ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Thin Serverless Functions with GraalVM Native Image

Master's Thesis Nr. 339

Systems Group, Department of Computer Science, ETH Zurich

by

Sutao Wang

Supervised by

Dr. Rodrigo Bruno (Oracle Labs Switzerland)

Dr. Ingo Müller

Prof. Dr. Gustavo Alonso

22.10.2020 - 22.04.2021

D INFK

Abstract

In the past two decades, cloud computing has reformed the IT service market. Serverless as its successor releases cloud users from heavy burdens of resources management and system administration. The auto-scaling and finer billing granularity make serverless easy-to-use and cost-efficient. However, serverless platforms are usually suffering the high latency caused by cold starts. Duplicated allocation of language runtimes, libraries and shareable program state for different invocations causes a huge waste of memory.

A recent work, *Photons*, enables collocation of functions of the same user inside the same runtime. *Photons* reduce the memory footprint and the number of overall cold starts by a lot without performance degradation. However, *Photons* are implemented in the normal JVM, which still suffers from slow starts and a large memory footprint. All objects of concurrent invocations are located on the common object heap. Heavy burdens of garbage collection causes high response tail latency.

In this project, we design and implement a serverless proxy runtime using GraalVM Native Image Isolate. Native Image Isolate supplies better isolation for collocated invocations. Separated heap spaces enable more efficient independent memory management, while state sharing among different isolates becomes difficult. Ahead-of-time compilation makes isolate proxy start up fast with low memory footprint. We propose isolate pooling and shared isolate to enable object caching and state sharing in our design. Different workloads are introduced to evaluate our isolate proxy. Isolate proxy can reduce runtime memory footprint by up to 59% and starts up 10x faster, with slightly a lower peak throughput compared to *Photons*. For memory-intensive workloads, distributed garbage collection reduces worst-case latency up to 50%. We use realistic serverless invocations pattern to perform a cluster-wide event-driven simulation, which shows that our isolate proxy can reduce the overall cluster memory usage by 30% while keeping slightly better overall response time compared to *Photons*. These results indicate that it is worthy to trade fast start up and low memory footprint for peak performance in serverless context.

Acknowledgements

The master thesis is an important milestone in my master study. The whole procedure was not easy but very fruitful.

I would like to give my best thanks to Dr. Rodrigo Bruno and Dr. Ingo Müller, who can always give me support on any problems I encountered during the thesis.

I want to thank Prof. Dr. Gustavo Alonso to give me the opportunity to work within the systems group. During our discussions throughout the project, he can always give me insightful guidance and point out the correct direction.

I also want to give special thanks to Vojislav Dukic, who generously shared his simulator code and helped me to set them up.

At last, I would like to thank my parents, Yilei Wang and Derong Su. They have been supporting me no matter what happened. Their eternal love and support gave me courage and made me strong.

Contents

List of Figures

List of Tables

Introduction

In the past two decades, cloud computing has reformed the IT service market. Cloud computing provides computing as a utility and allows users to pay as they use per second with a minimum of one minute [EC2]. However, in traditional cloud services, heavy burdens of resources scaling and system administration are left to cloud users. The emergence of serverless platform solved this problem. Serverless provides auto-scaling, adding system elasticity without server provisioning effort from end users. Finer pricing granularity down to even 1ms [Lam] also makes a big step forward to truly pay-as-you-use manner. Regular serverless platforms allocate independent VM instances for different invocations.

Allocating VM instances duplicates of language runtime, libraries and shareable program state such as machine learning model not only causes huge waste, but makes slow cold starts problem more severe. Dukic et al. [DBSA20] proposed a new sharing schema, Photons, to address these issues. Photons allow collocation of same functions from same user inside the same runtime. Using bytecode manipulation, Photons separate private states for each invocation to guarantee the correctness of their execution. Under proper resource configurations, Photons can scale out without performance degradation. Using Photons saves large fraction of system memory footprint for concurrent invocations and reduce the number of overall cold starts significantly.

1.1 Motivation

Although the design of Photons is simple and practical, there are still some problems. First, the Bytecode manipulation is prone to errors and difficult for debugging in source code level. Reflective operations have to be manually delegated in the framework as well. Second, concurrent invocations allocate objects on the common garbage heap of Java Virtual Machine (JVM). Accumulated burdens of garbage collection may cause high response latency. Just-In-Time compiler of normal JVM profiles and compiles in the background all the time, potentially hav-

ing resources contention with the executing invocation. Furthermore, normal JVMs still suffer slow start up and consumes large memory footprint.

GraalVM Native Image compiles Java application ahead-of-time into a standalone executable, with faster startup and much lower memory footprint. Native Image Isolates allow multiple VM instances with separated heaps within one process. Isolated heap space guarantees correct execution inside each isolate, while making it difficult for state sharing.

To achieve runtime sharing as in Photons, we use different isolates to execute concurrent invocations. Isolate separates private states of invocation automatically, ensuring the correctness execution of each invocation. During implementation, we find object caching is critical for performance, and state sharing is necessary for many workloads. Despite of the restrictions of isolate, we propose our solutions of isolate pooling and shared isolate to enable these features. Based on our implementation, we perform machine local evaluation using different workloads. We find that isolate proxy has significantly faster start up and much lower memory footprint. On the other hand, JIT compiler of fully-fledged JVM used in photons proxy can achieve higher peak throughput. In order to study this trade-off, we use realistic production traces together with our measurements to run cluster-wide event-driven simulation. Simulation result shows that, with lower overall memory consumption, isolate proxy encounters less cold starts under same restricted cluster size. Shorter cold start time and faster ramp-up let isolate proxy always have better response latency. Our work reveals that it is worthy to trade fast start up and low memory footprint for peak throughput in the serverless context.

1.2 Thesis Structure

In the second chapter, we introduce the evolution of cloud computing, pointing out the attractiveness and problems of serverless. We explain the design philosophy and system architecture of Photons, analyzing its problems in detail. Lastly, we introduce the GraalVM Native Image and Native Image Isolate, using sample code to illustrate Isolate life cycle.

In Chapter three, we start by explaining serverless system architecture for Isolate proxy. Then we study the overhead to use Isolate and argument passing between different Isolates to examine our design decisions. At the end of this chapter, we propose our solutions for object caching and state sharing.

In Chapter four, we introduce four different workloads, showing the strengths and weaknesses of using Isolate proxy based on results of machine-local evaluation.

In Chapter five, we introduce the simulation traces and simulator logic. Then we discuss the design trade-off based on the simulation results.

At last, we discuss related works and give conclusion and outlook for our project in chapter six and seven.

Background

In this chapter, we introduce more about serverless computing and serverless platform. Then we go into details about Photons [DBSA20], analyse how authors of Photons leverage runtime sharing and ensure the correct execution of each shared invocation. Based on the analysis of Photons, we also point out the drawbacks of their solution. To address those issues, we study Graal Virtual Machine (GraalVM) and GraalVM Native Image (GraalVM NI), highlight its features, and discuss the possibility it provides us.

2.1 Evolution of Cloud Computing

As we already introduced in the beginning of Chapter 1, serverless has come into more and more developers' mind. Due to a survey from *Serverless Inc* in 2018 [Pas], 82% participants indicate that they have used serverless at their work, whereas only 42% participants in the previous year (2017). There are many companies, which have no public cloud experience before, choosing to start with serverless as their first cloud computing service. What leads to the boost of serverless computing? To answer this question, we start by introducing the traditional serverful computing and then exploring the attractiveness of using serverless.

For many years, huge internet companies have been developing their own data centers to deal with their increasing business demand. Until early 2000s, large scale data centers could be equipped with a great number of commodity computers due to development of distributed system technologies. These computer clusters are designed to be highly elastic. They are capable to deal with highest peak demand and stay idle to save power when no requests come in. Although application of commodity computers has already reduced the data centers' cost by a lot, there are still many idle machines during off-season not being properly utilized. After building up data center infrastructures and technologies to manage large-scare distributed system, huge internet companies were seeking new revenue opportunity to increase utilization of their data

centers during off-season, for example by supplying services to third parties. Cost of building infrastructure would then be amortized, which in turn makes their public cloud service more price competitive than cloud users building their own computing facilities. Higher utilization increases power-efficiency of data centers as well. Since the success of Elastic Compute Cloud (EC2), which was first released by Amazon.com back in 2006, big companies like Microsoft, Google, IBM, Oracle etc. have released their own cloud computing platform one after another.

Cloud Computing provides computing as a utility, giving users the illusion of infinite amount of resources as they need [AFG+10]. Serverful Cloud Computing charges per allocated resources. This supplies the opportunity for many companies to save their cost. They can choose to use small amount of machines on the cloud at the start of their business, then increase their service capacity as their demand increases, instead of investing their own large data center. On the other hand, if the application is capable for parallel computing, such as batch processing, users could choose more computing resources for the exchange of shorter execution time with nearly the same cost. Cloud providers also charge resources usage using time-based pricing policy. This encourages users to release computing resource when they do not need them. Highly virtualization of hardware resources inside data center supplies the possibility for workloads multiplexing, which in turn increase data center utilization and reduce the cloud providers' cost. As predicted by [AFG+10], cloud computing has rapidly grown in the past ten years and has become one of the bases of IT industry. Main obstacles pointed by [AFG+10] have been addressed, and we are already in the Cloud Era.

Cloud platform, like EC2, provides users great flexibility to select Operating Systems, Libraries and Programming Languages as they like. However, administration of such an huge system is cloud users' responsibility. Aspects like scaling, deployment, fault tolerance, instance selection, security patches, monitoring, logging etc. still need cloud users to care about [JSSS+19]. Using serverful cloud computing reduces the difficulty and cost to build data center hardware facilities but does not reduce the difficulty to operate distributed applications for a huge size-changing cluster. Therefore, many open-source software showed up in the past years, trying to fill this gap. Kubernetes, developed from Google internal tools, aiming for container management, has been widely used for automatically deploying (micro)services. It supplies so called "container orchestration", which takes care of "scaling and failover" for the application running on distributed systems. For example, Kubernetes can be configured to add new containers and discard already existed containers automatically. When some container in the system is down due to system failure, Kubernetes replaces it in the background. What's more, after resources (such as number of CPUs and amount of RAM) of each container has been properly configured, Kubernetes tries to bin pack deployed containers, achieving a high usability of cluster resources [Wha].

Open-source automation software supplies convenience for container management, while a new cloud computing paradigm, Serverless, wants to release all resource management and system administration from end users. Serverless aims to reuse idle resouces in a much finer time-granularity, while supplying enormous automatic elasticity. Amazon.com first released their Function-as-a-Service production AWS Lambda back in 2015. AWS Lambda discards the management and administration of servers from cloud users and simplifies the development difficulties for cloud native applications by a lot [AWSb]. Right now, serverless users just need to write their application code using vendor supported programming languages, set up event triggers for their application and then make deployment to the cloud. All other parts of system management

would be taken care of by cloud vendors, and so is this manner of computing called serverless, though it does use servers to run the application code. Another innovative step of serverless computing is the finer pay-as-you-use pricing policy. Under serverful cloud computing pricing model, like in EC2, users can be billed per-second increments with a minimum of 60 seconds [EC2]. Serverless computing reduces this granularity further down to 100ms and recently even down to 1ms [Lam]. Billing time with serverless is very close to real execution time of the application. Finer billing granularity reduces the cost of cloud users by excluding unused milliseconds from their bill. Cloud users do not need to waste time to "optimize" their application to utilize that whole second. This is also the key difference to those serverful systems using Kubernetes. Those systems supply similar container management functionality as serverless. However, users are still charged per second or even per hour, even if the allocated resources are idle for most of the time [JSSS+19]. Serverless platform has the ability of automatically scaling up and down (to zero) with high elasticity. Just like the description given in [JSSS+19], serverless "represents an evolution that parallels the transition from assembly language to high-level programming language".

Serverless typically allocates different VM instances to execute different invocations. Start up time of VM instances is very high comparing to the execution time of invocations. Therefore, after each invocation, VM instances are kept alive for some time, avoiding cold start for potential successive invocations. However, due to the absence of infinite memory resources, warm VM instances cannot be kept forever in the memory. Memory footprint and startup latency are therefore the main aspects need to be improved in serverless platform.

While independent execution environment for each invocation supplies more safety and security, it creates a lot of resource duplication. Language runtime, libraries, shareable application state etc. are allocated multiple times in each VM instance as well, which causes huge waste of memory resources. Recent work Photons [DBSA20] proposed a new sharing schema, trying to alleviate this issue.

Although serverless is attractive enough to make more companies use cloud computing, there are still many improvements to be made. One good example is the large memory footprint of invocations and response time latency caused by cold start up. For safety and security reasons, today's serverless platforms [ABI+20] [Apaa] are scheduling each concurrent invocations into different executing environments, even if those invocations are running same code snippets triggered by the same cloud user. We take system architecture of Apache OpenWhisk [Apaa] as an example to illustrate this.

In Figure 2.1 we can see that, after serverless application being deployed to OpenWhisk, there is an event interface exposed to cloud users. If we take RESTful service as an example, REST API Interface would be applied here. Load balancer and controller sits behind REST interface, sending those requests further to message queues. Message queue dispatches the requests to different invokers, which takes care of assigning a container to execute that invocation. After invocation finishes, today's platform usually keeps those warm containers for some time according to different schedule policy. If there is any successive invocation coming in, warm container can directly be assigned to deal with it, without suffering from another cold startup. Keeping a pool of warm containers can reduce many cold starts and has been applied by many cloud providers. However, this does not reduce cold starts for concurrent invocations. Each one of concurrent invocation is dispatched into different containers in most of serverless platforms.

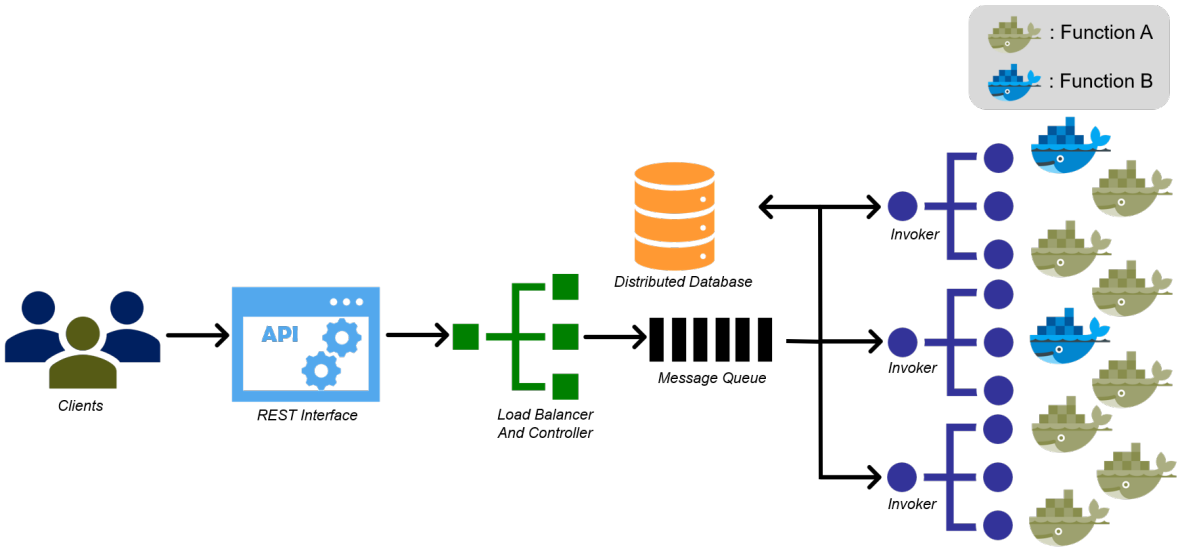

Figure 2.1: *Apache OpenWhisk System Architecture, adapted from [Apaa]*

Realistic serverless invocation statistics released by [SFIG+20] indicates that there are large number of invocations of the same function from same users are coming concurrently, so there would be a huge benefits to enable execution environment sharing among them. Since same functions from same user have same environment requirement, using different containers to run them causes a huge waste of memory. On the other hand, each concurrent invocation has to wait for runtime initialization and library loading etc. until it can be handled. This also causes significant higher response time latency.

2.2 Runtime sharing enabled by Photons

Dukic et al. study the above problem and propose a new framework called Photons [DBSA20], using language runtime sharing to reduce invocation memory footprint and reduce overall cold starts by a lot. In their work, authors start by studying the shareable memory components using Python and Java runtime running an image-classification workload. As illustrated in Figure 2.2, they find that most of memory usage consisting of runtime, libraries and machine learning model itself, which are same for each of the concurrent invocation. They argue that the isolation via different containers for *same functions of same user* is redundant, and it is possible to allocate parallel invocation inside same execution environment (container), as long as proper runtime-level isolation being performed to separate private state of each invocation, plus appropriate resource scaling configuration in order to keep minimal possible performance contention. Once these can be done, we may save a lot memory footprint and experience much fewer cold starts.

Based on Photons' design, there is a new sharing schema for same functions from same user. As illustrated in Figure 2.3, language runtime, invocation handler and shared state (such as machine learning model) are now shared among concurrent same invocations. Private state of each invocation will be separated by Photons automatically, such that each invocation could

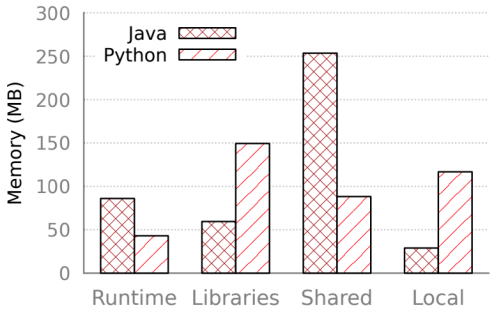

Figure 2.2: *Runtime Memory Breakdown for Java and Python [DBSA20]*

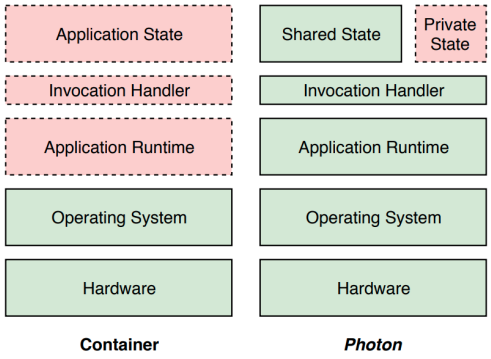

Figure 2.3: *Sharing Schema of Regular Container and Photons [DBSA20]*

be executed correctly. Using this abstraction, Photons enable collocation of same function invocations from same user inside a common execution environment. Under the experiments using Java Virtual Machine (JVM) as the runtime, Photons can "save memory consumption by 25% to 98% per invocation under at most 5% performance degradation and can reduce the overall memory utilization by 30%, and the total number of cold starts by 52%" [DBSA20].

Invocation-dependent private states consist of two parts: mutable static fields and static initializers. Mutable static fields need to be transformed to invocation-local fields, and static initializers need to be executed per invocation as well. On JVM, Photons use bytecode manipulation to achieve this separation. They design a class loader using Javassist [CN03], performing bytecode transformation during class loading (application initialization per jar file). For mutable static fields, class loader will generate corresponding variables for each invocation. Newly generated variables will be stored in a hash table, while original variable declaration would be removed. Besides the declarations of those static fields, any read or write access to them need to be detoured to the hash table. Transformed variables will be identified by invocation-distinguish identifiers. After this step, static initializers would be cloned into some method, which would be executed before each invocation, so that static fields can be properly initialized for each invocation. The original static initializers would be deleted to avoid duplicated initialization. Finally, to avoid possible memory leakage after each invocation finishes, authors of Photons select WeakHashMap in Java to store transformed static variables. Non-reachable objects after each invocation would be automatically removed by runtime garbage collector. Using this method, Photons ensures the correct execution of each concurrent invocation.

Since all the invocations are using the common heap, it is also easy to establish a global object store for convenient state sharing. Large objects, like data base connections, are expensive to be initialized. Using the global object store, Photons could cache pool of connections per invocation. Large shareable object, like machine learning model singleton, can also be easily shared using this data structure.

2.2.1 Photons System Architecture

To integrate Photons into existing platform, e.g. Apache OpenWhisk, several modifications need to be made. To allow collocation of different invocations of same function from same user inside a shared runtime, the logic of invocation dispatcher need to be changed. Besides that, the OpenWhisk runtime for Java needs to be modified by adding step for bytecode manipulation. Due to source code of Apache OpenWhisk runtime for Java [Apab], the application is nothing but a Hypertext Transfer Protocol (HTTP) proxy. The proxy instruments two different handlers to deal with service initialization and function invocation. Cloud users compile and build their serverless function (also called Action in OpenWhisk terminology) as a jar file and send the jar file to the proxy for action initialization. All OpenWhisk Java actions are supposed to have a main function with the following signature:

```
public static JsonObject main(JsonObject args);
```

During action initialization, proxy receives the jar file using initialization handler and saves it to local file system. A jar file loader searches and registers the main function with the above signature into a runtime class loader. Once initialization has been done, proxy is built into docker image and deployed to serverless system. When invocation comes, a Docker container would be instantiated. Invocation handler parses the input arguments of *com.google.gson.JsonObject* format, invokes the registered main function using reflection, gets return value of application code and send it back as HTTP response.

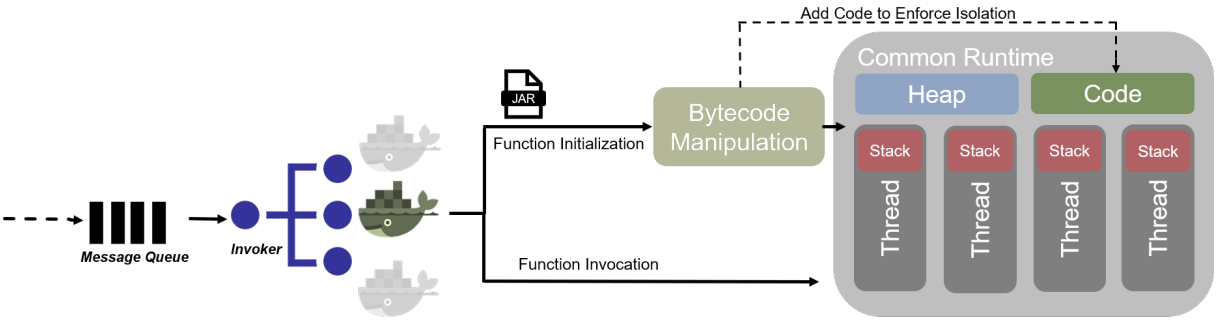

Figure 2.4: *Service Architecture for Photons*

As illustrated in Figure 2.4, Photons 1.0 [1] modifies the initialization handler of the proxy. In addition to registering main function, a modified class loader performs bytecode manipulation at the same time, separating private states in application code. When multiple same invocations are coming simultaneously, Photons 1.0 keeps pool of threads and dispatches different threads for different invocations. While all invocations have their thread-local invocation stack, all objects are allocated inside a common runtime heap space. As we mentioned before, this enables convenient state sharing among different invocations. To support this, Photons 1.0 uses a global Map data structure and assigns unique identifiers to each thread. These two parameters have been added to the main function signature [Pho]:

```
public static JsonObject main(JsonObject args, Map<String, Object> globals, int id);
```

[1] Since we are using same idea of collocation from Photons, we also refer original Photons work as Photons 1.0 interchangeably.

Using this key-value store, application state, such as image classifier object, can be stored into this map structure and retrieved by each invocation running inside the current runtime. We denote the modified proxy used by Photons as *photons proxy*.

2.2.2 Problems of Photons

Photons' design is clean and easy to implement. However, there are some problems with the current solution. Firstly, the correctness issues: bytecode manipulation is fallible to errors, making debugging difficult from source code level. Besides, using a modified class loader will trigger problems if application code refers libraries containing reflective operations. Such operations are not aware for modified class loader before application being executed and need manually configuration in Photons' framework code. Classes invoking reflections need to be delegated to modified class loader, such that they can be properly loaded when they are accessed. Second, the performance issue.

Using common heap space makes it easy for state sharing. However, it adds a lot burden to the runtime garbage collector. Objects from all invocations would be allocated onto the same heap. Accumulated non-reachable objects would make full garbage collection pause longer, which in turn increase the invocation response latency. At the same time, Java runtime, like HotSpot™ VM, uses Just-In-Time compiler to profile code execution statics and perform optimization via compiling hot code snippets down to native code. This extra work consumes CPU resources and may add response time latency as well.

Last, there are still some issues that Photons have not addressed. On the one hand, although only private state memory footprint needs to be allocated when concurrent invocations are coming, the "offset" memory footprint of runtime and libraries is still non-neglectable. On the other hand, startup time of JVM is also very high. To solve the above problems, we get inspiration from a newly developed Java Virtual Machine, the Graal Virtual Machine (GraalVM), especially the Ahead-of-Time compilation technology it supplies, the GraalVM Native Image.

2.3 Graal Virtual Machine

GraalVM consists of three main components: Graal compiler, Truffle framework and Native Image. Graal compiler is a high-performance Just-In-Time (JIT) compiler developed in Java, aiming to replace C2 compiler in Java HotSpot™ VM against Java-level JVM Compiler Interface (JVMCI) proposed in JEP243 [JEP]. Graal compiler integrates many innovative compiler optimization technologies and can be further optimized and transformed to a more efficient version of itself via JIT compilation.

On the other hand, to help other programming language leveraging advanced runtime technologies, Würthinger et al. [WWW+13] proposed the Truffle Language Implementation Framework (Truffle). The architecture of Truffle and GraalVM compiler can be found in Figure 2.5. Truffle is a library that supports implementing language interpreters for self-modifying Abstract Syntax Trees (AST). They modify Java HotSpot™ VM with Graal compiler, which uses speculative assumptions and deoptimization to produce efficient machine code in an adaptive manner.

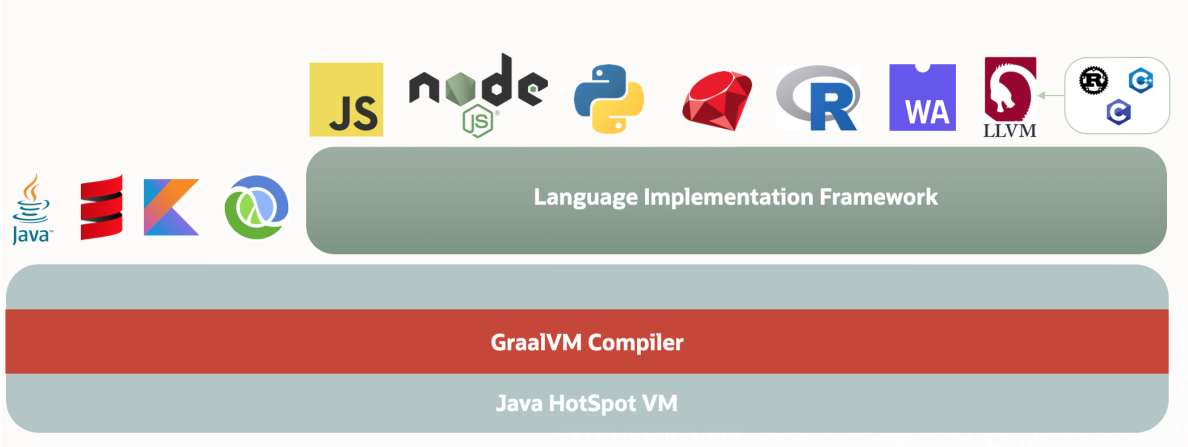

Figure 2.5: *GraalVM Architecture. [Int]*

Intermediate representation can be optimized and executed by the GraalVM.

Lastly, normal JVMs are suffering from slow start and high memory footprint. This comes especially problematic under serverless and microservice context, where runtime instances have to be frequently started up and memory usage is critical. Based on this, Wimmer et al. developed GraalVM Native Image to build Java application into standalone executable [WSH+19], aiming for faster star up and lower memory footprint.

2.3.1 GraalVM Native Image

Unlike traditional JVMs with JIT compilation, GraalVM Native Image transforms the compiled bytecode into a standalone executable (called a native image), which can be directly executed on the target platform. GraalVM Native Image does not have a JIT compiler, so it does not need the fully fledged JVM either. Instead, SubstrateVM would be integrated into Native Image executable, providing necessary subset of JVM functionalities such as memory management and thread scheduling [sub].

Native Image uses iterative points-to analysis [Hin01] [Ryd03] [SB15], heap snapshotting [Ung95] and Ahead-of-time compilation [PTB+97] to build the image. In the first step, Native Image builder performs the iterative points-to analysis, trying to find all reachable code and classes during running time by executing the application. Collected reachable classes in the current iteration will be start points of analysis in the next iteration. Points-to analysis would continue until there is no additional reachable class can be found. After the analysis reached convergence, all reachable safe classes inside heap, including the native image runtime, would be snapshotted into *Image Heap*, which is part of native image executable and directly accessible after native image starts [Claa]. This shifts parts of class loading and initialization to image build time, speed up executable start up. On the other hand, memory footprint of executable would be significant smaller, since only analysed reachable classes and code snippets would be included into native image. An AOT compiler, modified from the decoupled Graal compiler will then take result of points-to analysis as input, producing highly optimized machine code ahead of time. Modified AOT compiler performs many compilation optimizations like constant folding,

inline expansion, partial escape analysis etc. Analysis result would also help AOT compiler to determine which objects will not be modified during run time, even they are not declared as *final*. This will assist AOT compiler producing more efficient machine code. [WSH+19]

After executable starts up, Image Heap can be treated as part of normal Java heap. Initialized classes inside Image Heap will not be checked for initialization during run time, which reduce a lot runtime overhead. However, not all classes can be initialized during image build time. Methods that calling native code will not be considered as safe methods, since native code is not analysed by native image. For example, objects keeping TCP connections will not be initialized into Image Heap, as this kind of native resources would change during application run time. In addition, those virtual methods invocations that can not be reduced to specific single target, would be treated by native image as unsafe as well. Many dynamic dispatches can not be simplified before application execution and native image will not do safety analysis for every possible implementation in order to reduce analysis problem set size. [Claa]

The goal of GraalVM Native Image is to make Java application start up fast with low memory footprint. To achieve this, Native Image make a closed-world assumption, i.e. all possible classes need to be known during image build time. However, features like dynamic class loading, Java Native Interface (JNI), reflection, serializations, Dynamic Proxy etc. bring dynamic flexibility for the programmer together with uncertainty for the AOT compiler. Under the restriction of closed-world assumption, all accessible classes during run time need to be configured during image build time. [NIO] Besides manual configurations, Native Image supplies a trace agent [Tra], which can track all aforementioned accesses and generate necessary configurations. Before image building, users can run their applications on normal JVM with trace agent, generate necessary configurations, then build their applications into native image. Since cloud users need to profile their applications resource usage before deployment anyway, this automatic configuration theoretically will not add further complexity for users to deploy their application. Developers do not need to figure out reflections by themselves and add delegation code manually as they are using Photons.

Without run time profiling, AOT compilation can hardly produce the *best possible* JIT-compiled machine code, even though it may take more time to perform compilation optimizations. To further improve the performance of native image, GraalVM also supplies Profile-guided Optimizations (PGO). Users can run their applications on normal JVMs, collecting workloads profiles. Based on these statistics, native image can perform workload-specific optimization. So long as real workloads conform to that from profiling stage, optimized native image can improve execution speed. If the workloads are different than that being collected earlier, native image would have worse performance. Native Image also supplied Compressed References, "using 32-bit references to Java objects on 64-bit architectures" [Wim]. This can further reduce the memory footprint of native image.

Even with PGO, Native Image can only approximate the peak performance of a fully fledged JVM equipped with JIT compiler. By choosing Native Image, we are trading memory footprint and fast startup for peak throughput.

2.3.2 Native Image Isolate

GraalVM Native Image is fit for cloud native applications because of its low memory footprint and fast start up. We also want to collocate different invocations inside common runtime to leverage the benefits of Photons. In Photons, bytecode manipulation was applied for separating private state for different invocations. GraalVM Native Image supplies another layer of abstraction, called Isolates, providing runtime-level isolation, which would be a good match for this use case.

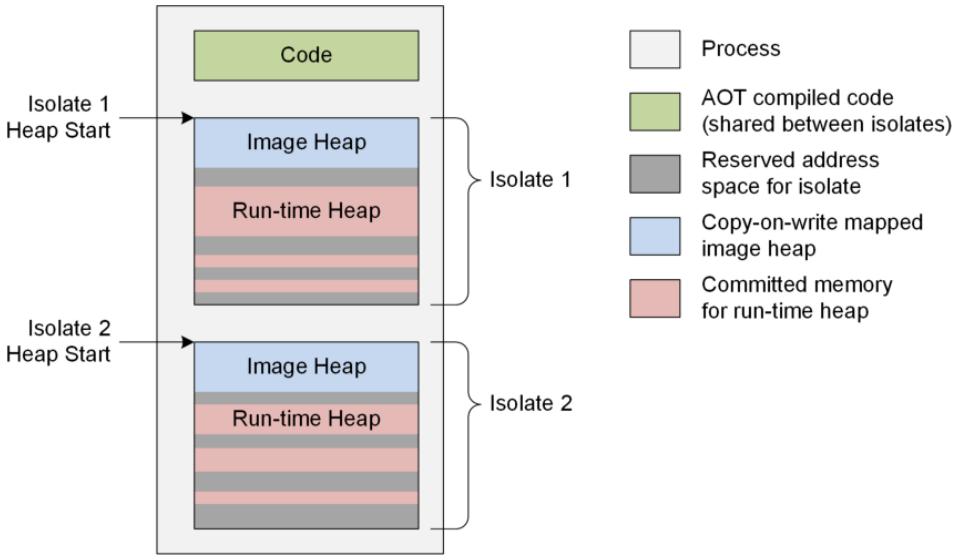

Figure 2.6: *GraalVM Native Image Isolate Memory Layout. [Wim]*

Isolates are lightweight VM instances based on SubstrateVM. As illustrated in Figure 2.6, each isolate has its own heap space. AOT compiled code are immutable and would be shared by all isolates. Image Heap with all initialized objects during building time would be shared by newly produced isolates under a Copy-on-write (COW) manner. Modification will be done in the copied version of original resource and unmodified parts will be directly referenced. Using this manner, creation of new Isolate can be done with high speed and minimal memory overhead [Wim]. Under COW, static mutable fields would have different instances in different Isolates, i.e. they will be copied to isolate-local when they are modified. Similarly, static initializers would also be executed inside each Isolate separately. When we run different invocations inside different Isolates, private state separation can be automatically achieved.

Since Isolate is lightweight VM instance, garbage collection is performed inside different Isolates independently. Comparing to common heap space for all invocations back to Photons, distributed manner of GC would have benefits for shorter GC pause and in turn better response time latency[WGW+11]. In additional to that, fully separated heap spaces provide security support as well, avoiding possible erroneous access from different invocations (even from different users). After invocation finishes, Isolate can be torn down, releasing all memory it allocated back to operating systems, which is much faster than garbage collection procedure. [WSH+19]

Native Image Isolate Java API

In this part, we want to combine the Native Image Java API to elaborate Isolate life cycle. Native Image Isolate supplies both Java and C API, while we only focus on Java API here. In Native Image Java API, there are mainly two types being used for Isolate management, i.e. *Isolate* and *IsolateThread*. Although they are declared as Java interface, they are *PointerBase* type, which is not Java objects [Wim]. These types are more similar to pointers in C/C++, where *Isolate* points to runtime data structure for an Isolate and *IsolateThread* points to the runtime data structure for a thread, which is associated to some Isolate [NIJ]. Different threads attaching to the same Isolate will have different *ThreadIsolate* values, while the *Isolate* value for all attached threads would be the same. In other words, *Isolate* is the main descriptor of one Isolate and one will have the full access to that Isolate using this value [Wim]. Although these types are similar to pointers in C/C++, there is no specification of the pointed data structure, so it is not allowed to dereference the underlying data structure using these values.

The following code snippet, inspired by [Wim], illustrates the typical life cycle of an Isolate. Method *run* is executed in the default Isolate and *stepIntoIsolate* is executed in the newly produced Isolate (denoted as working Isolate).

```
1  public static void run(int size){
2      IsolateThread processContext = Isolates.createIsolate(
3                                          Isolates.CreateIsolateParameters.getDefault()
4                                      );
5      stepIntoIsolate(processContext,size);
6      Isolates.tearDownIsolate(processContext);
7  }
8
9  @CEntryPoint
10 private static void stepIntoIsolate(@CEntryPoint.IsolateThreadContext
11                                       IsolateThread processContext,
12                                       int size){
13     dummy_workload(size);
14 }
```

In line 2-4, we create a new Isolate using default parameters and attach current thread to that Isolate. The returned *IsolateThread* points to the associated thread-local structure for working Isolate. At this point, we are still inside the default Isolate. Working Isolate is inactive for the current thread, though we have attached to it. The actual isolate-transition happens when we call a method denoted by *@CEntryPoint*.

As its name indicates, methods annotated by *@CEntryPoint* work as VM entry-point. They should be static and would be called as a C function. Exactly one parameter of this method is supposed to be the target execution context, which can either be *IsolateThread* or *Isolate* with corresponding annotation [NIJ]. When we invoke entry-point method, the current thread first leaves the default Isolate. At this point, both Isolates are inactive, until the thread enters and activates the target Isolate. This ensures that there is at most one active Isolate for a single thread and no thread can access object reference across different Isolates. Inside entry-point function, we can perform both native function call and Java method invocation. When entry-point function returns, we go back to the default (callee) Isolate by experiencing the same states transition.

If we want to discard an Isolate, we may tear it down as shown in line 6. Any attached thread may tear down the associated Isolate. By calling tear down, current thread sends notification

to all other attached threads and wait until all of them have detached. It's not possible to force tear down an Isolate while other threads are still attached, even if they are not running. Possible resources hold by other threads need to be released and therefore tearing down an Isolate with multiple attached threads needs to be carefully organized. In our case, if there is no other thread attaching to working Isolate after entry point function returns, we can tear down the working Isolate, releasing all memory it has allocated back to the operating system.

In additional to target execution context, entry-point function can also take other parameters. Those parameters can only be primitive Java values, enum values and word values (such as *IsolateThread* and *Isolate*). Since one thread can not access two different heap spaces simultaneously and therefore by no means access objects in other heap space via reference. Different heap spaces supply high quality isolation, also makes it impossible to have a global object share [WSH+19]. This adds additional overhead for arguments passing between Isolates. We study this overhead in the next chapter.

Using Native Image as runtime in serverless context, we can leverage its high startup speed and low memory footprint. Lower start up time speeds up the scale-out procedure, adding system elasticity. It can also reduce response latency to improve service quality. By using Isolate to execute concurrent invocations, we can benefit from its execution safety and efficient independent memory management. Distributed garbage collection would speed up memory-intensive applications while keeping the benefits of using runtime sharing. At the same time, separated heap spaces also add additional overhead for message passing and make it difficult to share state, since it is impossible to pass object references between isolates. In this project, we design and implement Native Image Isolate proxy, evaluating its performance in different dimensions. In addition to that, we also use cluster-wide simulation to show the design trade-off by using our approach under realistic serverless production trace.

System Design and Implementation

In the previous chapter, we discuss the motivation to integrate GraalVM Native Image Isolates into a serverless platform. In this chapter, we first elaborate the modified system structure using this technology. Based on Native Image Java API, we implement two mini-benchmark applications. Using the first application, we determine the overhead to create and tear down an Isolate. In the second one, we measure the speed of argument passing between different Isolates. Lastly, we give our solutions for object caching and state sharing for the Native Image Isolate proxy.

3.1 Serverless with Native Image

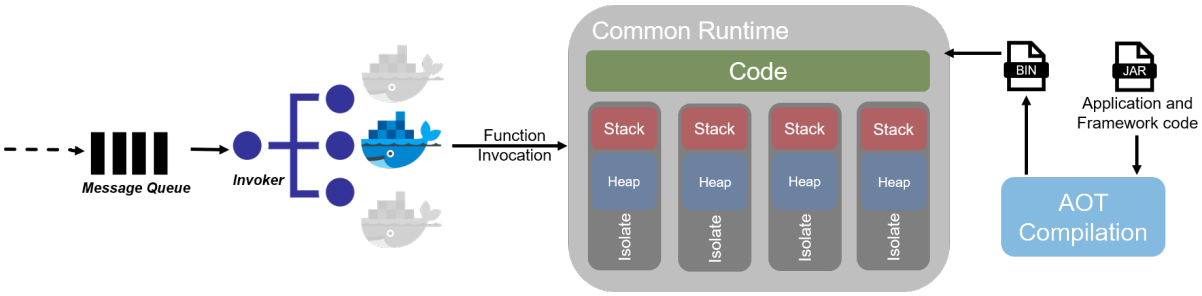

Figure 3.1: *Service Architecture for Native Image Isolate*

To integrate GraalVM Native Image to this system, we need to replace the OpenWhisk Java runtime with our customized standalone executable. Since Native Image uses AOT compilation, proxy code and application code need to be compiled together. In this case, we can invoke main function directly instead of using class loader to invoke main function via reflective operation.

Code for initialization, including class loader and initialization handler, can be discarded in our GraalVM Native Image Isolate Proxy (denoted as isolate proxy).

During action initialization, cloud users need to perform the same procedure as by regular OpenWhisk or Photons 1.0, i.e. compile and run their application locally on normal JVM to profile CPU and memory requirement of it. In addition to that, they also need to launch the proxy using Native Image trace agent to collect configurations for native image builder. GraalVM tracing agent attaches to JVM, recording Java application behavior and collecting usage of reflections, JNI, proxy, resource and serialization. It generates configuration files for native image builder.

Proxy needs to be tested by realistic invocations, such that all dynamic behavior can be captured by the tracing agent. If the closed-world assumption of native-image builder is violated, i.e. native image is going to instantiate any class during run time that it has not seen before, exception would be then triggered. With all the configuration files automatically generated by tracing agent, developers can use *native-image* tool to build the application into the standalone executable, without further manual configuration.

Since native image is going to be standalone, all libraries and project dependencies need to be included into jar file (a fat jar). *native-image* would perform analysis and optimization, building Java application into executable binary file. This procedure can be highly memory and CPU-consuming. The produced native image can be directly launched on target platform. We copy it into Docker image and finish the action deployment. As depicted in Figure 3.1, isolate proxy also deal with concurrent invocations using different threads as in Photons 1.0. Each thread attaches to an Isolate and finishes the invocation inside that Isolate. Different to Photons, each Isolate has its own heap space, while code and Image Heap are still be shared. Instead of using bytecode manipulation, execution in different Isolates would separate private state of each invocation automatically, which guarantees the correctness of each invocation. Garbage collection would also be performed separately for each Isolate, possibly reducing GC pause.

3.2 Overhead using GraalVM Native Image Isolate

Since using Isolate introduces additional complexity to the system, the performance of isolate is an important factor that influences decision making for our system design. Therefore, before we implement isolate proxy using GraalVM Native Image, we first study the overhead to use an Isolate. Native Image Java API creates a default Isolate for standalone executable before the main function is invoked. If we want to use another Isolate to run our code, we need to explicitly create one.

We implement our first mini-benchmark application where we create and tear down an Isolate. Since we want to study Isolate creation and tear-down overhead in relation to amount of memory being allocated inside it, we pass memory *size* as a parameter to the working Isolate. Inside the new Isolate, we allocate different size of Integer ArrayList, initialize ArrayList using a for-loop and calculate hash of the ArrayList. We measure the wall-time for creation and tear-down by setting checkpoints before and after the corresponding statements. We print the measured timestamps out and post-process the measurements using extra Python script. The measurement results are shown in the figures below.

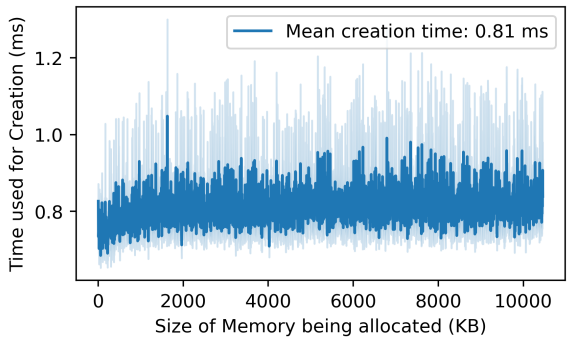

Figure 3.2: *Creation Time of new Isolate in relation to different amount of memory allocated inside that Isolate*

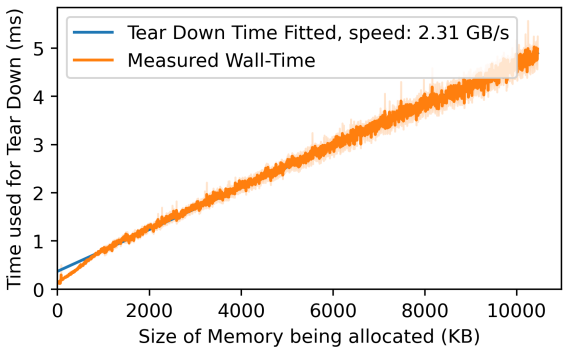

Figure 3.3: *Tear Down Time of new Isolate in relation to different amount of memory allocated inside that Isolate*

From the Figure 3.2, we can find that time used to create a new Isolate is relatively stable on our server machine [1] regardless the amount of allocated memory inside it. Creating an Isolate needs to map the Image Heap to the new VM instance. Image Heaps are shared under Copy-on-write manner. Code section is directly referenced. On our server machine, creation of a new Isolate costs around 0.8ms. On the other side, as depicted in Figure 3.3, the more memory being allocated inside the Isolate, the more time has to be taken to tear it down. On our server machine, the equivalent speed to release memory back to operating system is around 2.3 GB per second. When we dive into the source code for tear-down inside Substrate VM, we find a for-loop releasing all chunks have been allocated for that Isolate. This explains the proportional relationship between tear-down time and allocated memory. Nevertheless, using tear-down would be much faster than normal garbage collections inside Isolate, since garbage collector needs examine which objects can be discarded.

3.3 Message Passing between Isolates

Since we want to invoke a serverless function to run inside an isolate, we need to pass corresponding arguments to the entry-point method. However, the entry-point method cannot take object reference as parameters. How can we pass Java object as arguments to the working Isolate? Static class *CTypeConversion* of Native Image Java API supplies multiple static methods to enable data copying from one heap space to another. Object need to be serialized while being copied between different isolates.We take String-passing as an example and use the following code snippet inspired by [Wim] to elaborate how this works.

```
1  public static void run(String input){
2      IsolateThread processContext = ...; //initialize as before
3      ObjectHandle inputHandle = copyString(processContext,input);
4      stepIntoIsolate(processContext,defaultContext,inputHandle);
5      Isolates.tearDownIsolate(processContext);
6  }
7
8  @CEntryPoint
```

[1] Our server machine is equipped with AMD OpteronTM 6174 (4 sockets x 12 cores) 2.2GHz and 126GB Memory. All local experiments in this project are running on this machine.

```
9  private static void stepIntoIsolate(@CEntryPoint.IsolateThreadContext
10                                      IsolateThread processContext,
11                                      ObjectHandle inputHandle){
12      String input = ObjectHandles.getGlobal().get(inputHandle);
13      ObjectHandles.getGlobal().destroy(inputHandle);
14      //do jobs with input String
15  }
16
17  private static ObjectHandle copyString(IsolateThread targetContext, String sourceString) {
18      try (CTypeConversion.CCharPointerHolder cStringHolder = CTypeConversion.toCString(sourceString)){
19          return copyString(targetContext, cStringHolder.get());
20      }
21  }
22
23  @CEntryPoint
24  private static ObjectHandle copyString(@CEntryPoint.IsolateThreadContext
25                                         IsolateThread processContext,
26                                         CCharPointer cString) {
27      String targetString = CTypeConversion.toJavaString(cString);
28      return ObjectHandles.getGlobal().create(targetString);
29  }
```

String copying has been done by the two versions of *copyString* functions. In the Java function *copyString* (line 17-21), we extract C pointer of the Java String and pass it to a *CCharPointerHolder*. Inside try-block, we invoke another *copyString* method, where we first time enter the working Isolate.

Inside new Isolate, we cannot dereference String pointer as in an unsafe language such as C/C++. Instead, we have to copy an instance of the original object to the current heap space. During String copying, Substrate VM allocates a new byte array with the same length in the target Isolate and then copy data from the original heap space byte-by-byte. The pointer of new String is registered into a global store, called *ObjectHandles* in line 28. Objects registered inside global *ObjectHandles* are referred by different VM instances. Therefore, garbage collector would not collect those objects until they are explicitly *destroyed* as shown in line 17.

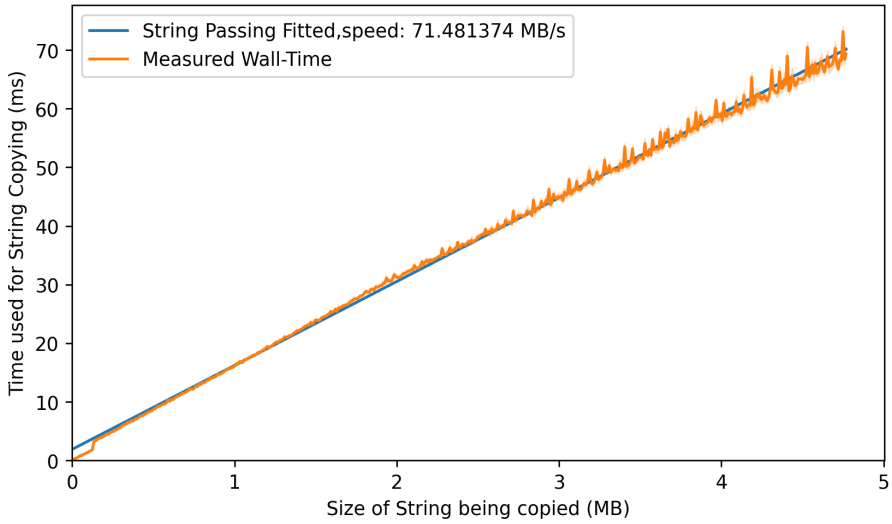

Figure 3.4: *Overhead for String Copying across Isolates, wall-time for copyString java method measured using different length of String. Orange line denotes measure wall time for argument passing, while blue line shows the fitted result using linear regression.*

Since byte-by-byte copying is implemented using a for-loop, time used to pass String is in

proportional to argument size. As illustrated in Figure 3.4, speed to copy arguments from one Isolate to another is only around 71.5 MB/s on our server machine.

In addition to String, *CTypeConversion* also provides helper methods for copying objects of type *Boolean* and *ByteBuffer*. For example, if some objects of other types need to be passed into an Isolate, they have to be converted into String or ByteBuffer at the first stage (via Serialization). Class meta data can then be copied using *CTypeConversion* low-level API. Deserialization has to be performed to retrieve the objects in the new Isolate. Serialization and deserialization add additional overhead to arguments passing. To meet the closed-world assumption of native image builder, serialization configurations has to be supplied to native-image during image build time as well. We need to keep this overhead in mind, designing invocation routine carefully to avoid copying large objects across Isolates.

3.4 Solution for Object Caching

The original design of Isolates assumes that execution inside Isolate are short-lived and there would not be too much memory being allocated inside each Isolate. In optimal cases, there is no garbage collection until an Isolate is torn down [Wim]. As we have shown in the previous section, releasing memory by tearing down Isolate is more efficient than relying on garbage collector. However, not all applications are suitable for this paradigm. For example, many applications interact with storage systems, such as a database. Establishing a database connection is very expensive such that many client-applications are applying strategy called connections pooling. Once established, database connections are kept alive and reused for further coming requests. In Photons, we can use the global object store for connection caching. Database connections can be cached per thread and kept alive after invocation finishes. Successive invocations reuse the cached connection, reducing response latency.

If we use isolate proxy to implement the above application, we need to establish the database connection inside the working Isolate. If we tear down the working Isolate after invocation finishes, no objects produced by that Isolate will remain, unless we copy it to the default Isolate, which seems to be very expensive as well. Without connection pooling, we have to suffer the overhead of creating a connection for each invocation, which can be well illustrated in the following figure.

In Figure 3.5, we break execution time into different parts of invocation in millisecond for both proxies. The stacked bar plot shows the results of first five serial invocations after proxy starts up, where we annotate different proxies using different hue palettes. Serverless function being invoked here is a File Hashing application, where each invocation downloads a 2 MB file from MinIO Server and calculates hash of file content. In the File Hashing application, we cache two objects per thread in Photons: the database connection and the byte buffer. From the plot we cannotice that Native Image starts up much faster than normal JVM. It takes much less time to establish the database connections as well. However, since we tear down the Isolate, we have to establish a new connection (denoted as *DefaultClient*, takes 13.32 ms) and allocate a new byte buffer (denoted as *getBuf*, takes 1.44 ms) for each of invocation. Total overhead using Isolate, including create Isolate, establish connection, allocate buffer, and tear down the Isolate even costs longer time than that for workload processing. Without caching, isolate proxy double the

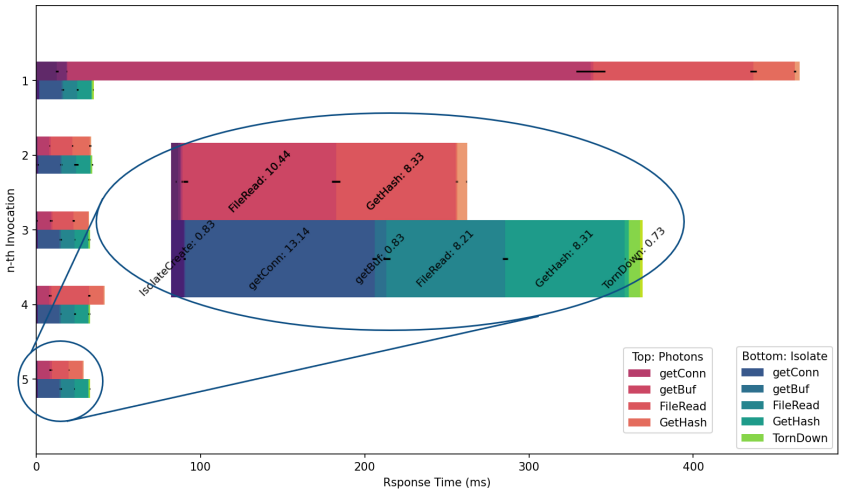

Figure 3.5: *Overhead for isolate proxy without object caching. The plot shows the execution time comparison between photons proxy and isolate proxy using File Hashing workload. Execution time of the first five invocations after proxy starts has been broken into different main components, where bottom stacked bars denote the results for isolate proxy.*

response time, reducing quality of service by a lot.

To enable caching and reduce overhead for each invocation, we decide to delay the tear-down of Isolates and keep warm Isolates in an object pool. In isolate proxy, we also use pool of threads to deal with different concurrent invocations. From the invocations' perspective, each thread would correspond to an Isolate. However, instead of binding thread and Isolate one to one, we keep a separated Isolate pool inside the default Isolate. Keeping a decoupled Isolate Pool supplies huge flexibility for independent resource management with neglectable minor overhead. As illustrated in Figure 3.6, after receiving an invocation, thread tries to borrow an Isolate from *Isolate Pool*. If there is no Isolate available, Isolate Pool would create a new Isolate using the Java API we introduced earlier and attach the current thread to that Isolate.

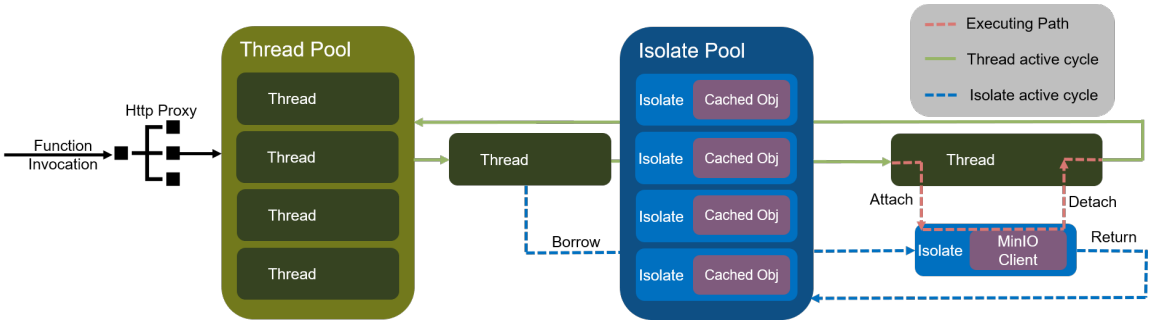

Figure 3.6: *Caching per Isolate via Isolate Pool*

Back to the mini-benchmark, we deal with Isolate mainly using *IsolateThread*. However, in this case, we need to pool the corresponding *Isolate* pointer, since *Isolate* value is the only descriptor supplying the full control of that VM instance and one Isolate may be used by different threads. Besides, the *Isolate* is not Java object, so we need pack it into wrapper class and implement boxing and unboxing interfaces. Once the thread has borrowed an Isolate, it attaches itself

to it, passing arguments to entry-point methods and execute the invocation inside that Isolate. Objects like buffer or database connection can be cached inside that Isolate e.g. using static fields. After the execution finishes, the thread would detach from the Isolate and send back the response. Warm Isolate would be kept alive and returned back to Isolate Pool, waiting for further invocations.

Besides object pooling, we can perform asynchronous eviction using a daemon thread. If an Isolate has been idle for a long time, daemon thread would attach to that idle Isolate and tear it down, release unused memory back to the operating system. Similar to warm containers kept by cloud providers, independent and adaptive management of Isolate Pool supplies additional elasticity with finer granularity. Transforming from "caching per thread" to "caching per Isolate" reduces the overhead of creating object while keeping efficient memory management via distributed garbage collection and adaptive eviction.

3.5 Solution for State Sharing

Another issue about isolate proxy is the difficulty to achieve state sharing, e.g. to share machine learning model among different Isolates. During implementing different workloads, we also find that invocation to native libraries (JNI) are encountering similar problems as well. For large static immutable objects, GraalVM Native Image allows to initialize and build the singleton of them into Image Heap during image build time. Using global key-value store called *ImageSingletons*, registered objects can be directly accessed during run time. Using this technique, Native Image is able to share large immutable object among different Isolates using Image Heap.

Since both JNI and machine learning model are invoking native codes, they are considered as unsafe methods and can not be initialized by native image builder. Therefore, instances of machine learning model can not be merged into Image Heap during image build time. To share machine learning model singleton, we decide to keep a shared Isolate that can be accessed by all worker Isolates to enable state sharing. Similar to Java Remote Method Invocation [Jav], we treat shared Isolate as an "remote server". Shared resources are only available inside shared Isolate. Worker Isolates send requests to the shared Isolate and get result back via argument passing we've introduced in 3.3.

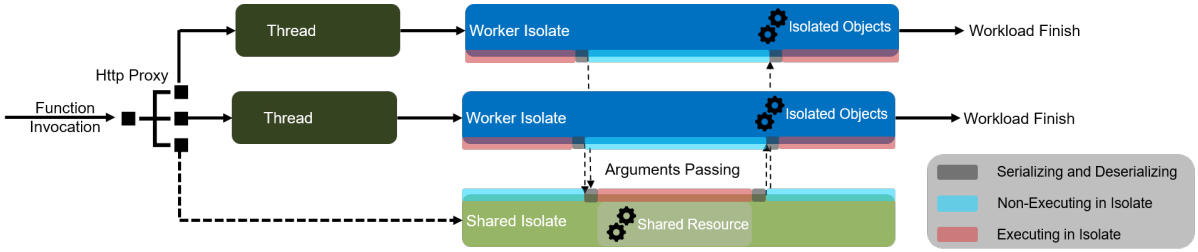

Figure 3.7: *State Sharing via global shared Isolate*

As illustrated in Figure 3.7, invocations are still processed by different worker Isolates. Invocation-dependent private states would be automatically separated in different worker Isolates. To execute code snippets using JNI or shared state, we detour the current execution around the shared

Isolate. We serialize the necessary arguments and attach current thread to the shared Isolate, retrieve arguments and invoke native interfaces (or use shared state) in the shared Isolate. After the critical part of codes being executed, we pass the result back to worker Isolate and detach from the shared Isolate. Under proper separation, arguments being passed can be kept minimal and the overhead is neglectable.

Only code snippets invoking JNI or shared state are supposed to be executed in the shared Isolate. Code snippets that are executed inside shared Isolate, either Java code or native invocations, are supposed to be thread safe. And it is developer' responsibility to avoid race condition inside shared context, e.g., through synchronization. Shared isolate separate shared execution code from independent executed code explicitly, easy for trouble shotting but also requiring extra data separation and serialization. We believe that in near feature, GraalVM Native Image would support JNI sharing globally and there would be better ways to do state sharing among different Isolates without much overhead.

Experimental Evaluation

In this chapter, we evaluate the performance of the isolate proxy using different workloads. We draw a comparison between the isolate proxy and the photons proxy, which runs on a normal JVM. Four different workloads are introduced for evaluation purposes. We start by introducing different workloads briefly, and explaining the design philosophy we mentioned in the previous chapter. We profile the minimal memory usage of workloads under different concurrency levels, while they are not encountering performance degradation. Then, we run experiments to evaluate different performance dimensions for both proxies inside isolated execution environment (docker container).

4.1 Workloads for Machine Local Evaluation

All experiments in this chapter are running on our server machine as described in Section 3.2. Therefore, we also denote the experiments as machine local evaluation. The four different workloads we study are: File Hashing, Image Classification, REST Requests, and a naive Memory Allocator. We take the first three workloads from Photons [DBSA20]. In addition, we introduce a new workload, memory allocator, to simulate a memory-intensive application. We introduce those workloads and point out the adoption we make for isolate proxy. In the first step, we need to profile memory requirement of each workload for both proxies. To make fair comparison, we deactivate the compressed reference explicitly for Photons and use the same serial GC algorithm for both proxies as well.

4.1.1 File Hashing

For the first workload, File Hashing, each invocation downloads a 2MB file from a local MinIO [Min] server and calculates the hash of its content. MinIO is an object storage server, which is

compatible to AWS S3 API [AWSa]. MinIO supplies a Java Client API to establish connections, and uses a thread pool to improve efficiency. The thread pool used by the MinIO client keeps a daemon thread alive, managing all the threads inside the pool. The daemon thread is attached to the working Isolate and we cannot deactivate it using the exposed interfaces. This prevents the adaptive eviction policy applied by our Isolate pool. We therefore implement our customized version of MinIO connection client without using connection pool and apply it for both version of proxies.

Besides the MinIO connection that is cached and reused, a byte buffer is also cached to store downloaded file stream. Those two objects are cached per thread inside global object store in Photons, and cached per Isolate using static fields in the isolate proxy.

In order to profile minimal memory usage for both version of proxies while keeping their peak performance, we need to carefully determine memory usage limit for each workload. In the first step, we launch proxy inside docker without memory limit and measure its peak throughput using consecutive invocations generated by Apache HTTP server benchmarking tool (ab) [ab] after a warm-up phase. Then, we shrink the size of memory limit until we observe performance degradation. We do this measurement by adding concurrent invocations, determining the minimal memory needed for each concurrency factor.

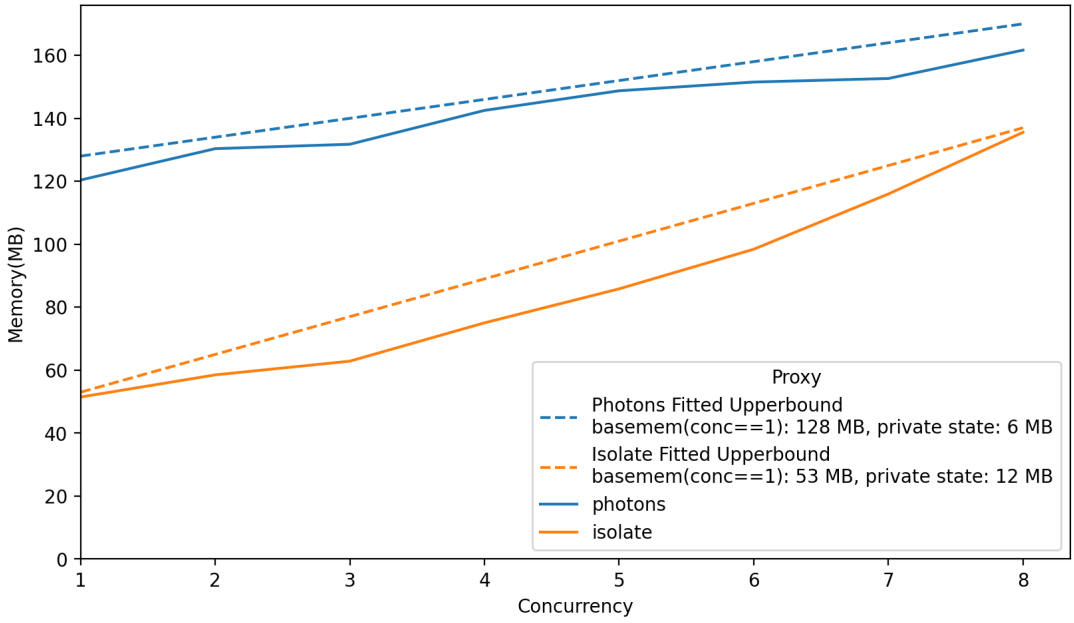

Figure 4.1: *Fitted memory usage profiles of both proxies for file hashing workload.*

For each workload, we split the memory limit into two components: base memory and memory increment. Base memory denotes the minimal necessary memory needed by proxy under serial invocations without scarifying the peak performance. Memory increment is the memory we need to add for each additional concurrent invocation during scaling out. We model the final memory limit for each workload using linear model using the following formula:

$$MemoryLimit(n) = BaseMemory + (n-1) \times MemoryIncrement$$

where n is the concurrency factor. Using linear regression, we determine the both factors as

depicted in Figure 4.1. Based on the measurements points, which is denoted in solid line in the plot, we first perform linear regression and take the minimal larger integer value of regression slop as *memory increment*. Then we go through all data points, finding the maximal possible intercept to calculate *base memory*, such that memory limits calculated by the above formula are always greater or equals to the values we have measured. Using this lowest upper bound as memory limit, we can reach a minimal cost regarding to memory usage while keeping minimal contention among different concurrent invocations under collocation.

From the plot we notice that each Isolate needs nearly double the amount of memory for each additional concurrent invocation, though it only requires 41.4% of the base memory to keep service for serial requests. Besides the minor overhead to create a new isolate, arguments and results are copied into each isolate, adding additional memory consumption. To compensate the overhead of argument passing, each isolate needs slightly more memory as well. File Hashing is a typical I/O intensive workload, imitating large-scale batch data processing using high number of concurrent invocations [JSSS+19]. We choose 1 CPU per concurrent invocation, which does not limit the peak performance for this workload.

4.1.2 Image Classification

Image classification represents typical machine learning workload applied in current serverless platforms. As machine learning services are getting more and more popular in cloud usage [IMS18] [AWSc], image classification workloads becomes one of the most basic machine learning applications in the cloud [Clac] [Clab] [Clad]. In this workload, the proxy uses Inception model to perform image classification. Weights of the pre-trained model are downloaded from local MinIO server and loaded into TensorFlow Graph during initialization.

After the model is initialized, each invocation requests an image downloaded from MinIO server as model input. The proxy performs the inference by calling native TensorFlow libraries and by returning the image category back to users. Weights of the classification model are immutable during prediction. Classification model can therefore be shared among different invocations.

While machine learning model is shared via the global object store in Photons, we need to carefully organize the invocation routine in isolate proxy. Although the pre-trained model is immutable large object, calling into native libraries prevents us from initializing all of TensorFlow classes during image build time. As a result, we cannot put the model into Image Heap for sharing across Isolates. On the other hand, Java Abstract Window Toolkit (AWT) uses BufferedImage to store and process images, which uses JNI. Since JNI is currently not shareable across Isolates, we therefore use the shared Isolate as described in Section 3.5 to hold Image Classifier singleton and detour the invocation around the shared Isolate for JNI calls and model access. Under proper separation, arguments passed between Isolates are only short Strings, without adding noticeable overhead.

Since classification is done inside native code, the JVM does not have much control over the memory being used inside the container. On the other hand, native code is not aware of the memory limit of the container, therefore we keep conservative while pushing down the memory limit, such that proxies would not be killed by container daemon. After measuring the memory usage, we get the memory profiles as shown in Figure 4.2. From the plot we can see that

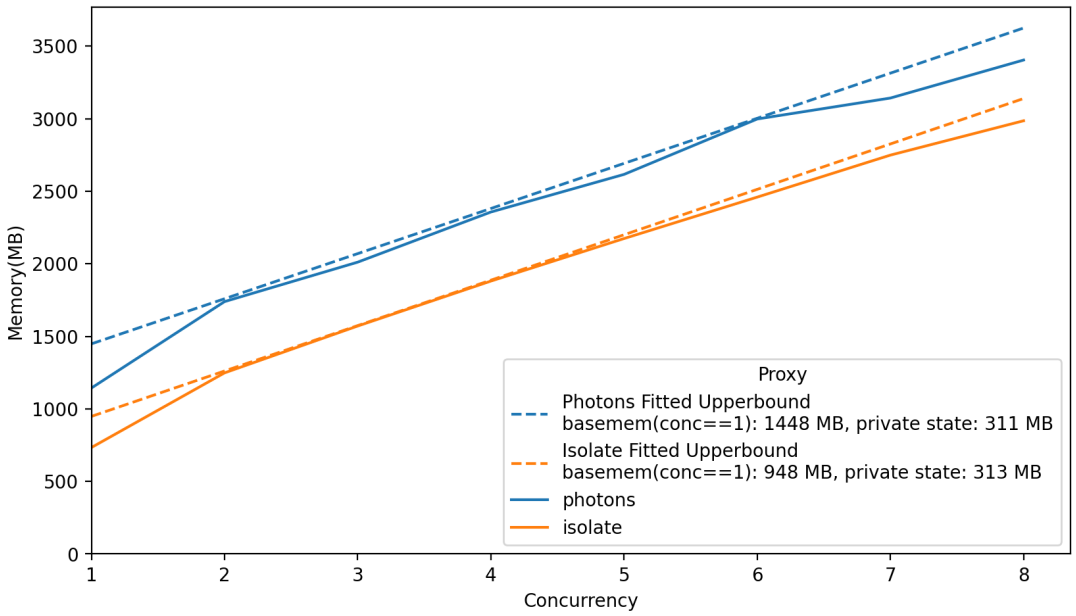

Figure 4.2: *Fitted memory usage profiles of both proxies for image classification workload.*

Native Image reduces base memory from 1.4 GB to 0.93 GB by over 34%. Since memory consumption comes primarily from native code, *memory increment* for both proxies are very close, while detour across different Isolates adds some overhead.

Image classification is typical CPU-intensive workload. Underlying native library is capable to use multi-thread to improve classification speed. We assign 3 CPUs per concurrent invocation due to limitation of our server machine, which also makes this workload CPU-bounded.

4.1.3 REST Request

This workload is a modified version of a microservice benchmark [GZC+19]. It receives HTTP requests from the user and performs a quick interaction with storage systems. In our case, a String is passed and compared with the version stored inside mongoDB [Mon] database. The comparison result is then returned to the user. The execution is very short and requires only few resources.

MongoDB database connections are also expensive to establish and therefore cached for both version of proxies. Similar to MinIO client, mongoDB client uses pool of threads as well. To leverage the adaptive eviction policy supplied by Isolate pool, we need to release all resources hold by the connection client and make sure there is no other thread still attaching to the Isolate before tear-down.

MongoDB client can be explicitly closed, releasing all resources it holds, including all resources and alive threads it keeps. We register a tear-down hook for the working Isolate in this workload, explicitly closing the mongoDB client. Using tear-down hook, idle Isolates in Isolate pool can be cleaned up and torn down, releasing allocated memory and keep proxy memory footprint as small as possible.

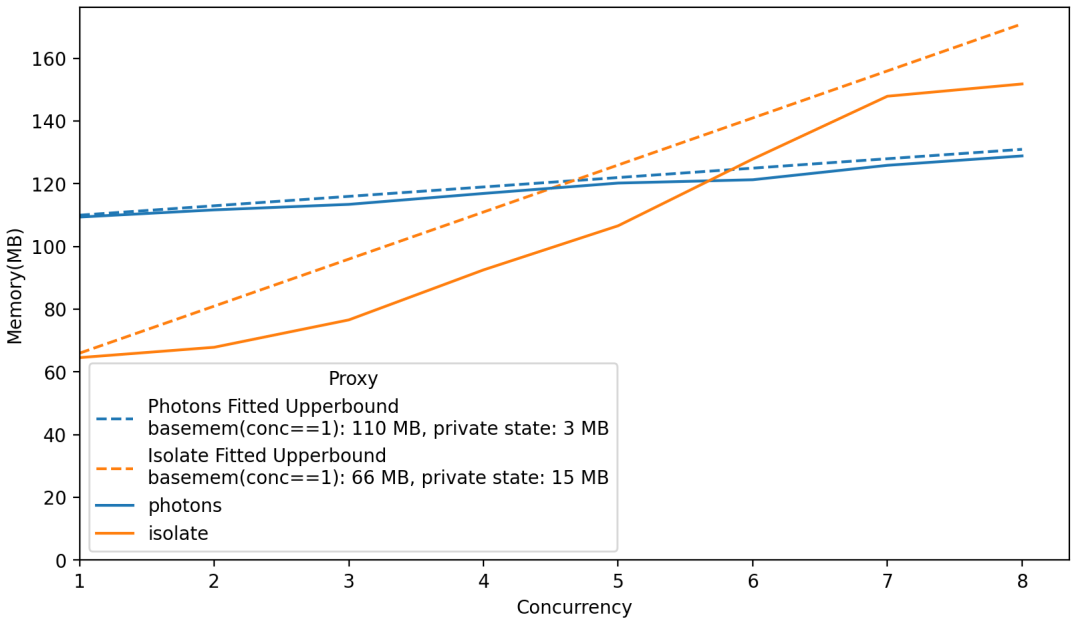

Figure 4.3: *Fitted memory usage profiles of both proxies for REST reqeust workload.*

We measure the memory profiles for this workloads using approach we introduced before. The results are shown in in Figure 4.3. From the plot we can find that isolate proxy requires more footprint increment when there is only very little memory allocated per invocation. Though base memory of isolate proxy is only 60% of photons proxy, higher memory increment makes total memory footprint larger for isolate proxy when there are more than 5 concurrent invocations. Under small memory usage, memory overhead to use Isolate is amplified.

On the other hand, as this workload is executed under 1ms, constant-time overhead of isolates, such as message passing, has a larger weight. This can only be compensated with a larger memory limit for each isolate. For this workload, we assign 1 CPU per concurrent invocation, which makes this workload again an I/O bounded workload.

4.1.4 Naive Memory Allocator

Since the before mentioned workloads are either I/O-bounded or CPU-bounded, we now introduce a simple workload to imitate the memory-intensive serverelss functions such as big data analytics, where large amount of memory is allocated to cache intermediate calculation results. To discard the side-influences of internet connection latency via interacting with database systems, we simply allocate small amount of memory inside each invocation and calculate its hash. Each invocation allocates an 10 MB byte array and register it into a HashMap. By removing it from the HashMap after calculation, the byte array is discarded and handled by runtime garbage collector.

Although we only allocate 10 MB memory per invocation, accumulated memory burden is very high under a batch invocation pattern using Apache *ab*. The base memory and memory increment for both version of proxies are denoted in Figure 4.4. From the plot we can see the

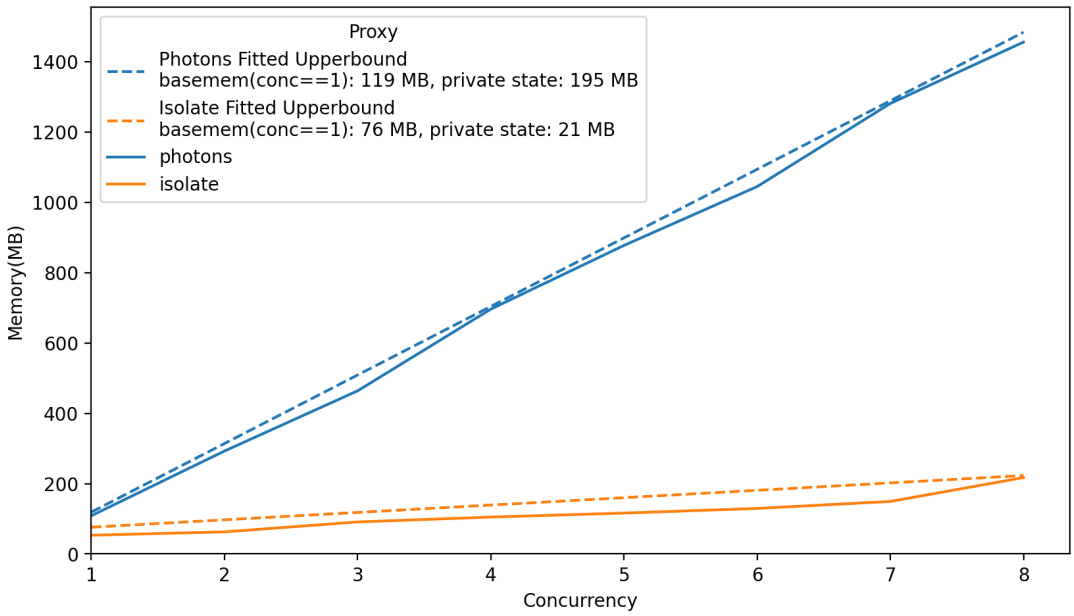

Figure 4.4: *Fitted memory usage profiles of both proxies for memory allocator workload.*

huge gap of *memory increment* between Photons and Isolate. While photons proxy requires 195 MB more memory to collocate a new concurrent invocation, isolate proxy can keep its service quality with just additional 21 MB. We notice that in the serial case, Photons can keep memory management efficiency just as for other workloads. However, once there are multiple concurrent invocations, accumulated non-reachable objects on the common garbage heap increases the difficulty for garbage collector significantly. In contrast, isolate proxy performs independent garbage collection. Only a relatively small heap to be analyzed by each independent garbage collector, which makes GC procedure more efficient.

Note here the *memory increment* of Photons is even larger than its *base memory*, which seems not worthy to still allow collocation. This is because we are using nothing but a hash function to do the calculation inside this workload. In reality, many more components and libraries are added into base memory, where collocation would save memory footprint and reduce cold start. The burden for JVM to collocate large memory is comparable to *memory increment* we showed here.

4.1.5 Workloads Resource Usage Profile

We summarize the resource usage profiles for the four workloads in Table 4.1. From the table we first notice that the base memory of isolate proxy is lower than that of photons proxy. For Image Classification, REST Request and Memory Allocator, the isolate proxy only requires under 65% base memory of Photons. For File Hashing, the isolate proxy reduces memory footprint down to 42%.

On the other hand, we can also see that using Isolate requires more memory increments for some workloads. This becomes more severe especially when there is only a small amount

of memory allocated in each invocation. Regarding the larger memory increment of isolate

	Base Memory (MB)		Memory Increment (MB)		
Workloads	Photons	Isolate	Photons	Isolate	CPU
File Hashing	128	53	6	12	1
Image Classification	1448	948	311	313	3
REST Request	110	66	3	15	1
Memory Allocator	119	76	195	21	1

Table 4.1: *Memory and CPU Usage Profiles for different Workloads.*

proxy, there are mainly two reasons. First, there is minor memory overhead to use Isolate, which is normally around 1 MB. Besides that, different Isolates split the entire heap space into different small cells. For each Isolate, we need to keep some amount of heap space for independent garbage collection. Proper size of Isolate garbage heap allows objects generated by each invocation to be accumulated for some extend and discarded together. Narrow heap space per Isolate increases the garbage collection frequency and can even makes each garbage collection longer. In contrast, Photons allocate all objects on the same garbage heap compactly without much spacing. The separation of whole heap space into exclude Isolates causes the heap space fragmentation, adding memory increment during collocation in exchange for execution security and efficient independent memory management.

In order to reduce the minimal size of heap we have to keep for each Isolate, a better paradigm for Isolate management should not rely on Isolate garbage collector, for example tearing down the Isolate after the work is done. However, even if object caching can be solved among different Isolates, overhead for creation and tear-down would still add up response time latency for invocations that are finished under 1ms like REST Request.

4.2 Cold start and Ramp-up Procedure

One benefit to use Native Image is its fast start up. Not only the time we need to wait until the service available, but also the time to spend before it can achieve peak performance, which is denoted as runtime ramp-up process.

In serverless platform like OpenWhisk, a new docker container is initiated when there is no available warm container for the invocation. This situation is called *cold start*. We measure this cold start time for both version of proxies. To measure this time, we set check points after proxy already starts and measure the wall time before starting the docker container till proxy has been started. As listed in Table 4.2, starting isolate proxy costs, on avereage, 576 ms less compared to photons proxy. Running machine code of AOT executable also reduces the variance of cold startup time by 16.6%.

Besides the faster cold start up, Native Image can achieve its peak performance much faster than Photons as well. We measure this procedure for different workloads. After we start up

	Time until Service Available (s)	
Proxy	mean	standard error
Photons docker	1.749	0.0368
Isolate docker	1.173	0.0307

Table 4.2: *Docker Cold Start Up Time for both version of proxies. Wall-time is measured here from starting docker container to HTTP proxy bas been started.*

the proxy inside docker with corresponding CPU and Memory profiles listed in Table 4.1, we use Apache *ab* to send continuous invocations to the proxy for 150 seconds. We aggregate the average response time per second and calculate the corresponding equivalent throughput. We take typical ramp-up procedure of file hashing workload as an example. The equivalent throughput for the first 140 seconds under concurrency level 8 is shown in Figure 4.5.

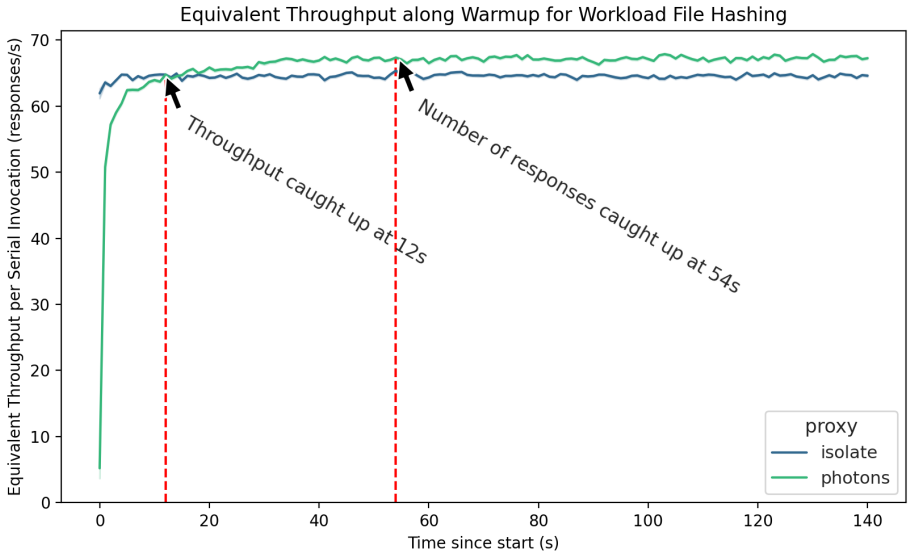

Figure 4.5: *Equivalent Throughput along warm-up procedure for file hashing workload under concurrent level 8. Throughput is calculated by mean response time per second without being multiplied with concurrency factor.*

As a full picture of what has been shown in Figure 3.5, we can clearly see the speed up slope for photons proxy. Isolate proxy not only reduces the first invocation of file hashing from 1.5s down to 33ms. It can also already reach its peak performance since the second invocation of each Isolate, i.e., after the MinIO connection and byte buffer are cached inside it. After very short time, isolate proxy reaches and keeps its peak throughput at around 64 responses per second per concurrent invocation.

On the other hand, Photons suffer from heavy class loading and inefficient bytecode interpretation at the start of its service. As the JIT compiler transforms bytecode into more efficient version, throughput of photons proxy catches up isolate proxy after 12 seconds. In this work-

load, profiles collected during run time makes JIT compiled code more efficient than AOT compiled executable and reaches its peak performance after around 30 seconds. After over 54 seconds, Photons finally catch up the total numbers of responses, while there are already more than 85,000 invocations being processed.

For less frequently invoked functions, the proxy can rarely be warmed up with so many invocations until they are stopped by eviction policy. Once the warm container is shut down, JVM has to experience the whole process of slow ramp-up again for next cold start. In contrast, Native Image can reduce number of slow invocations by a lot. Ideally, only the first invocation of each Isolate is executed slowly due to possible run time initialization. All successive invocations to isolate proxy can already be executed with its peak performance. If containers need to be frequently restarted, which is prevalent in serverless context, Native Image can increase the service quality, reducing response time latency by a lot.

For other workloads, isolate proxy also reduces the ramp-up time significantly. For image classification, model weights are downloaded and loaded during first invocation. Isolate speeds up this procedure from 3s down to 1.5s. The first invocation for REST request, including mongoDB connection client establishment, is executed by isolate proxy 100x faster than photons proxy, i.e., from over 900ms to 9ms.

4.3 Invocation Response Time Latency

Besides tail latency caused by cold starts, we study the response time latency for a warmed-up runtime in this section. Response time latency has always been used to evaluate quality of service and is part of service-level agreement of many serverless computing providers. Besides I/O access and internet connection, complex runtime systems like the JVM may introduce additional latency as well. Both JIT compilation and garbage collection consume computation and memory resources, which may interfere the execution of invocation.

We measure response time latency for different workloads on fully warmed-up containers with proper resource configurations as shown in Table 4.1. Before measurement, we use concurrent invocations to warm up each container for 150s. After that, we measure response time using Apache *ab*. The first plot in Figure 4.6 illustrates the percentage of the requests served within a certain time for file hashing workload, where horizontal axis shows the percentage and vertical axis denotes the corresponding percentile in milliseconds.

In order to discard the latency caused by MinIO server as much as possible, we measured the response time for multiple times and discard the outliers. Average values of measurements are annotated by dashed lines for photons or solid lines for isolate proxy. Variances of measurements are illustrated by the shadow area around those lines. Note that we use uneven scales for horizontal axis and start from 50 to show the inflation of tail-latency more clearly.

From the plot in Figure 4.6 we can see that response time of Photons are shorter than that of isolate proxy for more than 99% of all invocations for each concurrency level separately. JIT compiled code leverages run time profiles and therefore beats the AOT compiled native image. Since the workload is I/O-bounded, photons proxy also has comparable worst-case latency except for serial invocations. The spike for photons under serial invocations is causes by JIT

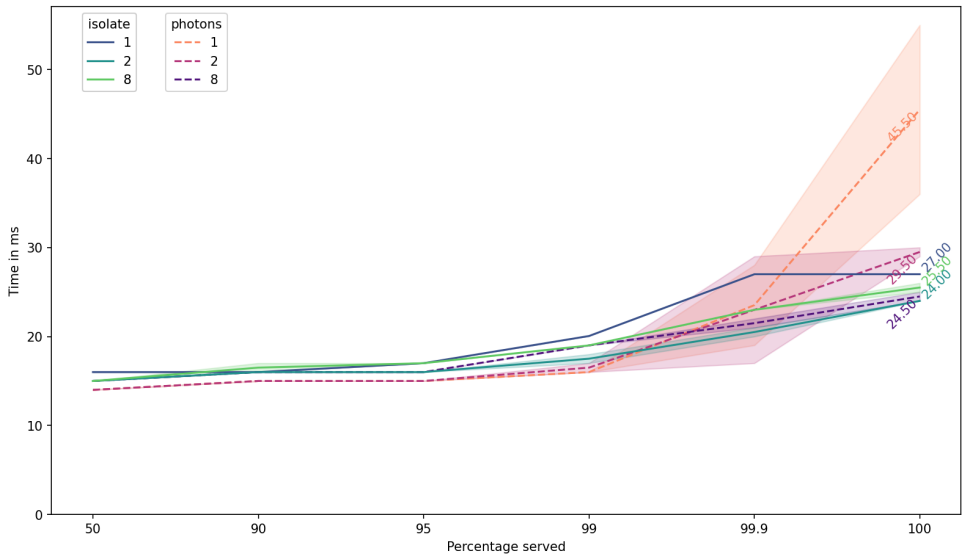

Figure 4.6: *Response time latency percentiles for file hashing workload under different concurrency levels. Dashed lines are for photons proxy while solid lines are for isolate proxy. Different concurrency levels are shown in different colors.*

compilation in the background. After 150s serial invocations, JIT compilation can still interfere the invocation execution. For higher concurrency levels, JVM are fed by more invocations in 150s. Additional CPU resources are used for JIT compilation during warm-up. Therefore, photons proxy does not experience high tail latency for concurrency level 2 and 8.

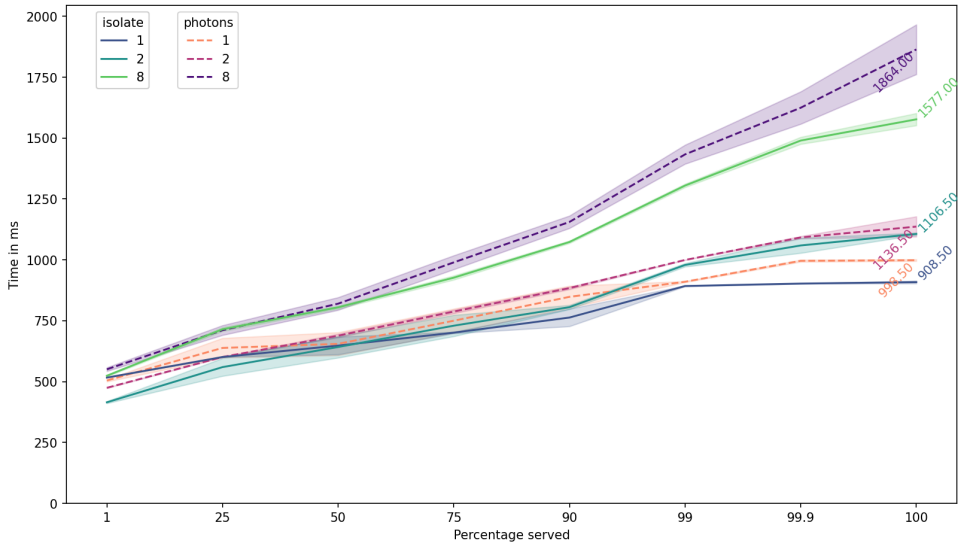

Figure 4.7: *Response time latency percentiles for image classification workload under different concurrency levels. Dashed lines are for photons proxy while solid lines are for isolate proxy. Different concurrency levels are shown in different colors.*

The second plot in Figure 4.7 shows the response time latency for image classification workload. As a CPU-intensive workload, main part of prediction is executed via TensorFlow native

code. JIT compilation therefore does not have huge impact on execution efficiency. In contrast, native image has slightly more efficiency to interact with native library. Therefore, isolate proxy has better latency performance. Since the configuration for this workload makes it CPU-bounded. Underlying native library tries to use as much CPU as they could. Contention for computational resources leads to unevenness of response time distribution. As concurrency factor increases, native library becomes the bottleneck as well. As shown in the plot, both proxies are experiencing performance degradation under higher concurrency levels.

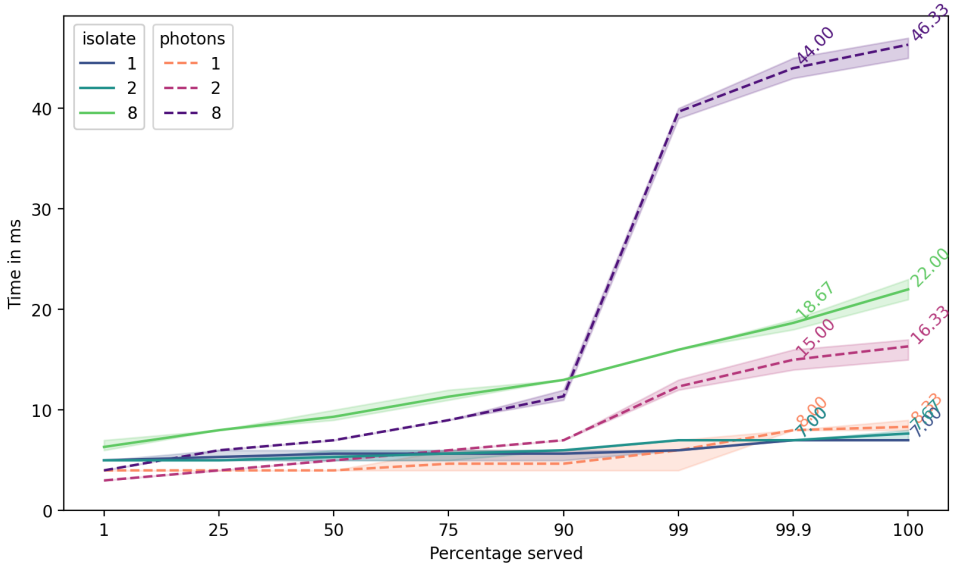

Figure 4.8: *Response time latency percentiles for memory allocator workload under different concurrency levels. Dashed lines are for photons proxy while solid lines are for isolate proxy. Different concurrency levels are shown in different colors.*

The last plot in Figure 4.8 shows the response time latency percentiles for memory allocator workload, where each invocation allocates 10 MB objects. From the plot we notice that best performance of JIT-ed code is always slightly better than AOT compiles native image. For example, under serial invocations, photons proxy has lower response speed for 99% of all invocations. However, for multiple concurrent invocations, independent garbage collection shows its efficiency. For concurrency level 2, photons proxy suffers large tail latency with longest request of 16.33ms, whereas isolate proxy can finish all requests in 7.67ms.

As the concurrency factor further increases, huge amounts of memory are allocated in short time. While both proxies are experiencing performance degradation, tail latency for photons proxy are more than double the worst case of isolate proxy. Photons proxy deals with its slowest 1% of invocations using almost 10 times of its peak performance.

One important thing to note is that, under concurrency level of 8, photons proxy consumes 1484 MB memory while there are only 223 MB memory available for isolate proxy. Distributed garbage collection substantially increases efficiency of memory management, potentially improving service quality under collocation of concurrent invocations.

Results of response latency test show that isolate proxy has more efficiency to invoke native code. Absence of JIT compiler minimizes runtime interference for executed invocation. Dis-

tributed garbage collection of isolate proxy is especially fit for memory-intensive workloads, reducing tail latency by 50% in our example workload.

4.4 Peak Throughput after full Warm-up

As indicated by the results of runtime ramp-up for file hashing workload in Figure 3.5, we notice that the JIT compiler of normal JVM may achieve better optimizations than Native Image. Without execution profiles during run time, the AOT compiler can hardly produce as efficient code as JIT compiler. We measure the peak throughput after full warm-up for different workloads to show this potential performance gap.

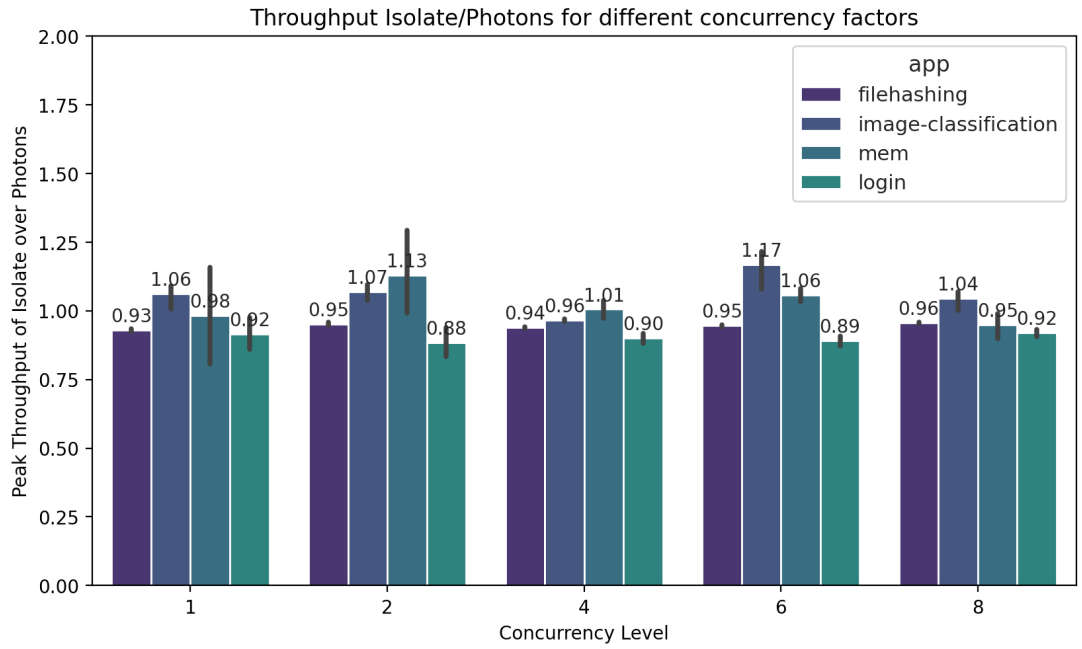

Figure 4.9: *Peak throughput of isolate divided by peak throughput of photons under different concurrency levels for different workloads. Different workloads are denoted in different colors, where mem refers to memory allocator workload and login refers to REST Request workload.*

In Figure 4.9, we plot the peak throughput comparison between two proxies. Numbers shown in the plot are calculated as peak throughput of isolate divided by peak throughput of photons proxy. We perform measurements for several times. Average performance factors are denoted as numbers over each bar, while variances of measurements are represented by the length of segments on the top of each bar. We show the throughput comparison for different workloads under different concurrency levels.

For I/O intended workloads, i.e., REST Request and File Hashing, the AOT compiler cannot produce as efficient code as JIT compiler. As depicted in the plot, isolate proxy experiences up to 7% degradation for file hashing, while runs up to 12% slower for REST request (denoted as login). Execution time of REST Request workload is around 1vms and only allocates few

memory. Using isolated heap spaces add too much overhead for such a tiny workload comparing to threads in photons, although it supplies more safety and security for each execution. For the image classification workload, native image performs more efficient interaction with native code than normal JVMs and therefore surpasses the peak throughput of photons proxy from 4% to 17%.

For memory allocator workload, JIT compilation of photons makes its execution slightly more efficient than isolate under serial invocation, where photons proxy runs 2% faster. When there is more than one invocation collocated in the same runtime, huge burden of garbage collection outweigh the efficiency of JIT compilation. Performance of both proxies for this workload are very close, while isolate proxy has lower tail latency and much less memory footprint.

From all above performance aspects, both isolate and photons proxy show their strengths and weaknesses. We discuss this design trade-off in summary in the next section.

4.5 Local Evaluation Summary

After evaluating different performance aspects of both proxies, we can conclude the advantages and disadvantages using isolate proxy.

- *For computation-intensive workloads, normal JVM could be more efficient.* In such workloads, Java bytecode are likely to be better compiled by JIT compiler using run time profiles. If such workload experiences high invocation frequency, i.e., runtime has enough time and enough number of invocations to be warmed up, a fully warmed-up photons proxy should be a better choice. Especially for tiny workloads with minor memory and time consumption, additional overhead introduced by isolate takes more weight. As the results shown in Section 4.4, isolate proxy may experience up to 18% peak throughput degradation.
- *Isolate proxy has much less base memory footprint.* As shown in Table 4.1, isolate proxy reduces runtime base memory from 35% up to 59%, while may require higher memory increment to keep compensate performance degradation. In the serverless cluster, if the overall memory footprint of isolate proxy can be smaller than photons proxy, it is possible to allocate more instances under restricted resources or just keep more containers warm to avoid cold starts.
- *Isolate proxy has much faster start up and ramp-up process.* Measurements show that isolate proxy reduce the cold start time by 576ms, 32.9% of the total cold start time of photons proxy. Isolate proxy also significantly improve the ramp-up procedure of the runtime. In file hashing workload, photons proxy can only catch up the total number of invocations isolate proxy after 54s of continuous non-stopping invocations, when over 85,000 of invocations have been processed. On the other hand, isolate proxy reduces the first invocation response time from 2x to 100x shorter. Under serverless context, where containers need to be frequently instantiated and shut down, isolate proxy can provide faster scale-out and smaller tail-latency.
- *Distributed garbage collection is more efficient for memory-intensive workloads.* For

memory-intensive workloads such as big data analysis, a distributed version of garbage collection can provide better latency with much less memory increment. As the results of Memory Allocator workload shows, isolate proxy reduces the worst-case latency by 50%, while only requiring 10% memory increment comparing to photons to achieve the same performance.

Combining all the factors above, we want to explore the possible benefits using Native Image under realistic serverless context. Without need to integrate Native Image into large-scale production environment, event-driven simulation is a useful tool to reveal insights of the real system state. We therefore run cluster-wide simulation to study the design trade-off for integrating isolate proxy into serverless platform.

Cluster-wide Event-driven Simulation

From the results of machine local evaluation, we know the strengths and weaknesses of GraalVM Native Image Isolate. Cluster-wide system state highly depends on nature of serverless invocations, i.e., distribution of invocation duration, memory footprint, arrival rate etc. Therefore, we use cluster-wide event-driven simulation with realistic production traces to explore the impact of integrating Isolate proxy. In this chapter, we first introduce how we generate simulation traces and scheduler logic of our simulator. We then interpret the simulation results, showing the trade-off of using Native Image in serverless platform.

5.1 Simulation Trace Generation

Since we are not able to build a realistic large-scale serverless platform integrated with Native Image, we use en vent-driven simulation to study the possible system state. The event-driven simulator takes a stream of events and dynamically schedules them based on the system state. Using realistic events traces can reveal realistic system behavior to a great extent. Researchers usually model coming events for such systems using Poisson distribution for independent arrivals or Markov distribution for dependent events [DBSA20]. In 2019, Microsoft Research study the function invocation statistics on their Azure platform [SFIG+20], showing the invocation statistics from cloud provider's perspective for the first time. Their study reveals that the duration, memory footprint and invocation frequency of different functions diverse greatly. For example, small parts of functions contribute to most of overall invocations and those invocations are arriving concurrently with high probability. This nature of invocation pattern indicates the efficiency of runtime sharing via collocation using Photons [DBSA20]. On the other hand, vast number of functions are invoked very infrequently. Keeping warm containers for them is expensive and even wasteful. In other word, containers for most of the functions are infrequently used and need to be frequently restarted, where fast cold start and ramp-up features of Native

Image can be very beneficial.

Besides the characteristics shown in the paper [SFIG+20], Microsoft Research also released a subset of the production traces they collected for public access [Azu]. Published data set consists of three parts: function duration distributions, function memory footprint distributions and function invocation counts. All identifiers of users, applications and functions are hashed using HMAC-SHA256 to protect users' privacy. Since the functions running on Azure are not available to us, we are not able to measure the real performance using Photons or Isolate proxy, such as startup speedup, memory footprint reduction and peek performance degradation etc. We therefore decide to combine the function invocation counts together with measurements from the four workloads in the last chapter to generate event traces for our simulation. To simplify the simulation, we assume CPU resources are not limited and take the execution time of each workload under default CPU setting from Table 4.1 for granted. We only consider memory footprint as the single limited resources among the cluster and invocation execution time helps scheduler to decide when and how much memory are occupied.

Within Azure traces, different applications are identified by their owners. Each application consists of different functions. While application is the basic schedule unit in Azure platform, we take function of same user as independent schedule unit in our case and do not consider their trigger types. Besides that, we map different function id from the production traces randomly to one of our four workloads. Therefore, all functions are sharing four different performance profiles, while they are treated and scheduled as different functions.

We take the memory footprint, execution time, cold starts and ramp-up procedure into account for each generated event. As the statistics of function invocation counts are aggregated per minute, we first assign timestamp for each invocation. We simply treat all invocations inside one aggregated minute as independent events and use uniform distribution to draw a random start time for each invocation in that minute. After that, we map corresponding function id randomly to one of our workloads, take the base memory and memory increment from Table 4.1. The exact memory footprint is dynamically calculated based on the collocation status during simulation time. One sample event generated using above approaches is illustrated as follows.

function_id	abs. start_time (s)	total_memory	private_state	startup_time (s)
538e5f...9191	50640.00007525826	132	6	1.780716656

Table 5.1: *Example of generated simulation event.*

In the Table 5.1, function_id is the unique function id used for scheduling, start_time denotes the absolute timestamp in seconds when the event arrives. total_memory and private_state are base memory and memory increment in MB. start_time is the docker cold start up time generated using Normal distribution based on the measured parameter in Table 4.2. Cold start time is added to total execution time during simulation, if a new container needs to be instantiated for that invocation.

Since we are considering ramp-up procedure of each workload, we cannot determine the exact execution time before we run the simulation. Therefore, execution time of each invocation is generated dynamically based on how the container warmed up during the simulation on the fly.

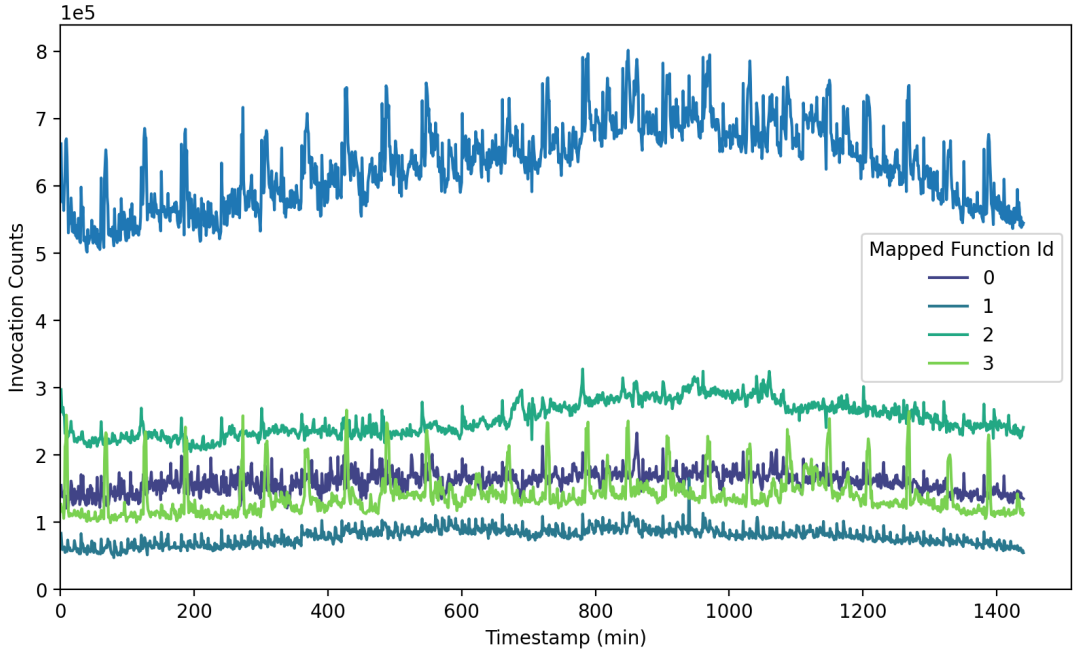

Figure 5.1: *Invocation Counts per minute for the first data from Azure Traces [SFIG+20], counts for each mapped function are also illustrated. Mapped Function Id denote four different workloads we have, 0 for file hashing, 1 for REST Request, 2 for Memory Allocator and 3 for Image Classification.*

In Figure 5.1, we show the number of invocations per minute for the first 24 hours, i.e. 1440 minutes from the production traces, together with the invocation numbers for each mapped function. From the invocation counts, it is clear that high number of invocations follows an hourly invocation pattern. Other than that, there is no significant invocation pattern for any time interval. Since the simulation for different cluster sizes is time-consuming, we therefore only choose the busiest 10 minutes in the first day for simulation, which requires the most resources for that day. We study three cases in the simulation: regular proxy without collocation, Photons proxy and Isolate proxy. We generate two versions of traces, where regular and photons proxy use measurements for photons proxy and Isolate proxy takes its specific memory footprint and execution time profiles.

5.2 Simulator Logic

Our simulator is adopted from the simulator used in [DBSA20]. It performs the resource management for incoming functions, which is the memory management of each machine in the cluster. Many serverless platforms use fixed time "keep-alive" policies for warm containers to reduce cold starts. After each invocation finishes, warm containers are kept alive for another 10-20 minutes [Keea] [Keeb], waiting for possible consecutive invocations. If containers are idle longer than the fixed time-threshold, they are shut down and release the allocated memory back to the cluster.

While there are more advanced adaptive policies like the adaptive scheduler using time-serial prediction described in [SFIG+20], we choose an infinite time "keep-alive" policy in our simulator. In other words, we do not set any fixed time-threshold and keep warm containers as much as we could. However, since we do not have infinite memory resources, eviction of warm containers is unavoidable. We only evict containers when we do not have enough memory to allocate new invocations. This lazy eviction policy represents the best possible case for fixed-time "keep-alive" policy under restricted resources. Other more advanced scheduling policy can always benefit from the faster ramp-up procedure supplied by Native Image as well.

After simulation traces are generated, the simulator takes the stream of events and schedule those incoming invocations into proper containers. The core component of the simulator is a priority queue sorted by timestamp of events. All arrived events are firstly pushed into this queue. During scheduling, scheduler polls the first event from the priority queue and try to find a proper container to execute it. To find a proper container, the scheduler needs to consider the following situations.

- *Collocation is allowed and there is a busy container executing the same function.* Scheduler tends to collocate concurrent invocation to the active containers in order to save memory usage. To collocate an concurrent invocation, scheduler needs to expand the container memory limit by *memory increment* required by the function. In this case, scheduler examines all machines holding such an active container and checks their available memory. If there is any machine having enough memory for the required *memory increment*, scheduler collocates that invocation directly.
- *Collocation is allowed but not directly possible due to memory shortage.* As we mentioned before, scheduler follows the lazy keep-alive policy. After some time of warm-up, machines among the cluster are keeping as many warm containers as they could. When the scheduler tends to collocate an invocation but the available memory on that machine is not enough, it tries to evict possible warm containers in exchange for needed memory. During eviction, all warm containers are sorted by the invocation frequency of the function they hold. Containers that are less likely invoked are shut down until there are enough memory for the collocation on that machine.
- *Idle warm container is available in the cluster.* If collocation is not allowed or the scheduler cannot get enough memory for the collocation, even via eviction, scheduler tries to find any warm containers for the function among the cluster.
- *No proper container exists for the incoming function.* If there is no warm idle container in the cluster, scheduler has to instantiate a new container for the invocation somewhere in the cluster. If this succeeds, cold start time is added to execution time.
- *No place available to allocate the invocation.* The worst case for the scheduler is that there is no memory resource among the cluster to execute this invocation. At this time, memory becomes the bottleneck of serverless platform. Scheduler can only push the current invocation in another waiting queue and schedule them with highest priority when there is available resource.

Once an invocation can be successfully scheduled, the execution time of it is generated as well. Since we are considering ramp-up procedure of each workload, the execution time of

each invocation depends on the age of container where it is executed. We define the age of warm container as the number of invocations have been executed in it. We only update the age of containers after invocation finishes. For example, all the concurrent invocations are experiencing cold start and treated as the first invocation as long as the first invocation for that container has not been finished.

The ramp-up procedure is divided into two phases. For the first one thousand invocations we generate the execution time based on container age. After age one thousand, we treat containers as fully warmed up. The total execution time of an invocation is then the sum of potential cold start time and the generated execution time.

After the event is successfully scheduled, we push the event of invocation finish into the same priority queue. When scheduler polls a finish-event, it updates state of machines correspondingly. We dump the *active memory usage* every simulated second and collect the invocation statistics. Since we are using lazy "keep-alive" eviction policy, warm containers are filling memory of each machine, we distinguish the memory usage from *active memory usage*, which only counts the total memory of active containers, where there are invocations being executed inside.

5.3 Simulation Result

Using the aforementioned simulator and generated trances, we run several simulations under different cluster size. We set up the cluster with one hundred machines and adjust the cluster size by setting available memory for each machine. At first, we analyse active memory usage dumped throughout the simulation. Then we count the total number of cold starts during the 10 minutes simulation time, studying the trade-off between total number of cold starts and cluster sizes. There are large amounts of concurrent invocations in the production trace. Runtime sharing improves overall performance of isolate and photons proxy over regular by more than 2 magnitudes. Therefore, we only make comparisons between photons and isolate proxy.

5.3.1 Memory Consumption

As listed in Table 4.1, isolate proxy has a lower base memory for all the workloads. However, for file hashing and REST request, isolate proxy requires a higher memory increment, which can lead to larger memory footprint than photons proxy under high concurrency level. We first show the cluster-wide active memory usage of each proxies under different cluster sizes. In Figure 5.2, we plot the total active memory usage throughout the cluster for both proxies under different cluster sizes. Simulation results under different cluster sizes are distinguished by different colors.

For this simulation, we have to setup an initial empty state for our simulator. As a result, at the very beginning of the simulation, high number of cold starts become system bottleneck and number measured at this time do not reflect the real performance of the cluster. We discard the cluster warm-up process and cool down process from the total 10 min simulation. The simulation process showed in the following figure is therefore shorter than 10 min.

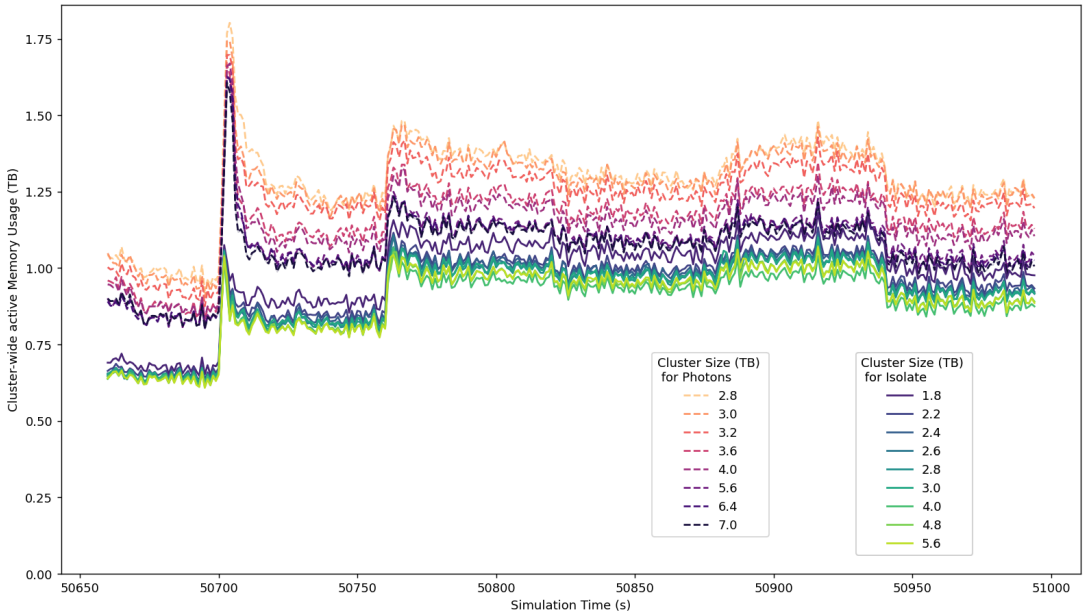

Figure 5.2: *Cluster-wide total active memory usage in TB. Dashed lines denote photons proxy, while solid lines denote isolate proxy. Simulations under different cluster size are distinguished by different colors. Simulation starts from absolute*

From the plot we notice that all solid lines are lower than dashed lines, which means total memory of running containers for isolate proxy are always lower. Base memory of active containers for each function dominates the active memory usage, which makes isolate proxy has overall lower memory footprint.

On the other hand, as cluster size gets larger, active memory usage goes down for both proxies. When there is less memory on the machine, many concurrent invocations cannot be collocated to active container due to memory limitation on that machine. New containers need to be instantiated on another machine, adding additional memory usage. As the available memory for each machine goes up, concurrent invocations can be collocated to fewer active runtimes, reducing overall active memory footprint as shown in the plot.

5.3.2 Cold Starts

During each simulation, we also count the total number of cold starts. We show these numbers in Figure 5.3. In this figure, number of cold starts are plotted as a function of cluster size. From the plot we notice that the more memory we give to the cluster, the less cold starts we are experiencing, until all warm containers can be kept alive for a while inside the cluster due to the infinite-time "keep-alive" eviction policy.

Comparing the knee points of both curves we can find that using isolate proxy reduces the memory requirement by around 30%. Cluster with 4 TB memory using isolate proxy is encountering the same number of cold starts as a 6.4 TB cluster equipped with photons proxy. This memory efficiency can reduce large amount of costs for cloud providers. The saved memory can also be leveraged to keep more warm containers to reduce cold starts. By suffering same number

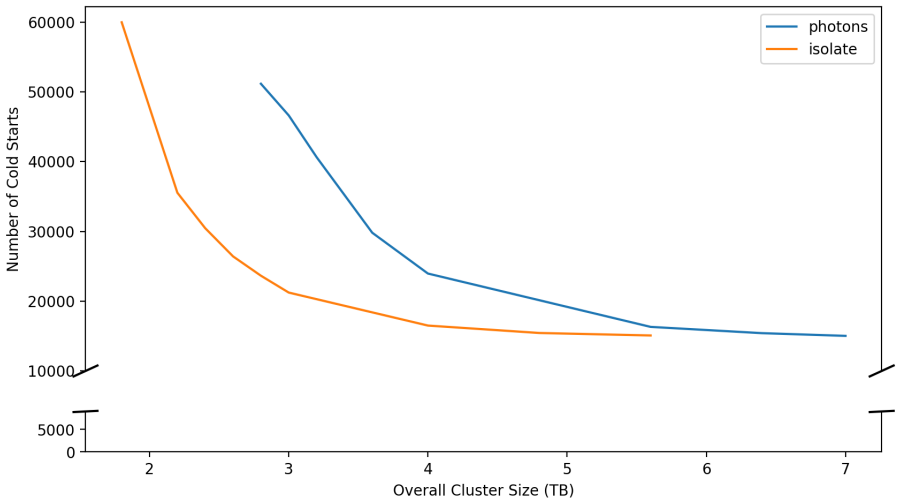

Figure 5.3: *Total number of cold starts during simulations under different cluster sizes. Note that for better visibility, x and y axis are not starting from 0.*

of cold starts, fast start up of isolate proxy container can still save over 500ms per cold start as shown in Table 4.2, which has huge impact for the performance of short-lived invocations.

5.3.3 Response Time Latency

To study the quality of service using the isolate proxy, we analyze the execution time for simulated invocations. Since all of the simulated functions are sharing profiles from four different workloads, we therefore aggregate results for all invocations that are mapped to the same workload. We use the median response time to indicate the quality of service. The metrics are firstly calculated for each simulated function. To aggregate results into the four mapped workloads, we take weighted average of the median response time, where invocation counts of each function are selected as weight.

Figure 5.4 shows the median response time of functions with profiles of different workloads. The top plot is for functions mapped to file hashing workload. From that plot we can see isolate proxy achieves better performance in general. We can also notice that there are invocation bursts at around 50700s, which causes large number of cold starts. Cold starts of photons proxy significantly degrade the median response time, while isolate proxy keeps invocation time stably small. As the cluster size goes up, spike for photons proxy goes down. On the one hand, more memory allow more warm container to reside in memory. On the other hand, it allows concurrent invocations to be more likely allocated on same machine. Both of these reduce the number of cold starts. For photons proxy, cold start takes nearly 1.7s based on our measurement listed in Table 4.2, which is very heavy overhead for file hashing workload with peak performance of 15ms.

As we have showed in Figure 5.2, isolate proxy has lower overall active memory usage, which allows warm start and collocation. In addition to that, native image takes around 500ms less for every cold start. Although the cold start time is still too high comparing to workload execution

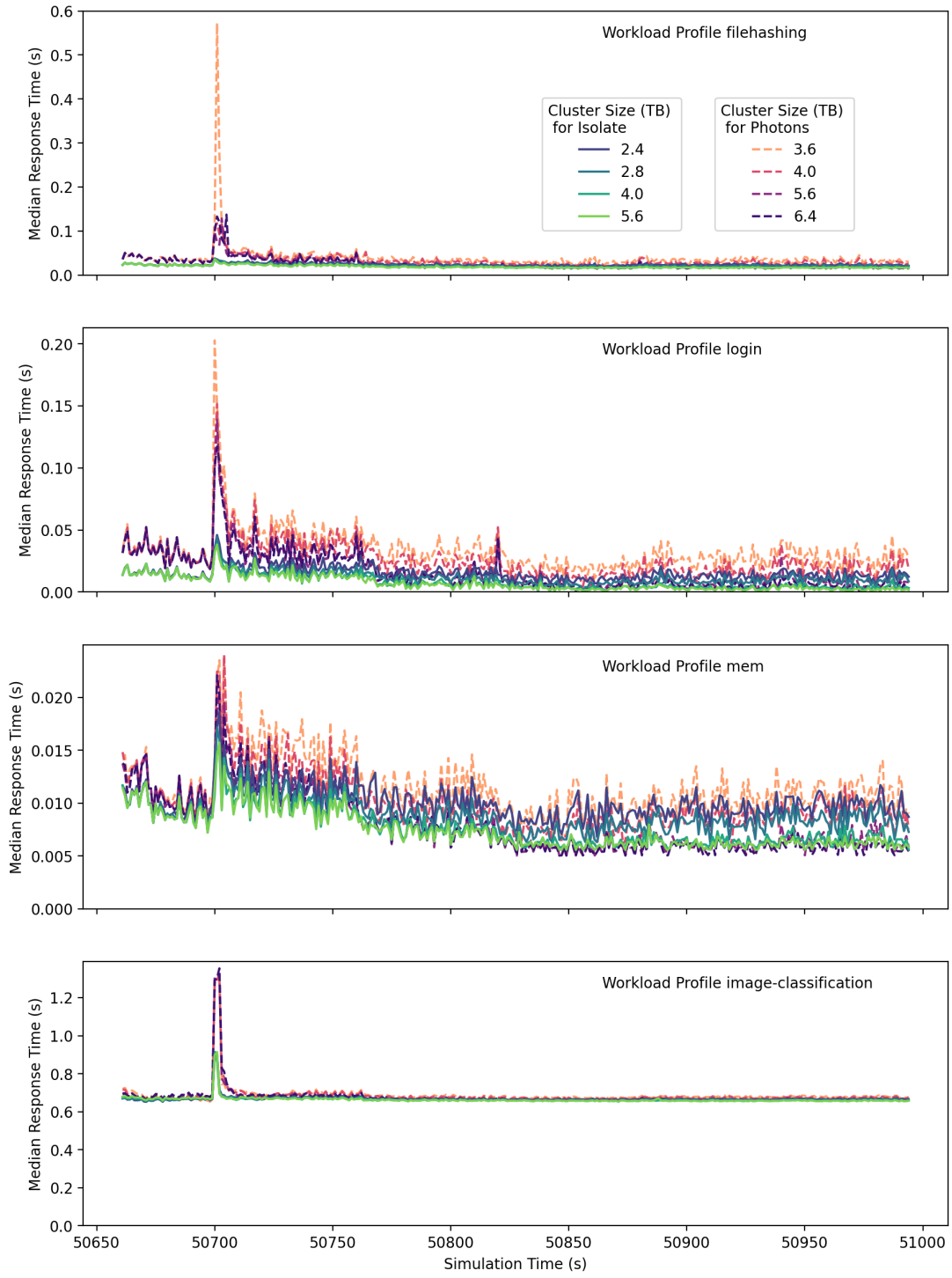

Figure 5.4: *Weighted average of aggregated median response time of all functions that are mapped to different workloads. Median response time is calculated separately for each function and averaged using their invocation counts as weight. login denotes the REST Request and mem denotes the Memory Allocator.*

time, faster start up by 500ms is difficult to be caught up by JIT optimization of photons proxy.

For functions taking profiles of REST Request workload, short execution time makes them extremely sensitive to cold starts. As illustrated in the third plot in Figure 5.4, performance of these functions has been improved by a lot as cluster size goes up.

The third plot in Figure 5.4 shows the median response time for functions that are using Memory Allocator workload profiles. As shown in Section 4.4, peak performance of both proxies for this workload is close to each other, while isolate proxy requests much less memory increment for collocating concurrent functions and better tail-latency.

As shown in Figure 5.1, functions mapped to Memory Allocator workloads have the largest invocation frequency comparing to other profiles. Large invocation frequency increases the survival rate of warm containers for these functions among the cluster due to eviction policy of our simulator. Large number of invocations also warm up the container as soon as possible, making peak performance of photons proxy close to isolate proxy, as shown in the third plot in Figure 5.4.

For simulated functions mapped to image classification workload, we can see their median response time in the last plot in Figure 5.4. Unlike previous workloads, image-classification takes over 600ms to be executed and therefore less sensitive to cold start time. Larger base memory of photons proxy creates higher spike during invocation bursts at around 50700s.

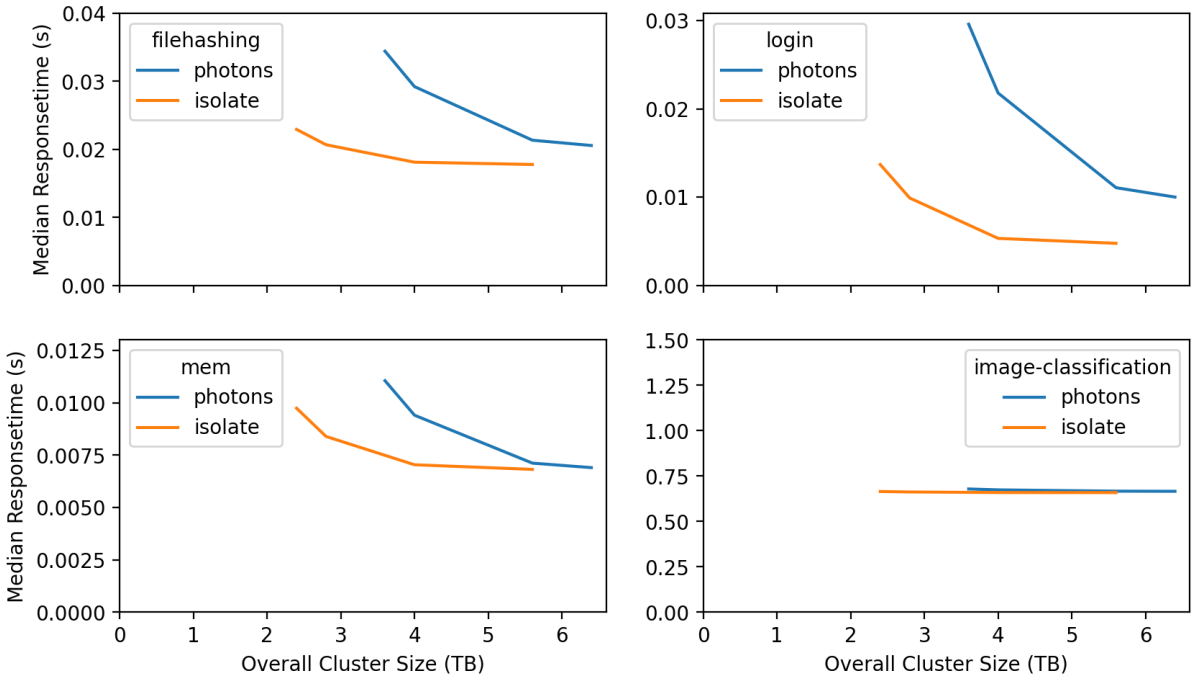

Figure 5.5: *Simulation average median response time for functions using profiles of different workloads under different cluster sizes. login denotes the REST Request and mem denotes the Memory Allocator.*

We calculate the mean of median response time over whole simulations and plot the average median response time as a function of cluster size for different workloads' profiles in Figure 5.5. As depicted in Figure 5.5, isolate proxy shows strictly better median response time than

photons, even with much smaller cluster size. From the plots we can also find that workloads with short execution time are more likely influenced by the additional number of cold starts for photons proxy, while workloads like image classification are less influenced.

For example, as we analyzed in Section 4.4, isolate introduces additional overhead to execute REST Request workload. Isolate proxy takes 10% longer time for each 1 ms invocation, which is around 0.1 ms. However, as shown in the simulation result, cold starts time dominates the latency of such short-lived workload. Reduction of 500 ms start up time of isolate proxy has to be amortized by more than 5000 invocations on fully warmed-up photons proxy. Similar to the ramp-up effect for file hashing work depicted in 4.5 back in Section 4.2, faster start up and ramp-up procedure of isolate proxy help to reduce the overall latency.

5.4 Cluster-wide Simulation Summary

From the simulation results we can see that using isolate proxy reduces the overall memory usage for the cluster by 30%. Under same restricted cluster size, isolate proxy experiences less cold starts and can always supply better quality of service. Lower memory footprint brings efficiency mainly in two ways. Allow more warm containers to reside in memory and allow more invocations to be collocated per runtime. Both of them reduce the number of cold starts.

Comparison of median response time between both proxies shows that runtime cold starts have huge impact on short-lived invocations. Less cold starts also help isolate proxy to deal with invocation bursts more smoothly, adding elasticity to serverless platform. To achieve same quality of services, isolate proxy requires 30% less memory resource, which can reduce much costs for cloud provider.

Our simulation results show that the nature of serverless functions invocation pattern conforms to the strengths of native iamge isolate proxy. It is worthy to use isolate proxy to trade faster start up and lower memory footprint for peak throughput in a serverless context. With less memory usage, the overall performance of the whole platform can be significantly improved.

Related Works

"Those who cannot remember the past are condemned to repeat it". Same problems are studied for many times as the technology develops. In the following, we discuss several works related to from virtualization technologies via runtime-level isolation to cloud simulation, comparing them with our work.

Virtualization technology like gvisior [gVi], firecracker [ABI+20] and kata container [Kat] supply light-weight virtualization while keeping security by allocating each invocations in different instances. Using these technologies have reduced the docker start time by a lot. However this does not reduce the start up time of the application language runtime such as JVM. From our experiments, native image proxy finishes start up in 20ms while normal JVM takes over 200ms. As the VM start up time decreases, the slow start of a JVM is amplified and becomes the optimization bottleneck. On the other hand, without runtime sharing, concurrent invocations are encountering more cold starts, leading to overall performance degradation. Duplicates of language runtime, libraries and shareable program states add additional memory usage to the serverless system, increasing costs for both cloud users and providers.

As an improvement to the existing invocation execution environment, SAND [ACR+18] allocates lambdas of the same function inside a common sandbox, leveraging a hierarchical message bus to enable communication between lambdas of same function, and among different applications. SAND uses process forking to deal with same invocations inside same sandbox, which is fast and memory efficient. However, as pointed out by Dukic et al. [DBSA20], forkink a fully fledged JVM modifies more memory than sharing invocations in same runtime. From our experiments, Copy-on-write of native image isolate does not pollute more memory than normal thread (under 1 MB), additional overhead for memory increment comes mainly from garbage heap hold by each Isolate, which is typically a performance trade-off. Communication across different processes is much heavier than message passing among different isolates as well.

Disjoint heap spaces have shown its efficiency in modern browsers for large amount of concurrent client-side sessions. Wagner et al. [WGW+11] proposed abstraction for disjoint JavaScript

heaps calles compartment. Objects are separated by their document origin in different heaps. Garbage collecting is executed for each individual compartment, increasing browser response time by a lot. Native image also keep disjoint Java heaps for different isolates, supplying secure isolation and efficient distributed garbage collection. Our tail latency measurement for memory-intensive workload shows that isolates reduce worst-case tail latency by 50%, which conforms to the observations from Wagner et al.

Sharing execution environment for different invocations via runtime-level isolation has also been explored before. For example CloudFlare Workers [Clo] leverage Isolate in JavaScript V8 engine [V8J] to allow different functions to be executed on the common JavaScript runtime. However, code is not shared among different Isolates in V8, while isolates in native image directly reference the shared part of code. In addition, CouldFlare restricts applications to be purely JavaScript while native image supports multiple dynamic languages like Python, R, JavaScript etc. Finally, Native Image has much faster start up time than V8 JavaScript VM [WSH+19].

There are some simulators to do simulations for cloud computing systems, such as CoudSim [The] and its extension ContainerCloudSim [PDCB17]. Those simulators are are also designed for studying systems states under different conditions. For example, ContainerCloudSim studies the system performance under different container scheduling and provisioning policies. However, such simulators are highly complicated and include many services we do not need, for example container deployment and destroy. Besides, we want to study the performance of each invocation depending on container state, e.g., the ramp-up process is related to container age we defined in Chapter 5. Modifying such a system to support our use case is too much overhead. We therefore developed our own simulator extended by the simulator from [DBSA20].

Conclusions and Outlook

In this project, we design and implement a serverless proxy runtime using GraalVM Native Image Isolate, leveraging runtime sharing for collocating same invocations from same user inside the same runtime proposed by Photons [DBSA20]. We let each invocation be executed in different isolates, which has independent isolated heap space. Isolate separates private states of each invocation automatically, ensuring the correctness of their executions. Using Native Image, isolate proxy achieves fast start up and lower memory footprint. Independent heaps allow garbage collection to be performed in a distributed manner, adding memory management efficiency for memory-intensive workloads. On the other hand, separated heap space limits the object sharing among different isolates. We propose our solutions of isolate pooling and shared isolate to enable object caching and state sharing.

We use different workloads to perform machine local evaluation. Isolate proxy reduces runtime base memory for serial invocations by up to 65%, while only need 10% memory increment for additional collocated memory-intensive invocation. Container equipped with isolate proxy also has shorter start up time. It costs over 500ms less to wait for service in container available. On the other hand, isolate proxy trades fast start up and lower memory footprint for peak throughput. JIT compiler of fully-fledged JVMs have 5% to 7% better performance for I/O intensive workloads, while for invocations with extremely short execution time and memory consumption, such as REST Request in Figure 4.9, JIT compiler can increase this performance gap up to 12%. Native Image has lower overhead to interact with native libraries, and can reduce the longest tail-latency of memory-intensive workloads by over 50%.

In the last part of this project, we use cluster-wide event-driven simulation with realistic serverless invocations pattern to study the efficiency to integrate isolate proxy. Simulation results show that isolate proxy can reduce the overall cluster memory usage by 30% while keeping slightly better performance comparing to photons proxy. Lower memory footprint further reduces number of cold starts, which in turn let isolate proxy have better response time latency under same restricted cluster size. From the results we can have the conclusion that it is wor-

thy to trade fast start up and lower memory footprint for peak throughput in serverless context. Integrating isolate proxy improves performance while reducing cost for memory consumption.

Although we have shown the advantages using isolate proxy via cluster-wide simulation, we make a lot of assumptions to simplify the simulation and only use statistics of four workloads we have measured, which is very restricted. In real-world serverless platform, execution duration, memory footprint and start up time of applications vary a lot [SFIG+20]. Trade-off between CPU configuration and execution time can also have huge impact for serverless platform scheduler. Deploying large numbers of applications in isolate proxy with various configurations in large-scale serverless system would reveal much more insights than our simulation.

On the other hand, using different isolates supply more security than needed for only allowing same invocations from the same user. However, there are still many open problems need to be addressed. While independent heap space of each Isolate can be used to limit memory consumption of each invocation, there is no restrictions for CPU resources that each isolate can access. As shown in image classification workload in Figure 4.7, contention for CPU resources increases tail latency significantly. What's more, all isolates inside common process are accessing the same file system, which leaves vulnerabilities for potential attacks. Further works can be extended to add scheduler to provide resources fairness and file system isolation to increase security, which would ultimately make isolate a lightweight abstract with fully virtualization functionality.

GraalVM supports polyglot programming. Besides JVM-based languages, many dynamic languages like JavaScript, Ruby, R or Python can also be executed in native image format [Nat]. Allowing multi-language application in different isolate using native image would further extend the capability of isolate proxy as a general purpose serverless runtime.

Bibliography

[ab] ab - apache http server benchmarking tool - apache http server version 2.4. https://httpd.apache.org/docs/2.4/programs/ab.html. (Accessed on 04/17/2021).

[ABI+20] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association.

[ACR+18] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker 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.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[Apaa] Apache openwhisk is a serverless, open source cloud platform. https://openwhisk.apache.org/. (Accessed on 03/29/2021).

[Apab] apache/openwhisk-runtime-java: Apache openwhisk runtime java supports apache openwhisk functions written in java and other jvm-hosted languages. https://github.com/apache/openwhisk-runtime-java. (Accessed on 04/10/2021).

[AWSa] Amazon s3 rest api introduction - amazon simple storage service. https://

docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html. (Accessed on 04/13/2021).

[AWSb] Aws lambda – serverless compute - amazon web services. https://aws.amazon.com/lambda/. (Accessed on 03/29/2021).

[AWSc] Use aws deeplens to give amazon alexa the power to detect objects via alexa skills | aws machine learning blog. https://aws.amazon.com/blogs/machine-learning/use-aws-deeplens-to-give-amazon-alexa-the-power-to-detect-objects (Accessed on 04/17/2021).

[Azu] Azurepublicdataset/azurefunctionsdataset2019.md at master · azure/azurepublicdataset · github. https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md. (Accessed on 04/18/2021).

[Claa] Class initialization in native image. https://www.graalvm.org/reference-manual/native-image/ClassInitialization/. (Accessed on 04/09/2021).

[Clab] How to deploy deep learning models with aws lambda and tensorflow | aws machine learning blog. https://aws.amazon.com/blogs/machine-learning/how-to-deploy-deep-learning-models-with-aws-lambda-and-tensorflow (Accessed on 04/17/2021).

[Clac] How to serve deep learning models using tensorflow 2.0 with cloud functions | google cloud blog. https://cloud.google.com/blog/products/ai-machine-learning/how-to-serve-deep-learning-models-using-tensorflow-2-0-with-cloud (Accessed on 04/17/2021).

[Clad] Use python and tensorflow for machine learning in azure | microsoft docs. https://docs.microsoft.com/en-us/azure/azure-functions/functions-machine-learning-tensorflow?tabs=bash. (Accessed on 04/17/2021).

[Clo] Cloudflare workers®. https://workers.cloudflare.com/. (Accessed on 04/22/2021).

[CN03] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, GPCE '03, page 364–376, Berlin, Heidelberg, 2003. Springer-Verlag.

[DBSA20] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo 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.

[EC2] Amazon ec2 pricing - amazon web services. https://aws.amazon.com/ec2/pricing/. (Accessed on 04/13/2021).

[gVi] gvisor. https://gvisor.dev/. (Accessed on 04/22/2021).

[GZC+19] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[Hin01] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, page 54–61, New York, NY, USA, 2001. Association for Computing Machinery.

[IMS18] V. Ishakian, V. Muthusamy, and A. Slominski. Serving deep learning models in a serverless platform. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 257–262, 2018.

[Int] Graalvm documentation. https://www.graalvm.org/docs/introduction/. (Accessed on 04/01/2021).

[Jav] Remote method invocation home. https://www.oracle.com/java/technologies/javase/remote-method-invocation-home.html. (Accessed on 04/15/2021).

[JEP] Jep 243: Java-level jvm compiler interface. https://openjdk.java.net/jeps/243. (Accessed on 04/06/2021).

[JSSS+19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing, 2019.

[Kat] Kata containers - open source container runtime software | kata containers. https://katacontainers.io/. (Accessed on 04/22/2021).

[Keea] Cold starts in aws lambda | mikhail shilkov. https://mikhail.io/serverless/coldstarts/aws/. (Accessed on 04/18/2021).

[Keeb] Cold starts in azure functions | mikhail shilkov. https://mikhail.io/serverless/coldstarts/azure/. (Accessed on 04/18/2021).

[Lam] Aws lambda changes duration billing granularity from 100ms down to 1ms. https://aws.amazon.com/about-aws/whats-new/2020/12/aws-lambda-changes-duration-billing-granularity-from-100ms-to-

(Accessed on 04/13/2021).

[Min] Minio | high performance, kubernetes native object storage. https://min.io/. (Accessed on 04/13/2021).

[Mon] Mongodb atlas: Cloud document database | mongodb. https://www.mongodb.com/cloud/atlas/lp/try2?utm_source=google&utm_campaign=gs_emea_switzerland_search_core_brand_atlas_desktop&utm_term=mongodb&utm_medium=cpc_paid_search&utm_ad=e&utm_ad_campaign_id=12212624569&gclid=CjwKCAjwjuqDBhAGEiwAdX2cj0aKyRV5DoFMRUco7NydE3R_2akwvh0rdZBzBMfiC9R_lsS0z5TwbBoCgiwQAvD_BwE. (Accessed on 04/17/2021).

[Nat] Native image. https://www.graalvm.org/reference-manual/native-image/. (Accessed on 04/22/2021).

[NIJ] Overview (graalvm sdk java api reference). https://www.graalvm.org/sdk/javadoc/overview-summary.html. (Accessed on 04/11/2021).

[NIO] Native image compatibility and optimization guide. https://www.graalvm.org/reference-manual/native-image/Limitations/. (Accessed on 04/09/2021).

[Pas] Andrea Passwater. 2018 serverless community survey: huge growth in serverless usage. https://www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage. (Accessed on 03/27/2021).

[PDCB17] Sareh Fotuhi Piraghaj, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, and Rajkumar Buyya. Containercloudsim: An environment for modeling and simulation of containers in cloud data centers. *Software: Practice and Experience*, 47(4):505–521, 2017.

[Pho] openwhisk-runtime-java/proxy.java at devel · rodrigo-bruno/openwhisk-runtime-java. https://github.com/rodrigo-bruno/openwhisk-runtime-java/blob/devel/core/java8/proxy/src/main/java/org/apache/openwhisk/runtime/java/action/Proxy.java. (Accessed on 04/10/2021).

[PTB+97] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications - a way ahead of time (WAT) compiler. In *Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS 97)*, Portland, OR, June 1997. USENIX Association.

[Ryd03] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In Görel Hedin, editor, *Compiler Construction*, pages 126–137, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[SB15] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69, 2015.

[SFIG+20] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider, 2020.

[sub] graal/substratevm at master · oracle/graal. `https://github.com/oracle/graal/tree/master/substratevm`. (Accessed on 04/09/2021).

[The] The clouds lab: Flagship projects - gridbus and cloudbus. `http://www.cloudbus.org/cloudsim/`. (Accessed on 04/22/2021).

[Tra] Build configuration. `https://www.graalvm.org/reference-manual/native-image/BuildConfiguration/#assisted-configuration-of-native-image-builds`. (Accessed on 04/09/2021).

[Ung95] David Ungar. Annotating objects for transport to other worlds. In *Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '95, page 73–87, New York, NY, USA, 1995. Association for Computing Machinery.

[V8J] V8 javascript engine. `https://v8.dev/`. (Accessed on 04/22/2021).

[WGW+11] Gregor Wagner, Andreas Gal, Christian Wimmer, Brendan Eich, and Michael Franz. Compartmental memory management in a modern web browser. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, page 119–128, New York, NY, USA, 2011. Association for Computing Machinery.

[Wha] What is kubernetes? | kubernetes. `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`. (Accessed on 03/29/2021).

[Wim] Christian Wimmer. Isolates and compressed references: More flexible and efficient memory management via graalvm. `https://medium.com/graalvm/isolates-and-compressed-references-more-flexible-and-efficient` (Accessed on 03/27/2021).

[WSH+19] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss, and Thomas Würthinger. Initialize once, start fast: Application initialization at build time. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[WWW+13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 187–204, New York, NY, USA, 2013. Association for Computing Machinery.