concrete measure of the RSS instead of just an estimate.

Acronyms

CNI Container Network Interface.

ETL Extract Transform and Load.

FaaS Function as a Service.

OCI Open Container Initiative.

RSS Resident Set Size.

VM Virtual Machine.

VMM Virtual Machine Monitor.

Glossary

Cold start A function execution that requires a new function instance to be created before executing.

Function A uniquely named piece of application code that is associated with a set of triggering events.

Function instance A place where a specific function's code can be executed.

Hot instance A function instance that is currently executing function code.

Operational isolation Function instances have operational isolation when function instances cannot access each other's state or change each other's execution flow.

Performance isolation Function instances have performance isolation when they can execute simultaneously with no performance degradation compared to when the functions are executed standalone.

Warm instance A function instance that is sitting idle and is ready to execute function code.

Warm start A function execution that executes on a pre-initialised and running function instance.

Bibliography

[1] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, Feb. 2020. USENIX Association.

[2] AWS. Amazon dynamodb. https://aws.amazon.com/dynamodb/. Date Accessed: 10/3/2023.

[3] AWS. Aws re:invent 2018: [repeat 1] a serverless journey: Aws lambda under the hood (srv409-r1). https://www.youtube.com/watch?v=QdzV04T_kec, Dec 2018. Date Accessed: 15/3/2023.

[4] AWS. Astrazeneca's genomics data processing solution runs 51 billion tests in 1 day on aws. https://aws.amazon.com/solutions/case-studies/astrazeneca/?did=cr_card&trk=cr_card, 2021. Date Accessed: 5/4/2023.

[5] AWS. Sky italia delivers real-time sports updates, optimizes data propagation 15x on aws. https://aws.amazon.com/solutions/case-studies/sky-case-study/?did=cr_card&trk=cr_card, 2021. Date Accessed: 5/4/2023.

[6] AWS. Taco bell satisfies customer delivery demand with serverless on aws. https://aws.amazon.com/solutions/case-studies/taco-bell/?did=cr_card&trk=cr_card, 2021. Date Accessed: 5/4/2023.

[7] AWS. Waitrose rises to pandemic challenge and amazes customers with aws. https://aws.amazon.com/solutions/case-studies/waitrose-case-study/?did=cr_card&trk=cr_card, 2021. Date Accessed: 5/4/2023.

[8] AWS. Aws re:invent 2022 - a closer look at aws lambda (svs404-r). https://www.youtube.com/watch?v=0_jfH6qijVY, Dec 2022. Date Accessed: 15/3/2023.

[9] AWS. Bosch thermotechnology accelerates iot deployment using aws serverless computing and aws iot core. https://aws.amazon.com/solutions/case-studies/bosch-case-study-and-video/?did=cr_card&trk=cr_card, 2023. Date Accessed: 5/4/2023.

[10] R. Berrelleza. Webassembly + openfaas, the universal runtime for serverless functions. Okteto, CNCF [Cloud Native Computing Foundation], 2020.

[11] J. Cadden, T. Unger, Y. Awad, H. Dong, O. Krieger, and J. Appavoo. Seuss: Skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[12] CNCF. Certified kubernetes software conformance. https://www.cncf.io/certification/software-conformance. Date Accessed: 9/4/2023.

[13] CNCF. Cncf 2022 annual survey. https://www.cncf.io/reports/cncf-annual-survey-2022/, 2022. Date Accessed: 15/3/2023.

[14] CNI. Cni - the container network interface. https://github.com/containernetworking/cni. Date Accessed: 7/4/2023.

[15] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, and H. Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[16] A. Ellis. Getting beyond faas: The plonk stack for kubernetes developers. OpenFaaS Ltd, CNCF [Cloud Native Computing Foundation], 2019.

[17] A. Ellis. Meet faasd. look ma' no kubernetes! OpenFaaS Ltd, CNCF [Cloud Native Computing Foundation], 2020.

[18] A. Ellis. How and why we rebuilt auto-scaling in openfaas with prometheus. OpenFaaS Ltd, CNCF [Cloud Native Computing Foundation], 2022.

[19] H. Fingler, A. Akshintala, and C. J. Rossbach. Usetl: Unikernels for serverless extract transform and load why should you settle for less? In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '19, page 23–30, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Firecracker. Secure and fast microvms for serverless computing. `https://firecracker-microvm.github.io/`. Date Accessed: 1/3/2023.

[21] Google. Life of a serverless event: Under the hood of serverless on google cloud platform (cloud next '18). `https://www.youtube.com/watch?v=MBBQ6P3GauY`, 2018. Date Accessed: 2/4/2023.

[22] Google. gvisor: Protecting gke and serverless users in the real world. `https://cloud.google.com/blog/products/containers-kubernetes/how-gvisor-protects-google-cloud-services-from-cve-2020-14386`, 2020. Date Accessed: 30/3/2023.

[23] gVisor. Performance guide. `https://gvisor.dev/docs/architecture_guide/performance/`. Date Accessed: 17/3/2023.

[24] gVisor. What is gvisor? `https://gvisor.dev/docs/`. Date Accessed: 16/3/2023.

[25] E. Jonas, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson. Cloud programming simplified: A berkeley view on serverless computing, 2019.

[26] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, c. Teodorescu, C. Răducanu, C. Banu, L. Mathy, R. Deaconescu, C. Raiciu, and F. Huici. Unikraft: Fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 376–394, New York, NY, USA, 2021. Association for Computing Machinery.

[27] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for

the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 461–472, New York, NY, USA, 2013. Association for Computing Machinery.

[28] OpenFaaS. Openfaas ® - serverless functions made simple. https://github.com/openfaas/faas. Date Accessed: 29/3/2023.