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

David Jean Frickert
david.frickert@tenico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

June 2022

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 [8] 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

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 [6]. 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 [14]. 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.

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 [8] 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.

Contributions The main goal of this project is to improve application execution in FaaS scenarios, by achieving a stricter data isolation than the original Photons [8] 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 Graal 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.

2. Related Work

In this section we present the most relevant work related to the main topics at hand.

2.1. 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 bursts and down if no requests are being received.

2.2. Language Runtime-Level Virtualization

We are looking for a finer-granularity virtualization than the one provided by Lightweight Virtualization such as Docker containers, 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.

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 interesting, 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 Environmnet (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. 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 [15].

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 Garbage Collection (GC) overhead on the application.

2.3. Relevant Systems

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.

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.

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.

SEUSS (Serverless Execution via Unikernel SnapShots) [7] attempted to reduce high function initialization cost present in the cold start problem of containerization-based serverless by using unikernel [10] snapshots that have al-

ready initialized runtimes for multiple language runtimes, 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. 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.

SOCK [12] 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.

SAND [4] 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. 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. 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 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.

3. Architecture

In this section we describe our proposal for Photons using GraalVM Native Image.

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 1 we show a simple view of 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.

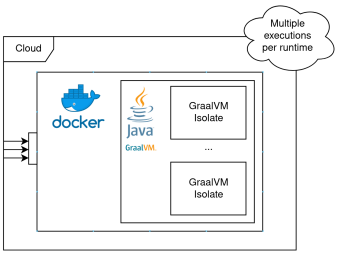

Figure 1: Overview of a Cloud using a Photon@Graal.

With this architecture, we aim to achieve a better solution at providing data isolation and faster startup than the original implementation of Photons[8], 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 baseline memory footprint than a regular Hotspot JVM based application.

3.1. 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.

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.

3.2. 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 or if resources are not properly closeable, as the mechanism only attempts to interrupt all running threads and waits for them to finish.

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.

Algorithm 1 Isolate Management Strategy

```
1:  constants
2:      MAX_ISOLATES, Compile-time constant
3:  end constants
4:  isolate_id_cache ← []
5:  procedure GET ISOLATE
6:      isolate_id ← isolate_id_cache.poll()
7:      isolate ← NULL
8:      if isolate_id IS NULL then
9:          isolate ← newIsolate()
10:     else
11:         isolate ← attachCurrentThreadToIsolate(isolate_id)
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
```

3.3. 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.

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 2 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.

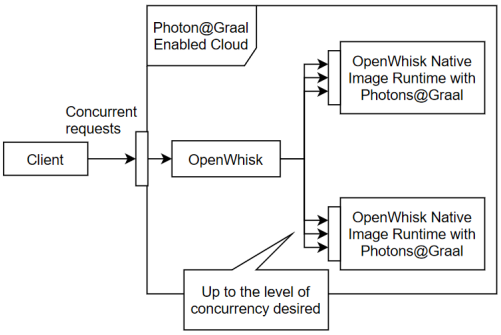

Figure 2: Photon@Graal Enabled Cloud. Using OpenWhisk as the serverless platform, this example works in a Private Cloud setting. OpenWhisk handles the scaling of Photons@Graal depending on the current load and using a custom built Native Image Runtime for OpenWhisk, each Runtime can execute multiple functions concurrently.

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 [3]. 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 3.1 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. After configuring the custom OpenWhisk deployment to allow concurrent requests, the developer just needs to configure for the function the maximum concurrency that he wants to allow.

4. Implementation

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

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.

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.

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 Section 3.1. The code is available at `https://github.com/davidfrickert/openwhisk-faas-graalvm-base`.

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, 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 JNI (Java Native Interface) calls or request proxy accesses [2]. The Agent then generates files that need to be used during build time of the Native 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.

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.

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: i) **Throughput**, measured by the number of requests processed in a time frame; ii) **Latency**, measured on the client's side, represented by the time the client perceives that it to took process a single request; iii) **Memory Usage**, measured for each container and as a global cluster-wide metric; iv **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.

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`.

We chose to use the recommended setup 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

[1] https://github.com/prometheus/pushgateway

[2] https://micrometer.io/

[3] https://github.com/google/cadvisor

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.

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 [8].

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[4] or MinIO[5] 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 [5, 9], 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 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 [11];

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 2. Then, we will also do the same experiment with the original Photons [8] 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.

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 5.1.

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.

5.1. 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.

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`, 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.

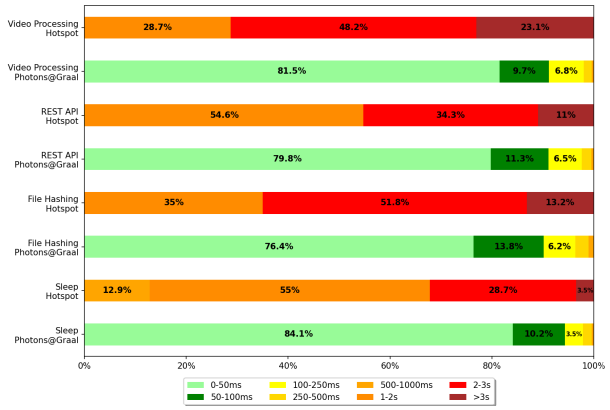

Figure 3: Distribution of the duration of cold starts of the selected functions deployed in a Photons@Graal and Hotspot Runtime in OpenWhisk

[4]https://aws.amazon.com/s3/
[5]https://min.io/

Results In Figure 3 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 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.

5.2. Evaluating Photons@Graal with Synthetic Benchmarks

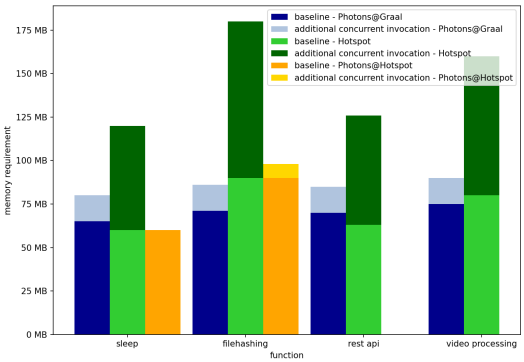

Figure 4: Memory requirements of Hotspot, Photons@Hotspot and Photons@Graal functions

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 [1] 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 4 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.

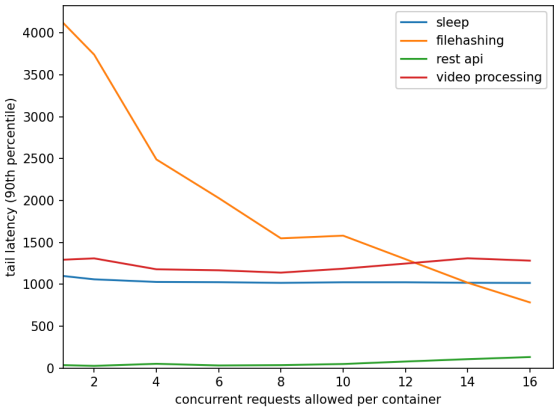

Figure 5: Tail latency of Photons@Graal functions with increasing level of concurrency allowed per container.

In Figure 5, 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.

5.3. 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 [13]. 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. 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.

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 6 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.

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 7, 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

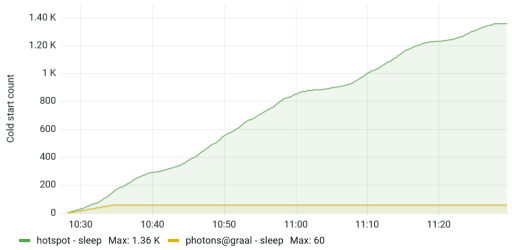

Figure 6: Cold start count over time - Sleep function.

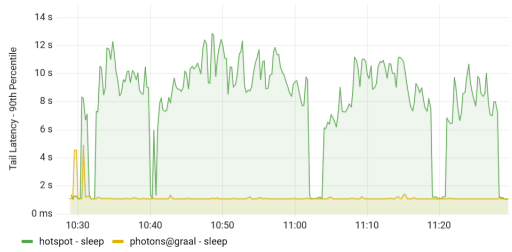

Figure 7: Tail latency over time - Sleep function.

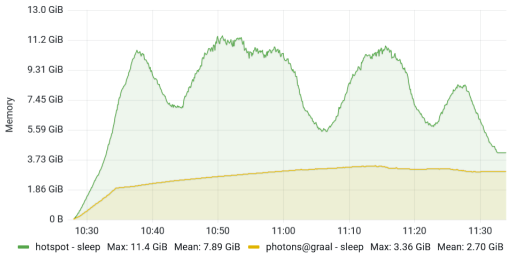

Figure 8: Cluster memory over time - Sleep function.

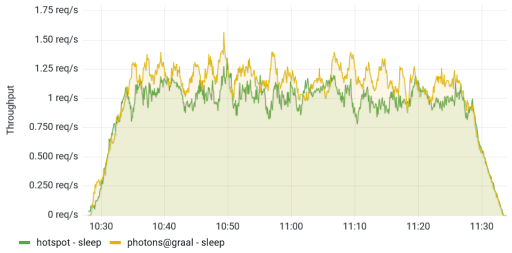

Figure 9: Throughput over time - Sleep function.

in Figure 3 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 8, 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.

File Hashing Function The results for the Cold Start count metric of experiment shown in Figure 10 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.

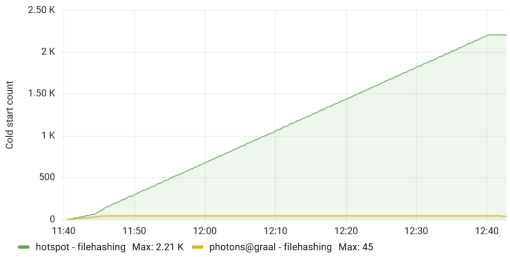

Figure 10: Cold start count over time - File Hashing.

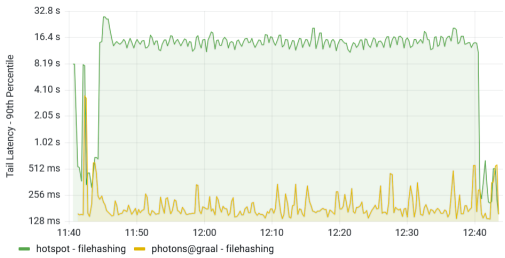

Figure 11: Tail latency over time - File Hashing.

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 14.

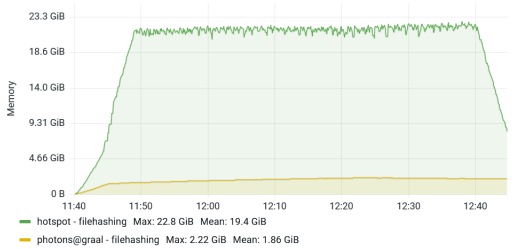

Figure 12: Cluster memory over time - File Hashing.

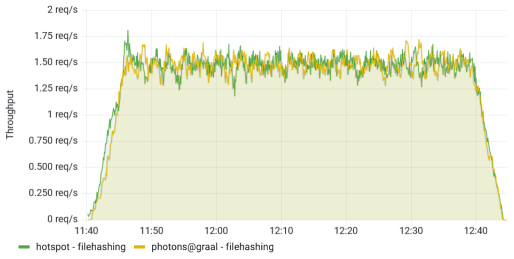

Figure 13: Throughput over time - File Hashing.

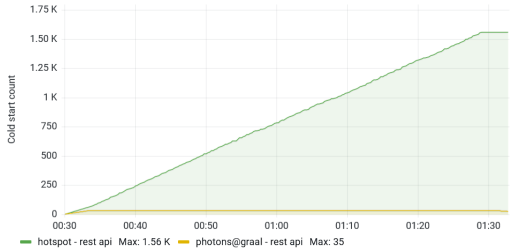

Figure 14: Cold start count over time - REST API.

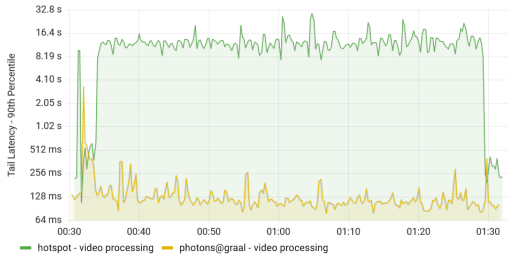

Figure 15: Tail latency over time - REST API.

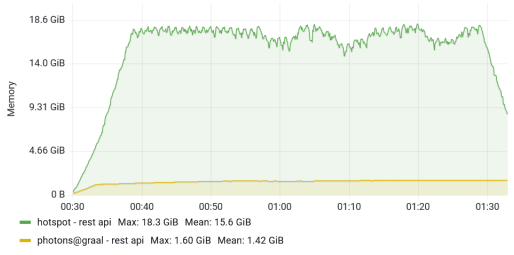

Figure 16: Cluster memory over time - REST API.

Due to this high level of cold starts during the exper-

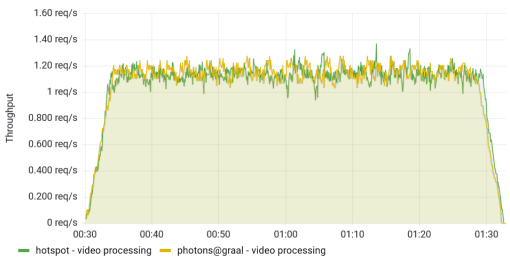

Figure 17: Throughput over time - REST API.

iment, 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 15.

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 16.

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 17.

The results recorded for this function are very similar to the results of File Hashing described in Section 5.3. 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.

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.

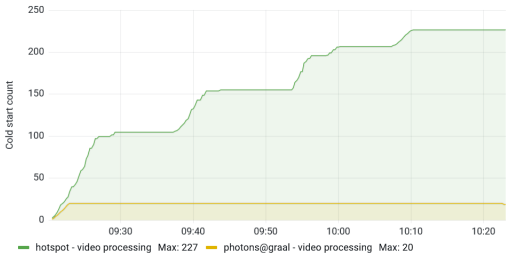

Figure 18: Cold start count over time - Video Processing.

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 18,

Figure 19: Tail latency over time - Video Processing.

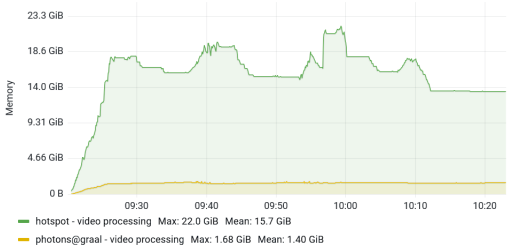

Figure 20: Cluster memory over time - Video Processing.

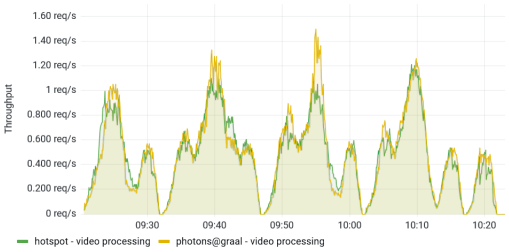

Figure 21: Throughput over time - Video Processing.

19 and 21. 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.

6. Conclusions

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 con-

texts 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.

References

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

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

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 24(3):68–79, jul 1990.

[15] 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.