Optimizing AOT Compilation Pipeline for Cloud Applications

Wallace Rocha dos Santos Garbim

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Rodrigo Fraga Barcelos Paulus Bruno

October 2024

Acknowledgments

I am deeply grateful to my family for their unwavering support over the past two years. A special thanks goes to Luciana Garbim, whose constant presence and encouragement gave me the strength to persevere.

I would also like to extend my heartfelt thanks to my advisor, Professor Rodrigo Bruno, whose invaluable insights, guidance, and knowledge were instrumental in making this thesis possible.

To each of you, my sincerest thanks.

Abstract

Serverless applications offer the abstraction of the entire IT infrastructure so that the user has the advantage of running simple applications very quickly. By not managing the infrastructure the customer receives scalability and high availability of the applications by default.

The most popular languages for serverless applications need a language runtime to run the application and rely on Just-in-time compilers (JIT compilers) for performance optimization. However, the startup time of the JIT compilers is very slow for a serverless application. In contrast, the binary generated by an Ahead-of-Time (AOT) compiler has a very reduced startup time, which is a better approach to use in serverless applications.

Despite having reduced startup time compared to the JIT compilation, the AOT compilation suffers from not being able to generate good optimize code in comparison to JIT-optimized code. Therefore, in this work, we want to develop a platform where it will be possible to send an artifact in `.jar` format and automatically build a Native Image as binary code using the AOT compiler. After collecting profiles during the execution of functions the platform will rebuild the application using the profiles to obtain an optimized version. Thus putting together the best of what exists in the profile-aware code compilation and AOT compiler.

Keywords

Serverless; Native image; Function as a service; JIT and AOT compiler; Cloud computing; Profile-guided optimizations;

Resumo

Os aplicativos serverless oferecem a abstração de toda a infraestrutura de TI para que o usuário tenha a vantagem de executar aplicativos simples muito rapidamente. Ao não gerenciar a infraestrutura, o cliente recebe escalabilidade e alta disponibilidade dos aplicativos por padrão.

As linguagens mais populares para aplicativos serverless precisam de um tempo de execução de linguagem para executar o aplicativo e dependem de compiladores Just-in-time (compiladores JIT) para otimização de desempenho. No entanto, o tempo de inicialização dos compiladores JIT é muito lento para um aplicativo serverless.

Em contraste, o binário gerado por um compilador Ahead-of-Time (AOT) tem um tempo de inicialização muito reduzido, o que é uma abordagem melhor para uso em aplicativos serverless. Apesar de ter tempo de inicialização reduzido em comparação com a compilação JIT, a compilação AOT sofre por não ser capaz de gerar um bom código otimizado em comparação com o código otimizado por JIT. Portanto, neste trabalho, queremos desenvolver uma plataforma onde será possível enviar um artefato em formato `.jar` e construir automaticamente uma Native Image como código binário usando o compilador AOT. Após coletar perfis durante a execução das funções, a plataforma reconstruirá a aplicação usando os perfis para obter uma versão otimizada. Reunindo assim o melhor do que existe na compilação de código com reconhecimento de perfil e no compilador AOT.

Palavras Chave

Serverless; Imagem Nativa; Function as a service; Compilador JIT e AOT; Computação na nuvem; Profile-guided optimizations;

Contents

1 Introduction 1

1.1 Context . . . 2

1.2 Problem . . . 3

1.3 Objective . . . 3

1.4 Solution . . . 3

1.5 Document Organization . . . 4

2 Background 5

2.1 Cloud computing . . . 6

2.2 Microservices . . . 6

2.3 Serverless . . . 9

2.4 Just-In-Time Compilation . . . 11

2.5 Ahead-of-Time Compilation . . . 12

2.6 Profile-Guided Optimizations . . . 13

3 Related Work 15

3.1 JITServer: Disaggregated caching JIT Compiler for the JVM in the Cloud . . . 16

3.2 HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale . . . 16

3.3 ShareJIT: JIT Code Cache Sharing across Processes and Its Practical Implementation . 17

3.4 Improving Virtual Machine Performance Using a Cross-Run Profile Repository . . . 18

3.5 Summary . . . 18

4 System architecture 21

4.1 Overview . . . 22

4.2 Controller Service . . . 22

4.3 Profiles merging . . . 25

4.4 Lambda Manager platform . . . 26

4.5 Lambda Manager pipeline . . . 26

4.6 Extending Lambda Manager's Pipeline to Support PGO . . . 28

4.6.1 Graalvisor . . . 28

4.6.2 Lambda manager 28
4.6.3 The profile Storage 29

5 Evaluation **31**
5.1 Benchmarks 32
5.2 Evaluation environment 32
5.3 Evaluation methodology 33
5.4 Metrics 33
5.5 The advantage of having Profile-Guided-Optimization 34
5.6 The minimum number of invocations of a function to have an effective profile 39
5.6.1 Throughput per function 39
5.6.2 Average and P90 latency per function 40
5.6.3 Binary size 42
5.7 Combining profiles from different functions 42

6 Conclusion **47**
6.1 Key Findings and Contributions 48
6.2 System Limitations and Future Work 48
6.3 Conclusion 49

Bibliography **49**

List of Figures

2.1 Model-View-Controller (MVC) Architecture. 7
2.2 Microservices Architecture. 8
2.3 FaaS Architecture. 10

4.1 Optimization AOT pipeline Architecture register event. 23
4.2 Optimization AOT pipeline Architecture invoke event. 24
4.3 .iprof file content example. 25
4.4 Lambda Manager pipeline. 28

5.1 Throughput Graalvisor vs Graalvisor PGO. 35
5.2 Average latency Graalvisor vs Graalvisor PGO. 36
5.3 90th percentile latency Graalvisor vs Graalvisor PGO. 37
5.4 Cold start latency Graalvisor vs Graalvisor PGO. 37
5.5 Binary size Graalvisor vs Graalvisor PGO. 38
5.6 Throughput Graalvisor PGO optimized. 40
5.7 Average latency Graalvisor PGO optimized. 41
5.8 P90 latency Graalvisor PGO optimized. 41
5.9 Binary size Graalvisor PGO optimized. 42
5.10 Hello Function throughput with merged profiles. 43
5.11 Sleep function throughput with merged profiles. 44
5.12 File hashing function throughput with merged profiles. 44

1

Introduction

Contents

1.1 Context 2
1.2 Problem 3
1.3 Objective 3
1.4 Solution 3
1.5 Document Organization 4

1.1 Context

Serverless is a cloud computing service where it is possible to execute applications on demand without the need to manage or maintain cloud infrastructure. All the main infrastructure-related tasks such as configuration, maintenance, update, and management are under the cloud provider's responsibility.

The serverless application execution model is based on the allocation and scaling of on-demand compute resources. When a serverless application needs to be executed, the infrastructure (which might be based on virtual machines, or containers) will be automatically provisioned for the execution of the application. When the execution is finished all available infrastructure will be automatically deactivated, reducing resource overheads as it can be used on another workload or even shut down. Users pay only for what is used and for the duration of the application execution time for the time it was used. In addition to the infrastructure benefits and costs, developers also benefit from the inherent dynamic behavior of serverless infrastructure, which is the ability to automatically activate and deactivate a resource, reducing the complexity of managing and developing complex distributed applications.

The most popular languages for serverless applications are Java, JavaScript, and Python. Due to the fact that these languages need a Language Runtime to run the application, it is natural to make use of Just-In-Time compilers (JIT compilers) for performance purposes. JIT is well known for it is great performance benefits for long-running applications [1].

When the application starts running, the bytecodes are interpreted by the language runtime. From the moment the code is running, the language runtime starts analyzing and profiling the code. Once the code reaches a certain number of executions the compiler marks it as warm. As the number of executions increases, the code is marked as hot. From that moment on, the JIT compiler performs the necessary optimization to improve performance and converts application bytecode into optimized machine code.

However, serverless applications running on language runtimes using a JIT compiler may not always be the best approach [2]. The JIT compiler needs to profile the use of the application for several executions. Since serverless functions may only run for a short time frame [3], applications may never be JIT-compiled at all. Moreover, since multiple function invocations may execute concurrently in different runtimes, multiple concurrent invocations will exercise multiple concurrent JIT compilers, leading to profiling and compilation effort redundancy.

To eliminate this redundancy, instead of using the JIT compiler, the Ahead-of-Time compilation (AOT) is a better option. Because the AOT compiler will transform the bytecode into machine language, without the need to do any kind of profiling and optimizations during program execution. In addition, an even more efficient combination can be used, Native Image with an AOT compiler (and without a JIT compiler).

Native Image uses an AOT compiler that compiles all the bytecodes in advance to machine code. A Native Image also includes only the code needed to run the application. AOT-compiled functions will have a better overall performance when comparing a non-optimized bytecode leading to a faster application startup time, and lower runtime memory overhead compared to a Java Virtual Machine (JVM). The AOT code won't do all the heavy work of compiling the code to the machine binary at runtime (executing) phase which slows down the application's initialization. As a result, AOT will eliminate everything that is not needed and generate a reduced binary file without the need for a JVM to run the code.

1.2 Problem

Despite having many advantages, the AOT compiler is not perfect and has some disadvantages compared to the features of the JIT compiler. JIT compilers can perform advanced optimizations which are based on learned knowledge about the application execution. AOT compilers cannot do those optimizations based on previous executions and must be conservative in their optimizations. As a consequence, AOT compilers often produce code that underperforms compared to JIT code. Also, AOT cannot revert decisions and optimize for different utilization and therefore must always produce code that always works. On the other hand, JIT compilers can de-optimize and optimize again which is not possible in AOT compilers, preventing important speculative optimizations from being utilized.

1.3 Objective

The objective of this work is to improve the execution performance and overall image size of the native image using profile-guided optimizations (PGO). The platform should be able to decide the best approach to do all its work with a minimal impact on the overall function execution. This work focuses specifically on serverless environments where functions are often executed rapidly and briefly. It aims to dynamically gather performance profiles over time, eventually accumulating enough profiling data to enable efficient recompilation of the application for optimized performance.

1.4 Solution

The purpose of this project is to build a fully automated platform where it will be possible to upload an artifact in Java Archive (JAR) format and the platform decides when and how the profile data will be collected and when a function should be optimized and rebuilt in a new optimized Native Image. In addition to that also how many invocations of a function are needed to have a good profile data. The platform should combine the low initialization and memory footprint of AOT compilers, with the profiling advantages of JIT compilers.

The system will analyze the generated executable and depending on the behavior of the function execution if it is invoked frequently, the system will automatically perform the profiling and carry out the entire process of rebuilding the Native image again, but this time, with the new profiling data. The system will also be responsible for knowing the minimum amount of profiling necessary to obtain the best reduction in memory consumption and the best gain in performance and throughput. Furthermore, the system will have the ability to use profiles of different functions and use these profiles as a basis for optimization.

1.5 Document Organization

In Chapter 2, it thoroughly examines the foundational concepts and technologies that serve as the backbone of this thesis. This chapter includes a detailed exploration of topics such as cloud computing, which provides the infrastructure for scalable applications; microservices, which allow for modular software development; serverless architectures, which enable on-demand execution of functions; native image ahead-of-time (AOT) compilation, which optimizes application startup times; and profile-guided optimizations, which enhance performance through data-driven adjustments.

In Chapter 3, it presents a comprehensive overview of related work in the field, offering insights into previous research that pertains to the themes and methodologies discussed throughout this thesis. This review not only highlights the contributions of prior studies but also identifies gaps in the existing literature that the work aims to address.

Chapter 4 delves into the architecture of the proposed solution, providing an in-depth explanation of the Lambda manager platform. It outlines the components and interactions within this system, detailing how the workflow is designed to optimize function execution and resource management.

In Chapter 5, it presents an evaluation of the proposed system, which encompasses a rigorous examination of the experimental environment, the specific workloads employed, and the metrics used for performance comparisons. This chapter aims to provide a clear understanding of the system's performance characteristics and its effectiveness.

Lastly, in Chapter 6, it wraps up the thesis by summarizing its primary focus and the significant findings uncovered throughout the research. It also proposes potential directions for future research in this domain, emphasizing the importance of continued exploration and innovation in optimizing serverless architectures and function execution.

2 Background

Contents

2.1 Cloud computing 6

2.2 Microservices 6

2.3 Serverless 9

2.4 Just-In-Time Compilation 11

2.5 Ahead-of-Time Compilation 12

2.6 Profile-Guided Optimizations 13

2.1 Cloud computing

Before the advent of cloud computing, companies owned their own servers. Usually, large companies had a huge amount of equipment, such as servers, firewalls, storage, routers, switches, and various network equipment.

Not only in terms of equipment but to manage all this equipment a team of professionals was needed. In order to maintain such a large amount of equipment and professionals, it was also necessary to bear a very high cost in relation to the initial investment to acquire the entire infrastructure, in addition to a monthly cost.

A problem when a company has total control of its equipment is that it is necessary to scale the IT infrastructure according to the demand it has and also add increments for future growth. At that time it was much more difficult to optimize the use of equipment for the following reasons. First, forecasting user demand is not a trivial task forcing developers to conservatively over-provision infrastructure. Second, it is desired to use the maximum of the computational resources within a controlled limit. For example, it is desired to use the maximum of the resources of a central processing unit (CPU) but not one hundred percent of it as otherwise, a system that is running can overload and simply stop working as expected.

What is usually done in these cases is to dimension the infrastructure a little beyond the maximum load, so that it can have a space capacity in case of unexpected demand, but even so there is still a risk of demand beyond the maximum load.

Over time, it was noted that most of the time the equipment was not used in its entirety with regard mainly to its processing capacity, memory, and storage.

One way to mitigate these problems is the use of could computing. In the beginning, cloud computing was nothing more than providing information technology infrastructure for companies, such as servers, storage, databases, and network equipment. Nowadays, cloud computing is not just hardware or basic software with a database, but a large number of services and technologies.

Large and small companies and startups can benefit from cloud computing as it offers flexible, on-demand resources. Using cloud computing, users pay for the resources that are being used, thus helping reduce upfront costs as well as easily scale up on demand. Among the various technologies and facilities that cloud computing can offer, the use of microservices and serverless computing can be highlighted.

2.2 Microservices

To talk about Microservices it is necessary to talk about Monolith applications. A Monolith application is a single application where all components, interfaces, and business rules are in the same deployable artifact.

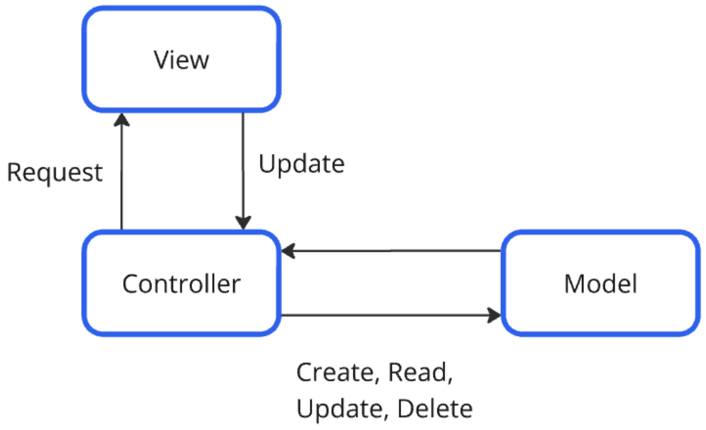

Figure 2.1: Model-View-Controller (MVC) Architecture.

Normally a Monolith application consists of a Model-View-Controller architecture as shown in Figure 2.1, where the Model usually consists of the data model layer and business rule, it is the core of the application. The View is the presentation layer, where an external agent such as a user, communicates with the Monolith application, it consists mostly of HyperText Markup Language (HTML) pages.

The Controller is the layer that makes the interface between the View layer and the Model layer, the View layer sends basic reads and writes commands. These requests are received by the Controller, which in turn knows where to send them in the Model layer and performs the right operation.

This triad forms the basic structure of a Monolith application. An important characteristic of a Monolith application is that it consists of a single artifact, which in turn implies that when some modification needs to be made to the code, such as adding new functionality or fixing a bug, all the code must be compiled, a new artifact must be created and deployed. Furthermore, when it is necessary to scale the application, all components or modules of the application are scaled at once, even if only one component, module or functionality has been modified.

This characteristic implies the constant challenge that whenever there is a new deployment, the entire system must stop to add a new version. In very large or complex systems it is common to have a very large code base, which makes any modification very hard and time-consuming. Additionally Monolith applications tend to have many people involved, including many developers which makes it very difficult to manage changes that do not affect other adjacent components of the application.

The advent of cloud computing was an enabler for the world of Microservices, making it much easier to decompose a Monolith into multiple applications with different responsibilities interacting with each other in a distributed architecture, where for example a shopping cart component could be modified and deployed independently without depending on other components of a Monolith application and also be able to scale horizontally with multiple instances.

Microservices are independently deployable services modeled around a business domain. [4] Mi-

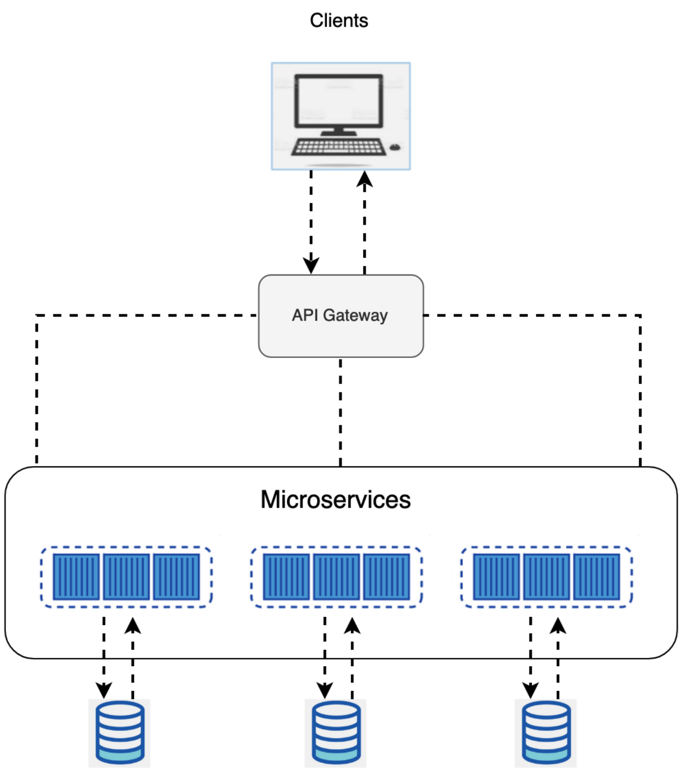

Figure 2.2: Microservices Architecture.

croservices are not a new idea or concept, Microservices are any independent service that exposes an interface to communicate with the outside world.

However, when it comes to cloud computing, the most used Microservices model is the client-server model, where a service (the server) is exposed through an Application Programming Interface (API) usually a Representational State Transfer (REST API) over Hypertext Transfer Protocol (HTTP protocol) and the clients consume their API.

A basic Microservices architecture consists of an API Gateway that communicates between clients that are externally connected to the Microservices cluster as shown in Figure 2.2. The API Gateway, in turn, communicates with the Microservices through a Rest API. Naturally, the API Gateway has a very important role in knowing how to translate a request from an external client and knowing exactly which Microservice is responsible for that request. The Microservices can also communicate with other Microservices, databases, and other resources internal to the cluster.

A small service that only performs tasks under a specific domain is a very interesting idea instead of traditional Monolith systems, where all the functionality is in a single big artifact. With the advent of Microservices and their popularity, their benefits and not-so-good consequences became evident.

The benefits are diverse, for example:

- A specific functionality can be developed in a programming language more suitable for a specific domain, for example: using Python or R for Data Science systems instead of Java;
- Systems can be divided into small teams rather than large teams of programmers;
- New functionality can be more easily integrated into small systems, and new functionality can be integrated without having to deploy a complete application;
- In case of a problem in a specific part of the system, for example, the shopping cart of an online shopping system is unavailable, but the search or recommendation system may be working. This is because the systems are independent, which is often not the case with a Monolith system;

The undesired consequences are also evident, such as:

- Managing and monitoring many services becomes a much more complicated task, knowing where there was a failure and tracking the entire path taken until discovering the problem is much more complicated.
- Managing many programming languages adds a greater component of difficulty in a company.
- Monitoring services, collecting metrics, scalability and the entire IT infrastructure becomes a crucial task for the success of Microservices.

As mentioned above Microservices bring many benefits, but also some less desired consequences of complexity. However, it can be observed that the not so desired consequences are mostly with regard to the IT infrastructure.

Several prominent companies, including Amazon, Netflix, Uber, Airbnb, eBay, Spotify, Twitter, and SoundCloud, have successfully migrated from monolithic architectures to microservices to address scalability, flexibility, and development speed challenges. Amazon and eBay moved to microservices to handle increasing complexity and scale, while Netflix and Uber needed more reliable systems to support rapid user growth. Spotify, Airbnb, and SoundCloud adopted microservices to improve deployment frequency and feature management, and Twitter's migration helped resolve stability issues and better handle high traffic loads. Each company used microservices to enable faster development cycles, better fault isolation, and improved scalability.

2.3 Serverless

Serverless is a cloud-based application development model. Unlike Microservices, in the development of applications based on serverless, the developer or the development team does not need to take care of the management, monitoring, and scalability of the infrastructure.

All this work is done by the cloud provider, so the developer can only take care of developing the application, that is, the business rule, and leaving the heavy work of systems administration to the cloud provider.

In this way, with the use of serverless, the main negative point of using Microservices and having to manage the infrastructure can be eliminated. In short, when it comes to serverless as shown in Figure 2.3, the cloud provider is responsible for automatically running the application when a client requests it in event 1 and the serverless platform in event 2 will dynamically spin up a container and execute the function.

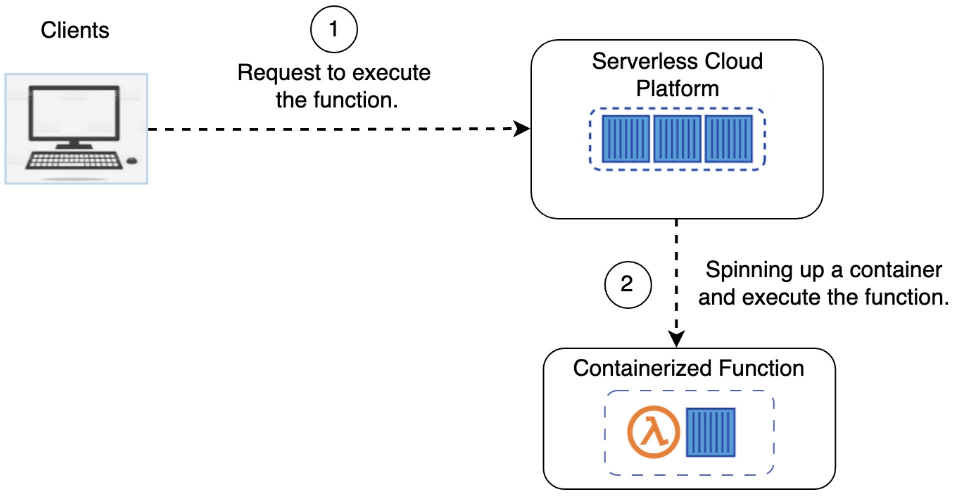
Clients
1
Request to execute the function.
Serverless Cloud Platform
2
Spinning up a container and execute the function.
Containerized Function
λ

Figure 2.3: FaaS Architecture.

Due to the fact that a serverless application is usually smaller in size compared to a Microservices application, it is even easier and faster to scale independently. Through many invocations of the same application, the cloud provider is also responsible for scalability, scaling up on high demand or scaling down when the demand is not so high. When it is noticed that there are no more requests for a given application, the cloud provider is also responsible for shutting down all instances.

The cloud provider uses a payment model per millisecond of use, as well as a predefined amount of vCPU and Random Access Memory (RAM). In this way, the end user only pays for what they use in the period that is used.

Serverless computing is fundamentally powered by language runtimes that have evolved over the years, primarily driven by the demands of application performance and resource efficiency. Historically, the Java Virtual Machine (JVM) was designed for long-running applications, leveraging its Just-In-Time (JIT) compilation capabilities to optimize performance dynamically over extended execution periods. This design philosophy capitalized on the ability of JIT to analyze runtime behavior and apply optimizations, thus improving the execution efficiency of applications that are continuously active.

However, with the rise of Function as a Service (FaaS) and serverless architectures, a significant shift in application design has occurred. FaaS is inherently focused on short-lived applications that execute in response to events, resulting in rapid creation and destruction of execution environments. In this context, the traditional JVM, optimized for prolonged runtime scenarios, is ill-suited for managing these ephemeral workloads [1], [2]. The startup overhead associated with the JVM, combined with the loss of optimization benefits after each function invocation, makes it challenging to meet the performance requirements of serverless applications effectively.

As a result, there is a growing need for more lightweight and efficient execution environments that can support the rapid scaling and transient nature of serverless computing. This evolution reflects a broader trend in software development, where traditional paradigms must adapt to the demands of modern architectures, ensuring that applications can run efficiently and responsively in a serverless context.

2.4 Just-In-Time Compilation

Just-In-Time (JIT) compilation is a dynamic optimization technique used by modern runtime environments to enhance the performance of applications. Unlike Ahead-Of-Time (AOT) compilation, where code is fully compiled before execution, JIT compilation occurs during the program's execution. This allows the system to make real-time decisions about how to optimize the code based on its actual runtime behavior.

In a JIT-compiled environment, the application initially runs using an interpreter or minimal compilation to start quickly. As the application executes, the runtime system collects performance data, typically through profiling. This profiling information includes details about frequently executed code paths, branch prediction patterns, and memory usage. The longer the application runs, the more detailed the profile becomes, allowing the JIT compiler to make informed decisions about which parts of the code should be optimized.

The JIT compiler then recompiles sections of the code based on this profiling data. Hotspots, or frequently executed code paths, are identified and compiled into highly optimized machine code. By doing so, JIT compilation balances initial execution speed with long-term performance improvements. As the program continues to run, the JIT can re-optimize the code if the application's behavior changes over time, ensuring that the compiled code remains efficient throughout its lifecycle.

This approach is especially beneficial for long-running applications, where the overhead of profiling and compilation is amortized over time. As the application is continuously profiled, the JIT compiler can adapt to its evolving runtime characteristics, resulting in significant performance improvements. The key advantage of JIT compilation is that it optimizes the code based on the actual workload, ensuring that resources are utilized efficiently and code execution is tailored to real-world use cases.

In summary, JIT compilation leverages runtime profiling to compile code dynamically, providing optimized performance for applications that run over extended periods. By continuously analyzing and adapting to the execution patterns of an application, JIT ensures that the code is compiled in the most efficient manner based on actual runtime conditions.

2.5 Ahead-of-Time Compilation

Ahead-of-Time (AOT) compilation is a method of converting high-level programming languages into machine code prior to execution. This process occurs during the build phase, as opposed to Just-In-Time (JIT) compilation, which translates code at runtime. AOT compilation offers significant advantages, particularly in performance and predictability. By eliminating the need for on-the-fly code generation, AOT improves startup times and overall execution speed. The compiled code remains consistent, allowing for predictable performance, which is especially valuable in environments where latency is critical.

However, AOT compilation also presents disadvantages. The fixed nature of AOT-compiled code limits its ability to adapt and optimize based on real-time execution patterns. Additionally, the build process can be lengthy, particularly for large applications, as optimizations must be determined without the benefit of runtime profiling.

In contrast, JIT compilation, while adaptable, has limitations that become particularly evident in serverless environments. One major issue is the profiling overhead that JIT compilers require. The need for a profiling phase can introduce significant startup latency, adversely affecting initial response times. Furthermore, in serverless architectures, once an execution context is terminated, any optimized code generated during that session is lost. This ephemeral nature means that the benefits of JIT optimization cannot be retained across invocations, ultimately leading to performance inconsistency.

One notable approach within AOT compilation is Native Image AOT. This technique compiles Java applications into native executables. The binary includes a light-weight runtime that includes a Garbage Collector (GC) and will perform the normal tasks a JVM would do, except that it does not include a JIT compiler. In addition, Native Image will also pre-load and pre-process all classes that will be used at runtime. Normally will have a better overall performance when comparing a non-optimized bytecode, better application startup time, and lower runtime memory overhead compared to a Java Virtual machine. Native Image AOT improves performance by reducing the memory footprint and startup time, making it particularly suitable for cloud and serverless environments. This method leverages static analysis to determine which parts of the code are needed at runtime, resulting in smaller, optimized binaries that execute more efficiently.

Due to these characteristics, Native Image is a great solution to the problems faced when using a traditional JVM in conjunction with serverles. [5], [6], [7] As mentioned above, Native Image generates a reduced size binary executable file without the need to use a traditional JVM to run the code. To run the application, it is only necessary to execute the binary file. The advantages of Native Image are irrefutable compared to JIT, the main problem of slow startup faced earlier can be eliminated, however Native image with AOT does not solve all problems.

In summary, while AOT compilation excels in providing performance and resource efficiency, its lack of adaptability contrasts sharply with JIT's flexibility. Nonetheless, JIT compilation faces substantial challenges in serverless contexts, where profiling overhead and the temporary execution environments hinder its effectiveness.

2.6 Profile-Guided Optimizations

The Native Image technology as an AOT compiler is not able to perform advanced optimization. The way to solve the lack of code optimization feature is to use Profile-Guided Optimization (PGO). PGO is an additional Native Image tool responsible for collecting application data (profiles) when in execution mode. [8]

Native Image PGO works as follows. The first step is to build a Native Image binary that includes instrumentation code to collect profiling information. This is achieved by passing a special flag `pgo-instrument` to the compiler.

The second step is to launch the binary and allow it to execute for some time while collecting profiling data. Profiling data is saved in a file with the `.iprof` extension. Data collection ends when the application is terminated. With the `.iprof` file, it will be necessary to run the Native Image build again, now passing the path to the profiling file as a parameter. This way a new binary will be created based on the data collected by profiling.

Profiling data is extremely important to address the problems of non-optimized code and binary low performance generated by the AOT compiler.

PGO is very useful because the main performance issue of Native Image is solved. When PGO is used, the new binary will be optimized towards the same non-optimized function, similar to what a JIT compiler would do. In addition, there could be a reduction in the total file size as well as the heap memory.

3 Related Work

Contents

3.1 JITServer: Disaggregated caching JIT Compiler for the JVM in the Cloud 16
3.2 HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale 16
3.3 ShareJIT: JIT Code Cache Sharing across Processes and Its Practical Implementation 17
3.4 Improving Virtual Machine Performance Using a Cross-Run Profile Repository . . . 18
3.5 Summary 18

3.1 JITServer: Disaggregated caching JIT Compiler for the JVM in the Cloud

JITServer [9] uses a widely used concept, caching code already optimized by JIT. Its difference is that it uses an external server where it is possible to send a request requesting that the JITServer compile the code to optimized machine binary.

The JITServer works as follows: a client (the application) generates a request for compilation, and this request is sent to a queue that waits to be sent to the JITServer and processed. Once the request has been sent to the JITServer, the request is sent to a new queue where it waits to be compiled. Each request only contains the most used data in most compilers, this means that not all optimizations that a modern JVM can do will be done.

Data is not always sent in the optimization request. When necessary, the JITServer can request more data from the client. Once the process is finished, the body of the compiled code and the metadata are sent to the client, which is relocated and installed.

Despite studies showing a reduction in CPU usage, RAM memory, and additionally a reduction in startup time and warm-up time, JITServer is not able to solve the serverless problem.

Since the JITServer is a common JIT compiler, the client needs to be executed many times until it reaches the threshold and sends the code to be remotely optimized in the JITServer. This model would not work with serverless, because normally a cloud provider provides a function on demand in a container and when the execution ends, the container is terminated, so it is not possible to have the threshold. In addition, there is still another problem, which is the network latency, if it were possible to send a request to the JITServer. A possible latency would revert to an additional cost since serverless is computed in milliseconds.

3.2 HHVM Jump-Start: Boosting Both Warmup and Steady-State Performance at Scale

In this work the author uses a strategy of sharing VM profile data on a large scale to solve a very common problem which is the runtime overhead during the application's start-up of JIT compilers, however, this strategy was designed based on very specific criteria of HipHop Virtual Machine (HHVM) [10].

A specific case in point is the Facebook website. Facebook uses HHVM to run its website, HHVM employs a traditional VM architecture where the source code is compiled to a bytecode representation offline before the application starts to execute. HHVM uses a custom bytecode designed specifically for representing PHP and Hack programs.

A problem faced by Facebook is that when there is a new release of a new version of the website, it takes a long time in the warmup phase of the HHVM due to the fact that the codebase contains more than 100 million lines of code. This makes the warmup phase take approximately 30 minutes.

As a consequence, during the warmup phase, Facebook loses a large part of its ability to respond to user requests, because, during this entire period of time, the HHVM has to perform all the steps to reach its maximum performance, which is profiling of the code and later compile to machine code.

This problem is solved by adding what the author calls a Jump-start. The purpose of Jump-start is to pre-compile the JITed code before starting server requests so that the server can achieve high performance from the start.

Due to its architecture, HHVM provides the possibility to choose between two pre-compiled code options: both optimized and live code, or just the optimized code. The last option was chosen because otherwise, it would take a long time to have both at the same time, approximately 30 minutes. However, with only the optimized code, 90 percent of the total performance is achieved.

Despite significantly reducing the warmup phase, this approach does not solve the serverless with Native Image problem. Jump-start makes constant use of the JIT compiler and has the JIT compiler as a pillar for its functioning, however, Native Image does not use a JIT compiler, making its implementation unfeasible. Jump-start uses profiles from a version prior to the version being optimized, however, the proposal is to always use the most up-to-date code.

3.3 ShareJIT: JIT Code Cache Sharing across Processes and Its Practical Implementation

ShareJIT [11] uses the concept of share JIT code cache, a widely used technique, but this study has a very specific focus on reducing memory usage focused on the context of the Android Runtime (ART). ShareJIT is a global code cache for JITs that can share code across multiple applications and multiple processes.

This study was motivated by the increased memory consumption in Android applications. Every Android application has duplicate copies of compiled code that are kept in process-private code caches. The idea of ShareJIT is to make these duplicate caches shareable between processes and applications.

ShareJIT consists of two key components: a global JIT cache and a global sharing map. Shared memories across processes are implemented as regions of Android Shared Memory, a component provided by Android OS to facilitate memory sharing. ShareJIT uses the initial zygote process to create the shared memory space it needs during the zygote's startup. The Zygote is a process that is started its execution during the Android OS startup; it is initialized by preloading all the runtime Java classes and other shared resources.

Despite having achieved its goal of reducing memory consumption and consequently improving performance, this approach does not solve the problem of serverless with Native Image. Native Image does not use the JIT compiler, which is crucial in the ShareJIT solution, which makes its use completely unfeasible. ShareJIT was designed to be used in a VM with multiple processes and applications that benefit from sharing code in common, which is not the case with a serverless function that has a very short lifecycle and does not share code with multiple processes because it is intended to be executed normally only once in its life cycle and must be isolated, that is, it has no communication with external functions or processes. Another disadvantage is that in order to have more shareability, ShareJIT has limited use of some JIT optimizations.

3.4 Improving Virtual Machine Performance Using a Cross-Run Profile Repository

Online profiling is a technique used by the Java Virtual Machine (JVM) to collect data on a Java application in order to do dynamic optimizations to improve performance. This collection is made from the moment the application is initialized and depending on its use, an analysis is made, and later this analysis of code executions the JVM makes several optimizations and generates a new binary code.

However every time the JVM is shut down or rebooted or even when there is a new deployment of an application, it is necessary to do the profiling and optimization process again. The purpose of this study is to implement an automated architecture where profiles are collected and stored in memory and later the JVM performs optimizations based on one or more profiles [12]. A great advantage of this approach is that even if the application has not yet provided any profile data, it can still take advantage of the optimizations obtained through profiling other instances.

This approach is very interesting for short-lived applications because it can take advantage of all these profiling mechanisms. This is typically the behavior of a serverless, as it is a very short-lived function.

However, it is not an approach that can be used in serverless design with AOT compilation. Because the profile data collected is used only when the application is initialized and not before as in the case of AOT. There is additional overhead in parsing profiles as well as extra consumption of JVM resources.

3.5 Summary

The related work proposals try to somehow solve the main problem of slow startup and binary optimization in the JIT compiler. Either in the widely used form of caching where they try to reuse previous optimizations in order to improve startup time and performance. Using an external JIT server in order to reduce the need for processing and memory consumption and use it as caching or even sharing profiles of previous JIT executions.

The main problem is that none of them use the AOT compiler, the solutions are very focused on the JIT compiler which is designed to be used in medium or long-running applications and consequently cannot be used in an AOT approach for short-lived serverless applications.

4

System architecture

Contents

4.1 Overview 22

4.2 Controller Service 22

4.3 Profiles merging 25

4.4 Lambda Manager platform 26

4.5 Lambda Manager pipeline 26

4.6 Extending Lambda Manager's Pipeline to Support PGO 28

4.1 Overview

The purpose of this project is to create a fully optimized platform, where the user sends an artifact in `jar` format and the platform executes the entire build process for the Native Image, collects the profile, and rebuilds the application with profiling.

The platform will be divided into the following components:

- Controller Service: Orchestrates the overall system operation;
- Build service: Handles the generation of native images;
- Profile storage: Stores the function profiles collected during code instrumentation with PGO enabled;
- Function Storage: Manages the storage of functions in various formats, including native images and binaries;

The main component of the platform is the Controller service. It will be the platform's entry point, responsible for receiving applications in `.jar` format and sending them to the Function storage. Once in the Function Storage, the Controller will trigger the build process in the Build Service of two types of Native Images, one without the profiling flag and the other with profiling enabled.

These two Native Images are important because when executing the function, it is likely that the function with profiling enabled will have a lower performance than the function that does not have profiling enabled. This is relevant because we only want to run a fraction of the total runs with profiling enabled.

Once the function calls start, the controller will be responsible for determining how many times the profiled function will be executed. Through a predefined threshold, it will be possible to determine when we have enough profiling data for a well-optimized build. The `.iprof` file, which is the output file of the profiling, will be stored in the Profile Storage.

When the global threshold used across multiple containers is reached the controller will send an event to the Build Service to merge the profiles and use it to build a new optimized Native Image.

4.2 Controller Service

The controller is the core of the platform, it is responsible for orchestrating system events, establishing the profiles threshold, determining when a function with profiling enabled or not should be executed and when an optimized Native Image build should be executed.

The platform will have mainly two events: the first is the install event when a client sends the `.jar` artifact and the second is the invoke when it receives a request to execute the function.

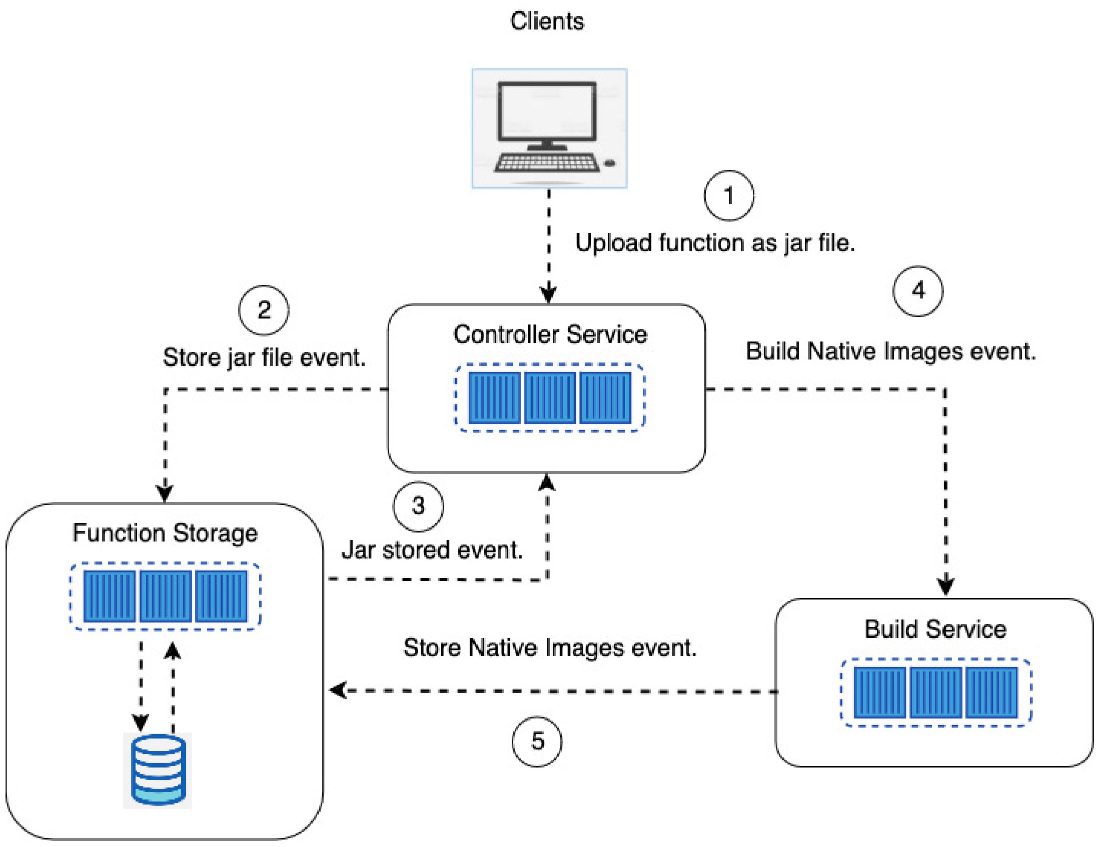

Figure 4.1: Optimization AOT pipeline Architecture register event.

The install event is represented in Figure 4.1. In event 1, the client sends the jar artifact to the platform, the Controller Service receives the file, and in event 2 stores it directly in Function Storage. When the Controller Service receives a confirmation event in event 3 that the file was properly stored, the Native Image build process starts.

The Controller Service in event 4 sends a request to the Build Service to create two Native Images, one with profiling enabled and another with profiling disabled. The two Native Images are important because in event 5 the Build Service will send the files to the Function Storage and in the invoke event the Controller Service will determine when to use the Native Image with profiling enabled or not. Not all invocations should have profiling enabled due to possible performance issues. Only a certain fraction of the total runs should be profiling enabled.

The invoke event happens after the end of the register event. During the interval in which the platform is creating the Native Image, the function is in disabled mode and cannot be invoked until the build is finished. When the build is finished the client can send requests to execute the function as Native Image. As shown in Figure 4.2, in event 1 the client requests the execution of the function, the Controller Service receives the request and immediately checks if the function already has an optimized Native

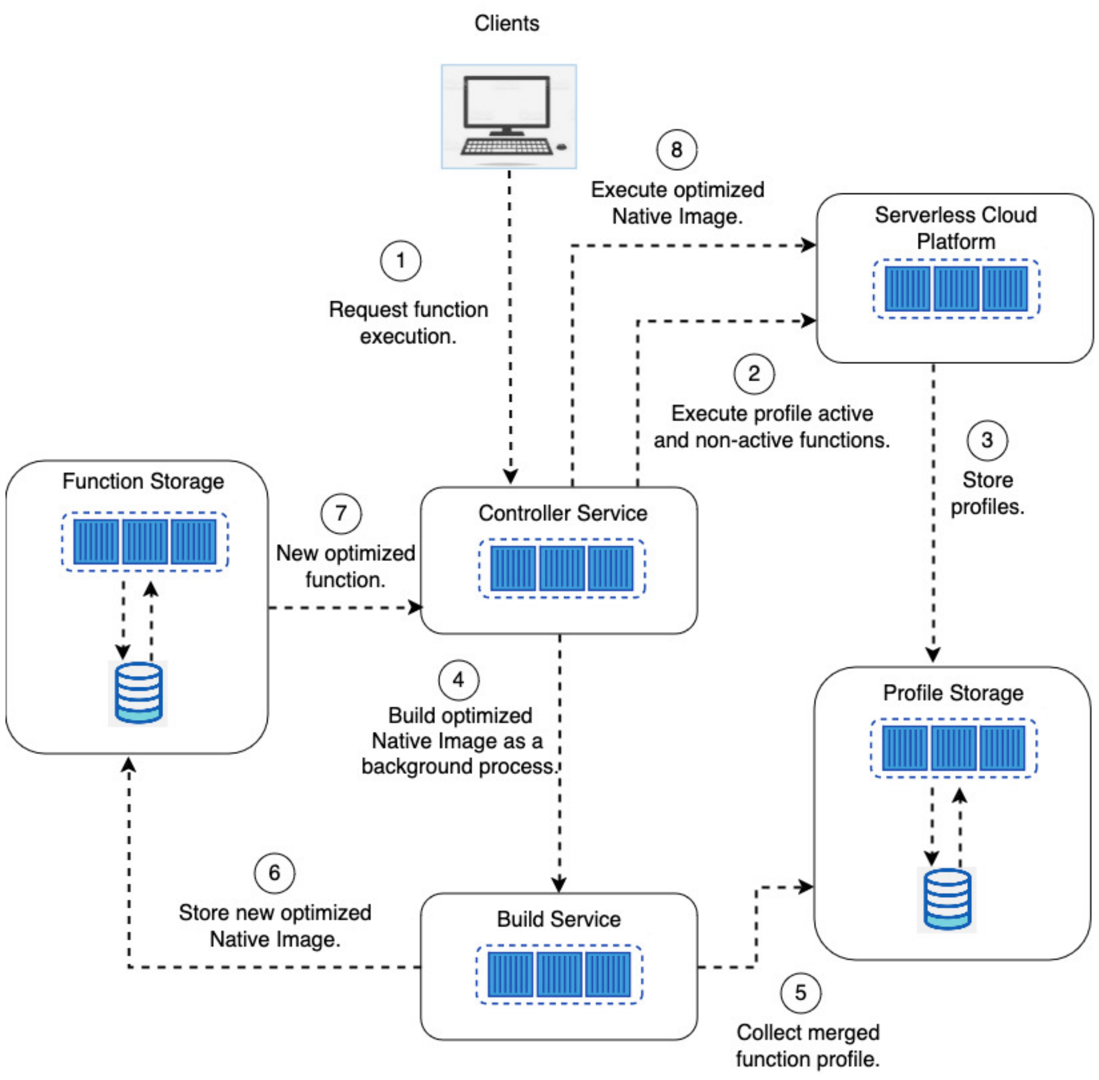
Clients
8
Execute optimized Native Image.
Serverless Cloud Platform
1
Request function execution.
2
Execute profile active and non-active functions.
3
Store profiles.
Function Storage
7
New optimized function.
Controller Service
4
Build optimized Native Image as a background process.
Profile Storage
6
Store new optimized Native Image.
Build Service
5
Collect merged function profile.

Figure 4.2: Optimization AOT pipeline Architecture invoke event.

Image version, if so, it executes the optimized version otherwise launches the unoptimized function and monitors the executions to know when to execute functions with profiling enabled or not.

In event 3, eventually, some functions will be executed with profiling enabled and consequently, the profile files will be sent to the Profile Storage.

Once the Controller Service verifies that enough profiles have been collected, event 4 will be sent to the Build Service to collect the merged profile in event 5 and then build a new optimized Native Image for the function. So in event 6, the Build Service will send the file to be stored in Function Storage.

Finally, in event 7, the Function Storage notifies the Controller Service that there is a new optimized function and the Controller stops executing non-optimized functions, and in event 8 executes only the function with the optimized Native Image.

4.3 Profiles merging

Profile Storage is responsible for storing profiles, but it is also responsible for merging profiles of the same function running in different instances at the same time or separated time. The merge is necessary because each execution of a function will generate a new `.iprof` file when the profiling is enabled. When the Controller Service determines that the profile threshold has been reached then the Profile Storage will merge the files. As shown in Figure 4.3, the final `.iprof` structure.

```
{
  "version": "0.2.0",
  "types": [
   { "id": 4472, "typeName": "[Ljava.security.Provider;" },
    { "id": 2747, "typeName": "sun.nio.ch.FileLockImpl" },
    ...
  ],
   "methods": [
   { "id": 32768, "methodName": "address", "signature": [ 4051, 1, 1724, 0 ] },
    ...
   ],
   "hotnessPoints": [
   { "methodBciList": "7822:0,11372:13,11355:22", "records": [ 19 ] },
    { "methodBciList": "7461:0,7458:25", "records": [ 11 ] },
    ...
   ],
  "conditionalPoints": [
  { "methodBciList": "8632:58", "records": [ 71, 0, 1, 61, 1, 0 ] },
    { "methodBciList": "4416:42,2520:1", "records": [ 50, 0, 0, 24, 1, 2 ] },
    ...
  ],
  "invokePoints": [
  { "methodBciList": "7934:18,7716:92", "records": [ 3380, 3 ] },
    { "methodBciList": "6063:1,4279:5", "records": [ 2573, 5 ] },
    ...
  ],
  "monitorsProfile": {
  "records": [ 2, 4, 112, 1, 118, 7, 886, 11, 1607, 7, 1782, 4, 2426, 15 ]
  }
}
```

Figure 4.3: .iprof file content example.

The `.iprof` file generated contains detailed profiling information collected during the execution of a Java application. This file captures essential performance data such as frequently executed methods (hotspot methods), execution counts, branch prediction data, and inlining decisions. Additionally, it includes information on memory allocation patterns, object usage, garbage collection events, and call frequencies. These metrics provide a comprehensive overview of the application's runtime behavior, highlighting which parts of the code are performance-critical and how resources are utilized.

The data within the `.iprof` file is leveraged to optimize future runs of the application by informing the native image generation process. By understanding which methods are hot and how often they are called, the system can make informed decisions about inlining methods, optimizing branch predictions, and managing memory more efficiently. The profiling information also aids in identifying deoptimization events, allowing for adjustments in compilation strategies to maintain optimal performance. Ultimately, the `.iprof` file enables the production of highly optimized binaries that enhance both startup times and overall execution speed based on real-world usage patterns.

4.4 Lambda Manager platform

The platform used to implement the architecture described in Section 4.1 was GraalServerless (Hydra) [13].

Hydra is a new serverless platform that showcases how advanced runtime techniques can enhance serverless application performance. Rather than introducing a new interface or function scheduling method, Hydra uses language runtimes as a virtualization tool to manage large-scale function executions efficiently.

Like other serverless platforms, Hydra offers an endpoint accessible via web, allowing users to register and invoke functions. When a function is invoked, Hydra assigns it to a cluster node and a lambda executor (a VM). Frequently called functions, or "hot functions," are compiled into Native Images and run within a Native Image Unikernel (NIUk) VM, a lightweight virtual machine that minimizes startup time and memory usage by eliminating the need for an operating system.

Functions that are frequently invoked are processed within Native Image Isolates in a single NIUk VM, which uses a language runtime abstraction to manage these executions. For multi-language applications, Hydra uses Truffle, allowing different language engines to run within separate Isolates, enabling concurrent function execution within one NIUk VM.

At the core of Hydra, the Lambda Manager is the component responsible for deciding which lambdas should be launched and when they should be terminated. The Lambda Manager also decides when and which functions should be optimized by progressing in a function pipeline (see next section).

4.5 Lambda Manager pipeline

The Lambda Manager orchestrates the execution pipeline of lambda functions. It is a Java-based application built with the Micronaut framework, providing a REST API for interaction.

Lambda Manager works as follows: once initialized, the user must send a configuration file that contains information such as: gateway address, maximum memory used by the lambda function, timeout, health check interval and lambda pool that defines the number of standby instances for a function to be executed.

Once this is done, the next step is to upload the function to be executed. From this moment on, Lambda Manager is ready to accept requests.

Every time a function is registered in Lambda Manager, this function will go through a pipeline. The pipeline consists of the following steps:

- Hotspot with agent (collect metrics)
- Hotspot
- Native Image
- Graalvisor PGO enabled (generates profiles)
- Graalvisor PGO optimized

As shown in Figure 4.4, the first step, Hotspot with agent, the function is executed in a container with a Hotspot Java Virtual Machine (JVM) with an agent enabled. The function is executed a predefined number of times and an agent is used to collect execution metrics of the function.

In the second step, after the predefined threshold is reached, the agent is no longer required, and the function is executed directly on the Hotspot JVM without instrumentation. While the function runs on Hotspot, a background build process compiles the function into a Shared Object (`.so`) file. This `.so` file includes the function code combined with a minimal Java Runtime (Native Image). This compiled file will be used in the third step.

In the third stage, Graalvisor, the function undergoes a process similar to the second stage. After reaching a predefined threshold, a build process begins to compile the function into a binary format. Once this process completes, the function advances to the fourth stage

In the fourth stage, Graalvisor PGO enabled, the function is executed and after reaching a predefined threshold, the function moves on to the last stage of the pipeline. During the fourth stage, the new file in binary format is instrumented to be able to collects execution data and after each execution is finished, an `.iprof` file is created. This file will be used later in the last stage of the pipeline. In the fourth stage, the execution time is much longer than in the previous stages and therefore, it must be as efficient and brief as possible and collect as much data as possible to move on to the last phase.

In the fourth stage, Graalvisor with PGO enabled, the function is executed, and upon reaching a predefined threshold, it progresses to the final stage of the pipeline. During this stage, the binary file is instrumented to collect execution data, generating an `.iprof` file after each run. This file will be used in the final stage. Execution time in this phase is significantly longer than in previous stages, so it must be as efficient and concise as possible while collecting comprehensive data for the final phase.

In the fifth and last stage, Graalvisor PGO optimized, the function has already been optimized in a build process that uses the `.iprof` files collected in the previous stage. These files are used as an entry point in the build process of the new `.so` file.

Below is the simplified representation of the pipeline as shown in Figure 4.4.

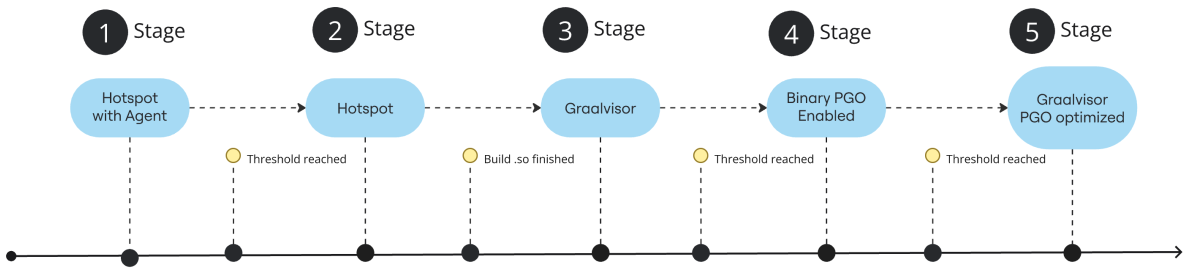

Figure 4.4: Lambda Manager pipeline.

4.6 Extending Lambda Manager's Pipeline to Support PGO

Lambda Manager originally had a three-stage pipeline: Hotspot with Agent, Hotspot, and Graalvisor. To enhance the system, two additional stages were introduced: Binary with PGO enabled and Graalvisor PGO optimized. This required modifications to two core modules of the Lambda Manager platform: the Graalvisor module and the Lambda Manager module.

4.6.1 Graalvisor

The primary role of the Graalvisor module is to execute functions, while the Lambda Manager handles orchestration. In the Graalvisor module, additional functionality was added to invoke functions in PGO-enabled mode in binary format, enabling the collection of `.iprof` files, which are then sent to profile storage. It also manages function execution in PGO-optimized mode using these profile files.

4.6.2 Lambda manager

To maintain portability and ease of execution, the entire architecture is containerized using Docker. At each stage of the pipeline, a pool of containers is ready to execute functions across various stages.

To accommodate the two new stages, two scripts were added to the Lambda Manager. One handles building the executable binary, and the other builds the shared object with PGO optimization using Docker containers.

Another key part of the platform that was updated is the Lambda Manager scheduler. The scheduler orchestrates when a function progresses through the stages and defines how many times it must be executed before advancing to the next stage. Properly promoting functions through the stages, especially with the two new ones, is crucial for achieving the optimization goals of this implementation.

4.6.3 The profile Storage

The profile storage was designed with portability and ease of use in mind. Minio was selected as the storage application. MinIO is an object storage solution that offers an API compatible with Amazon Web Services S3 and includes all key S3 features. It is designed for flexible deployment in a range of environments, including public and private clouds, bare-metal infrastructure, orchestrated platforms, and edge systems [14].

Integrating Minio Storage with the container pool was seamless. Whenever a function is executed with PGO enabled, a background process automatically transfers the generated file to the corresponding function's storage location once execution is complete.

During the final stage of the pipeline, the Graalvisor PGO optimized file is built after the generation of the `.iprof` files, based on the predefined threshold. A new process then initiates the download from Minio Storage, using these files as the foundation for optimization.

5 Evaluation

Contents

5.1 Benchmarks 32
5.2 Evaluation environment 32
5.3 Evaluation methodology 33
5.4 Metrics 33
5.5 The advantage of having Profile-Guided-Optimization 34
5.6 The minimum number of invocations of a function to have an effective profile 39
5.7 Combining profiles from different functions 42

To assess the performance of our proposed solution, we used a set of three benchmarks, as described in Section 5.1. These benchmarks are widely used for serverless and Function-as-a-Service (FaaS) systems, for example in Photons [7] and SeBS [15], making them well suited for evaluating our testing strategy. The chosen benchmarks represent a diverse and comprehensive sample of functions commonly used in modern serverless environments. They include Hello World, Sleep, and File Hashing, which collectively cover a range of workloads from I/O-intensive to mixed and CPU-intensive tasks. Other types of testing are also valuable and will be considered for future work, as time constraints limited their inclusion in the current evaluation.

5.1 Benchmarks

To verify the performance improvements of this work, a number of experiments were conducted. The three benchmarks that were used are:

- **Hello world function:** This benchmark illustrates the most basic and simple type of function. When executed, prints the classic 'Hello World' message, which is then returned to the user as a response.
- **Sleep function:** This benchmark simulates a function execution with a pause defined in the request body in milliseconds and after the pause, returns a message with the value of the wait time that was sent in the request.
- **File hashing:** This benchmark simulates data processing when downloading files and concurrently processing portions of the data. This function downloads an image file from a remote web server. After downloading, it hashes the bytes of the file and returns the hash value.

5.2 Evaluation environment

The test environment for the evaluation of the benchmarks was designed to ensure consistency, performance reliability, and scalability. The hardware setup was based on an enterprise-grade server, providing sufficient computational power and memory to accommodate various test scenarios, including performance benchmarks and stress tests. The server specifications: Processor: Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz, and 64 GB of memory RAM. The server utilized a powerful Intel Xeon Gold 6138 processor, which features: 20 physical cores and 40 threads per processor, with a clock speed of 2.00 GHz.

5.3 Evaluation methodology

The performance evaluation will be conducted by running a series of controlled tests on three distinct functions: Hello World, Sleep, and File Hashing. Each function will be executed five times, with each round processing 1000 requests. The objective is to gather comprehensive performance data under a consistent load for each function, enabling a robust comparison between two configurations of the system: Graalvisor and Graalvisor optimized with Profile-Guided Optimization (PGO).

Following each test iteration, detailed performance metrics will be extracted from the files generated during execution. The metrics under consideration include throughput, average response time, the 90th percentile (P90) response time, another key metric is the cold start latency, and the binary size of each function will be recorded, as the size of the compiled binaries can affect both deployment time and cold start performance. Smaller binaries tend to result in faster startup times, whereas larger binaries can introduce delays, particularly during the initial loading phase.

The primary goal of these tests is to compare the performance of the baseline Graalvisor with that of the Graalvisor optimized using PGO for each function. PGO enhances performance by utilizing runtime data (profiles) to optimize frequently executed code paths. By comparing these two versions, we aim to quantify the benefits of PGO in terms of throughput, response times, cold start improvements, and binary size reductions.

During the execution of each function, it will pass through the entire Hydra platform pipeline. This pipeline involves multiple stages, including request handling, execution scheduling, and function instantiation, before reaching the final stages where the function is either executed by Graalvisor or Graalvisor PGO-optimized. By ensuring that each function undergoes the same processing steps through the platform pipeline, the comparison between Graalvisor and Graalvisor PGO-optimized will be fair and consistent.

Through analyzing the results of the five test rounds for each function, we expect to uncover insights into how PGO impacts various performance aspects under different workloads. This analysis will help demonstrate the trade-offs between standard compilation and profile-guided optimization within the context of serverless computing, and ultimately determine which configuration offers the best overall performance across a range of use cases.

5.4 Metrics

In evaluating the performance of a system, several key metrics are critical to understanding how well the system operates under various workloads. Among these metrics, throughput, average response latency, and 90th percentile latency (P90) provide comprehensive insights into the system's capacity, efficiency, and user experience. By considering these metrics in combination, we can gain a thorough understanding of the system's performance characteristics and its ability to handle various workloads efficiently.

- **Throughput:** Throughput is a crucial metric in performance testing that measures the volume of transactions or requests handled by the system in a specific period. It helps assess the system's capacity, identify bottlenecks, and ensure the application can scale effectively under load. In the context of benchmarks, the number of requests the application can process every second will be the unit of measurement to validate the tests.

- **Average response latency:** Average latency in the context of performance testing refers to the average time it takes for a system or application to process a request and return a response. It is a key metric that is used to evaluate how quickly the application reacts to user requests or transactions. Response latency is the time between a client sending a request to the server and the client receiving a response. The average response latency is the mean value of all individual response times measured during a test. The unit used for the average latency is milliseconds.

- **90th percentile latency (P90):** 90th percentile latency (P90), also known as tail latency, is a key performance metric used to assess the distribution of response times in an application. It refers to the maximum latency (or response time) for 90% of all requests made during a performance test. In other words, 90% of the requests have a latency less than or equal to this value, while the remaining 10% may take longer. Monitoring tail latency is crucial because it highlights the performance experienced by the slower subset of requests, which can significantly impact user satisfaction. The unit used for the 90th percentile latency is milliseconds.

- **Binary size:** Final size of files generated after the Shared Object build process with and without optimization. It will be used to find out if there is also any advantage related to the final file size. The unit is in mega bytes (MB).

- **Cold start:** In the context of performance testing a JVM application, the Cold Start metric refers to the time it takes for the application to initialize and become fully functional when started from scratch. Cold start performance is critical for applications that need to scale dynamically or restart frequently, such as microservices in cloud environments, serverless functions, or applications using container orchestration. The unit used is milliseconds.

5.5 The advantage of having Profile-Guided-Optimization

Using Profile-Guided Optimization is crucial for the Native Image performance to be equivalent (in terms of throughput) to a JIT-compiled application. Using one or several profiles will enable the possibility to hint the AOT compiler the best way to compile the code to binary. The goal of this section is to measure the performance difference between Graalvisor, and Graalvisor PGO optimized.

The first graph shown in Figure 5.1, illustrates the throughput of the benchmarks presented in Section 5.1 between Graalvisor vs Graalvisor PGO optimized.

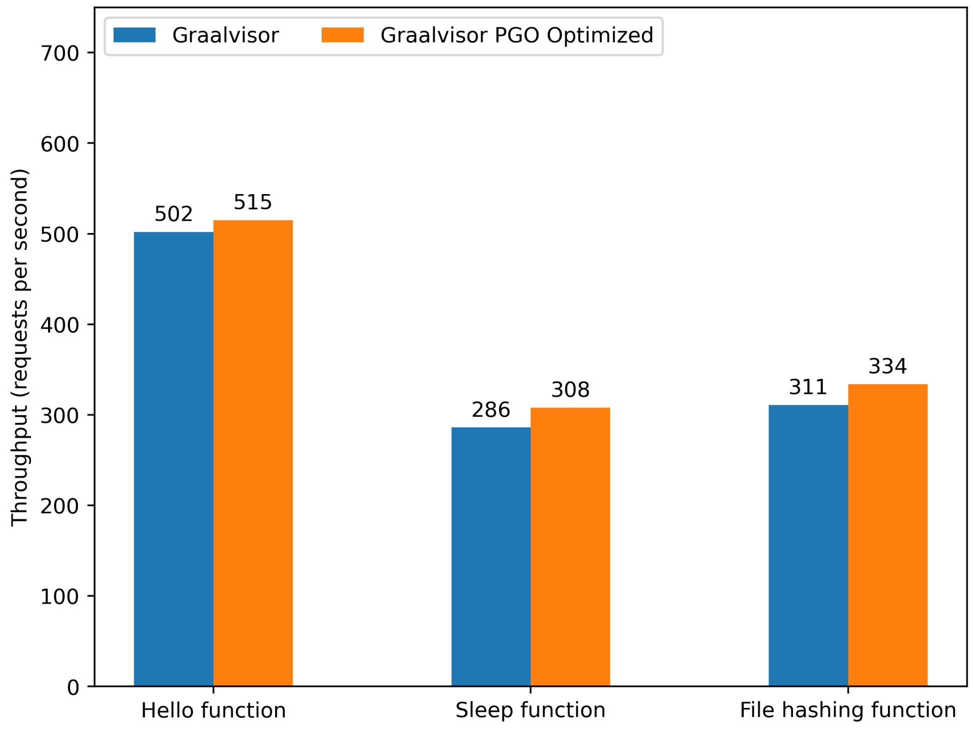

Figure 5.1: Throughput Graalvisor vs Graalvisor PGO.

In this benchmark, only one profile was used as a basis for optimization and it can be seen that the Hello, Sleep and File hashing functions made it possible to obtain a gain of 2.58%, 7.69% and 7.39% in throughput respectively.

The metric depicted in Figure 5.2 represents the average latency. Lower average latency values indicate that, on average, requests are being processed more quickly, leading to a more responsive and efficient system.

In the graph, the Hello, Sleep, and File Hashing functions exhibit reductions in average latency. Specifically, the Hello function achieved a reduction of 4.52%, indicating a slight but measurable improvement in response. The Sleep function saw a more substantial latency reduction of 7.65%, while the File Hashing function also showed a meaningful decrease of 7.16%.

Lower average latencies demonstrate an improvement in overall system efficiency, particularly in everyday usage, where reducing the average processing time leads to smoother interactions and a better user experience.

The metric represented in Figure 5.3 corresponds to the 90th percentile (P90) latency, a critical

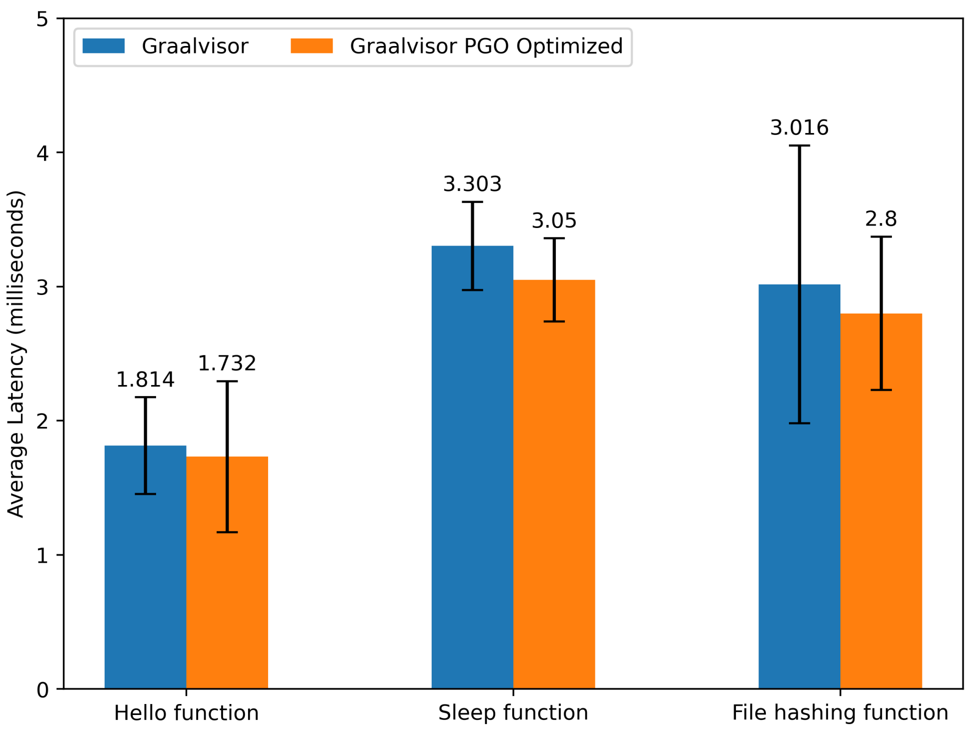

Figure 5.2: Average latency Graalvisor vs Graalvisor PGO.

measure of system performance that reflects the time by which 90% of all requests are completed. Lower values for this metric indicate better performance, as they suggest that the majority of requests experience shorter delays.

In the chart, it is evident that the Hello, Sleep, and File Hashing functions show notable reductions in latency. Specifically, the Hello function achieved a modest decrease in latency of 3.87%, while the Sleep function demonstrated a more significant reduction of 9.61%. The File Hashing function, which processes more intensive computational tasks, also improved with a 7.45% reduction in latency.

These improvements suggest that optimization strategies applied to these functions have been effective, especially in the context of high-load scenarios where minimizing latency spikes can greatly enhance the user experience. The consistent reduction across all three functions indicates that they are now better equipped to handle requests more efficiently, particularly when subjected to high-traffic or resource-intensive operations. This contributes to smoother overall system performance and potentially higher throughput.

The metric shown in Figure 5.4 represents the cold start latency, a critical performance measure for applications running in environments such as serverless computing where the time it takes to initialize an instance can greatly impact response times.

Cold start latency reflects the delay introduced when a function is invoked for the first time, particularly in environments like the JVM where initialization overhead can be significant.

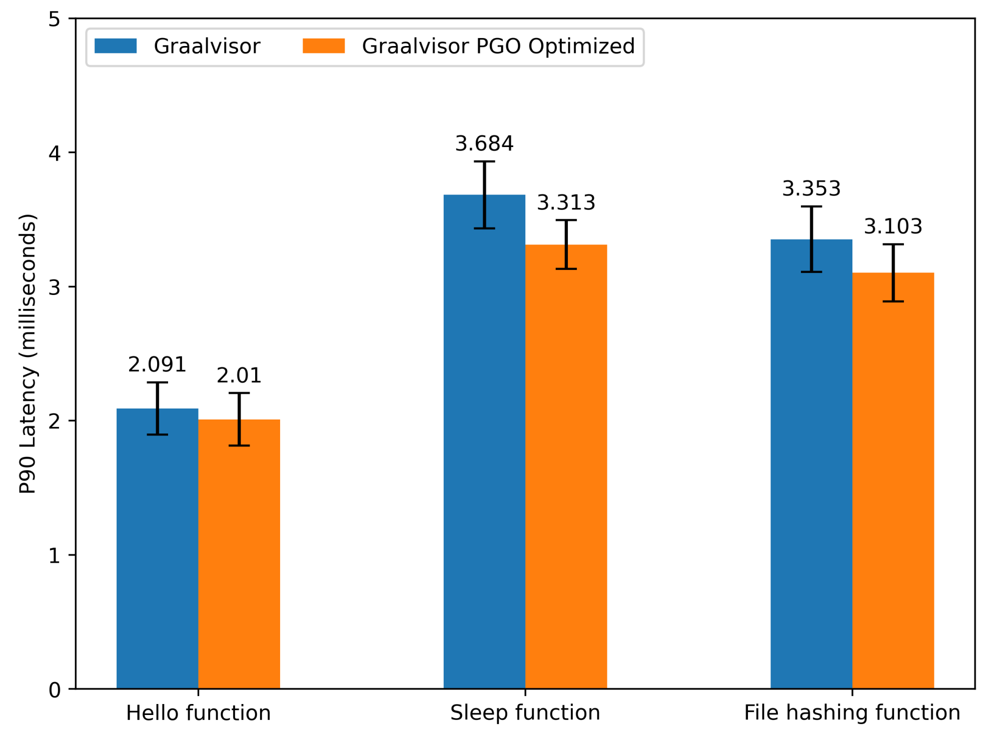

Figure 5.3: 90th percentile latency Graalvisor vs Graalvisor PGO.

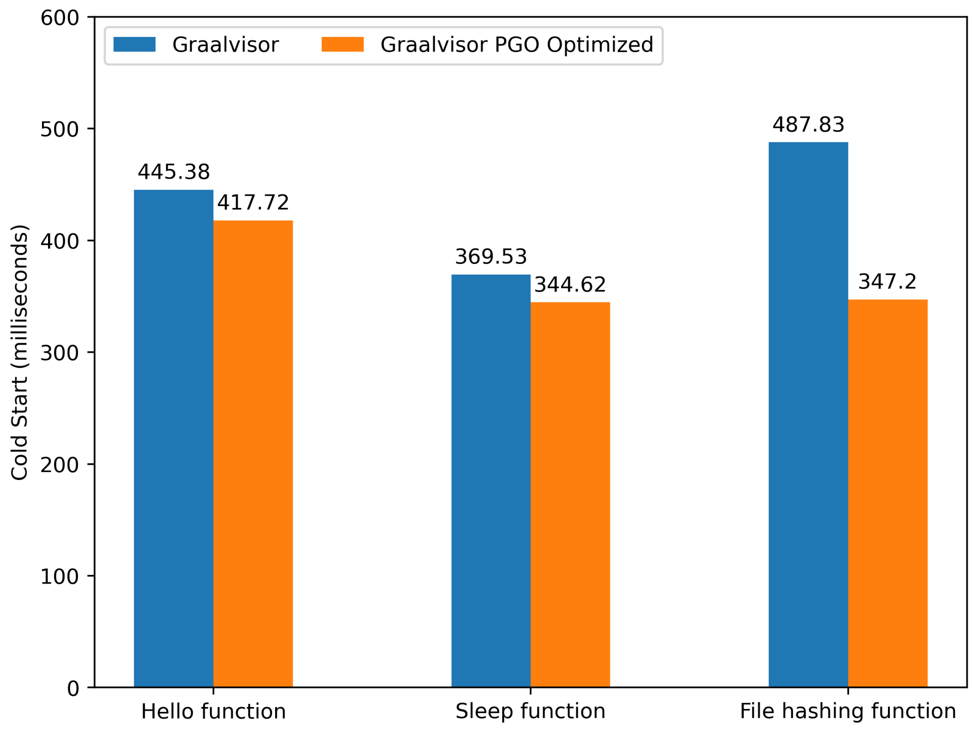

Figure 5.4: Cold start latency Graalvisor vs Graalvisor PGO.

In the chart, it is clear that the Hello, Sleep, and File Hashing functions have achieved notable reductions in cold start latency. The Hello function saw a decrease of 6.21%, indicating a moderate improvement in initialization speed. The Sleep function experienced also a similar reduction of 6,74%, demonstrating that optimizations have lowered the overhead involved in starting this function. The File Hashing function showed a considerable decrease in cold start latency, with a reduction of 28.82%.

These improvements are crucial, particularly in environments where cold starts can severely impact performance during high-demand periods. By reducing the cold start latency, these functions can be invoked more quickly after periods of inactivity, improving overall system responsiveness and decreasing the time users spend waiting for services to initialize.

The graph in Figure 5.5 shows a reduction in the final file size. The reduction in binary file size when using profile guided optimizations in GraalVM can be attributed to several optimizations that target dead code elimination, better inlining decisions, class and method pruning, and more efficient use of memory and resources. These optimizations result in a smaller and more efficient native image tailored to the actual runtime behavior of the application.

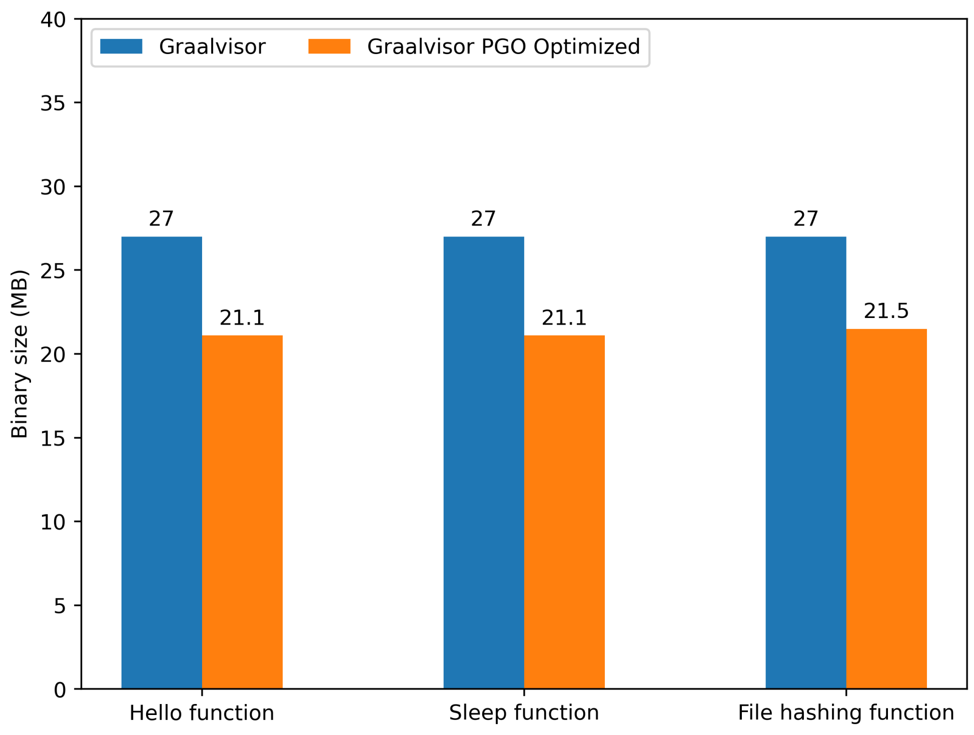

Figure 5.5: Binary size Graalvisor vs Graalvisor PGO.

5.6 The minimum number of invocations of a function to have an effective profile

The platform must know the minimum number of invocations necessary to have a good profile. It is unreasonable to have a platform that performs more profiling than necessary. Detecting the balance between the number of executions and a good profile is crucial for the overall performance and response time of the function. Executing a function with profiling enabled could generate a reduction in performance and increase the overall cost, therefore excessive execution must be avoided. In sum, the next experiments show the metrics that compare optimization using only one profile, ten profiles, a hundred profiles, and a thousand profiles. Therefore, it is possible to identify the trade-off of using different amounts of profiling information. This way, it is possible to identify the best threshold when moving from one stage in the pipeline and minimize the impact during the profile collection period.

In the following experiments, throughput, average latency, P90 latency, and binary size were evaluated by running 1, 10, 100, and 1000 profiles, each conducted over 5 rounds of 1000 requests. This approach allows for a comprehensive analysis of how varying the number of profiles impacts these key performance metrics. The results of these experiments will provide insight into the relationship between profile count and performance, helping to identify optimal strategies to utilize profiling data in optimization efforts.

5.6.1 Throughput per function

As shown in the Figure 5.6 the results indicate that, despite the increasing number of profiles utilized during the optimization, the throughput remains largely consistent across all tested scenarios, exhibiting only minor variations. This suggests that the function's performance is stable and that the additional profiles do not lead to significant improvements in throughput.

The findings imply that the benefits of using a higher number of profiles may be limited in this context, as the throughput does not show substantial enhancement beyond the use of a single profile. This observation underscores the importance of efficiency in the profiling process; gathering excessive profiles may not yield the expected performance gains and could lead to unnecessary complexity.

Overall, these results highlight the effectiveness of PGO in optimizing function throughput while also suggesting that, after a certain point, increasing the number of profiles may not contribute to further performance benefits. Future optimizations may focus on refining the selection and use of profiles rather than simply increasing their quantity.

that multiple profiles do not negatively affect performance.

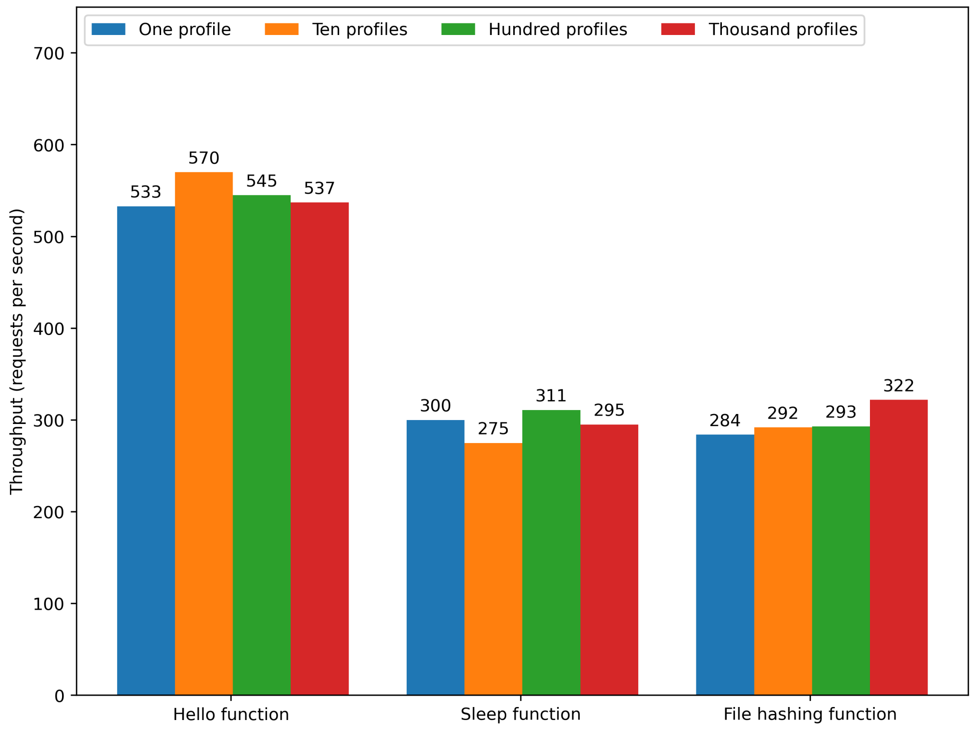

Figure 5.6: Throughput Graalvisor PGO optimized.

5.6.2 Average and P90 latency per function

As shown in Figure 5.7 for the Hello function, the average latency remains consistently low across all profile configurations, with values hovering around 1.58 to 1.686 milliseconds, demonstrating minimal variance regardless of the number of profiles. For the Sleep function, a slightly larger difference is observed depending on the number of profiles. The File hashing function shows similar behavior, with latencies ranging between 2.919 and 3.324 milliseconds, suggesting the marginal impact of additional profiles on this function's performance. The results of this graph suggest that, in general, the number of profiles used for optimization has minimal impact on latency as more profiles are introduced.

As shown in the Figure 5.8 the P90 latency is slightly higher than the average latency across all functions and profile counts, reflecting the higher latency observed in the slowest 10% of executions. For the Hello function, the P90 latency varies from 1.853 milliseconds to 1.93 milliseconds, a small range that indicates stable performance across different profile configurations. The same behavior can be observed for the other Sleep function ranging from 3.337 to 3.769 milliseconds and File hashing function ranging from 3.229 to 3,741 milliseconds. Overall, this graph underscores the importance of performance optimization for improving the tail-end latency of executions, particularly for functions that involve more complex operations.

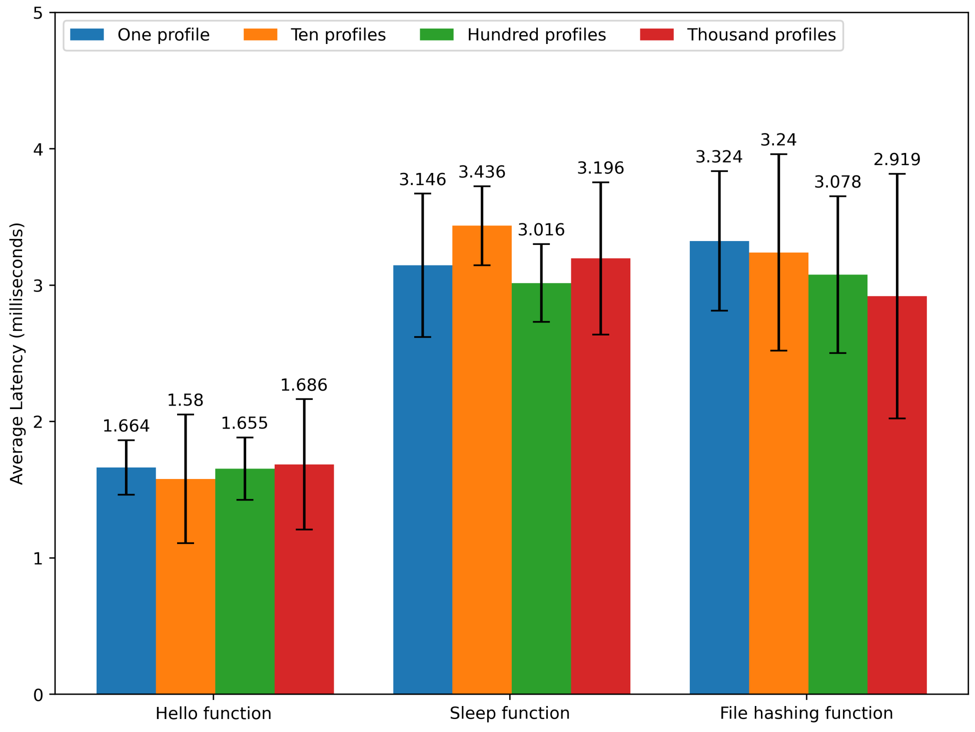

Figure 5.7: Average latency Graalvisor PGO optimized.

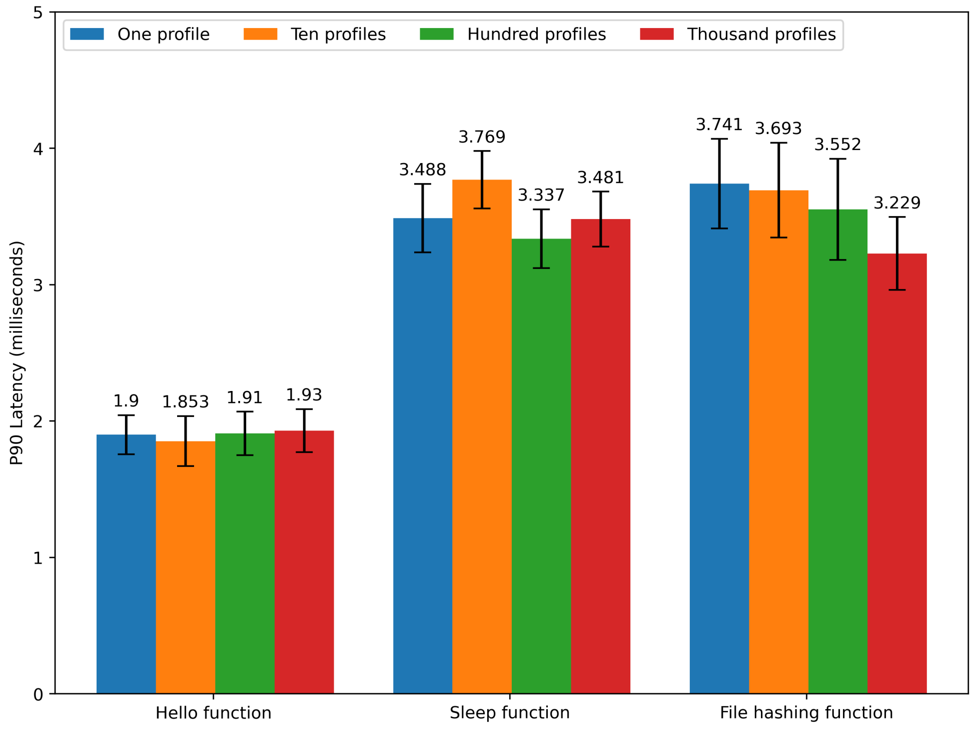

Figure 5.8: P90 latency Graalvisor PGO optimized.

5.6.3 Binary size

As shown in the Figure 5.9 the binary size remains unchanged even with the inclusion of additional profiles. This observation suggests that, regardless of the number of profiles used during the optimization process, there is no noticeable reduction in the final file size. The size of the binary is unaffected by the profiling strategy, implying that the increased number of profile data does not translate into a more compact executable.

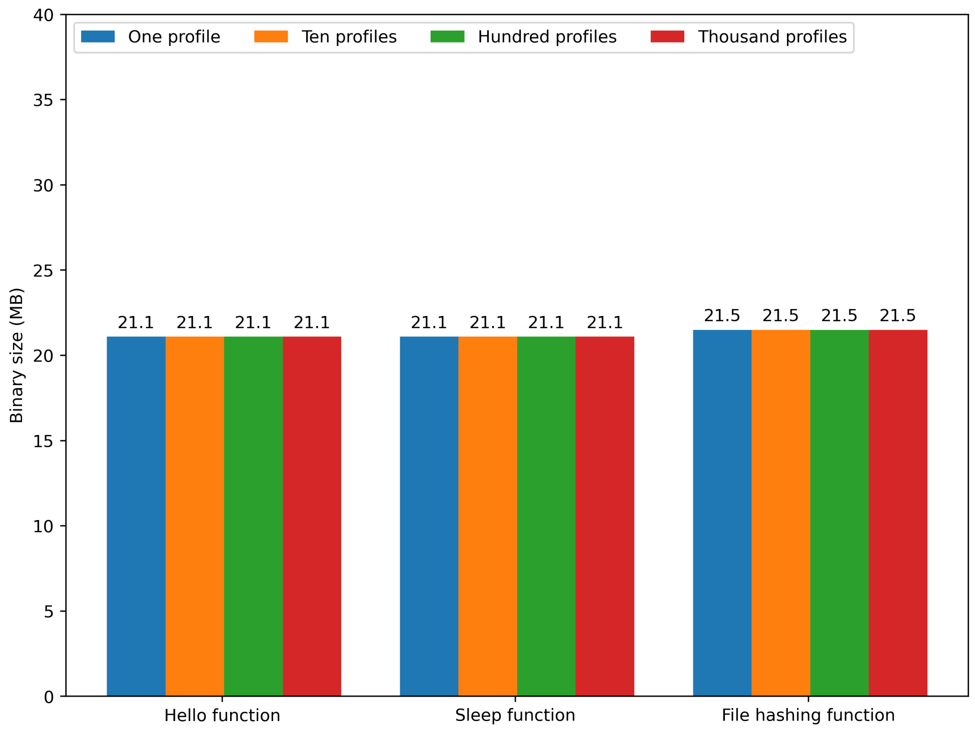

Figure 5.9: Binary size Graalvisor PGO optimized.

5.7 Combining profiles from different functions

It is crucial to determine whether multiple profiles from different functions can be combined, given that serverless infrastructure can deploy a containerized function on various types of machines with differing hardware. The quality, in terms of throughput of the final compiled code, will be assessed by comparing merged profiles with non-merged profiles. The following sequence of graphs shows the throughput for the hello, sleep, and file hashing functions combined with different profiles and also with the profile generated by it's own function execution.

In the first graph, Figure 5.10, the Hello function runs in Graalvisor without optimizations, using the Sleep function profile, the File hashing function profile, and its own profile. Despite using different

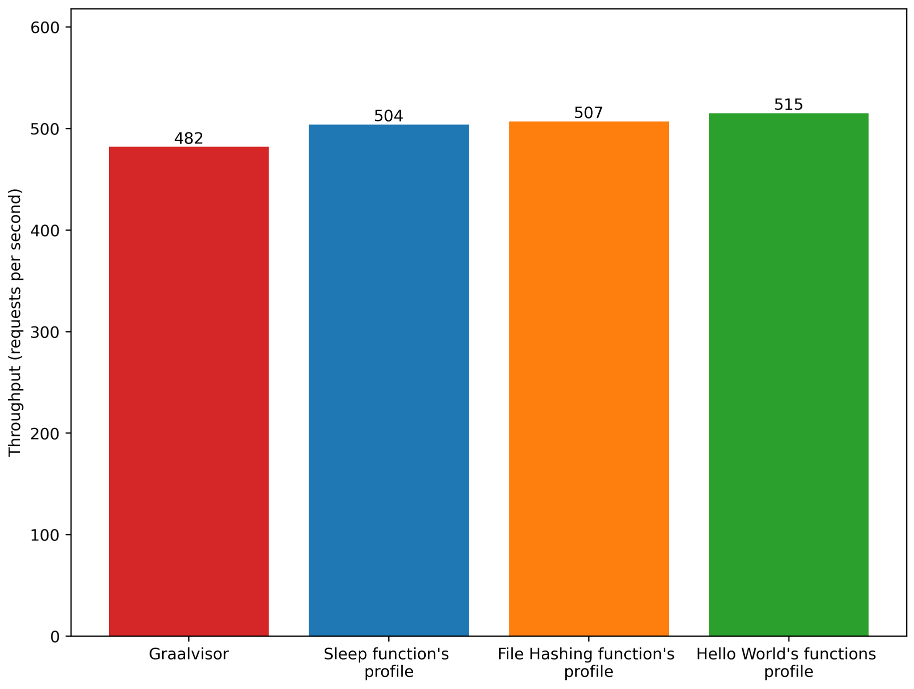

Figure 5.10: Hello Function throughput with merged profiles.

function profiles, the performance gain remains within a similar range when compared to the profiles of the function being executed. This suggests that even with mismatched profiles, there is still a notable performance improvement. It is important to note that, in this comparison, the profile generated directly by the function's own execution delivers better performance than the others.

In the second graph, Figure 5.11 and in the third graph, Figure 5.12 a similar combination of profiles is applied. In the second graph, the Sleep function uses the Hello and File hashing function profiles, while in the third, the File hashing function uses the Hello and Sleep function profiles. For both functions also includes its own profile. The behavior remains consistent, with only slight differences observed when combining the File hashing profiles with the Sleep function and vice versa. These tests indicate that performance is stable even when using profiles from other functions.

It is worth noting that, in this comparison, the profile produced from the function's own execution delivers higher performance than the other profiles. This indicates that while it can be beneficial to use profiles from other functions to potentially enhance performance and save time on profile collection, even greater throughput can be achieved when the optimization utilizes profiles generated from the function itself.

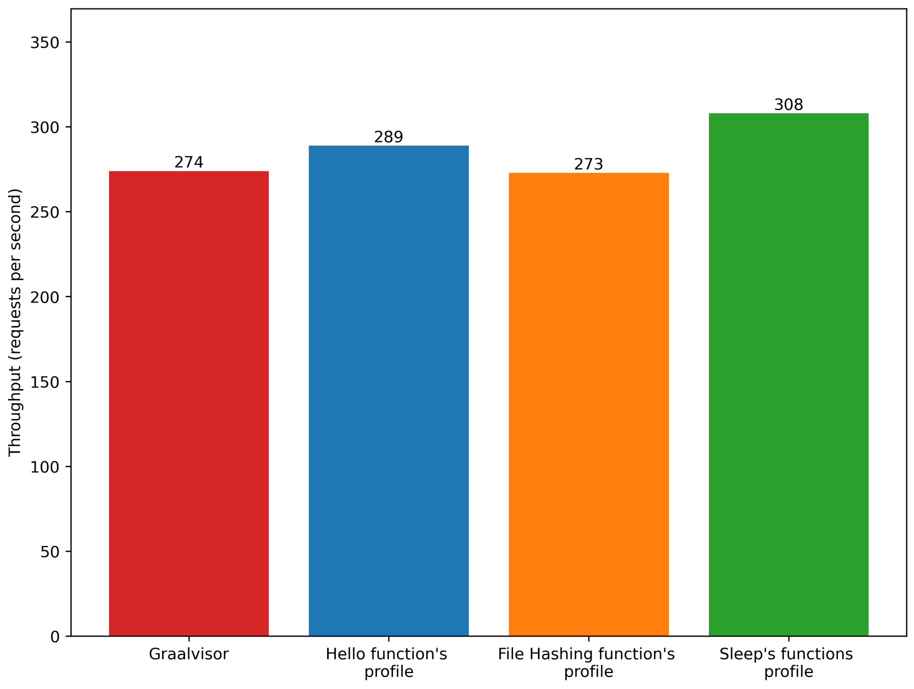

Figure 5.11: Sleep function throughput with merged profiles.

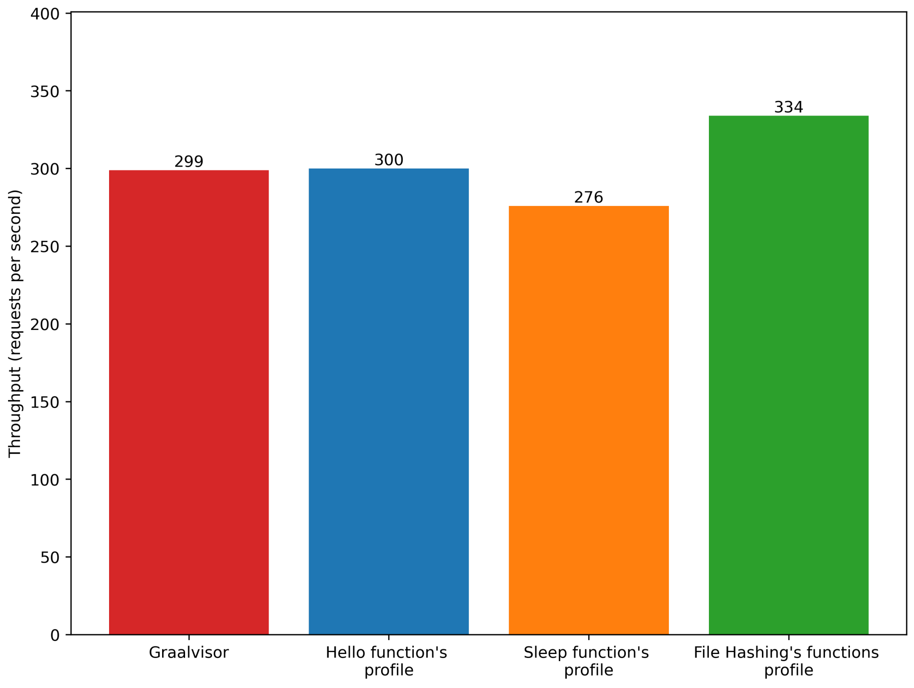

Figure 5.12: File hashing function throughput with merged profiles.

All tests were carried out using a single profile file for each function, with each function going through a total of five rounds of one thousand executions each. The ability to combine profiles from different functions presents a particularly valuable feature. This capability becomes highly advantageous in scenarios where the Lambda Manager already has a diverse set of functions running and maintains an extensive portfolio of stored profiles. In such cases, these pre-existing profiles can be utilized to optimize performance even before the system has collected any specific profile data from the newly executing function. This not only saves time in the profile collection process, but also allows for immediate optimizations, enhancing efficiency and reducing potential delays in function execution. By leveraging profiles from previously executed functions, the system can potentially achieve improved performance right from the start, without the need to wait for the completion of a new profiling phase.

6

Conclusion

Contents

6.1 Key Findings and Contributions 48
6.2 System Limitations and Future Work 48
6.3 Conclusion 49

6.1 Key Findings and Contributions

The main contributions of this thesis lies around improving the performance of serverless computing through a novel approach that leverages GraalVM's capabilities and profile-driven optimization. In serverless environments, function execution typically relies on Just-In-Time (JIT) compilation, where optimizations are performed dynamically during runtime. However, these optimizations are lost after each execution due to the stateless nature of serverless architectures, leading to recurring cold starts and a lack of accumulated performance benefits over time. On the other hand, Ahead-of-Time (AOT) compilation addresses cold starts by compiling functions in advance but does not achieve the same level of optimization as JIT, particularly for frequently executed functions.

The key finding of this work is that by using GraalVM, along with execution profiles collected via code instrumentation, it is possible to recompile serverless functions to achieve a performance that surpasses traditional AOT methods. The thesis demonstrated, through extensive experimentation, that functions optimized with profiles exhibited significant improvements in throughput, response time, and cold start performance. Importantly, the study showed that even a single execution profile is sufficient to achieve these optimizations, offering equivalent performance to multiple profiles while reducing the overhead associated with profile collection in instrumented mode, which is typically slower.

The experimental results also revealed that the system could use profiles from different functions to generate optimized code, indicating that previously stored profiles could be reused effectively to reduce overhead. This suggests a potential for broader application of profile-based optimization across various serverless functions. Overall, the thesis makes a substantial contribution by showing how serverless computing can be optimized without sacrificing the flexibility and scalability that make it appealing for cloud-native applications.

6.2 System Limitations and Future Work

While the proposed system demonstrated promising results, the system handled relatively simple serverless functions. These functions lacked the complexity of more robust applications such as multi-layer web applications, APIs, or data-intensive workloads. As such, the experiments did not explore how the optimization techniques might perform in real-world scenarios involving more sophisticated and resource-demanding serverless functions.

Another limitation is that the thesis did not include a comparison between the optimized binary files generated by GraalVM and shared object files, which could provide additional insights into how the choice of compilation output affects performance, file size, and memory usage. Furthermore, the benchmark tests were conducted on an x86 server architecture. While the entire platform is compatible with ARM architecture, it remains uncertain whether profiles generated from different architectures can be combined to optimize a function.

For future work, there are several opportunities to extend and enhance the proposed system. One major area for improvement is applying the optimization approach to more complex applications and testing it under different workloads.

Another potential direction for future research lies in the refinement of profiling methods. Current profiling techniques require the system to run in an instrumented mode, which introduces some overhead. Exploring advanced machine learning techniques to predict optimal profiles for functions could also be a valuable extension, allowing for proactive optimizations rather than reactive ones.

6.3 Conclusion

In conclusion, this thesis demonstrated that the performance of serverless computing could be significantly improved by leveraging profile-driven optimizations through GraalVM. The experiments showed clear benefits in throughput, response time, cold start times, and overall file size, confirming that serverless functions can be optimized in a way that maintains both the flexibility of the serverless model and the high performance typically associated with Just-In-Time compiled applications.

The approach of using a single profile for optimization proved effective, offering a streamlined solution to reduce overhead while still reaping the benefits of optimized code execution. The results indicate that serverless environments can benefit from advanced compilation techniques, particularly when applied thoughtfully with profile-based insights.

Despite its limitations, such as its focus on simple functions and the lack of comparison between the optimized binary files generated by GraalVM and shared object files, the thesis paves the way for future work that could extend these findings to more complex applications. Further exploration could yield even greater performance improvements, making serverless a more attractive option for a wider range of applications in cloud computing. This work represents a key step in optimizing serverless architectures, potentially enabling them to combine flexibility and scalability with high-performance execution.

Bibliography

[1] S. Kohli, S. Kharbanda, R. Bruno, J. Carreira, and P. Fonseca, "Pronghorn: Effective checkpoint orchestration for serverless hot-starts," in *Proceedings of the European Conference on Computer Systems (EuroSys '24)*, Edinburgh, Scotland, UK, April 2024.

[2] J. Carreira, S. Kohli, R. Bruno, and P. Fonseca, "From warm to hot starts: Leveraging runtimes for the serverless era," in *HotOS '21: Workshop on Hot Topics in Operating Systems*. Ann Arbor, Michigan, USA: ACM, June 2021, pp. 58–64.

[3] M. Shahrad, R. Fonseca, I. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 205–218. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad

[4] S. Newman, *Monolith to Microservices, Evolutionary Patterns to Transform your Monolith.* Sebastopol, CA: O'Reilly Media, Incorporated, 2019.

[5] F. G. C. de Sousa, "Performance isolation in graalvm native image isolates," Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, November 2022. [Online]. Available: https://fenix.tecnico.ulisboa.pt/downloadFile/1970719973969181/81120-Filipe-Sousa-dissertacao.pdf

[6] S. Wang, "Thin serverless functions with graalvm native image," Master's thesis, ETH Zurich, July 2022. [Online]. Available: https://www.research-collection.ethz.ch/handle/20.500.11850/480335

[7] D. J. Frickert, "Photons@graal - enabling efficient function-as-a-service on graalvm," *Instituto SUperior Técnico*, pp. 124:1–124:23, 2022. [Online]. Available: https://web.tecnico.ulisboa.pt/~ist14191/repository/MSc_Extended_Abstract___FaaS_GraalVM_FINAL.pdf

[8] G. D. Team. (2022) GraalVM Native Image: Profile-Guided Optimizations (PGO). Accessed 18th June 2024. [Online]. Available: https://www.graalvm.org/22.0/reference-manual/native-image/PGO/

[9] A. Khrabrov, M. Pirvu, V. Sundaresan, and E. de Lara, “Jitserver: Disaggregated caching JIT compiler for the JVM in the cloud,” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 869–884. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/khrabrov

[10] G. Ottoni and B. Liu, “Hhvm jump-start: Boosting both warmup and steady-state performance at scale,” in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE Computer Society, 2021, pp. 340–350. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CGO51591.2021.9370314

[11] X. Xu, K. D. Cooper, J. Brock, Y. Zhang, and H. Ye, “Sharejit: Jit code cache sharing across processes and its practical implementation,” *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 124:1–124:23, 2018. [Online]. Available: https://doi.org/10.1145/3276489

[12] M. Arnold, A. Welc, and V. Rajan, “Improving virtual machine performance using a cross-run profile repository,” in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. San Diego, CA, USA: ACM, October 2005, pp. 297–311.

[13] S. Ivanenko, J. Stevanovic, V. Jovanovic, and R. Bruno, “Hydra: Virtualized multi-language runtime for high-density serverless platforms,” *arXiv preprint arXiv:2212.10131*, 2022. [Online]. Available: https://arxiv.org/abs/2212.10131

[14] I. MinIO, “Minio container documentation,” 2024, accessed: 2024-10-03. [Online]. Available: https://min.io/docs/minio/container/index.html

[15] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, “Sebs: A serverless benchmark suite for function-as-a-service computing,” in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 64–78. [Online]. Available: https://doi.org/10.1145/3464298.3476133