Photons@Graal - Enabling Efficient Function-as-a-Service on GraalVM

David Jean Frickert

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor(s): Prof. Luís Manuel Antunes Veiga
Prof. Rodrigo Fraga Barcelos Paulus Bruno

June 2022

Acknowledgments

I would like to thank my thesis supervisors, Professors Luís Veiga and Rodrigo Bruno, for the continuous guidance and time spent on weekly support during the past months. I would also to thank my parents for the support during this time.

Resumo

A GraalVM é uma implementação de VM de Java moderna desenhada para obter melhor *performance* a nível de *throughput*, memória e latência de inicialização que também pode ser usada para suportar aplicações que misturam código escrito em diferentes linguagens de programação. Hoje em dia, na GraalVM, funções *Serverless* não podem ser executadas concorrentemente num só *runtime* da linguagem. O Photons [1] propôs um mecanismo de isolamento automático de dados que permite partilha de um *runtime* da linguagem. Contudo, o Photons depende de manipulação de *bytecode* em tempo de carregamento para forçar o isolamento de dados, o que é indesejável por razões de manutenibilidade e *performance* e também deixa que o espaço da *heap* seja partilhado, o que leva a gestão de memória ineficiente e causa perdas de *performance* e latência. O objetivo deste trabalho é estudar e tirar partido das características únicas da GraalVM Native Image, como (i) compilação antes-do-tempo de código Java, levando a um tempo de inicialização muito baixo, e (ii) *Isolates*, uma alocação separada de àrea de memória que pode ser ligada a funções para suportar a implementação de funções *Serverless* finas na GraalVM Native Image. Isto pode ser alcançado incorporando o Photons na arquitetura da GraalVM e implementado-o tirando partido dos *Isolates* e da compilação antes-do-tempo.

Palavras-chave: Function-as-a-Service, Serverless, GraalVM Native Image, Isolates, Cloud, Photons@Graal

Abstract

GraalVM is a novel Java VM implementation designed to achieve better performance (throughput, memory, and start-up latency) that can also be used to support applications that mix application code written in different programming languages. Currently, in GraalVM, Serverless functions cannot be executed concurrently within the same language runtime. Photons [1] proposed automatic data isolation to allow sharing the language runtime. However, Photons relies on bytecode manipulation at load time to enforce data isolation, which is undesirable for maintainability and performance reasons and it still allows functions to share the same heap space, leading to inefficient memory management, and causing performance and latency overheads. The goal of this work is to study and take advantage of GraalVM Native Image unique features, such as (i) Java ahead of time compilation, providing very low startup time, and (ii) Isolates, a separate allocation area that can be attached to functions in order to support the implementation of thin Serverless functions in GraalVM native image. This can be achieved by incorporating Photons in the GraalVM architecture and implement them while taking advantage of Isolates and ahead of time compilation.

Keywords: Function-as-a-Service, Serverless, GraalVM Native Image, Isolates, Cloud, Photons@Graal

Contents

List of Tables

List of Figures

List of Algorithms and Code Snippets

Chapter 1

Introduction

The increasing level of abstraction provided by Cloud providers, initiated by the shift from Infrastructure as a Service (IaaS) to Platform as a Service (PaaS), has led to the development of Function as a Service (FaaS) solutions that now is offered through Serverless platforms. These platforms allow programmers to write small stateless functions, containing only the necessary logic code and which are executed only when a request for the invocation of the function arrives. Such approach aims at reducing costs by charging costumers only during the time the function is actually being run [2]. This makes Serverless desirable for asynchronous and event-based workloads that don't have demand for continuous operation, resulting in cost savings when no events are occurring since the customer is only billed when the functions are being invoked. Serverless is also more elastic than other lower level platforms, as the platform will scale the amount of workers along according to the incoming invocation rate automatically, being able to handle large bursts and sporadic invocations equally well.

Serverless has a similar paradigm to Remote Procedure Call (RPC) which was a popular paradigm in distributed systems for invocation of functions with the code located in a remote server [3]. But, unlike Serverless, RPC requires a Server to be online to process the function invocation, not having the same potential for elasticity and resource freeing during idle time.

1.1 Current Shortcomings

Serverless functions are typically executed in virtualized environments, specifically, containers. In a typical scenario, every function execution creates a container with the function code, executes it, and then the container is destroyed. The overhead is quite significant, reaching hundreds of milliseconds to start a container. To solve this problem, which is commonly referred to as ”cold start”, serverless platforms keep containers alive after execution finished for a set amount of time; if more executions of the same function arrive while the container is still alive, it can be reused, skipping the cold start. Still, when no ”warm container” is available, the cold start is unavoidable.

Photons [1] aims to allow execution of concurrent requests in a container, while also introducing a global cache to share common data that can be used by the function executions. By allowing concurrent execution in containers and sharing data, less memory is used overall and cold starts are also greatly reduced for workloads with high concurrency.

However, Photons doesn't enforce a strict memory separation - all functions execute in the same heap and the memory separation that Photons employs is actually built upon bytecode manipulation at class load time. This has many downsides, such as introducing performance and security issues due to the modification of classes while the application is running. It makes the modified code harder to debug. Further, it also introduces a big dependency on the bytecode manipulation framework and conflicts with other bytecode manipulation techniques that the application might be using.

1.2 Objectives and Deliverables

The main goal of this project is to improve application execution in FaaS scenarios, by achieving a stricter data isolation than the original Photons[1] design while providing similar memory benefits when executing applications with high concurrency degrees.

Thus, we present Photons@Graal. By using GraalVM Native Image Isolates as means of providing data isolation, it provides disjoint heap allocation through the Isolate API, enabling a strict memory isolation of each task execution. It also greatly reduces Garbage Collection activity and overhead in low memory task scenarios, in which Isolates are simply destroyed at the end of the execution without any Garbage Collection being carried out during the execution.

Photons@Graal allows GraalVM functions to have multiple concurrent executions within the same runtime by having each function invocation thread attach to an Isolate. Data isolation between concurrent function executions is ensured due to Isolates having disjoint heaps, making it impossible for function invocations to interfere with the memory of other function invocations.

In order to evaluate and tune the implementation of Photons@Graal, we will integrate with Apache OpenWhisk, which is an Open-source Serverless framework and is the backbone of the proprietary service IBM Cloud Functions, possibly with added proprietary code. This allows us to replicate a Cloud Serverless environment using local machines with a system that is highly customizable, which is needed to be able to create and use a custom Runtime that supports Photons@Graal. With some modifications we can also allow OpenWhisk to forward concurrent requests to a single runtime which isn't normally available in Cloud environments.

We perform both synthetic and realistic workload benchmarks, using some selected functions that are meant to represent multiple types of functions that are widely used in Serverless platforms to assess the quality of the solution when comparing with some existent systems.

1.3 Document Roadmap

This document is structured in the following way. Chapter 2 describes the research on the state of the art of the multiple relevant themes and how they are all connected into the work that is proposed. Chapter 3 contains the proposed architecture of the system that aims to solve the issues described in the current shortcomings shown earlier in this Chapter and Chapter 4 contains the implementation details. In Chapter 5 we describe the evaluation environment, the synthetic and cluster-wide benchmarks designed to evaluate our system and compare with existing alternatives. Finishing, we draw some conclusions in Chapter 6 and establish some improvements for a future iteration.

Chapter 2

Related Work

In this section we present the most relevant work related to the main topics at hand. In Section 2.1 we present the evolution of Distributed Application Architecture over time. Section 2.2 describes the evolution of Cloud Service Levels. Section 2.3 addresses the various types of Cloud Deployments. Section 2.4 studies Managed Runtimes, and in Section 2.5 Relevant Systems, i.e., systems that achieve similar goals as this work, will be discussed.

2.1 Distributed Application Architecture

This section addresses different distributed application architectures starting from Web Services to Serverless, showing the evolution of monolithic Web Services into smaller components with microservices, and ending on the small logic units with serverless.

2.1.1 Web Services

Web Services started as a standard to provide a web interface that can be formally described in a contract file, such as Web Services Description Language (WSDL). Typically, such web services exchange Simple Object Access Protocol (SOAP) serialized XML messages while serving requests. Web Service developers would hand over an WSDL file to potential consumers of the API of the service, so that they know what resources are available, what parameters and data structures are used [4].

Nowadays, Web Services are more of a broad term that represent any kind of service that has a web interface that can be documented, such as the popular RESTful over JSON or just plain HTTP. For example, for RESTful APIs, the OpenAPI Specification became a popular specification to describe the API for resources, parameters and data structures, similar to what WSDL is for SOAP services.

2.1.2 Microservices

The microservices architecture came as a form of separating high level responsibilities in smaller independent services, that communicate among themselves using lightweight protocols.

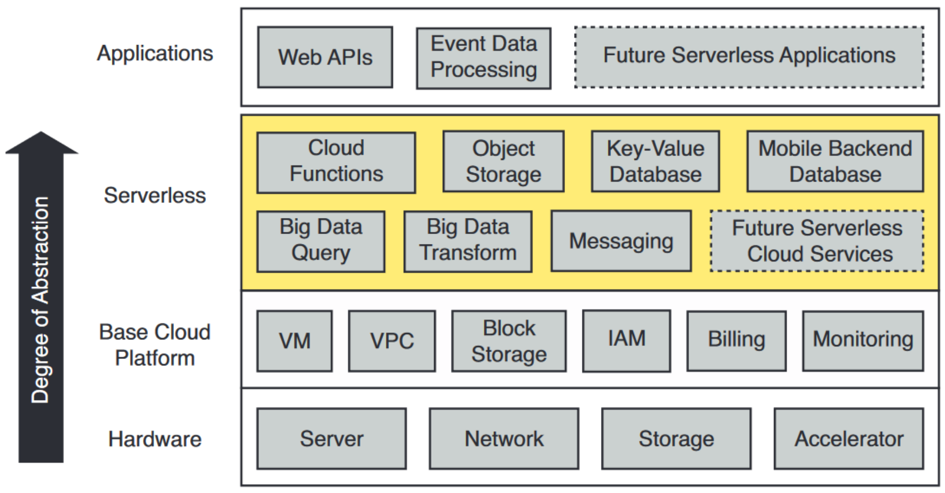


Figure 2.1: Overview of Serverless Architecture. (Source: Eric Jonas et al. [6])

With a correct microservice architecture applied, these smaller services become more scalable and maintainable over time. This decreases the cost and time to market for new features, that would otherwise become more expensive and longer to develop adopting a classic monolithic application architecture. With a code base that keeps getting larger and more complex over time, it gets increasingly difficult to add new features without breaking existing functionality or adding even more complexity [5].

2.1.3 Serverless

Serverless computing is a new distributed application architecture in which applications are composed by a collection of small logic units, functions. It aims for an even more fine-grained architecture design and management than microservices, putting the line of separation at the function level. In serverless, the programmer only needs to manage the function code that he wants to execute; all the infrastructure, even up to the web servers running the code, is managed by the provider of the platform being used, which is typically done in a public cloud. Besides, serverless platforms are built to be extremely elastic, ensuring that applications can scale up rapidly to accommodate load burts and down if no requests are being received.

The Serverless layer in Figure 2.1 shows what kind of services the Serverless platform is formed of. Serverless requires an entire ecosystem of components, with the Cloud Functions serving as the main service to deploy code that interacts with other external components such as fetching and storing files in Object Storage, reading or inserting data into a Key-Value Database, executing functions triggered by events from a Message Queue and others. This allows the developer to only having to write the required business code and interacting with other services, whose infrastructure is managed by the Cloud, using SDKs (Software Development Kit) or APIs (Application Programming Interface) that are provided by the Cloud. The downside is that all of these services are billed individually and the total operational cost can be high if the service is regularly used, as the billing of most of the services are per-usage, but it's the price to pay for convenience for the developer to have almost zero infrastructure management. In

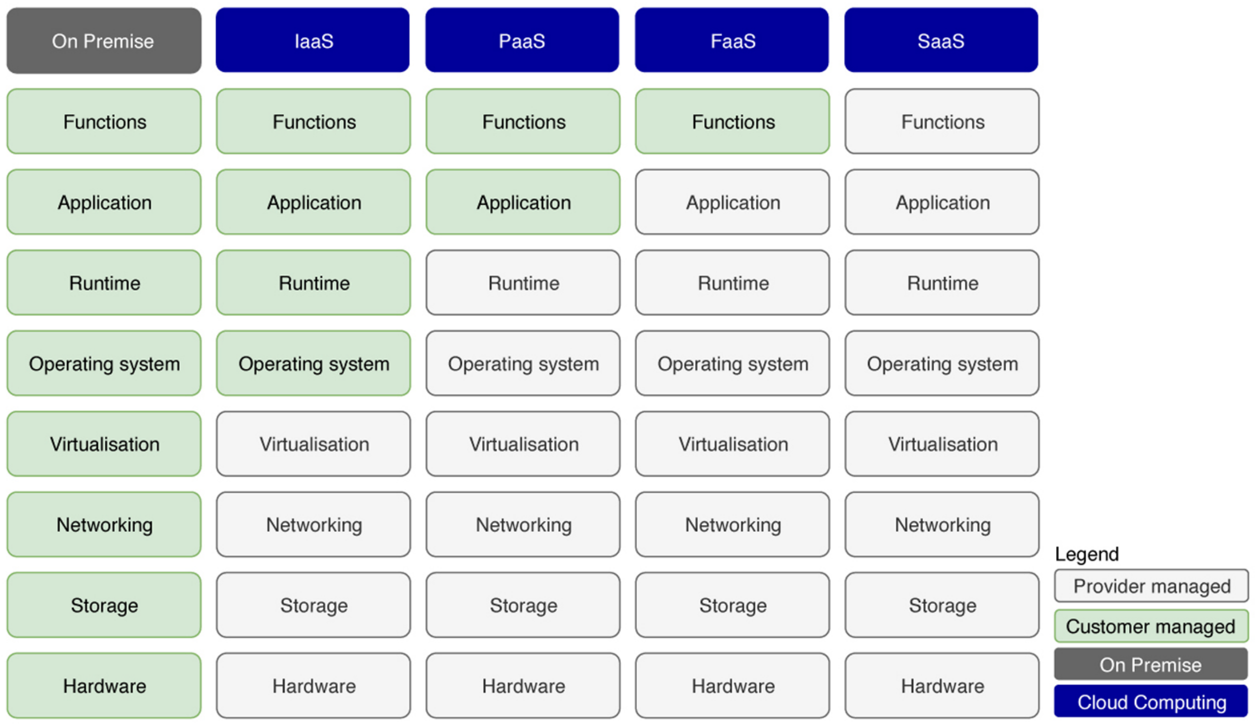


Figure 2.2: Abstraction level of various cloud service levels offered. (Source: [7])

the AWS Cloud, Lambda (Cloud Functions) interacts with other services such as S3 (Object Storage), DynamoDB (Key-Value Database), Simple Queue Service (Messaging) and others.

Looking at the lower layer, the Base Cloud Platform, we can see simplified view of lower level cloud services that serverless abstracts from the developer, such as virtual machines (VM), private networks in the cloud (VPC), block storage, access control (IAM), billing and monitoring of resources.

2.2 Cloud Service Levels

It's also important to look at Cloud Service Levels in terms of the abstractions provided to the developer. We'll explore these in this section, with summarized description of each of the most commonly used levels and an in-depth analysis of current FaaS (Function-as-a-Service) offerings.

2.2.1 From IaaS to SaaS

Clouds provide multiple levels of abstraction on their compute services. Figure 2.2 provides a comprehensive view of the relevant resources and entities in cloud offerings (e.g., hardware, storage, operating system) and the way they are managed across different service levels (i.e. on premise with own resources, and from IaaS (Infrastructure as a Service) to SaaS (Software as a Service)). As we move up in the service level list, more responsibilities are shifted from the developer to the Cloud Provider, removing the need to manually manage infrastructure such as the OS (Operating System) and the application

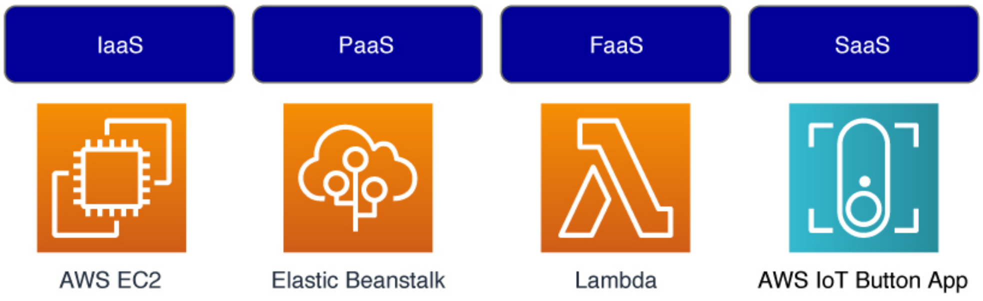


Figure 2.3: Example Apps in AWS for the various service levels. (Source: [7])

runtime.

In IaaS service level, the customer is responsible for managing which OS to deploy and all the configurations associated with it, starting from the system kernel upwards. At this service level, only the hardware is virtualized and managed by the cloud provided.

This low level management implies that some responsibilities can be allocated to the customer, such as redundancy: the customer can select how many machines he wants to deploy and manage them accordingly.

In PaaS (Platform as a Service), the customer receives an increased abstraction. There is no longer the responsibility of managing the OS directly. The customer only has to manage the application deployed using the platform provided by the cloud provider. This cloud service level, opposed to IaaS, which will supply automatic scaling and high availability automatically.

Finally, in FaaS (Function as a Service), the level of abstraction rises once again. Instead of having to manage a web server and the business logic, the customer only needs to manage the business logic.

Figure 2.3 shows examples of popular cloud service products, and the cloud service level each offer is being provided at. In AWS EC2 (IaaS platform), the user manages virtual machines, he needs to decide which OS, how much resources the machine needs for the workload and everything from configuring the OS to deploying the application is the user's responsibility. Auto-scaling solutions are available provided that the user creates a custom EC2 image that already contain means to auto-start the application.

Elastic Beanstalk (EB), AWS's main PaaS platform provides an increased abstraction over EC2, in EB the user is only responsible of submitting the application code to deploy and choosing what type of machine needed depending on the resources required. EB deals with auto-scaling, load-balacing and health monitoring. Even though the main goal of EB is to allow the programmer to focus on development first instead of cloud infrastructure, it doesn't forbid the developer to customise the environment, all the infrastructure that is created by EB, such as the compute instance which are regular EC2, are accessible and can be customized.

Lambda, AWS's main FaaS platform provides an increased abstraction over the previously described EC2 and EB platforms, abstracting the developer from typical application infrastructure, such as an HTTP server and authentication handling, allowing the developer to focus on coding the application's

Provider	Requests (per Million)	Memory-Duration	CPU-Duration
AWS Lambda	$0.20	$0.0000166667/GB-sec	N.A.
Azure Functions	$0.20	$0.000016/GB-sec	N.A.
GC Functions	$0.40	$0.0000025/GB-sec	$0.0000100/GHz-sec

Table 2.1: Metrics that affect cost billed to FaaS users [8–10]

logic only. FaaS is analyzed more in-depth in sub-section 2.2.2.

Lastly, SaaS represents the highest abstraction of the services described in this Section. In SaaS, the provider delivers a complete application, the developer is only a user of the application and is only responsible for configuring the application if possible, the provider is responsible for managing the rest of the stack. An example is the AWS IoT Button, a service that allows integration of an IoT button with other AWS services - such as AWS Lambda for function execution or other IoT devices automatically through AWS IoT Core system, simply by registering a compatible hardware IoT button with the AWS IoT Button service.

2.2.2 Existing FaaS platforms

The three big cloud providers offer FaaS, such as AWS Lambda, Azure Functions and GCF (Google Cloud Functions). Although they are all FaaS, each of them has specific restrictions such as the programming languages available to deploy and the billing is also different.

The most common metrics used to bill are the **Number of Requests**, **Memory-Duration** and **CPU-Duration**, as shown in Table 2.1. **Number of Requests** is commonly defined by a fixed price per million requests executed, **Memory-Duration** is commonly defined as 1 GB-sec being the price of running a function with 1GB of memory during 1 second and **CPU-Duration** is a metric used only by GCF that is similar to Memory-Duration but 1 GHz-sec represents a function executing in a machine with a vCPU running at 1GHz during 1 second.

AWS and Azure don't provide direct means of provisioning CPU like Google Cloud does, instead, the CPU is scaled proportionally to the amount of requested memory. A 256MB function will have 2x the CPU of a 128MB function.

Although looking to this table we might think GCF is more expensive due to the seemingly higher cost of Number of Requests and introduction of a metric not used by AWS Lambda and Azure Functions, it's not that simple since there is another factor to be detailed which is the free tier offered by each platform. This will be better detailed in the appropriate paragraphs for each platform.

When invoking functions from any of these providers, the first time we invoke the function it will incur a cold start in which a container will be created, the runtime environment will be started and only after these steps the function is actually invoked. To mitigate this problem to a certain extent, providers attempt to keep "used" containers for a set amount of time after the function is invoked to attempt to reuse containers for subsequent invocations. The time these containers are kept alive is a trade-off that providers have to make between between the memory used and the performance gains of reduced cold starts. It is highly dependant on the function invocation frequency and its more difficult to predict accurately if the function is invoked in irregular loads.

AWS Lambda Programming Languages supported natively are Java, Go, PowerShell, Node.js, C#, Python and Ruby.

Unsupported Languages can be added by creating a custom runtime [11].

Lambda's billing granularity is much finer than other providers, billing by increments of **1ms** of execution time and **1MB** of memory [12], having a greater advantage for extremely quick functions that execute in less than 100ms and also allows the customer to choose a level of memory more accurate to what the function will use with the 1MB increments. In Lambda, 1,792MB of allocated memory is equivalent to 1 vCPU, which is designed as a Thread of either an Intel Xeon or AMD EPYC core. If the programmer decides to allocate a lower or higher amount of memory, then the designated vCPU is adjusted as well.

Free tier is available, up to 1 Million requests and 400 thousand GB-sec of Memory-Duration.

Azure Functions Azure Functions billing granularity is counted in increments of **100ms** of execution time and **128MB** of memory. It gives less flexibility to choosing function memory since there isn't any advantage of choosing values that aren't multiples of 128. In terms of execution time it isn't very good for very fast functions ($<$ 100ms) but otherwise is pretty standard pricing.

It includes a free tier, up to 1 Million Requests and up to 400 thousand GB-sec of Memory-Duration.

They also include a different plan that aims to reduce completely cold starts by having warm instances always available. This is the Premium Plan and is billed differently from regular functions. The billing in this plan is, understandably, closer to IaaS or PaaS, as it is billed by hour of CPU and Memory used: **$0.185 vCPU/hour** and **$0.0132 GB/hour**.

Google Cloud Functions Includes a free tier for all the metrics used, up to 2 Million Requests, 400 thousand GB-sec of Memory-Duration and 200,000 GHz-sec of CPU-Duration per month are not billed to the user, making the calculations of total cost a bit more complicated.

2.3 Cloud Deployment Models

Cloud computing is essential to the ease of use and viability of serverless: the cloud provider manages all hardware and software required to run the provided code. The programmer is only responsible for managing the logic code and has no need to worry about managing web servers nor complex deployments [13].

In this section we will present how the various models of cloud deployments enable serverless. Starting with the most known **Public Clouds** that provide their own serverless platforms, moving to **Private Clouds** which are deployed in a private environment and require the owner to setup the software infrastructure for serverless, **Hybrid Clouds** that use both public and private clouds elements [14], and ending with more recent cloud deployments enabling edge computing (**Edge Computing Clouds**).

2.3.1 Public Cloud

Cloud Computing in Public Clouds is the easiest form to deploy serverless functions without having to incur in upfront costs in hardware, and without having to manage all the software complexity behind serverless orchestration, but it has some drawbacks such as high operational costs and low control over data security. Examples of popular Public Clouds are AWS (Amazon Web Services), Google Cloud and Microsoft Azure.

Today, many small to big companies use Public Cloud computing to host their services due to its flexibility and huge number of available services making it extremely powerful and easy to start. Applications running on AWS include Netflix, Facebook, Twitch, LinkedIn, Twitter and many more.

2.3.2 Private Cloud

Private clouds are mostly used when the need for data security and computation trust is high and Public Clouds can't be trusted with this data, but there is still a desire to have some of the ease of use of Public Clouds, such as automatic provisioning of virtual machines to deploy applications. Example applications that might run on Private Clouds are Government agencies' applications that access sensitive citizens' data, financial organizations, healthcare organizations or other organizations that are required by law to secure data. These Clouds can typically be deployed in an in-house setup of bare-metal server-grade machines or in rented hardware in a data-center.

Serverless is seldom used in Private Cloud environments, one of the big advantages of serverless is the apparently easy deployment and high abstraction of hardware and software. In Private Cloud environments, the owner of the cloud needs to manage the operational aspect, from the hardware high initial cost, to managing the whole Serverless software infrastructure, using any of the Open-Source serverless frameworks, such as Apache OpenWhisk [15], OpenFaaS [16] or Fn [17], or non-free solutions such as VMWare Cloud Director [18].

2.3.3 Hybrid Cloud

Hybrid Clouds represent the combined usage of Private Clouds, or plain in-house hardware, and Public Cloud services. Companies that already have in-house hardware, and want to use Public Cloud services for many reasons such as high peak loads, big data processing and others, are typical adopters. Other typical use for Hybrid Cloud setups is for an iterative Cloud migration of their systems, having services running in the Private Cloud and services running in the Public cloud during the migration process.

An example of a company leveraging Hybrid Cloud to improve their services is Blackline, an USA-based company that provides financial automation solutions. Blackline migrated some of their services to Google Cloud, which allowed them to scale their services to maintain their user experience while growing the number of customers [19]. It also allowed them to use the powerful services that Clouds provide.

2.3.4 Edge Computing Cloud

Traditional cloud computing infrastructure is housed in big data centers which are far from people's devices, suffering from moderate latency levels and cost of bandwidth of data transferred between the devices and the cloud.

Edge Computing promised to be a better solution for latency sensitive applications by moving the computation from the center of the internet (Cloud) into the edge of the internet, which ranges from intermediate devices such as routers, switches and others to end-user devices such as smartphones, Raspberry Pi's, and other devices with computing abilities. Cloud deployments reaching towards the Edge aim to provide low latency and lower bandwidth cost by offloading tasks closer to the source of data, while also improving resource utilization of possibly underused devices in the edge and decreasing the centrality of the internet in cloud data centers [20].

Serverless computing can be instantiated at the Edge, executing serverless functions in the same device that generates data. Two main types of Edge Computing are available, Content Delivery Network (CDN) based and Internet of Things (IoT) based.

CDN-based serverless edge platforms allows custom functions to run in the same location as the CDN, achieving lower latency than cloud since the edge network has more available locations that are likely to be closer to the end user than cloud regions are. CloudFlare Workers [21] use the JavaScript V8 Engine that provides Isolates, having the possibility to invoke the function concurrently in the same runtime, achieving little overhead of high concurrency levels when compared to traditional single container per invocation approaches. Use cases can go from providing access control, modifying request headers, fetching external resources and others.

Other popular CDN-based serverless platforms include Akamai EdgeWorkers [22], IBM Edge Functions [23] and Lambda@Edge [24] (with CloudFront [25]).

IoT-based serverless edge platforms aim to execute functions in client-side, instead of server-side. Current platforms require the user to provide their own devices to run the functions, being attractive only for companies that already have large amounts of IoT devices ready to use and want a platform to integrate with the devices. Available choices are AWS GreenGrass [26] and Azure Functions in Azure IoT [27] provided by public clouds or FogFlow [28] and Open-Whisk Light [29] that require a private setting to deploy. Since IoT devices have very limited resources, Serverless can be difficult do use, as the algorithms used in the cloud to scale workers automatically to handle request load may overload the devices. With this in mind, solutions that attempt to minimize the resource impact of Serverless and do not compromise on the data isolation and security, such as Photons or the work presented in this document, Photons@Graal become very useful on this type of Edge Computing, maximizing the usefulness of these limited IoT devices.

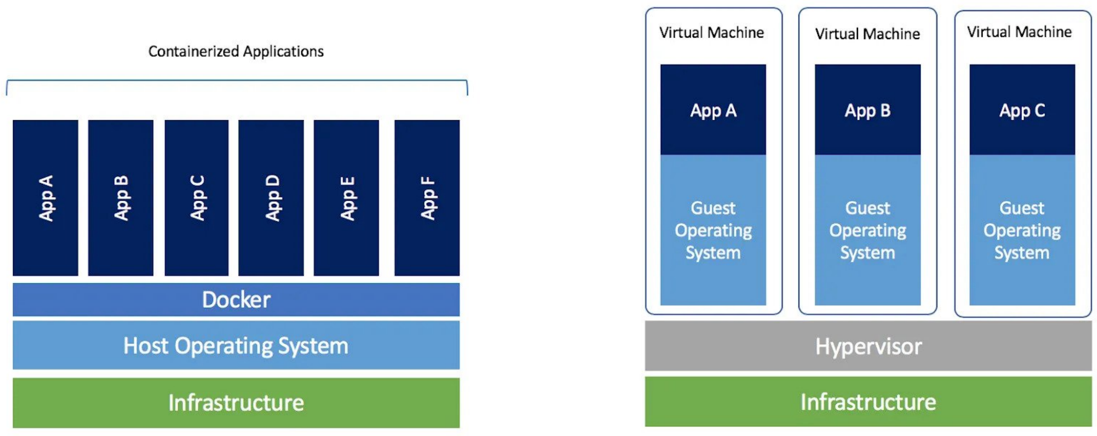


Figure 2.4: Container architecture using docker running apps in sandboxed processes under a single OS and traditional Type 1 Virtual Machines each running an OS for a single app that run on top of an Hypervisor. (Source: [30])

2.4 Managed Runtimes

Serverless functions are run in isolated environments, providing security and an easy way for the infrastructure manager to limit the function's access to resources such as CPU, Memory and disk. Traditionally, this isolated environment would be a Virtual Machine, providing strong security and isolation from other VMs. Due to the nature of serverless relying on easily bootstrapped and disposable machines, VMs with high startup latency would introduce too much overhead.

2.4.1 Lightweight Virtualization

Instead of virtualizing the whole environment up to the Operating System, a simpler solution was required. This solution should still provide isolation and the ability to limit resource usage, in order to allow cloud providers to offer easily customised products that they can bill accordingly.

The solution is using lightweight virtualization techniques, such as containers. Containers provide sandboxed processes that run with its own virtual filesystem and can have limits imposed on resources such as CPU, Memory, networking bandwidth and I/O.

LXC (Linux Containers) is an operation-system-level virtualization method that allows the creation of multiple isolated virtual environments within the same host that was first released in August 6, 2008. It leverages Linux's `cgroups` features, which allows for a fine-grained resource limitation, by limiting the CPU, memory, disk I/O, network usage to the level desired, avoiding processes from using too much resources. LXC provides an isolated system by using `chroot`, which changes the root filesystem to a specific directory, for example: `/mnt/container1/data` becomes / inside the LXC container, ensuring that the container only has access to it's own data.

Docker is an example of an open source containerization framework that allows users to build and run custom images as containers that was first released in March 20, 2013. Docker initially used LXC as its default execution environment, but eventually created it's own virtualization mechanism and removed

LXC support. The main advantage that Docker provides over LXC, is the build and distribution system. Docker can build Images which contain the filesystem and configurations of the container that can be published in Docker Hub[1], a service maintained by Docker that hosts the images and allows users to easily grab them to use. This is a big selling point of Docker, images can be created by users or by developers of applications and anyone willing to install the application can just download the image and run it, without having to worry about installing application dependencies, increasing system bloat or anything of the sort, everything is contained [31]. The main advantages (less resource overhead for each application running on top of virtualization) and limitation (mandatory usage of a given operating system in Docker, against free choice when using an hypervisor) are illustrated in Figure 2.4.

2.4.2 Language Runtime-Level Virtualization

We are looking for a finer-granularity virtualization than the one provided by Lightweight Virtualization described in the previous subsection which only guarantees that an application runs in an isolated process, it provides no guarantee for isolation of concurrent executions within that process. To guarantee that concurrent executions within a single Java runtime are properly isolated we leverage GraalVM Native Image. To give an insight into the major differences between a regular Java VM and GraalVM, we will describe the base VM architecture of both.

Java VM To execute Java code, it is first compiled into bytecode, a special instruction set that can be interpreted by a Java Virtual Machine (JVM) such as OpenJDK's HotSpot JVM. By compiling to bytecode instead of machine code, it gives Java the ability to run the same compiled bytecode on any hardware that can run the JVM, since the JVM acts as a translation layer between the hardware and the bytecode. The files containing this bytecode regularly have the format of `class_name.class` for regular classes.

When running a bytecode compiled Java program, the JVM's Class Loader loads the bytecode into the runtime data structures containing the classes and methods represented in the input bytecode. These data structures are used by the runtime components of the JVM, specifically the interpreter, Just-in-time (JIT) compiler and Garbage Collector. The interpreter is the component that interprets bytecode and executes the corresponding native hardware instructions. For highly executed code, the interpreter needs to re-interpret every time it is executed, so, the JIT compiler is used to compile the bytecode into native code that can be executed multiple times and is also faster to execute than the interpreted instructions. This means that the first few times that code is executed after a runtime is initialized it can be quite slow, depending on the complexity of the code executed, which can be a problem for Serverless functions that are seldom invoked. The Garbage Collector is a mechanism that manages memory and attempts to automatically remove unused memory from the heap without programmer intervention as is necessary in lower level languages, such as C. A simple visual representation of these components is show in Figure 2.5.

[1] https://hub.docker.com/

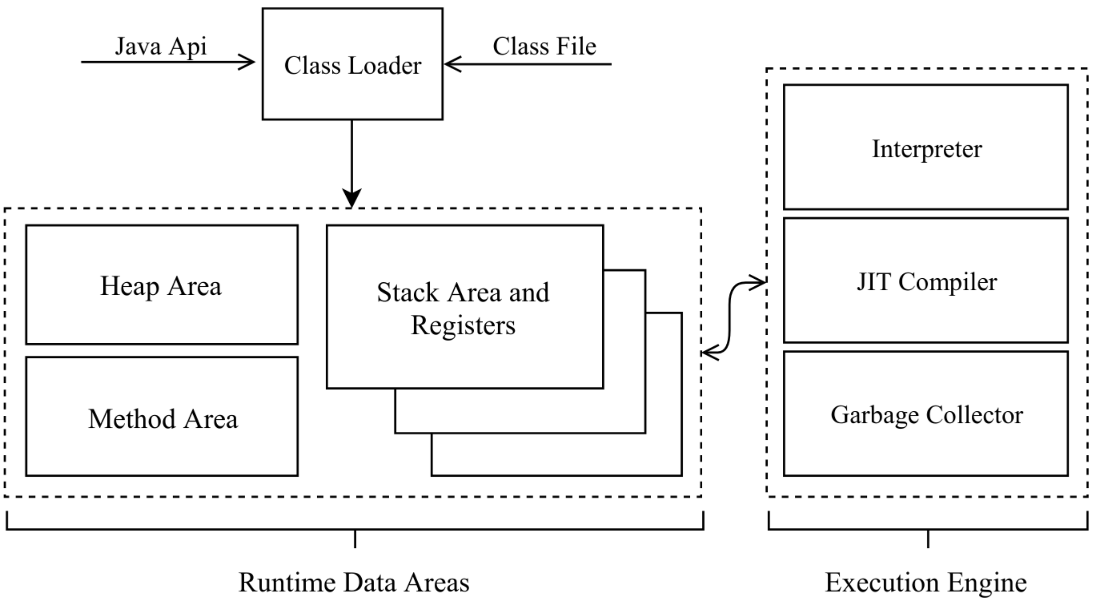


Figure 2.5: Hotspot JVM Architecture. (Source: C. Gonçalves [32])

GraalVM GraalVM is a novel Virtual Machine (VM) that allows interoperability between multiple programming languages such as Java, JavaScript, Python, C, Rust and many others in a shared runtime. It includes runtime components such as a JVM, Node.js and LLVM runtimes to allow execution of programming languages that require these runtime environments and also includes interpreters for interpreted languages such as Python, Ruby and JavaScript.

Although the polyglot features of GraalVM are relevant, it is not the focus of this work. The most important GraalVM component is the **Native Image**, a technology developed to reduce Java startup time, initializing the application at build time by loading all classes that belong to the application and to the Java Runtime Environment (JRE). To avoid loading unnecessary artefacts, only reachable classes and methods are loaded from the code being compiled. This approach assumes that everything is already known at build-time, so, no runtime modifications are expected. Unlike regular JVM artefacts, the resulting artefact of the build, the native image, is a native executable that can be run only on the operating system and hardware that built the image [33], similarly to C artefacts.

The heap that is built during native image build stage contains the pre-loaded classes of the application, it is called the ”image heap”. To have a fast start of isolated tasks (Isolates) and low memory footprint these tasks use copy-on-write sharing to have a reference to the image heap.

A feature that GraalVM Native Image provides that is crucial for the success of this work are Isolates, which provide ability to run tasks within a disjoint heap that is created on demand. Due to the strict memory isolation, Isolates can’t access any sort of shared memory, requiring programmers to gather data from external sources or copying it into the Isolates. The only memory that can be shared is the ahead-of-time compiled code. Since Isolate code executes within a separate heap, garbage collection can run only on Isolates that need collection, reducing GC overhead on the application. This is demonstrated in Figure 2.7, which shows a Native Image process, with the AOT compiled code that is available to all Isolates, and 2 Isolates that use the map the heap with pre-loaded classes, the ”image heap”, and

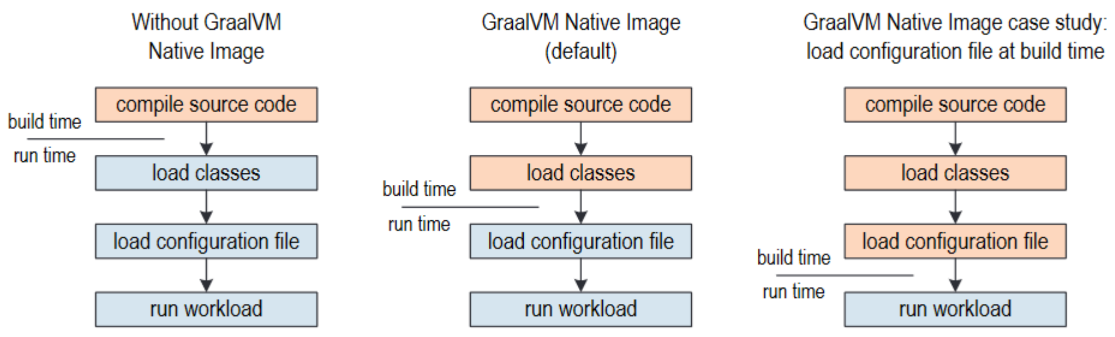

Without GraalVM Native Image
GraalVM Native Image (default)
GraalVM Native Image case study: load configuration file at build time
compile source code
load classes
load configuration file
run workload
build time
run time

Figure 2.6: Simplified application life-cycle in a classic JVM app that compiles code but only loads classes after application is running compared to GraalVM Native Image's build system that loads all classes into the executable providing much faster startups. Native Image also provides means to load immutable resources at build time, such as a configuration file. (Source: Christian Wimmer et al.[33])

reserve extra memory for their own Objects, the ”run-time heap”. It's important to note that the main Thread in Native Image runs inside an Isolate as well, using the same mechanisms.

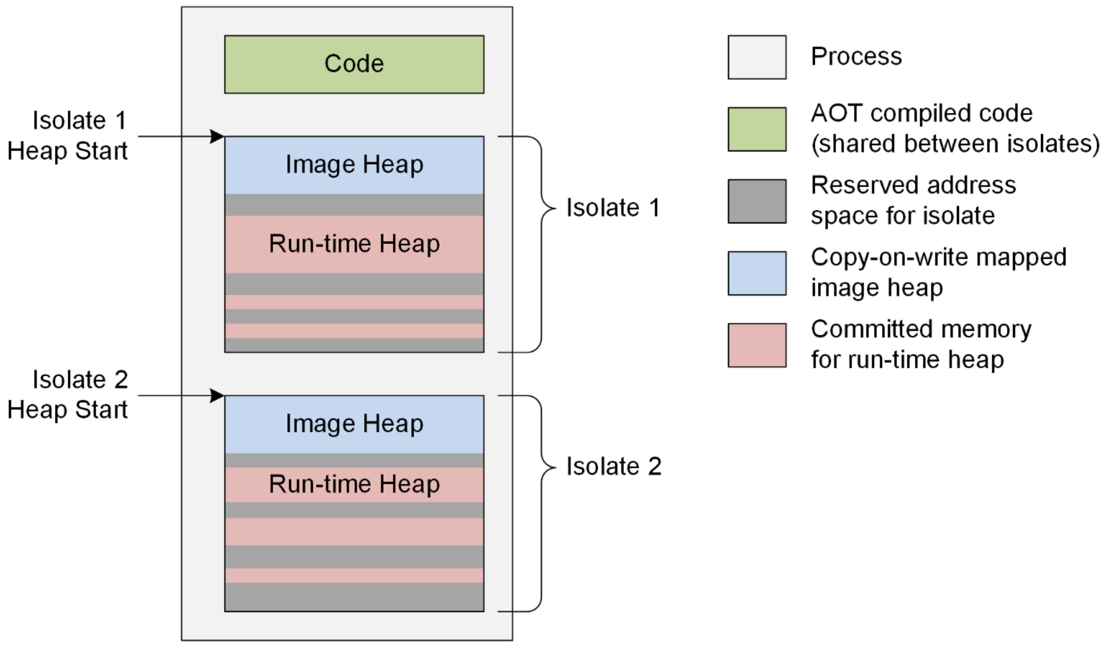

Code
Isolate 1 Heap Start
Image Heap
Run-time Heap
Isolate 1
Isolate 2 Heap Start
Image Heap
Run-time Heap
Isolate 2
Process
AOT compiled code (shared between isolates)
Reserved address space for isolate
Copy-on-write mapped image heap
Committed memory for run-time heap

Figure 2.7: GraalVM Native Image memory division. (Source: Christian Wimmer [34])

2.5 Relevant systems

Serverless is a recent hot topic and many works have been trying to optimize the current inefficiencies in commercial platforms, in this section we will talk about some of them.

2.5.1 Photons

Photons is a framework that allows concurrent serverless function executions to be co-located in a single docker container, attempting to improve several inefficiencies in today's public cloud platforms, such as the high number of cold starts due to the single concurrent invocation per container policy and high memory utilization due to every container requiring some application state, such as machine learning model for example, in Photons this model is shared by all invocations in the same runtime. Throughout the document, we might reference this system as Photons@HotSpot to help differentiate with Photons@Graal.

Data Isolation To ensure that Photons remains as secure and fault tolerant as current clouds are, the concurrent invocations in the same runtime must not access or alter other functions memory, they need to be isolated. Since baseline Java doesn't provide any means of isolation, Photons developed a proxy that intercepts user code at bytecode load time and using bytecode manipulation. Static fields, methods and initialization blocks are properties of a class, being application-wide, this breaks isolation. Every time a function is executed in a runtime, a bytecode transformation is made to change the static fields to be related to the specific invocation instead of global.

To avoid memory leaks by having reachable unused memory, the local static fields resulting of the bytecode manipulation are stored in weak tables, which is a concept that keeps a value reachable as long as there are strong references to it. When a function finished execution the references to the static fields get out of scope and the mechanism of the weak table marks them as unreachable, then, garbage collection will ensure that they are disposed and no memory leak occurs.

State Sharing Photons also provides a memory sharing mechanism resembling a global cache, a shared object store that is a simple key-value map that all photons can read and write to. It's useful to share data that is used by all invocations, such as machine learning models, reducing overall memory usage.

2.5.2 SEUSS

Serverless Execution via Unikernel SnapShots (SEUSS [35]) attempted to reduce high function initialization cost present in the cold start problem of containerization-based serverless by using unikernel [36] snapshots that have already initialized environments, providing a much faster startup time. In unikernels, the application and other system components (file system, networking) are packaged in a single address space. SEUSS uses UC's (Unikernel Contexts), which are custom Unikernels that contain a language Runtime, such as Node.js or Python that is configured to import and run function code. Figure 2.8 shows the overall architecture and operations applied by SEUSS OS. On boot of the OS, SEUSS populates a cache of UC's for the desired application runtimes (**B** in Figure 2.8) that are then available to be copied into worker nodes when a function invocation arrives in the system. When a function invocation arrives for the first time, a runtime snapshot is fetched and the function is initialized (**C** in Figure 2.8) and the function UC is then cached (**S** in Figure 2.8). Subsequent invocations for the same function but without any active worker nodes are then initialized from the cached UC (**W** in 2.8). When an invocation arrives and there is a live worker node for such function, it's directly used (**H** in 2.8).

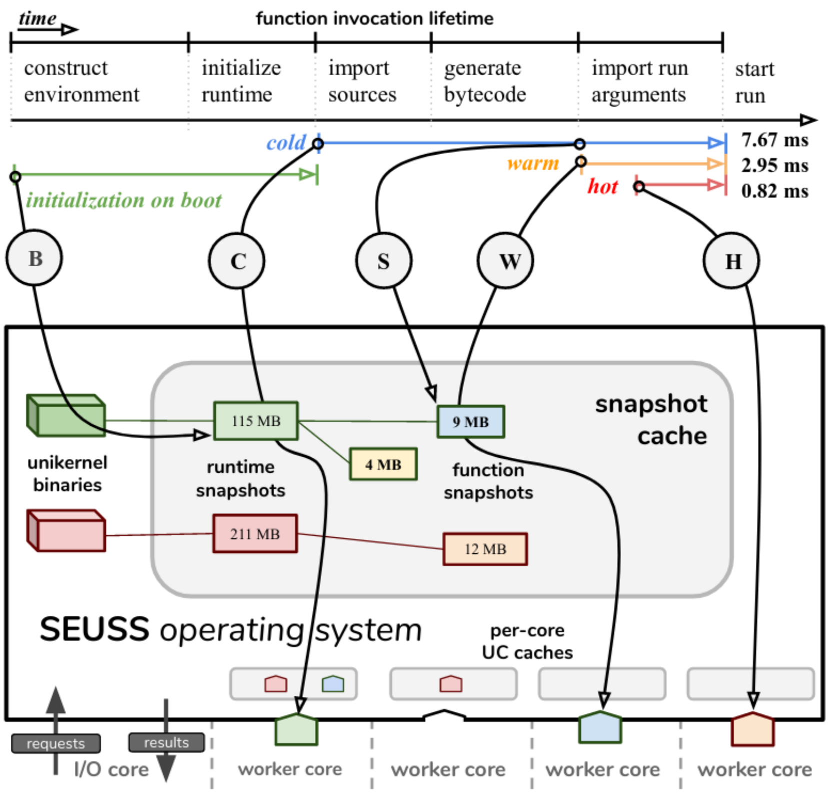


Figure 2.8: SEUSS high-level architecture and operations. (Source: James Cadden et al. [35])

This architecture allows for extremely fast startup times of functions, between 3-8ms without using too much memory in caches when comparing with container provisioning which normally take from 500ms-3s.

2.5.3 SOCK

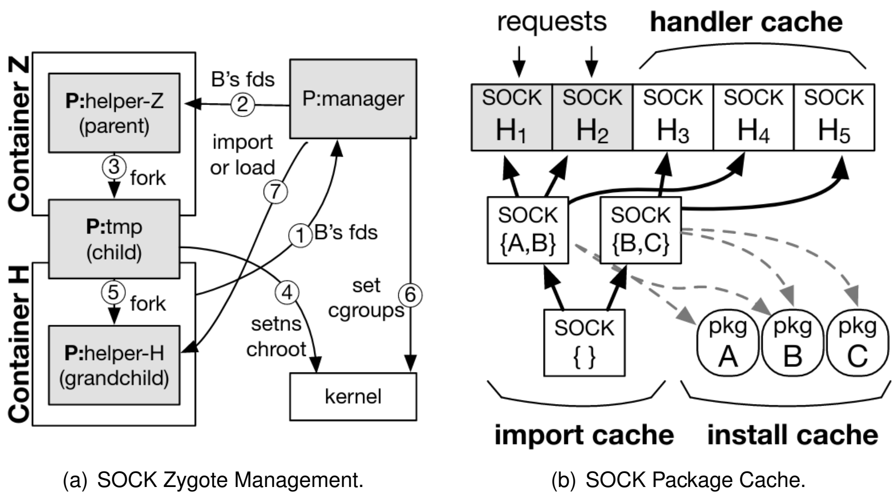


Figure 2.9: Overview of SOCK's Zygote and Cache architecture. (Source: Edward Oakes et al. [37])

SOCK [37] also aims to reduce cold start initialization cost by using Zygote provisioning, in which new processes are forked from the main process, the Zygote, that already has imported libraries that are needed by the application and done some initialization work, reducing the initialization work needed to be done by the child processes. This means that the system must maintain a set of Zygote processes with the different sets of preinstalled packages, which could prove difficult in an environment with many different types of applications being executed. SOCK is based around the Python ecosystem.

Figure 2.9 shows the strategies used by SOCK for the process forking using Zygotes and package caching. In sub-figure (a), the flow of spawning a warm runtime (Container H) using a Zygote process (Container Z) is described. In sub-figure (a), the process of maintaining Zygotes with preinstalled packages is described in the **import cache** section and the **install cache** represents a large set of packages pre-installed on disk, for a fast bootstrap of processes using commonly-used packages.

2.5.4 SAND

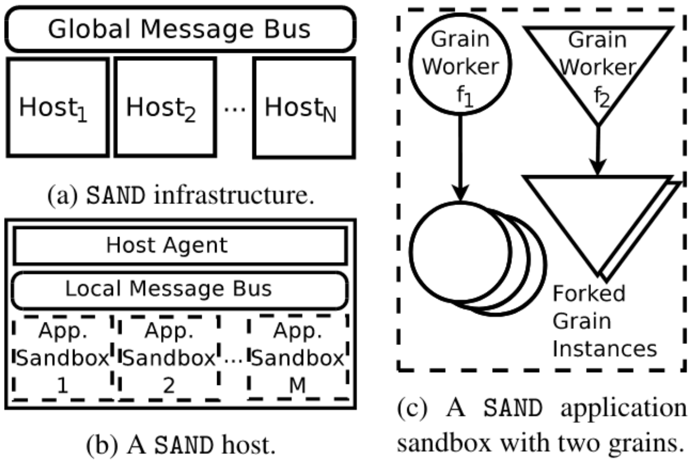


Figure 2.10: SAND high-level architecture. (Source: Istemi Akkus et al. [38])

SAND [38] aims to reduce startup latency and improve latency in function chaining workloads, in which certain functions are called after other type of functions finishes computing. In 2.10 the high-level architecture of SAND is shown. It provides strict application sandboxing by providing isolated containers for invocations of different applications and allows a less strict separation between invocations of different functions of the same application, encouraging some data sharing inside the container to reduce redundant memory usage, as it can be seen in sub-figures (a) and (b) of Figure 2.10. To reduce function chaining latency, a two-level message queuing approach was taken, a global message bus is implemented that allows all function executions of any application to produce and consume messages (sub-figure (a) of Figure 2.10) and a local message bus available only for applications in the same host (sub-figure (b) of Figure 2.10).

The result is that SAND achieves fast startup times, reducing the total execution time in function chaining workloads. But, the architecture proposed doesn't limit the sandboxed application and they

System	Problem(s) Tackled	Runtimes Supported	Concurrent Invocations	Isolation Mechanism
Photons [1]	Runtime duplication	Java HotSpot	Yes	Bytecode Manipulation
SEUSS [35]	Cold start latency	Any, but requires a custom OS	No	Unikernel
SOCK [37]	Cold start and package installation latencies	Python	No	Container
SAND [38]	Cold start and Function chaining latencies	Languages that support forking, such as C, Python	No	Container
Photons@Graal	Runtime Duplication and cold start latency	Java GraalVM Native Image	Yes	Native Image Isolates

Table 2.2: Comparison of presented relevant systems.

might compete for resources, diminishing performance. SAND also relies on process forking to achieve fast startup times, being incompatible with runtimes that don't have native forking such as Java and Node.js.

2.6 Discussion

In this chapter we provided a background on related topics of the proposed project, defining the multiple types of Clouds, tracing the evolution of Cloud Service Levels - with greater detail on FaaS due to its relevance, explaining the advantages of lightweight virtualization using containers and detailing the components of a HotSpot Java VM and the GraalVM. We then explored some relevant systems that tackled some of the inefficiencies of today's Cloud FaaS platforms, some attempting to reduce the latency of cold starts and some attempting to reduce the number of active runtimes by providing data isolation of concurrent request execution.

The relevant systems presented are compared in Table 2.2 along with our solution, Photons@Graal. Only Photons attempts to solve the problem of Runtime duplication on highly concurrent loads which is the problem we wish to tackle with the proposed system, the other presented systems attempt to focus more on the high latencies of starting new runtimes, the cold start, which in Clouds involve the provisioning of containers which is slow, ranging from 500ms-3s depending on system load plus the time that it takes to initialize the function, which varies depending on the language runtime used.

While Photons allowed Java Runtimes to process concurrent function invocations within a single language runtime, avoiding spawning duplicate instances of these runtimes, the data isolation provided is based on bytecode manipulation which can be problematic - it can be buggy, since the bytecode modifications are done during the runtime phase and can introduce bugs that are hard to debug and fix that are only detected in runtime. Photons also allows the concurrent invocations to share the same Heap, which allows some memory sharing, but does mean that the data isolation mechanism is entirely dependent on the correctness of the bytecode manipulation code. With our solution, we aim to improve the data isolation of Photons by using strict data separation, allocating an Isolate to each function invocation, having an entirely disjoint heap for each invocation which allows for a much stronger isolation. We also aim to improve the cold start latency of runtimes leveraging the AoT builds that provide extremely fast function initialization.

Chapter 3

Architecture

In this section we describe our proposal for Photons using GraalVM Native Image. In Section 3.1 we describe the architecture of a single photon. In Section 3.2 we describe the proxy developed and how it interacts with the GraalVM Isolate API. In Section 3.3 we describe the algorithm that was created to manage the Isolates. Finally, in Section 3.4 we describe how Photons can be deployed using any open source function orchestrator, such as Apache OpenWhisk.

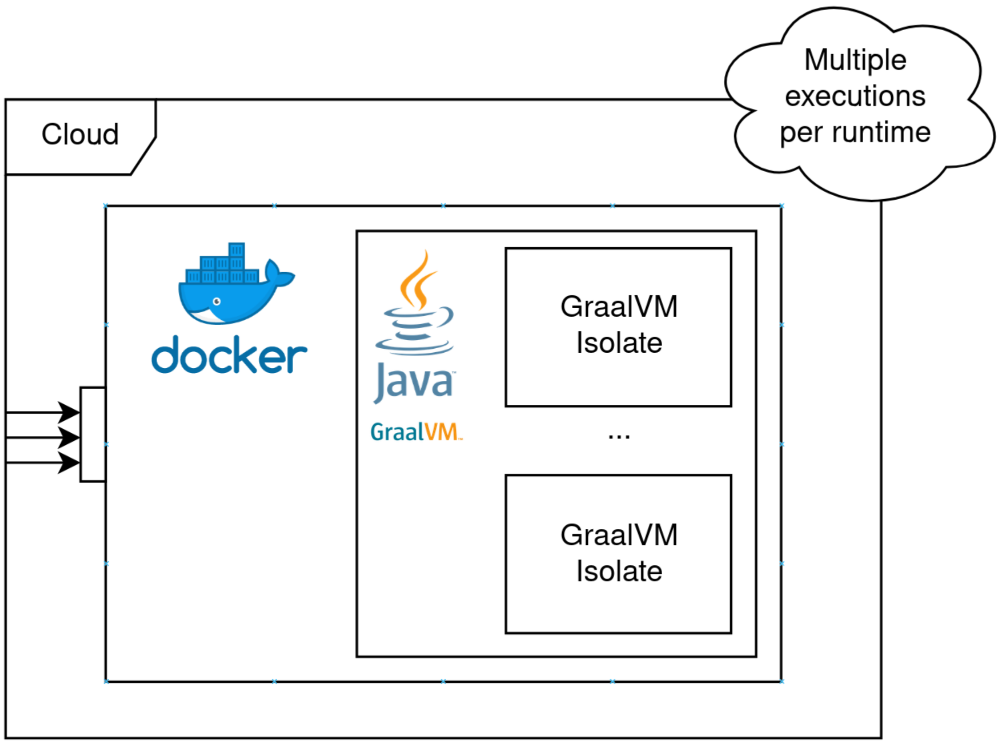
3.1 Photon Architecture

Just like regular functions on the Public Cloud, a Photon@Graal runs in a container, ran by a container engine such as Docker. The striking difference is that Public Cloud functions don't allow concurrency in each function, so, for every concurrent function invocation a new container has to be created. In Figure 3.1 we show a simple view of both a regular cloud running Java functions, and a cloud running a Photon@Graal function. To support concurrent invocations, the cloud must provision extra containers, as many as the concurrency of the requests, but in the Photon@Graal function this is not necessary as the framework will simply allocate more isolates for these requests in a single container.

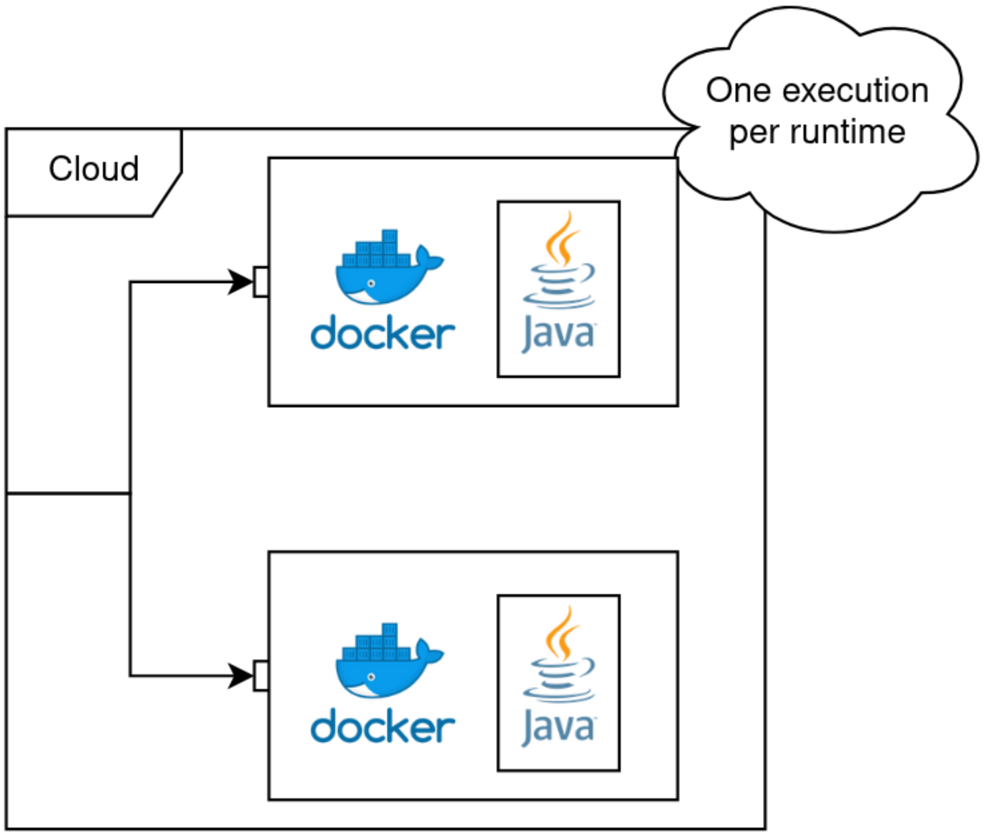
This architecture allows that a single Photon@Graal execute functions concurrently in the same Java runtime, aiming at reducing the excessive cold starts due to the high amount of containers required to initialize for highly concurrent workloads. Each concurrent function invocation will have its execution associated with an Isolate which ensures a separate heap from the other Isolates and greatly reduces memory usage when compared with the common scenario of scaling horizontally to process the incoming load by provisioning more containers with fresh runtimes.

With this architecture, we aim to achieve a better solution at providing data isolation and faster startup than the original implementation of Photons[1], which had data isolation but still allowed the functions execution to share the same heap.

An inherent benefit from using GraalVM's Native Image tool is that we use Ahead-of-Time compilation to pre-initialize our application at build time. This makes it possible to have extremely fast startup times, making cold starts faster than regular Java function cold starts. Native Image also has a much smaller



(a) Photons-Graal Enabled Cloud



(b) Regular Cloud

Figure 3.1: Overview of a Cloud using a Photon@Graal runtime and a Cloud using traditional Java runtimes.

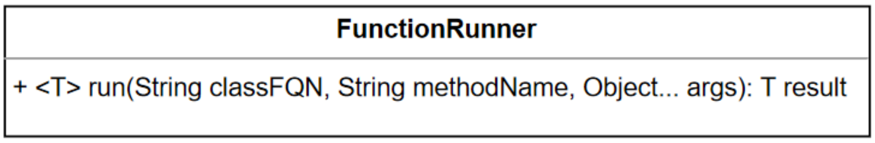

FunctionRunner
+ <T> run(String classFQN, String methodName, Object... args): T result

Figure 3.2: UML class diagram of the user-facing interface FunctionRunner provided by Photons@Graal with the method to run a given method by using reflection. Photons@Graal will handle all the orchestration of Isolates and interaction with Native Image API.

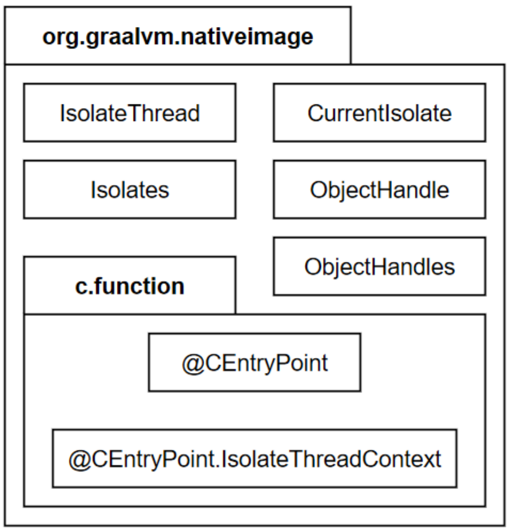


Figure 3.3: Simplified Native Image API Package View.

baseline memory footprint than a regular Hotspot JVM based application.

3.2 Function Runner Library

Photons@Graal is delivered as a library that integrates with the Native Image APIs. It manages Isolate lifecycle automatically without the users of the library needing to manually intervene. It also manages all the data transfers from and to Isolates, which need to be translated to and from C types.

This library will be based on reflection, to allow any regular function to be executed seamlessly in serverless fashion. The user only needs to provide the Class Fully-Qualified Name (FQN), the method name and the values to use as parameters for the function to be invoked. In Figure 3.2 we show a simple diagram of the interface provided.

To make use of GraalVM Isolates we need to use the provided API to:

- Create and destroy Isolates;
- Create handles for data being copied into or out of an Isolate;
- Mark methods that are Isolate entry-points with appropriate annotation.

Figure 3.3 represents a simplified view on the most important interfaces provided by GraalVM Native Image API that we will be using. Our proxy will abstract the usage of this API from users. Creating and

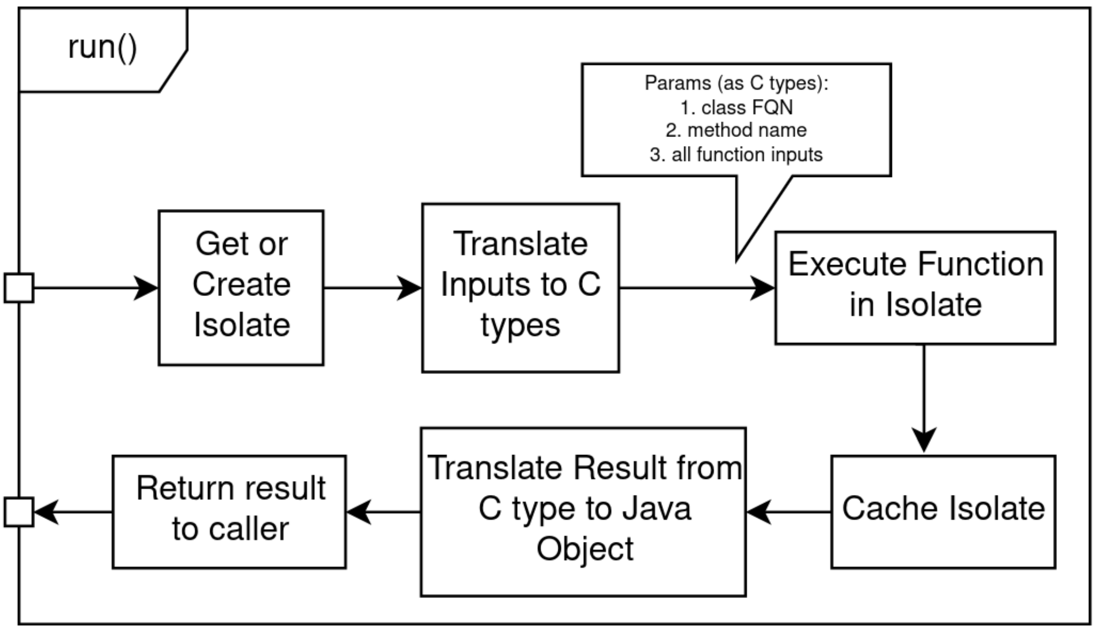


Figure 3.4: Diagram that represents FunctionRunner.run, the main entry-point of Photons@Graal. It handles the translation of the function inputs into Native Image's ObjectHandle to allow copying into the Isolate, handles the Isolate creation and caching, and invokes the function within the newly created Isolate. Then, it returns the result to the caller.

destroying Isolates via the class Isolates, managing threads with `IsolateThread` and `CurrentIsolate` and managing data transferred in and out of Isolates with `ObjectHandles` and `ObjectHandle`. The method that will act as a generic function executor inside Isolates will have the annotation `@CEntryPoint`, necessary by Native Image to know that it is an VM entry-point.

In Figure 3.4 we show a general activity diagram of a function invocation in Photons@Graal. It first attempts to fetch a valid Isolate, either reusing a cached one, or creating a new one if there are no available cached Isolates. Then, it translates the function JSON (JavaScript Object Notation) payload, the class FQN and method name into C Types, as it is necessary to pass data to the Isolate. After the function execution finishes, the used Isolate is returned to the cache, the outputs are translated and the result is returned to the caller.

A simplified application using the logic described is shown in Listing 3.1. First, the function inputs need to be translated into Native Image's ObjectHandle. This allows copying data into the Isolate, but requires a conversion of Objects into C Types due to the low-level link between Isolates, for this, we take advantage of some abstractions that are part of Photons@Graal tools that do this conversion. Then, an Isolate is provisioned for the code execution. In this example we use the Isolate Pooling strategy, but there are others available. The abstraction used provides the necessary `IsolateThread` instance, which represents a Thread associated with an Isolate. The function is then invoked within the provisioned Isolate. To run code inside Isolates, we need to create a `static` method annotated with the `@CEntryPoint` annotation. This annotation marks the method as a VM entrypoint, allowing code to be executed in a different Isolate that the current one. To specify which Isolate we want the code to run on, we use the `@IsolateThreadContext` annotation. When inside the new Isolate, we must unwrap the

`ObjectHandle` instances which is quite simple, execute the wanted code and then convert the output into an `ObjectHandle` using the same abstraction used before to convert it to a C Type. Imports of Native Image classes have been excluded from the sample for brevity. In order for this application to be executable in non-Native Image mode, we check first if we are in Native Image mode before attempting to use Isolate code.

Listing 3.1 Example simple application leveraging Isolates to perform a String reversion

```java
import pt.ist.photon_graal.runner.utils.conversion.registry.TypeConversionRegistry;
import pt.ist.photon_graal.runner.utils.management.IsolateStrategy;
import pt.ist.photon_graal.runner.utils.management.SimpleIsolatePool;

public class IsolateCodeExecutionExample {

  private static final IsolateStrategy isolatePool = new SimpleIsolatePool(4);

  public String run(final String input) {
    final String output;

    if (ImageInfo.inImageCode()) {
      final IsolateThread childIsolate = isolatePool.get();
      final ObjectHandle inputHandle = TypeConversionRegistry.getInstance()
                                                           .createHandle(childIsolate, input);
      output = unwrapHandle(reverseInIsolate(childIsolate, CurrentIsolate.getCurrentThread(), inputHandle));
      isolatePool.release(childIsolate);
    } else {
      output = reverse(input);
    }

    return output;
  }

  @CEntryPoint
  private static ObjectHandle reverseInIsolate(@IsolateThreadContext IsolateThread executingIsolate,
                    IsolateThread parentIsolate,
                    ObjectHandle str) {
    String input = unwrapHandle(str);
    String output = reverse(input);

    return TypeConversionRegistry.getInstance().createHandle(parentIsolate, output);
  }

  private static <T> T unwrapHandle(ObjectHandle handle) {
    T result = ObjectHandles.getGlobal().get(handle);
    ObjectHandles.getGlobal().destroy(handle);
    return result;
  }

  private static String reverse(final String input) {
    return (new StringBuilder(input)).reverse().toString();
  }
}
```

3.3 Isolate Management

To have Isolates available exclusively for each function invocation, we need to have an algorithm to manage these Isolates. We decided on going for a caching algorithm, that doesn't attempt to delete isolates for every invocation. There are a few reasons for this, first of all, we found out by doing some performance tests, that constantly deleting isolates by keeping them only for one invocation is expensive and can slow down the application if it is under considerable load and we also experienced some problems with functions that use HTTP Clients to invoke REST APIs, the isolate deletion mechanism has some issues. It may become stuck if there are some threads that ignore interrupts, as the mechanism waits for all the threads executing inside the isolate to stop. We experienced this problem while using `OkHTTP` Client, as this client creates some daemon threads that ignore interrupts, so, the isolate deletion mechanism would wait 60s for these daemon threads to finish, which is the time that these threads auto shutdown if there is no work to process. Another problem that we experienced, also with multiple HTTP Clients, is after each isolate deletion, a stream handle would not be closed - this does not cause immediate problems, but if the application stays alive for a long time and processes a considerable number of requests, it will crash due to too many files open, as Linux has a limit for the number of files open.

The data structure used to implement this cache should be a Thread-safe Java `java.util.Map` implementation, such as `java.util.concurrent.ConcurrentLinkedQueue`. Since there's a possibility that multiple threads attempt to fetch a valid isolate, we needed to use a Thread-safe implementation here. In order to guarantee that the system doesn't attempt to cache a absurdly large number of Isolates, the Function Runner library requires that a maximum number of Isolates is provided through configuration file. We keep the Isolate identifier in the cache, which is used to either attach the Isolate to a Thread that wants to use it, or to destroy it.

The caching algorithm described in Algorithm 1 allows us to not lose performance in scenarios with a high number of invocations and to have a stable environment, although with a drawback which is increased memory usage. The Algorithm has two procedures, one to fetch a valid Isolate to execute a function (Get Isolate), which is called on receiving a function invocation, and one to release an Isolate from the current Thread after finishing the invocation (Release Isolate). The **Get Isolate** procedure works by attempting to fetch a valid Isolate to attach to from the cache, if no result is obtained, then a new Isolate is created, else we attach the current Thread to the returned Isolate by it's id. The **Release Isolate** procedure works by first detaching the Isolate from the current Thread, then checking if the cache has space for the Isolate, if it has then the Isolate id is stored in the cache, if not, it is asynchronously destroyed to not affect the latency of the function invocation.

3.4 Function Orchestration

To use Photons@Graal we either need a photon-enabled cloud provider, which doesn't exist yet, or use any open source serverless platform to orchestrate function invocation. This can be done using one of the open source Serverless frameworks available such as OpenWhisk, OpenFaaS or Fn.

Algorithm 1 Isolate Management Strategy - Caching Isolates

```
1: constants
2:     MAX_ISOLATES, Compile-time constant
3: end constants
4: $isolate\_id\_cache \leftarrow []$
5: procedure GET ISOLATE
6:     $isolate\_id \leftarrow isolate\_id\_cache.poll()$
7:     $isolate \leftarrow$ NULL
8:     if $isolate\_id$ IS NULL then
9:         $isolate \leftarrow newIsolate()$                                ▷ Provided by Native Image API
10:    else
11:        $isolate \leftarrow attachCurrentThreadToIsolate(isolate\_id)$   ▷ Provided by Native Image API
12:    end if
        return $isolate$
13: end procedure
14: procedure RELEASE ISOLATE(isolate)
15:     $detachIsolateFromThread(isolate)$
16:     if $isolate\_id\_cache.size() <$ MAX_ISOLATES then
17:         $isolate\_id\_cache.add(isolate.getId())$
18:     else
19:         $isolate.destroyAsync()$
20:     end if
21: end procedure
```

Independently of what serverless platform is chosen to integrate Photons@Graal, some work will be necessary to implement a compatible GraalVM Native Image unit to run the serverless functions. In OpenWhisk for example, there are Docker Images with a Java 8 runtime, but nothing exists yet for GraalVM Native Image.

In Figure 3.5 we describe a Photon@Graal deployment using OpenWhisk. It can handle the the scaling of Photons@Graal based on the continuous assessment of the CPU load. By virtue of employing a custom built Native Image Runtime, each Runtime is able to concurrently execute multiple functions.

The custom runtime is very similar to the base Java runtime. It contains a simple HTTP Server than provides the `/init` and `/run` endpoints, which are part of the OpenWhisk Action interface [39]. This HTTP Server processes incoming requests using an unbounded Thread Pool that scales according to the number of requests. For each request, it extracts the function parameters from the request JSON payload and deserializes it into Java Objects, it then dispatches the work to the Function Runner library described in Section 3.2 which will fetch or create a valid isolate, execute the function with the provided arguments and return the result to the client.

But, having a custom runtime that can process multiple concurrent requests is not enough. OpenWhisk by default does not allow a single runtime to process concurrent requests, it needs to be configured to do so.

The following Ansible configurations were applied in our OpenWhisk deployment:

1. `container_pool_akka_client: true`
   (a) enables the akka http client at invoker config.
   (b) required to allow concurrent requests to a single runtime

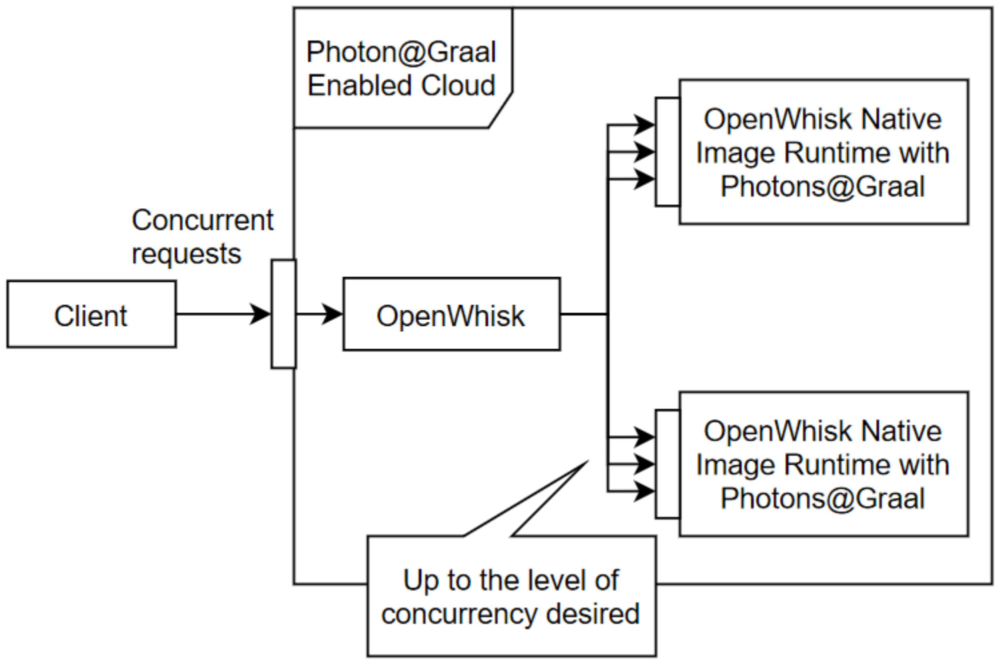

Photon@Graal Enabled Cloud
Concurrent requests
Client
OpenWhisk
OpenWhisk Native Image Runtime with Photons@Graal
OpenWhisk Native Image Runtime with Photons@Graal
Up to the level of concurrency desired

Figure 3.5: Example of a Photon@Graal Enabled Cloud. Using OpenWhisk as the serverless platform, this example works in a Private Cloud setting. OpenWhisk handles the scaling of Photons@Graal depending on the current load and using a custom built Native Image Runtime for OpenWhisk, each Runtime can execute multiple functions concurrently.

2. `userLogs_spi: "org.apache.openwhisk.core.containerpool.logging.LogDriverLogStoreProvider"`
   (a) disables processing logs at invokers.
   (b) required to allow concurrent requests to a single runtime
3. `controller_blackbox_fraction: 1.0`
   (a) configures 100% of invoker instances to accept blackbox functions.
   (b) there are two types of functions at openwhisk - normal runtimes, such as Java, .NET, NodeJS and black box functions which can be custom made to run with anything in docker. Since our custom runtimes are built onto custom Docker images, they all fall in the black box type. To avoid wasting resources, we allow all invokers to process black box functions.
4. `invoker_allow_multiple_instances: true`
   (a) allow multiple invokers in the same machine.
   (b) this allows us to deploy multiple invokers in the same machine, dividing the load between them.

By applying the configurations described above, an OpenWhisk deployment can now dispatch concurrent requests to a single runtime, the developer just needs to configure for the function the maximum concurrency that he wants to allow.

Chapter 4

Implementation

In this chapter we discuss the details of integrating with GraalVM Native Image APIs, in Section 4.1 and integrating our solution into a compliant OpenWhisk Runtime, in section 4.2. We also discuss some of the tools used and challenges faced to collect metrics from inside the application in Section 4.3.

4.1 GraalVM Native Image Integration

One of the challenges of this work was to integrate with GraalVM Native Image APIs, to manage Isolates and use them to execute code with isolated memory. Since Native Image APIs are low-level, we can't send regular Java Objects into isolates, all Objects being sent between isolates, inputs or outputs must be converted to basic C types. To simplify these operations, we built a module with some common types converters, with a simple interface that receives the Isolate and the Object, and returns the converted handle. It fetches the appropriate converter by looking at the class of the received object and selecting the most applicable Converter and converts the received Object.

The UML diagram for the Converter module can be seen in Figure 4.1. The interface `TypeConverter` is the base interface for a converter, it receives the Object to be converted along with the recipient `IsolateThread`, the class representing an Isolate currently attached to a Thread and returns an `ObjectHandle`, the class that represents a handle of an Object that was passed through to an Isolate. We implemented some converters, for classes such as `String`, `Boolean`, `Object` and `ByteArray` but more can be created, although it shouldn't be needed since every Object should be convertible through the `ObjectConverter`. The `TypeConverterRegistry` is a singleton class that maintains a registry of associations of classes and converters. It's initialized with the default converters created but more can be added. The class of this module of highest abstraction is the `ObjectHandleCreator`, which receives the IsolateThread and the Object to convert and returns the ObjectHandle. This class uses the `TypeConverterRegistry` to fetch a converter for the received class and uses it to convert the Object.

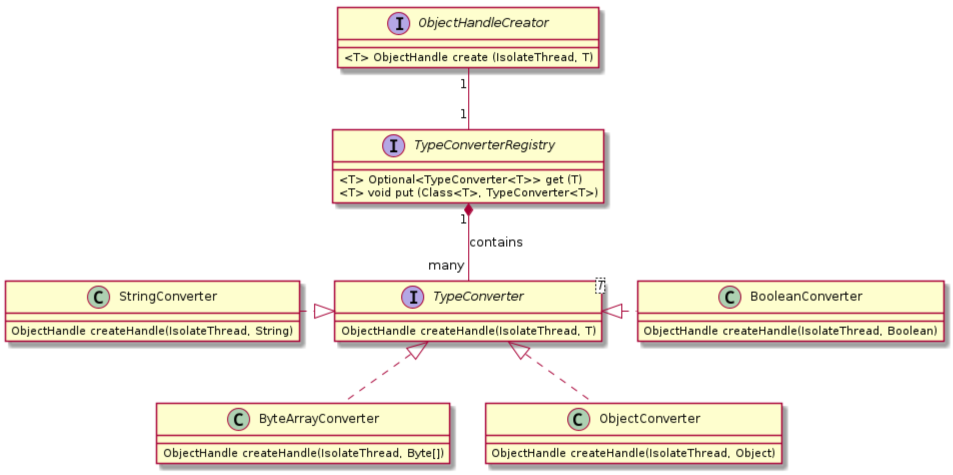

ObjectHandleCreator
<T> ObjectHandle create (IsolateThread, T)
1
1
TypeConverterRegistry
<T> Optional<TypeConverter<T>> get (T)
<T> void put (Class<T>, TypeConverter<T>)
1
contains
many
StringConverter
ObjectHandle createHandle(IsolateThread, String)
TypeConverter
T
ObjectHandle createHandle(IsolateThread, T)
BooleanConverter
ObjectHandle createHandle(IsolateThread, Boolean)
ByteArrayConverter
ObjectHandle createHandle(IsolateThread, Byte[])
ObjectConverter
ObjectHandle createHandle(IsolateThread, Object)

Figure 4.1: UML Diagram of the Converter classes and Interfaces.

4.2 Apache OpenWhisk Integration

After implementing the module that manages the Isolates and provides an interface to run functions in its own Isolate, we needed to integrate this with the OpenWhisk runtime standard.

4.2.1 Custom Runtime

The custom runtime is based on OpenWhisk Java 8 runtime available at `https://github.com/apache/openwhisk-runtime-java`. We upgraded it to Java 11 and replaced the code running mechanism with our FunctionRunner module described in Figure 3.2 and Section 3.2. The code is available at `https://github.com/davidfrickert/openwhisk-faas-graalvm-base`.

Ahead of Time Building

The standard OpenWhisk Java functions have a common, generic OpenWhisk-specific code and a separate JAR with the function Code. When starting a new container for a function, OpenWhisk initializes it by sending the JAR to the initialization API - that is part of the OpenWhisk Java Runtime. This makes the startup quite slow since before the function can start processing requests, it needs to do this initialization step, but it does allow for a clear separation between the platform-specific code and the function code.

Since GraalVM Native Image requires an ahead-of-time build, we can't initialize functions during the runtime, so, we opted for a different method. To create a Photons@Graal enabled function that can be deployed in OpenWhisk, we need to generate the native image executable by building the function along with the custom OpenWhisk Runtime that we developed.

We used GraalVM Java Agent to generate the required configuration files shown in the pom.xml snippet, most importantly the configuration for reflection. This Java Agent records all the actions on the Java VM that perform lookup of classes, methods, fields, use Java Native Interface (JNI) calls or request proxy accesses [40]. The Agent then generates files that need to be used during build time of the Native

Listing 4.1: Snippet of Maven pom.xml to build Native Image.

```
<plugin>
    <groupId>org.graalvm.nativeimage</groupId>
    <artifactId>native-image-maven-plugin</artifactId>
    <version>21.1.0</version>
    <executions>
        <execution>
            <goals>
                <goal>native-image</goal>
            </goals>
            <phase>package</phase>
        </execution>
    </executions>
    <configuration>
        <skip>false</skip>
        <imageName>app</imageName>
        <mainClass>pt.ulisboa.ist.Main</mainClass>
        <buildArgs>
            --no-fallback
            --enable-https
            --enable-http
            -H:ReflectionConfigurationFiles=conf/reflection.json
            -H:SerializationConfigurationFiles=conf/serialization.json
            -H:ResourceConfigurationFiles=conf/resource.json
            -H:JNIConfigurationFiles=conf/jni.json
        </buildArgs>
    </configuration>
</plugin>
```

Image, most importantly for Reflection, Serialization, JNI and Resources. These configurations are very important to be properly configured as without them, the application may fail at runtime unexpectedly. Since our framework depends on Reflection to execute the functions and on Serialization to transfer data to and from Isolates, these configurations are necessary to be properly configured for the stability of the application. A snippet of a Maven configuration that passes these configuration files into a Native Image build is shown in Listing 4.1.

By running the application in normal JVM mode with the Java Agent and invoking functions, the agent can analyze which classes are needed for reflection and used for JNI or proxy accesses. Since our application has code that only runs in Native Image mode - such as the code for Isolate Management, we had to implement some logic to make the isolate managing code not run while in normal JVM mode to run this agent. Due to the impossibility of running the Isolate Management code with the agent, we had to manually edit the reflection configuration file with classes that the library depends on.

4.3 Metrics Collection

Since we chose Prometheus as the metrics server, which scrapes applications for metrics instead of the applications pushing the metrics to the server, we had to find an alternative method to send the metrics to Prometheus. Since our functions may be ephemeral and not last long, and the number of functions available and their endpoints are not deterministic, it was unfeasible to have Prometheus scrape the functions for the metrics. To counteract this problem, Prometheus has the PushGateway[1] system which enables applications to push metrics to this intermediate system that Prometheus scrapes the data from.

To collect metrics inside the functions, we integrated Micrometer[2] into our custom OpenWhisk Runtime. With Micrometer, we can measure custom metrics with vendor-neutral code, having Prometheus-specific code only in the code that sends the metrics to PushGateway. We collected data for the **Memory usage**, **Execution Time** and **Current** / **Max Concurrent Requests**.

To collect metrics on the cluster, we used cAdvisor[3] which is an open-source tool developed by Google that collects data on running containers, such as CPU usage, memory usage, etc. These metrics are then collected by our Prometheus instance and visualized in Grafana. Since both our OpenWhisk and the functions run on containers, these metrics can be used to fine-tune both the OpenWhisk deployment (increase or decrease number of instances of certain components) and to monitor OpenWhisk and the functions.

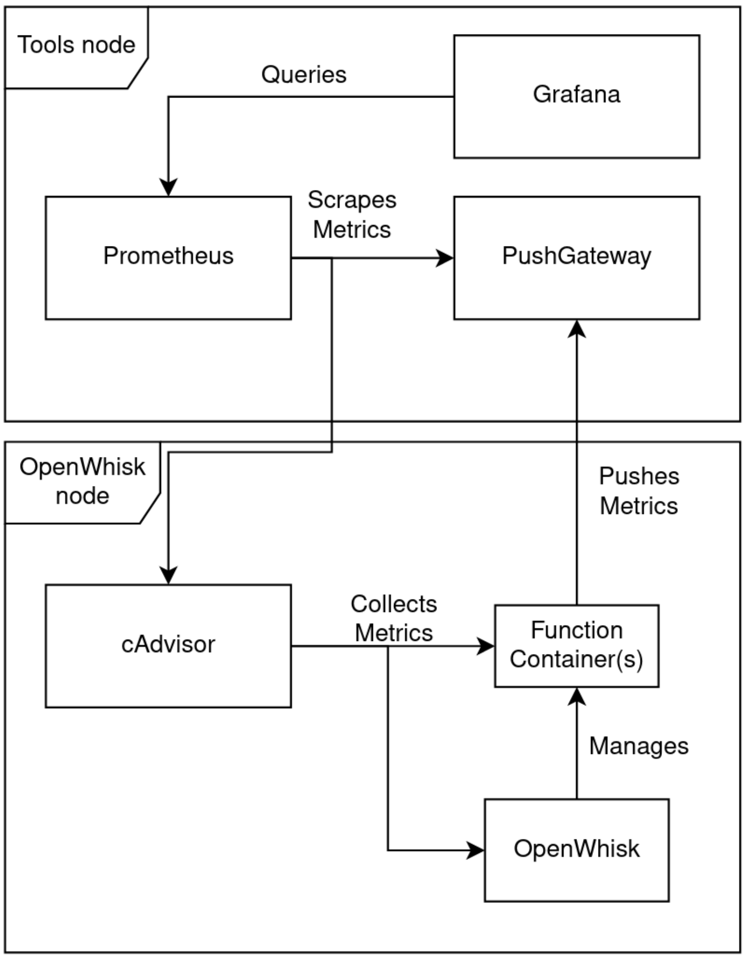


Figure 4.2: Metrics collection flow in the system.

[1]https://github.com/prometheus/pushgateway
[2]https://micrometer.io/
[3]https://github.com/google/cadvisor

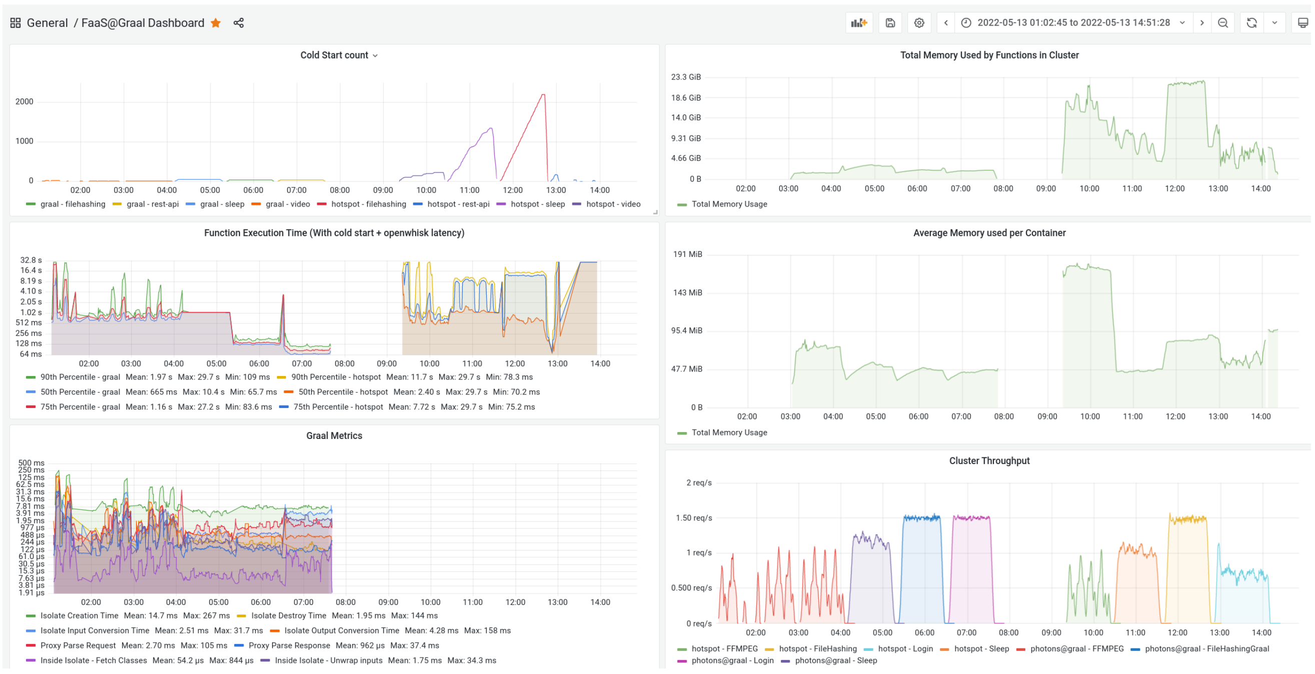

General / FaaS@Graal Dashboard
2022-05-13 01:02:45 to 2022-05-13 14:51:28
Cold Start count
graal - filehashing
graal - rest-api
graal - sleep
graal - video
hotspot - filehashing
hotspot - rest-api
hotspot - sleep
hotspot - video
Total Memory Used by Functions in Cluster
Total Memory Usage
Function Execution Time (With cold start + openwhisk latency)
90th Percentile - graal Mean: 1.97 s Max: 29.7 s Min: 109 ms
90th Percentile - hotspot Mean: 11.7 s Max: 29.7 s Min: 78.3 ms
50th Percentile - graal Mean: 665 ms Max: 10.4 s Min: 65.7 ms
50th Percentile - hotspot Mean: 2.40 s Max: 29.7 s Min: 70.2 ms
75th Percentile - graal Mean: 1.16 s Max: 27.2 s Min: 83.6 ms
75th Percentile - hotspot Mean: 7.72 s Max: 29.7 s Min: 75.2 ms
Average Memory used per Container
Total Memory Usage
Graal Metrics
Isolate Creation Time Mean: 14.7 ms Max: 267 ms
Isolate Destroy Time Mean: 1.95 ms Max: 144 ms
Isolate Input Conversion Time Mean: 2.51 ms Max: 31.7 ms
Isolate Output Conversion Time Mean: 4.28 ms Max: 158 ms
Proxy Parse Request Mean: 2.70 ms Max: 105 ms
Proxy Parse Response Mean: 962 µs Max: 37.4 ms
Inside Isolate - Fetch Classes Mean: 54.2 µs Max: 844 µs
Inside Isolate - Unwrap inputs Mean: 1.75 ms Max: 34.3 ms
Cluster Throughput
hotspot - FFMPEG
hotspot - FileHashing
hotspot - Login
hotspot - Sleep
photons@graal - FFMPEG
photons@graal - FileHashingGraal
photons@graal - Login
photons@graal - Sleep

Figure 4.3: Grafana Dashboard for Photons@Graal.

The overall system composition with the metrics collection flow can be viewed in Figure 4.2 and an example of a Grafana dashboard showing these metrics is shown in Figure 4.3. The Grafana dashboard plots relevant metrics of the systems being evaluated over time using the metrics available in Prometheus. It also plots specific Native Image metrics, such as Isolate creation and destruction time and conversion time of inputs and outputs between Java Objects and C types for Isolate data transfer, allowing us to visualize if these operations slow down the application.

Chapter 5

Evaluation

To evaluate our solution, we aim at gathering some metrics to compare the original Photons implementation, which offered no strong memory isolation between concurrent requests, and a traditional Serverless framework, that forces concurrent function invocations to be handled in separate containers and language runtimes.

We study the following metrics:

- **Throughput**, measured by the number of requests processed in a time frame;
- **Latency**, measured on the client's side, represented by the time the client perceives that it to took process a single request;
- **Memory Usage**, measured for each container and as a global cluster-wide metric;
- **Cold Starts**, i.e., the number and time to start a new execution environment.

We believe these metrics represent the most important performance indicators of a Serverless platform and we intend to study Photons@Graal's impact on each of these metrics.

5.1 Evaluation Environment

To evaluate Photons@Graal, we needed a setup similar to a Cloud that could be deployed in local servers. As we have mentioned before, we chose Apache OpenWhisk platform. We used a virtual machine allocated with **32 virtual cores** and **32GB of RAM** as the machine to host OpenWhisk, based on a physical machine with a **Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz**, that has a total of **40 virtual cores** and **64GB of total RAM**. We also used a smaller virtual machine, with 8 virtual cores and 16GB of RAM with some tools needed for the functions, such as MongoDB and MinIO. This smaller machine also contains our tools for metrics collection and visualization: Prometheus and Grafana.

We deployed OpenWhisk using a slightly modified version of the base ansible deployment available at `https://github.com/apache/openwhisk/tree/master/ansible`.

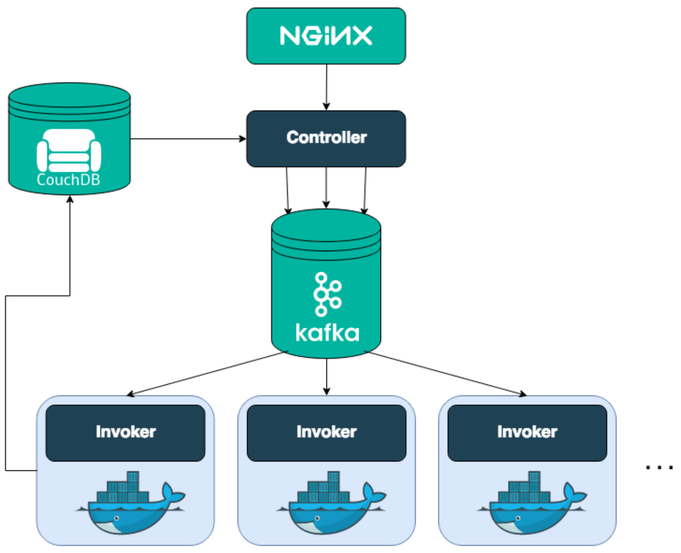


Figure 5.1: Openwhisk component architecture diagram.

We chose to use the recommended setup as described in Figure 5.1, which includes 1 nginx instance, 1 controller instance, 1 kafka instance, 1 CouchDB instance and 3 invokers. The component that is most important that it is replicated is the invoker, which is the component that handles the life-cycle of the function containers and forwards and receives the invocations to the functions. We also attempted to increase the number of available CouchDB and Controller instances but didn't get any noticeable improvement, these are mostly only replicated for fault-tolerance.

5.1.1 Workloads

In this subsection we will introduce the functions that will be used throughout the experiments to evaluate the system. These functions represent common use-cases for Serverless usage and were based on functions already used in previous work, such as in the Photons paper [1].

- **File Hashing:** Serverless is being used in distributed data processing, in which data is split in chunks and then processed in parallelizable serverless functions. To simulate this kind of workload, we fetch a small file from an external object storage such as S3[1] or MinIO[2] and apply a hash function to it;
- **REST API:** We test a simple function serving as a REST API that provides a resource. It receives some parameters and passes them to a back-end database. This serves as a benchmark for functions that execute quickly without using much resources;
- **Video Encoding:** Another possible use case for serverless is video transformation, previous work has explored this idea [41, 42], by chunking a video in smaller parts and applying a serverless function to each chunk with extremely high concurrency a video can be transformed much faster than in a regular single file transformation. We aim to test this by chunking a large video in small

[1]https://aws.amazon.com/s3/
[2]https://min.io/

files and having each function invocation fetch the appropriate file chunk from an external object storage, process the file with the function desired and submit the output. A typical case study in embarrassingly parallel workloads in distributed computing[43];

- **Sleep:** To test the overhead of the infrastructure we propose a function that sleeps for a set amount of time. This is useful because we know exactly how long the function itself took to execute, the remaining time is overhead in the serverless platform and infrastructure.

To run the workloads described above to test our solution we will deploy a modified version of OpenWhisk that can support GraalVM Native Image runtimes as described in Figure 3.5. Then, we will also do the same experiment with the original Photons[1] OpenWhisk environment, and with a regular OpenWhisk deployment with plain OpenJDK Java functions that do not support concurrency.

We will test multiple levels of concurrency in the invocations to see how our solution behaves at low, medium and high levels of concurrency when compared to Photons and OpenJDK Java. It will also important to know how many concurrent invocations Photons@Graal can execute before performance starts to deteriorate and compare the same metric with Photons.

5.1.2 Overall Methodology

In order to obtain the data for the plots that will be presented in the next sections we had to collect it using the metrics system that was described in Section 4.3.

For **Throughput**, we used a metric that counted the number of processed requests and then used PromQL, the Prometheus Query Language to calculate the throughput by simply calculating the sum of the request rate over all active functions.

For **Latency**, we collected the execution time of all invocations as a metric and then specifically calculated the **Tail Latency** as the 90th percentile of the metric using PromQL.

For **Memory Usage**, we relied on cAdvisor to collect the metrics of the containers and used the `container_memory_usage_bytes` metric, that represents the total memory usage of the container, including the memory used by the container's kernel and the application.

Finally, for **Cold Starts**, we relied on the presence of OpenWhisk's `initTime` to know if a cold start happened and how long it took. This is described more in-depth in Section 5.2.1.

For each experiment, we ran it 3-5 times, to guarantee that the results don't variate too much, and used the results of the last experiment.

5.2 Evaluating Photons@Graal Cold Starts in Cluster-Wide Deployment

One of the theoretical gains of Native Image over traditional Hotspot Java Runtimes is the much faster startup due to the Ahead-of-time compilation that produces an application that starts up almost instantly, compared to Hotspot applications that are quite slow to start due to the JVM having to load the classes and optimize the generated code with the JIT compiler, operations that in the Native Image are done in the build phase.

5.2.1 Methodology

We simulated a competitive environment in our OpenWhisk deployment by creating multiple functions and having sporadic executions for each of the functions, forcing cold starts to be happening often.

In order to know that a cold start was happening we took advantage of the verbose results when invoking a function with the `blocking` parameter, which outputs a big payload with metadata such as `initTime`[3], which is only available in the payload if the invokation required a cold start of a runtime. It represents the time that OpenWhisk took to initialize a function and doesn't contain the time spent in the internal OpenWhisk components and the time spent provisioning the container, which both take non-negligible time and are the same in both systems since both depend similarly on Docker containers.

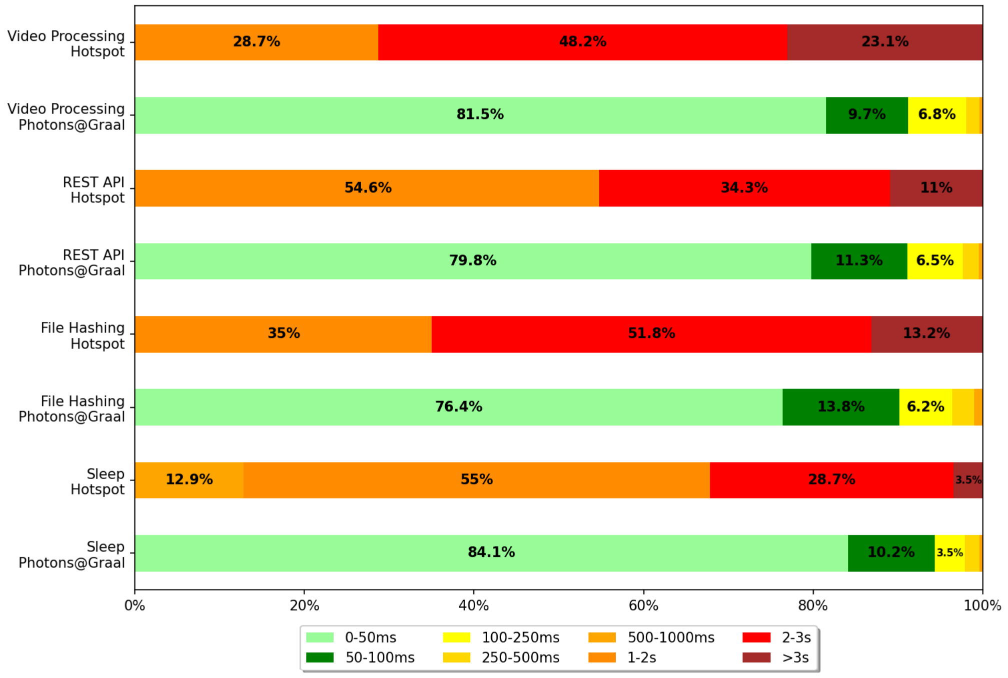


Figure 5.2: Distribution of the duration of cold starts of the selected functions deployed in a Photons@Graal and Hotspot Runtime in OpenWhisk.

5.2.2 Results

In Figure 5.2 we show a distribution of the time each function takes to start in Hotspot and Photons@Graal platforms. Since Photons@Graal takes advantage of the Ahead-of-time compilation, functions running in this platform achieve extremely fast startups, having overall 90% of the startups under 100ms. On the other hand, Hotspot functions are quite slow to initialize, most taking at least 1 second and a big chunk of them taking over 2 seconds. Since HotSpot runtimes need to initialize the function JAR along with the Runtime and load all required classes while starting it is considerably slower than

[3]https://github.com/apache/openwhisk/blob/master/docs/annotations.md#annotations-specific-to-activations

Photons@Graal, which compiles and loads the classes during the AOT build phase. These results show the suitability of Native Image-based applications for Serverless, as big chunk of the startup-time is eliminated, allowing for sporadic latency-sensistive functions to have faster overall execution.

5.3 Evaluating Photons@Graal with Synthetic Benchmarks

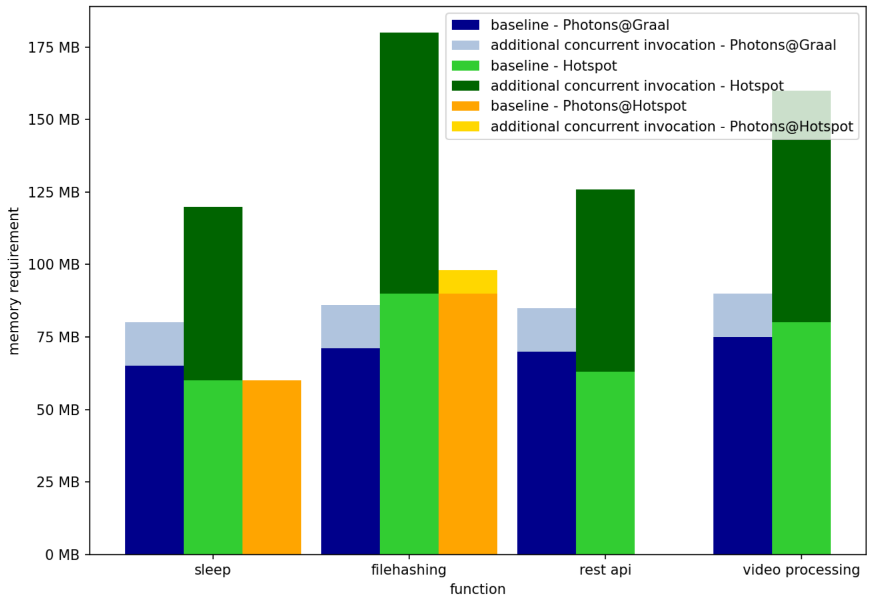


Figure 5.3: Memory requirements of Hotspot, Photons@Hotspot and Photons@Graal functions.

In this section we explore some synthetic experiments by sending a constant invocation rate (i.e., a constant workload) for function handlers. These experiments were done using the Apache Benchmark [44] tool, which is a simple CLI (Command Line Interface) that allows an HTTP Server to be benchmarked. This tool can be configured to run for N number of requests or a specified amount of time, it can also be configured to send concurrent requests up to the specified level of concurrency. This enables us to test the multiple functions with various levels of concurrency and see how they perform. In order to use this tool with OpenWhisk, we call OpenWhisk's HTTP API directly to invoke the functions.

Figure 5.3 contains a representation of the baseline and incremental concurrent invocation memory requirements of functions deployed in traditional Cloud environments, represented as **Hotspot**, functions deployed in a **Photons@Hotspot** environment and functions deployed in a **Photons@Graal** environment. The baseline represents the maximum memory usage detected in the synthetic benchmarks, and the additional concurrent invocation represents the average memory requirement of an additional concurrent request. In the case of the Hotspot environment, each runtime doesn't allow concurrent requests, so, OpenWhisk has to start up a new container for each concurrent invocation received - $total\_mem\_hotspot = baseline\_mem * concurrency$ Photons@Hotspot and Photons@Graal both support handling concurrent requests in a single runtime, making the additional memory load much smaller. Photons@Hotspot's additional memory requirements are entirely application dependent, as we can see that the sleep function has no additional requirements since it does nothing, but filehashing has a visible increase. Since Photons@Graal depends on Isolates, for each additional concurrent invocation we are creating (or reusing) a small heap that has memory requirements even if the application does nothing. Due to some problems with Photons@Hotspot, we could not run the REST API and Video Processing functions as we had runtime errors executing these functions related to Javassist.

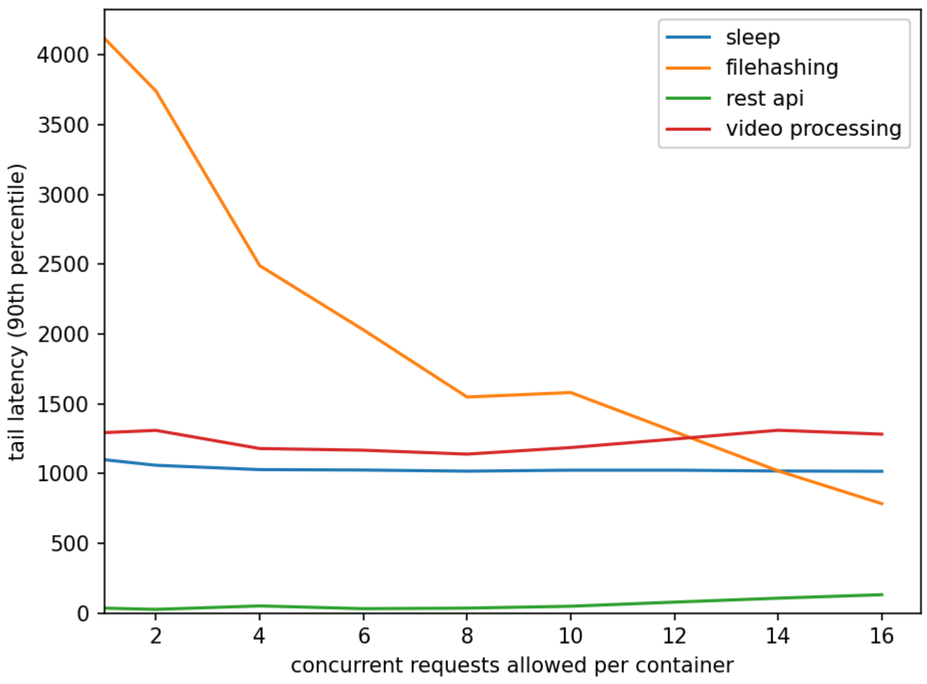


Figure 5.4: Tail latency of Photons@Graal-enabled functions with increasing level of concurrency allowed per container.

In Figure 5.4, we explore the impacts of allowing increasing level of concurrency in a single runtime, to see if we would have performance gains or losses. In order to perform this experiment, we used the same script mentioned in the beginning of this section with a slight modification - on each iteration of the script, we change the function parameters in OpenWhisk to be able to process more requests concurrently, starting at 1 and moving to 2, 4, 6, 8, etc... The request rate arriving to the OpenWhisk cluster for the whole experiment is always constant, the only variable is the number of requests that we allow each container to process. By increasing this value, we require less containers to be spawned to process the same load.

We observed a small loss of performance in functions with little application logic, such as rest api call and video processing. The rest api function only executes a rest api call to a MongoDB server with the provided credentials and is usually very fast (around 30ms), and ends up degrading as the concurrency level is increased, up to around 100ms at the highest level of concurrency measured. The file hashing function experienced a much different scenario than the other functions, the performance increased as the concurrency increased. This function fetches a remote file from MinIO and performs the hashing purely in Java code. This increase of performance can be explained on network usage optimization. Each invocation of this function fetches a 4MB file from MinIO, considering that this experiment is constantly invoking these functions, this file is being download over and over in up to 16 different processes. As the concurrency enabled within each container is increased, this file is still being downloaded at the same rate, but in fewer processes. The video function fetches a remote file from a MinIO server as well and executes a shell script using the binary tool `ffmpeg` to lower the resolution of the file. The majority of the processing here is executed outside of the Java runtime, explaining the slight performance degradation as the concurrency increases. Another reason that explains the difference between this function and the file hashing function, is that due to the compute intensity of running `ffmpeg`, the file is only 100KB, 40 times smaller than the file used in file hashing, making the network gains not as relevant.

5.4 Evaluating Photons@Graal with Azure Traces Cluster-Wide Benchmarks

To evaluate Photons@Graal in a more realistic scenario, we used a public data-set that consists of real-world traces of Serverless functions executed in Azure Functions[45]. These traces contain data on the memory usage, execution duration and number of executions per each minute in a 24h interval.

For each of our 4 functions, we searched the data-set for a function with similar memory requirements and with a number of invocations that would fit the resources of the system in the first 60 minutes, the time of the experiment.

To simulate competition in the cluster and stimulate potential cold starts due to removal of containers, for each function that we wanted to test, we created multiple functions in OpenWhisk, running the same code and the same trace. The number of functions created varies according to the requirements of each function and the number of requests of the trace selected, going from 20 (video processing) to 60 (sleep).

Since we could not get all functions running in Photons@Hotspot system, we opted to do the experiments of this section only with regular Java Hotspot functions and Photons@Graal functions.

Each Section presented below will present the results of the experiment for one of the functions listed in Section 5.1.1. Beware that the time represented in the graphs is merely for reference, it doesn't represent the exact time that both experiments were done. The experiments for the functions deployed using Hotspot and Photons@Graal were not run at the same time. They are just displayed in the same time-frame for comparison.

5.4.1 Sleep Function

Due to the nature of the Hotspot function configuration, mimicking Cloud setups, it does not allow concurrent invocations within a single runtime, so, in Figure 5.5 we can see a steady growth in the number of cold starts in Hotspot mode. As the requests arrive in the system, if at any moment there's more than 1 concurrent requests, OpenWhisk needs to start a new container to process any extra request. Since there's a limited number of containers that can be running in the cluster, the startup of a container may mean that another must be shut down, causing cold starts to happen quite often.

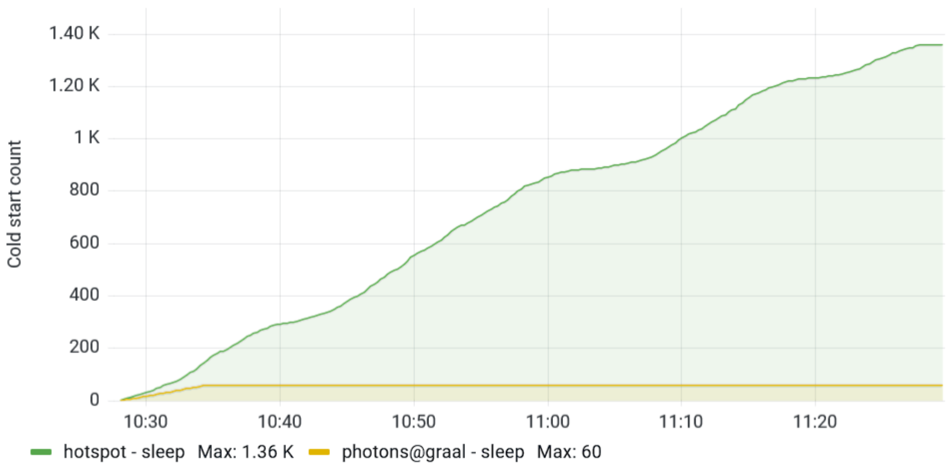


Figure 5.5: Cold start count over time - Sleep function.

In the Photons@Graal mode, we see a different scenario, a steady growth in cold starts for the first 5 minutes while the various OpenWhisk functions are being initialized, and then the growth stops. Since this system allows concurrent requests to be processed in a single runtime, in most cases, a single runtime is enough for a function, which means that it's quite unlikely OpenWhisk will start new containers for concurrent requests.

When looking at the tail latency of both platforms in Figure 5.6, that is measured on the client side, it's apparent that in the case of Hotspot platform the tail latency suffers from the cold starts. Since cold starts are quite expensive due to having to create a docker container which isn't normally fast plus initializing the function as we've seen in Figure 5.2 is also slow in an Hotspot platform, the tail latency can go from 6 to 10 seconds depending on the load on OpenWhisk. Since Photons@Graal only experiences some cold starts at the beginning of the experiment, the tail latency overall is quite good.

In Figure 5.7, we can see the evolution of the memory used solely by the function containers in the cluster over time. The memory used is significantly higher in Hotspot functions, due to the higher number of runtimes required to process all the incoming invocations due to the inability to process concurrent requests within a single runtime. Since the trace that was selected for this function varies the number of requests over time, there are times where the memory usage goes up and down, due to the higher/lower number of concurrent requests that makes OpenWhisk start and destroy containers when needed / unneeded. Overall, due to the lower number of containers active, Photons@Graal achieves a much lower memory utilization on the cluster.

In terms of the overall cluster Throughput, that was measured with metrics collected inside the functions, Photons@Graal has a slightly higher throughput, mostly due to using a lower number of containers and achieving a more efficient use of resources.

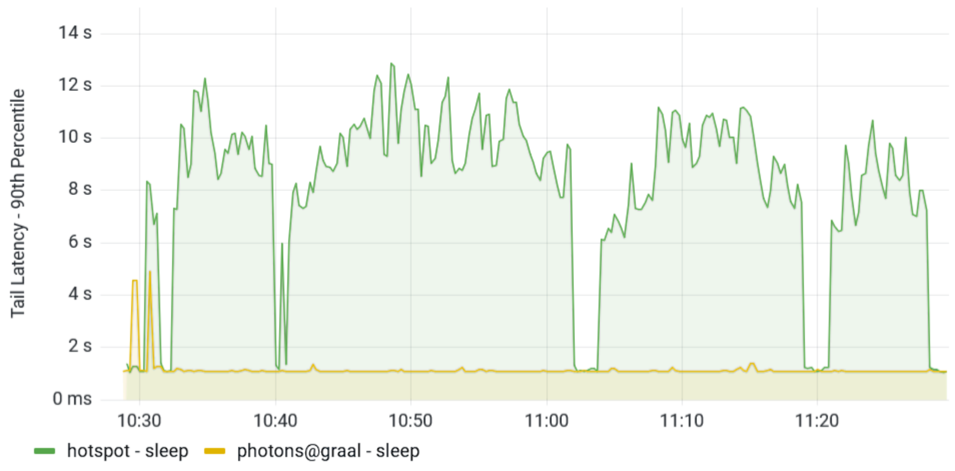


Figure 5.6: Tail latency over time - Sleep function.

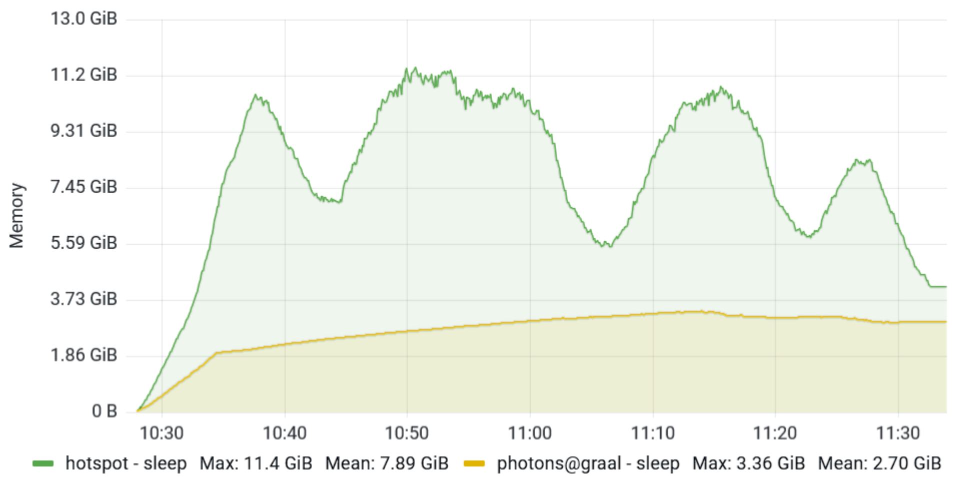


Figure 5.7: Cluster memory over time - Sleep function.

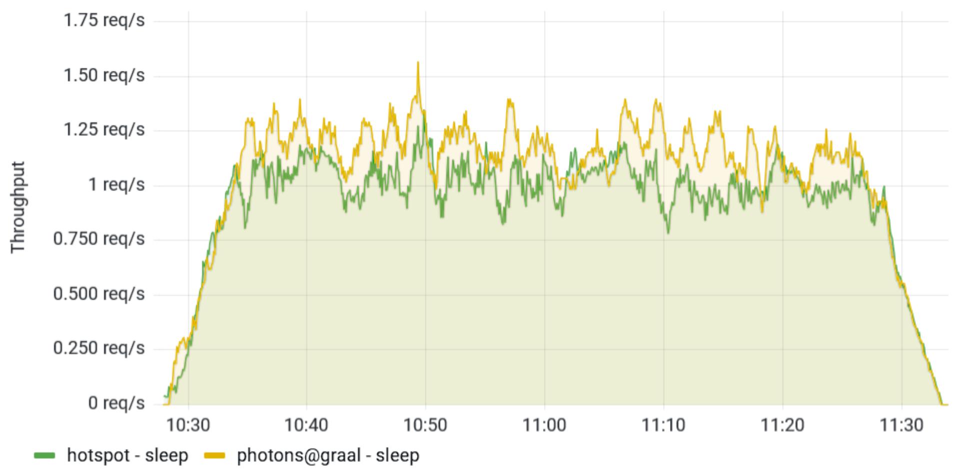


Figure 5.8: Throughput over time - Sleep function.

5.4.2 File Hashing Function

The results for the Cold Start count metric of experiment shown in Figure 5.9 show that in the Hotspot mode, the system experienced a much higher number of cold starts as expected. The increase of cold starts over time is linear in Hotspot platform, this is due to the trace selected having a linear distribution of requests, which all the replicated functions deployed in OpenWhisk follow. Due to the constant competition in the cluster, cold starts happen often in Hotspot platform.

The tail latency experienced in the Hotspot-based function remains steadily high during most of the experiment due to the common cold starts. Since there is high competition for space for containers in the cluster, cold starts can take very long. Since Photons@Graal is not affected by cold starts due to being able to handle the load by processing concurrent invocations, the tail latency remains low during the experiment, with only a slight increase in the beginning, due to the initial cold starts.

The throughput of the cluster is very similar between the two platforms as expected, since processing power should be largely the same between Hotspot and Photons@Graal considering the high rotation of HotSpot containers.

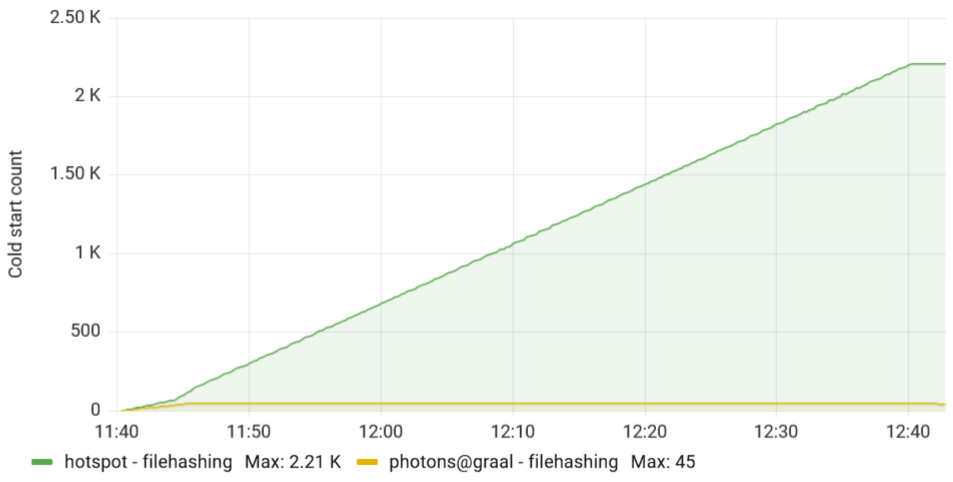


Figure 5.9: Cold start count over time - File Hashing function.

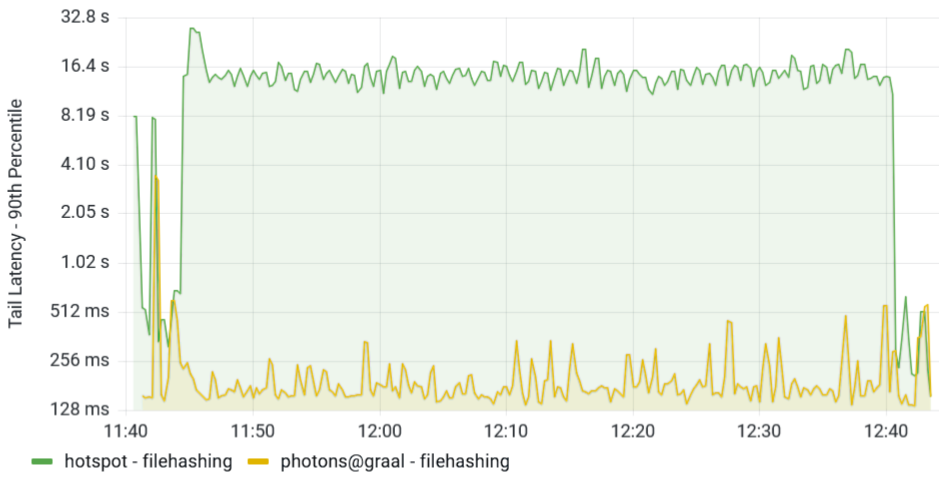


Figure 5.10: Tail latency over time - File Hashing function.

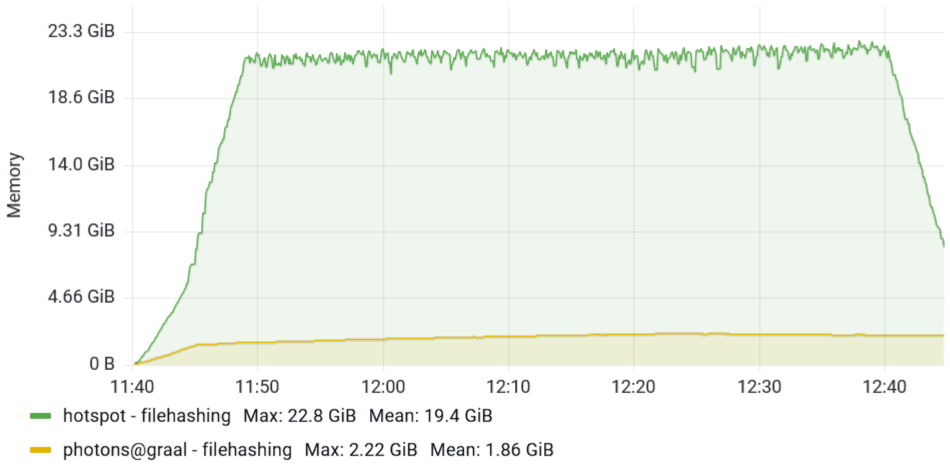


Figure 5.11: Cluster memory over time - File Hashing function.

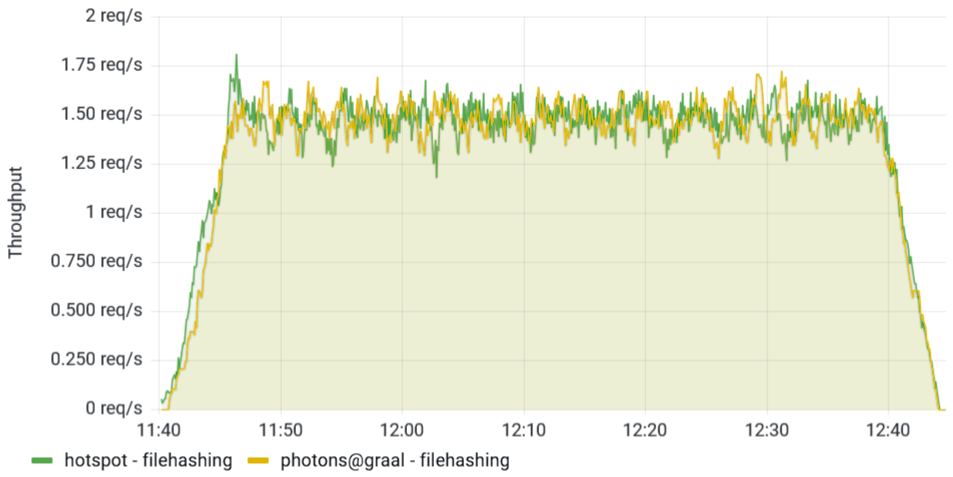


Figure 5.12: Throughput over time - File Hashing function.

5.4.3 REST API Function

In this function, we experienced a high number of **cold starts** in Hotspot mode function compared to Photons@Graal function, this can be seen in Figure 5.13. Due to the high competition for space in the cluster for containers, very often containers that are idle are removed to make space for containers to process function invocations that need that space. Since Photons@Graal processes concurrent invocations within a single runtime, this doesn't happen so often.

Due to this high level of cold starts during the experiment, the **tail latency** experienced by clients invoking functions is constantly high for Hotspot mode function. In Photons@Graal since most of the runtimes are maintained alive during the whole experiment, there is only a slight spike of tail latency in the beginning of the experiment due to the starting of the initial containers, during the rest of the experiment, the tail latency stays low. The data can be seen in Figure 5.14.

Due to the linear trace selected, the memory usage experienced during the experiment stays high for Hotspot. This is due to a constant demand of concurrent invocations during the experiment which causes a high number of containers to be spawned, showing a much increased memory usage than Photons@Graal which uses a limited number of containers which are enough to handle the load. The data can be seen in Figure 5.15.

The throughput experienced is very similar between the two functions. This is expected since Photons@Graal did not aim to improve the compute performance compared to Hotspot. The data can be seen in Figure 5.16.

The results recorded for this function are very similar to the results of File Hashing described in Section 5.4.2. This is mostly due to the trace selected being the same, as the memory requirements of both functions are very similar which is the selector argument for our script that selects a compatible trace.

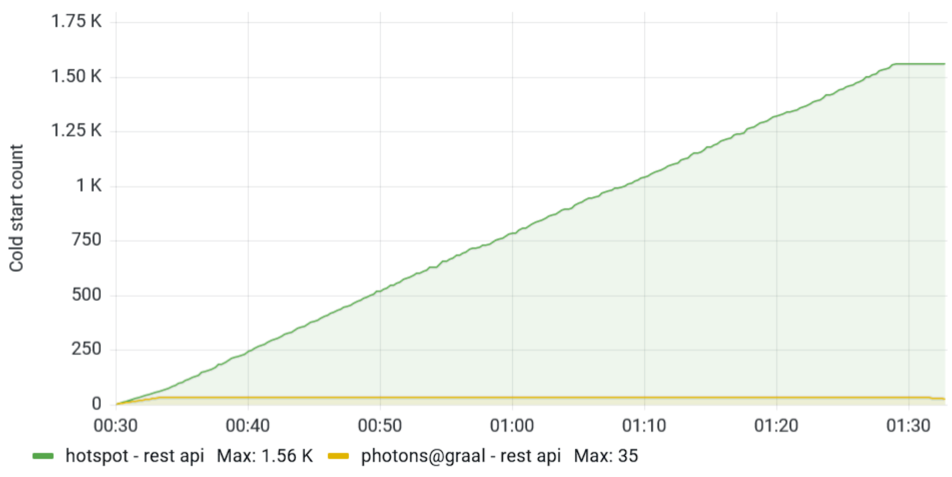


Figure 5.13: Cold start count over time - REST API function.

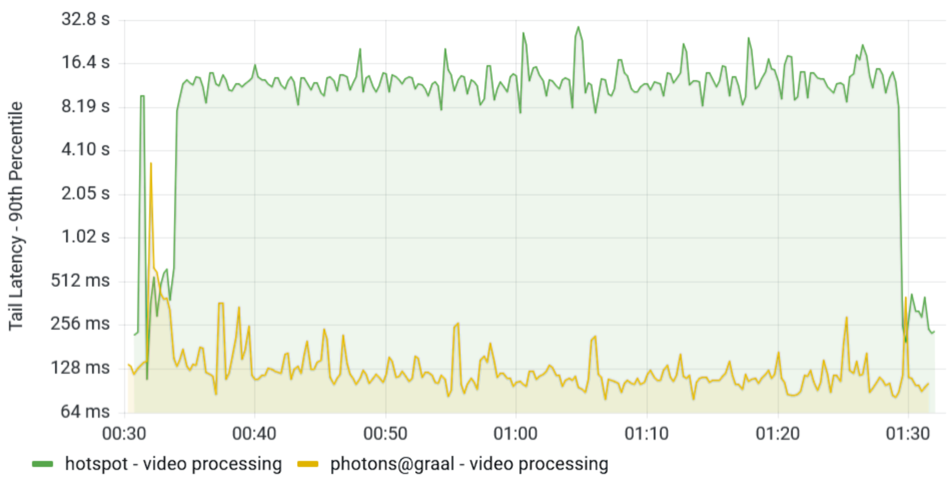


Figure 5.14: Tail latency over time - REST API function.

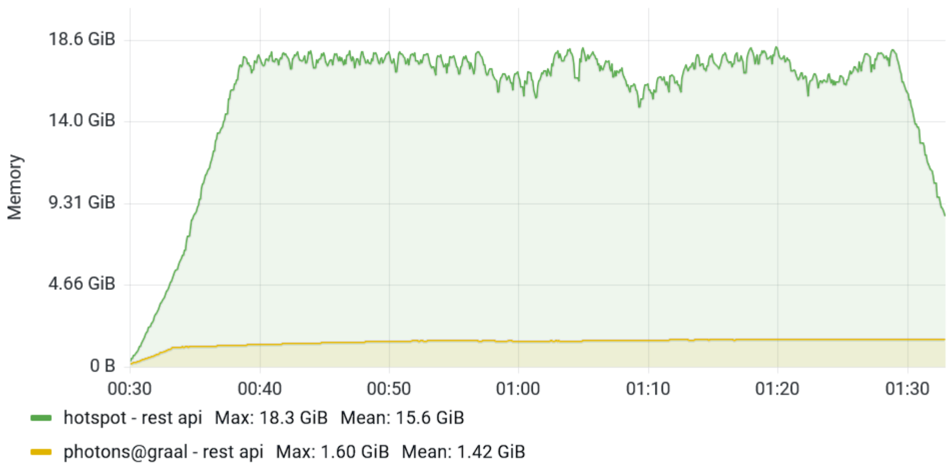


Figure 5.15: Cluster memory over time - REST API function.

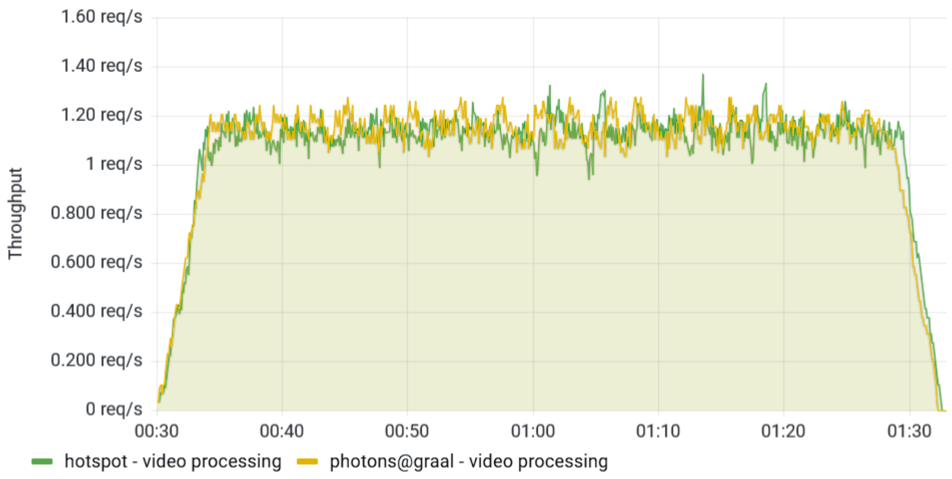


Figure 5.16: Throughput over time - REST API function.

5.4.4 Video Processing Function

The trace selected by our script for this function is quite different from the others, due to this function having higher memory requirements. The trace has periods with high number of requests and periods with little to no requests.

We can see that in the periods with high number of requests, the tail latency of the Hotspot-based function is high and the cold start count increases and in periods with a lower number of requests, the tail latency is more or less the same, or sometimes lower than Photons@Graal-based function. This can be validated by viewing the Figures 5.17, 5.18 and 5.20. The tail latency in periods of high load is up to 30 times higher in Hotspot compared to Photons@Graal.

In terms of memory, the scenario is very similar to the one described in the other functions, having high overall memory usage on the Hotspot-based functions, which dips slightly in periods with lower requests due to OpenWhisk removing unnecessary containers but is still overall more than 10 times higher than the memory used by the function in Photons@Graal.

The throughput during the experiment was very similar between the two platforms, but during periods of high load Photons@Graal ended up having a slightly higher throughput. The trace selected for this function contained some sections where the client would invoke the function up to 5 times concurrently, which in the case of Hotspot means that if the function doesn't have 5 runtimes ready to process, it will have to start them up, which takes much longer than what happens in Photons@Graal which requires a single runtime to process those 5 concurrent requests.

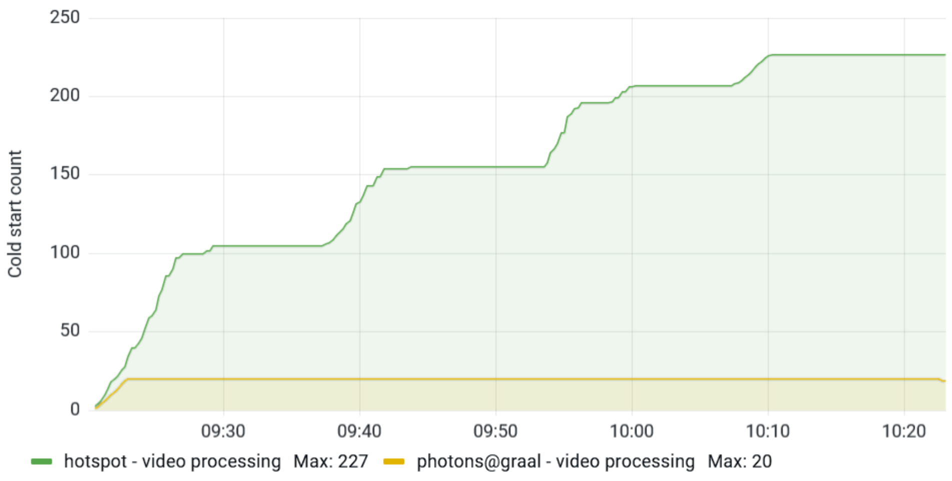


Figure 5.17: Cold start count over time - Video Processing function.

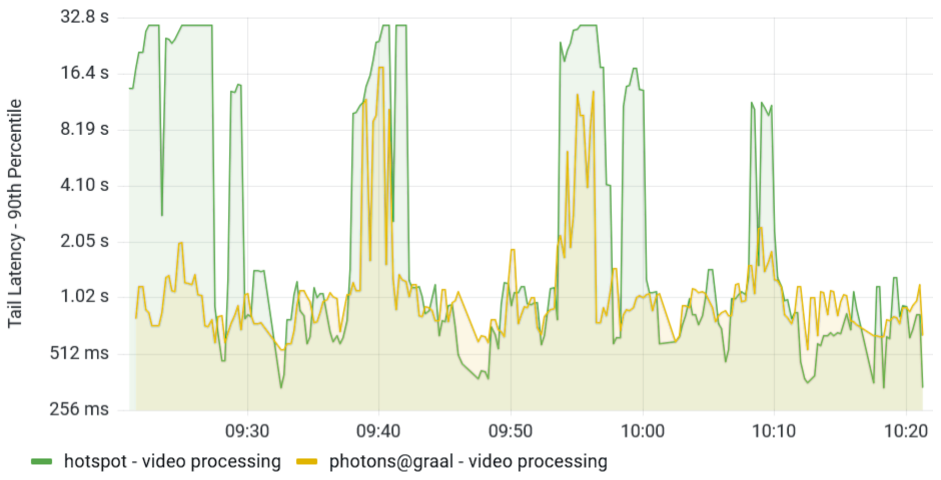


Figure 5.18: Tail latency over time - Video Processing function.

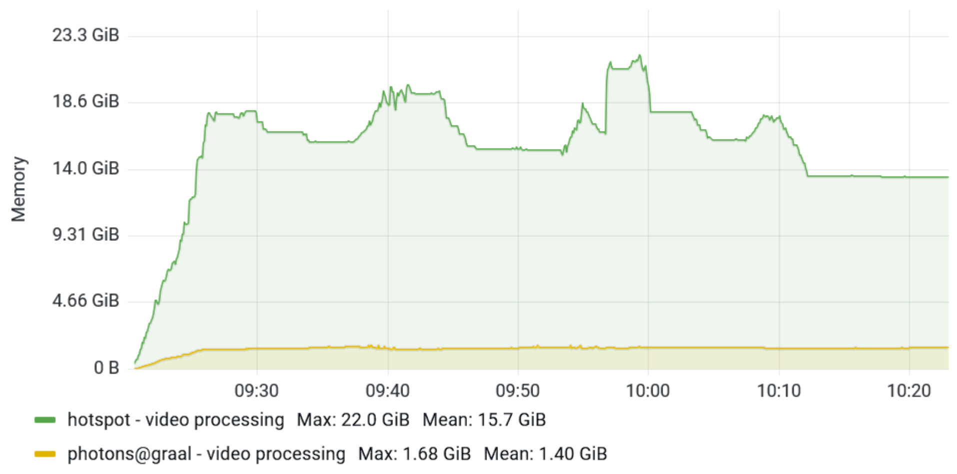


Figure 5.19: Cluster memory over time - Video Processing function.

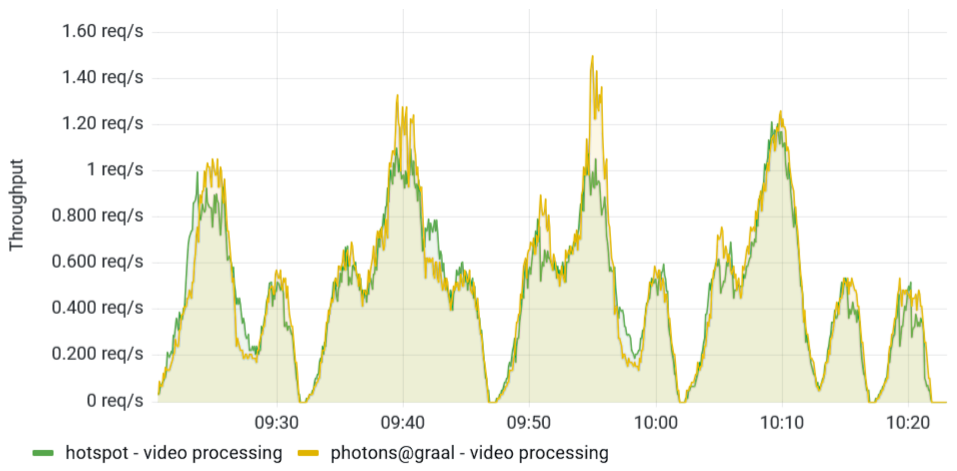


Figure 5.20: Throughput over time - Video Processing function.

5.5 Discussion

As we used Photons@Graal to deploy functions in OpenWhisk throughout this chapter, we achieved good improvements in some of the problems that usually make serverless slow.

We achieved a big reduction in **Cold Starts**, having less **91.2%** (Video Processing function) to **98%** (File Hashing function) cold starts, depending on the function. We also managed to make cold starts extremely fast, compared to functions running in Hotspot platform, which brings a big value to functions that are rarely invoked which means that the containers of these functions are rarely re-utilized, benefiting hugely from fast startups.

In terms of **Memory**, we also experienced a big reduction due to the lower number of active runtimes spawned. We achieved a reduction of from **34.2%** (Sleep function) to **90.4%** (File Hashing function) of the average memory used in the cluster during the experiment. As we've seen in Section 5.3, the cost of adding a concurrent invocation in Photons@Graal is much smaller than in Hotspot, since in the case of Photons@Graal we just need to allocate a new isolate which has non-negligible memory costs, but is still way lower than the cost of starting a new runtime. Each Isolate ends up costing around **10-15MB**, depending on the application, which is quite low compared to the cost of a whole runtime which can go from **50-90MB**.

Throughput experienced is more or less the same, as expected, since our solution did not aim to improve compute performance in Serverless applications. Although, we did see some improvement in the experiments done in Section 5.3 with the File Hashing function where we achieved a reduction on latency as we allowed more concurrent requests to be processed which did increase the throughput. This result is counter intuitive as HotSpot has a mature JIT compiler that optimizes code as the application is running and executing code, something that Native Image does not, it only pre-compiles code and pre-loads classes once during the initial AoT building process. But, since in our tests HotSpot runtimes did not have a very long lifetime due to fast executions and competition on the cluster causing a frequent removal of runtimes, the HotSpot runtimes may never reach the fully-optimized state in these workloads. We found that in these scenarios, the JIT optimizations are out-weighted by the faster startup and steady execution of Photons@Graal functions.

As of **Latency**, we managed to improve the tail latency experienced by the clients by reducing the number of cold starts that happen in the cluster. Since starting a function involves some expensive steps, such as provisioning a container and initializing the function, by avoiding cold starts we avoid doing these expensive steps which ultimately make some invocations take far longer than normal. We achieved a reduction of the tail latency during the experiment of from **80.5%** (Video Processing function) up to **98.7%** (File Hashing function).

Chapter 6

Conclusions

6.1 Achievements

In the beginning of this document, we described FaaS and its key service, Serverless, which enabled the execution of code without common code infrastructure, such as HTTP server, allowing programmers to focus on the application-specific code only. We then listed some of the inefficiencies existent in today's Cloud Serverless platforms, including the prominence of slow cold starts and the redundant memory used by the need to spawning multiple runtimes to handle concurrent invocations.

In order to give some context to the necessary concepts surrounding our work, we introduced in Chapter 2 the evolution on the architecture of applications that culminated in Serverless, described multiple types of Clouds available and the service levels they offer and analyzed the runtime virtualization levels, from VM to Container to Language-level virtualization, ending with some context on GraalVM Native Image, the tool that made this work possible. Before finishing this chapter, we looked at a few systems that tackled some of the same problems, showing a high-level overlook on their architecture and ending with a comparison on the problems of the systems along with our system.

In Chapter 3, we introduced the architecture of Photons@Graal, our solution to allow concurrent execution of Serverless functions within a single Native Image runtime leveraging Isolates. We designed the solution to work as a library that automatically manages isolates according to a Pooling mechanism, that can be integrated into any Serverless platform that supports custom runtimes, but decided on focusing more on the OpenWhisk integration as that was our selected environment. We also provided some background on the modifications made on OpenWhisk for it to support concurrent executions, as it doesn't by default.

In Chapter 4, we provided some implementation details on the tools developed to simplify the usage of Native Image APIs, due to the requirement of converting Objects to C Types. The custom runtime that was created to support Photons@Graal functions in OpenWhisk was also described more in-depth, along with the process of configuring a function to build a Native Image executable. To finish this chapter, we provided an overview of the tooling used to provide observability of the system in regards to metrics storage and collection using Prometheus and dashboards using Grafana.

Then, in Chapter 5, we presented the results of the experiments performed to evaluate the solution when compared to regular HotSpot functions and Photons@HotSpot functions. We performed both synthetic benchmarks and benchmarks following real-world usage using Azure Traces and recorded relevant metrics for each experiment that were used in the plots.

With Photons@Graal, we provided a framework to deploy functions that can be executed concurrently within the same runtime, reducing memory usage due to the lower number of containers that are provisioned and having faster cold starts due to the pre-initialization that Native Image does at build time with AoT compiling. We provided a stronger isolation of the function execution contexts which each have its own isolated Heap space, allowing the framework to control the amount of memory it wants to reserve for each function execution, compared to Photons where all invocations share the same Heap space and there is no control on each execution memory usage. But, Native Image is still a growing environment, it currently requires a semi-manual configuration to work properly, that is partially fed by running the application in non-native mode along with a Java Agent and partially manually edited in certain cases. It also provides a very bare-bones Java runtime which can be undesirable. The Native Image runtime has no support for Java Agents, JMX (Java Management Extensions), no JIT compiler to optimize code, semi-manual configuration on reflection, serialization and other features, Serial GC (Garbage Collector) instead of a more modern implementation and no Thread/Heap dump. Some of these features are available in the Enterprise version of GraalVM, such as a modern GC and Thread/Heap dumps, but these are features that are normally available in the Community Edition of regular Java VMs such as HotSpot.

6.2 Future Work

We explored some of the features provided by GraalVM Native Image that could be used to improve resource utilization in Serverless Java functions without reducing security or performance. But, there are some more interesting features that we didn't explore and could be useful to increase even further the usability and adaptability of the solution we explored. In this Section we will propose future work in the form of improvements to the user experience of creating and deploying functions with Photons@Graal and an extension to allow Photons@Graal to run functions of non-Java code, while maintaining the framework in Java code.

In this project we didn't explore much the user-facing experience of Photons@Graal. Currently, the users need to manually build the Native Image executable by building their function together with the Base OpenWhisk runtime developed, along with manually modifying the Native Image configurations for their function to work properly. A few improvements can be made here, such as developing a tool that receives the User code in a known format, such as JAR and receives some metadata such as the entrypoint class and method and then automatically generates a Native Image executable and builds it into a docker image that can be deployed to OpenWhisk, mimicking the user experience of deploying regular Java functions into a Cloud. This has a few non-trivial challenges linked to the way Native Image works, such as managing the Native Image configuration files automatically without user intervention in order to maintain the same user experience as regular Hotspot Java functions.

We used the regular GraalVM CE (Community Edition) for the experiments of this project without any Web frameworks. It could be interesting to port this implementation to a Web framework such as Quarkus. These frameworks have the added value of more features that the base Native Image SDK doesn't have, possibly helping with the developing experience. One example of such features is the `RegisterForReflection`[1] annotation that Quarkus provides which allows developers to assign classes that they want to register for reflection without having to add it to the manual configuration files.

An interesting feature that wasn't explored in this project is the Polyglot features of GraalVM. It could be interesting to port Photons@Graal to be able to run code of many different languages with a single runtime, it could be a very powerful solution to optimize memory utilization in the Cloud for non-java functions. It's possible to use Isolates in Polyglot environments, although at the moment of writing this document still it's still an experimental feature[2] and is only available on the GraalVM Enterprise version. GraalVM also has a current limitation that only one language can be used per runtime, so, if we choose to invoke a Javascript function in a Polyglot Photons@Graal runtime, this runtime is now locked for Javascript functions only during its lifetime.

[1] https://quarkus.io/guides/writing-native-applications-tips#registerForReflection
[2] https://www.graalvm.org/22.1/reference-manual/embed-languages/#polyglot-isolates

Bibliography

[1] V. Dukic, R. Bruno, A. Singla, and G. Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 45–59, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450381376. doi: 10.1145/3419111.3421297. URL https://doi.org/10.1145/3419111.3421297.

[2] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter. *Serverless Computing: Current Trends and Open Problems*, pages 1–20. Springer Singapore, Singapore, 2017. ISBN 978-981-10-5026-8. doi: 10.1007/978-981-10-5026-8_1. URL https://doi.org/10.1007/978-981-10-5026-8_1.

[3] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 24(3): 68–79, jul 1990. ISSN 0163-5980. doi: 10.1145/382244.382832. URL https://doi.org/10.1145/382244.382832.

[4] What is a web service? Web, URL https://www.w3.org/TR/ws-arch/#whatis. Accessed: 04/01/2021.

[5] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. Microservices: yesterday, today, and tomorrow, 2016. URL https://arxiv.org/abs/1606.04036.

[6] 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. URL https://arxiv.org/abs/1902.03383.

[7] On premise vs. serverless. Web, URL https://customers-love-solutions.com/?p=784. Accessed: 04/01/2021.

[8] AWS Lambda pricing. Web, URL https://aws.amazon.com/lambda/pricing/, . Accessed: 04/01/2021.

[9] Azure functions pricing. Web, URL https://azure.microsoft.com/en-us/pricing/details/functions/. Accessed: 04/01/2021.

[10] Google cloud functions pricing. Web, URL https://cloud.google.com/functions/pricing. Accessed: 04/01/2021.

[11] Custom AWS Lambda runtimes. Web, URL https://docs.aws.amazon.com/lambda/latest/dg/runtimes-custom.html, . Accessed: 04/01/2021.

[12] AWS Lambda changes duration billing granularity from 100ms down to 1ms. Web, URL https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-changes-duration-billing-granularity-from-100ms-to-1ms/, . Accessed: 04/01/2021.

[13] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, University of California at Berkeley, February 2009. URL http://berkeleyclouds.blogspot.com/2009/02/above-clouds-released.html.

[14] P. Mell and T. Grance. The nist definition of cloud computing, 2011-09-28 2011.

[15] Apache OpenWhisk is an serverless, open source cloud platform. Web, URL https://openwhisk.apache.org/, . Accessed: 23/05/2022.

[16] OpenFaaS - Serverless Functions Made Simple. Web, URL https://www.openfaas.com/, . Accessed: 23/05/2022.

[17] Fn Project - The Container Native Serverless Framework. Web, URL https://fnproject.io/. Accessed: 23/05/2022.

[18] VMWare Cloud Director — Leading Cloud Service Delivery Platform. Web, URL https://www.vmware.com/products/cloud-director.html. Accessed: 23/05/2022.

[19] With Google Cloud AI, BlackLine integrates real-time intelligent data services for finance and accounting customers. Web, URL https://cloud.google.com/blog/topics/customers/with-google-cloud-ai-blackline-integrates-real-time-intelligent-data-services-for-finance-and-accounting-customers. Accessed: 23/05/2022.

[20] B. Varghese, N. Wang, S. Barbhuiya, P. Kilpatrick, and D. S. Nikolopoulos. Challenges and opportunities in edge computing. In *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, pages 20–26, 2016. doi: 10.1109/SmartCloud.2016.18.

[21] Cloudflare Workers. Web, URL https://workers.cloudflare.com/, . Accessed: 23/05/2022.

[22] EdgeWorkers — Akamai Developer. Web, URL https://developer.akamai.com/akamai-edgeworkers-overview. Accessed: 23/05/2022.

[23] IBM Edge Functions. Web, URL https://www.ibm.com/cloud/blog/edge-computing-for-serverless-applications. Accessed: 23/05/2022.

[24] Edge Computing — CDN, Global Serverless Code, Distribution — AWS Lambda@Edge. Web, URL https://aws.amazon.com/lambda/edge/, . Accessed: 23/05/2022.

[25] Low Latency Content Delivery Network (CDN) - Amazon CloudFront - Amazon Web Services. Web, URL https://aws.amazon.com/cloudfront/, . Accessed: 23/05/2022.

[26] Intelligence at the IoT Edge - AWS IoT Greengrass - Amazon Web Services. Web, URL https://aws.amazon.com/greengrass/, . Accessed: 23/05/2022.

[27] Tutorial: Deploy Azure Functions as modules - Azure IoT Edge — Microsoft Docs. Web, URL https://docs.microsoft.com/en-us/azure/iot-edge/tutorial-deploy-function?view=iotedge-2020-11. Accessed: 23/05/2022.

[28] FogFlow - FogFlow v3.2.6 documentation. Web, URL https://fogflow.readthedocs.io/en/latest/. Accessed: 23/05/2022.

[29] kpavel/openwhisk-light: Lightweight OpenWisk-compatible runtime to run OpenWhisk actions on a local Docker engine. Web, URL https://github.com/kpavel/openwhisk-light, . Accessed: 23/05/2022.

[30] Are containers replacing virtual machines? Web, URL https://www.docker.com/blog/containers-replacing-virtual-machines/. Accessed: 04/01/2021.

[31] D. Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239), mar 2014. ISSN 1075-3583.

[32] C. D. O. Goncalves. A performance comparison of modern garbage collectors for big data environments. 2021. URL https://rodrigo-bruno.github.io/mentoring/77998-Carlos-Goncalves_dissertacao.pdf.

[33] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger. Initialize once, start fast: Application initialization at build time. *Proc. ACM Program. Lang.*, 3 (OOPSLA), oct 2019. doi: 10.1145/3360610. URL https://doi.org/10.1145/3360610.

[34] C. Wimmer. Isolates and compressed references: More flexible and efficient memory management via graalvm, Jan 2019. URL https://medium.com/graalvm/isolates-and-compressed-references-more-flexible-and-efficient-memory-management-for-graalvm-a044cc50b67e.

[35] 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. ISBN 9781450368827. doi: 10.1145/3342195.3392698. URL https://doi.org/10.1145/3342195.3392698.

[36] 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. *SIGPLAN Not.*, 48(4):461–472, mar 2013. ISSN 0362-1340. doi: 10.1145/2499368.2451167. URL https://doi.org/10.1145/2499368.2451167.

[37] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. SOCK: Rapid task provisioning with Serverless-Optimized containers. In *2018 USENIX Annual*

Technical Conference (USENIX ATC 18), pages 57–70, Boston, MA, July 2018. USENIX Association. ISBN 978-1-931971-44-7. URL https://www.usenix.org/conference/atc18/presentation/oakes.

[38] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. SAND: Towards High-Performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, Boston, MA, July 2018. USENIX Association. ISBN 978-1-939133-01-4. URL https://www.usenix.org/conference/atc18/presentation/akkus.

[39] OpenWhisk Action Interface. Web, URL https://github.com/apache/openwhisk/blob/master/docs/actions-new.md#action-interface, . Accessed: 21/05/2022.

[40] Assisted Configuration with Tracing Agent. Web, URL https://www.graalvm.org/22.0/reference-manual/native-image/Agent/. Accessed: 23/05/2022.

[41] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 263–274, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450360111. doi: 10.1145/3267809.3267815. URL https://doi.org/10.1145/3267809.3267815.

[42] S. Fouladi, R. S. Wahby, B. Shacklett, K. V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, Mar. 2017. USENIX Association. ISBN 978-1-931971-37-9. URL https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/fouladi.

[43] J. a. Morais, J. a. N. Silva, P. Ferreira, and L. Veiga. Transparent adaptation of e-science applications for parallel and cycle-sharing infrastructures. In *Proceedings of the 11th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, DAIS'11, page 292–300, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 9783642213861.

[44] ab - Apache HTTP server benchmarking tool. Web, URL https://httpd.apache.org/docs/2.4/programs/ab.html, . Accessed: 21/05/2022.

[45] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider, 2020. URL https://arxiv.org/abs/2003.03423.