

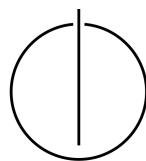
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis

**Towards Demystifying Intra-Function  
Parallelism in Serverless Computing**

**Michael Kiener**







DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis

# **Towards Demystifying Intra-Function Parallelism in Serverless Computing**

## **Entmystifizierung von Intra-Funktion Parallelismus in Serverless Computing**

Author:	Michael Kiener
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	Anshul Jindal, Mohak Chadha
Submission Date:	15.09.2021



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich,

Michael Kiener

## Acknowledgments

I would first like to thank my thesis advisors Mohak Chadha and Anshul Jindal, who made it possible for me to research this topic. They provided me with regular feedback and were always willing to quickly help me when I had questions or got stuck.

I would also like to express my gratitude to Prof. Dr. Michael Gerndt, who sparked my interest in the topics of cloud computing and parallel computing, with his very interesting and engaging lectures and other courses.

A very special thank you to my dear friend Julian Frielinghaus, who always had an open ear to my problems and gave me advice throughout the entire process of this thesis and beyond.

Lastly, I would like to thank my family, which supported me throughout my whole studies and made it possible for me to pursue my studies in the first place. This thesis would not have been possible without them.



# Abstract

Serverless computing offers a pay-per-use model with high elasticity and automatic scaling for a wide range of applications. One of the key concepts of serverless is that the cloud provider manages and allocates all necessary infrastructure for the application to run. Since commercial cloud providers abstract most of the underlying infrastructure, these services work similarly to black-boxes. Due to this, developers can quickly allocate more resources, especially virtual CPUs (vCPUs), than they effectively use without knowing. By not utilizing the full hardware that is available, the developer pays more money than necessary.

In this thesis, we analyze the impact of parallelization of compute-intensive workloads within single function and container instances for AWS Lambda, Google Cloud Functions (GCF), and Google Cloud Run (GCR). Furthermore, we give insights into the underlying infrastructure and research the correlation between allocated hardware threads and vCPUs in serverless environments. We conduct our experiments with 5 different benchmarks and collect metrics, such as execution time and memory consumption, about each service and environment. Our results show, that the amount of allocated vCPUs does not always equal the number of hardware threads. Furthermore, we show that by parallelizing, applications can be speedup close to the amount of vCPUs allocated. We measured the execution times between single- and multi-threaded applications and measured speedups up to 5.4x for AWS, 2.1x for GCF, and 3.7x for GCR. The resulting maximum cost savings we could observe were up to 81% for AWS Lambda, 49% for GCF, and 71% for GCR.





# Contents

<b>Acknowledgments</b>	<b>iii</b>
<b>Abstract</b>	<b>v</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Problem . . . . .	3
1.2 Motivation . . . . .	4
1.3 Outline . . . . .	5
<b>2 Background</b>	<b>7</b>
2.1 Serverless Services . . . . .	7
2.1.1 Function-as-a-Service . . . . .	8
2.1.2 Container-as-a-Service . . . . .	8
2.2 Parallel Computing . . . . .	9
2.2.1 Processing Units . . . . .	9
2.2.2 Threads . . . . .	10
2.2.3 Shared Memory Parallelism . . . . .	10
<b>3 Related Work</b>	<b>11</b>
3.1 Distributed Parallelism using FaaS . . . . .	11
3.2 Performance evaluations of FaaS platforms . . . . .	12
<b>4 Evaluated Services</b>	<b>15</b>
4.1 AWS Lambda . . . . .	15
4.1.1 Invocation Architecture . . . . .	15
4.1.2 Worker Component . . . . .	16
4.1.3 Firecracker . . . . .	17
4.1.4 Instance Life-cycle . . . . .	18
4.1.5 Function Configuration . . . . .	18
4.2 Google Cloud Functions . . . . .	19
4.2.1 Google Cloud Platform Cluster Architecture . . . . .	19
4.2.2 gVisor . . . . .	20
4.2.3 Function Configuration . . . . .	21
4.3 Google Cloud Run . . . . .	21
4.3.1 Knative . . . . .	22
4.3.2 Container Configuration . . . . .	22
4.3.3 Pricing Model . . . . .	23

<b>5</b>	<b>Evaluation Setup</b>	<b>25</b>
5.1	Runtimes . . . . .	25
5.1.1	C++ . . . . .	25
5.1.2	Go . . . . .	26
5.1.3	Java . . . . .	28
5.2	Benchmarks . . . . .	29
5.2.1	Microbenchmarks . . . . .	30
5.2.2	Applications . . . . .	31
5.2.3	Input Overview . . . . .	33
5.3	Service Configurations . . . . .	34
5.4	Evaluation Process . . . . .	35
5.5	Metrics . . . . .	36
5.5.1	Execution Time . . . . .	38
5.5.2	Memory Usage . . . . .	38
<b>6</b>	<b>Results</b>	<b>39</b>
6.1	Observations . . . . .	39
6.1.1	Threads and Virtual CPUs . . . . .	39
6.1.2	Memory . . . . .	40
6.2	Performance Evaluations . . . . .	42
6.2.1	Sequential Speedup . . . . .	42
6.2.2	Parallel Speedup . . . . .	44
6.2.3	Function Execution Times . . . . .	47
6.2.4	Cost Comparison . . . . .	48
6.2.5	Impact of Cold starts . . . . .	52
<b>7</b>	<b>Discussion</b>	<b>55</b>
7.1	Costs . . . . .	55
7.2	Runtimes . . . . .	57
7.3	Tradeoff Cost vs. Speedup . . . . .	57
<b>8</b>	<b>Conclusion</b>	<b>59</b>
8.1	Shortcomings . . . . .	59
8.2	Future Work . . . . .	60
8.2.1	Services . . . . .	60
8.2.2	Runtimes . . . . .	60
8.2.3	Benchmarks . . . . .	60
8.2.4	Inter- and Intrafunction parallelism . . . . .	60
	<b>List of Figures</b>	<b>63</b>
	<b>List of Tables</b>	<b>65</b>
	<b>Listings</b>	<b>67</b>

**Bibliography**

**69**



## **Acronyms**

<b>API</b>	Application Programming Interface
<b>AWS</b>	Amazon Web Services
<b>ACF</b>	Azure Cloud Functions
<b>CaaS</b>	Container-as-a-Service
<b>CLI</b>	Command Line Interface
<b>EC2</b>	Elastic Compute Cloud
<b>FaaS</b>	Function-as-a-Service
<b>GC</b>	Garbage Collector
<b>GCF</b>	Google Cloud Functions
<b>GCP</b>	Google Cloud Platform
<b>GCR</b>	Google Cloud Run
<b>HPC</b>	High Performance Computing
<b>JIT</b>	Just-in-Time
<b>KVM</b>	Linux Kernel-based Virtual Machine
<b>LB</b>	Load Balancer
<b>MVM</b>	Micro Virtual Machine
<b>OCI</b>	Open Container Initiative
<b>OS</b>	Operating System
<b>SMT</b>	Simultaneous Multithreading
<b>S3</b>	Simple Storage Service
<b>vCPU</b>	virtual CPU
<b>VM</b>	Virtual Machine
<b>VMM</b>	Virtual Machine Monitor



# 1 Introduction

Cloud computing has been growing steadily, especially since the introduction of Amazon Web Services (AWS) back in 2006 [53]. The major commercial cloud providers offer a variety of cloud services, from virtual machines (VMs) to storage, networking, and more. There are several reasons why many companies choose to use different cloud computing services. The first one is time to market [34]. Especially for new companies, it can take a long time to buy the hardware and set up the necessary infrastructure required to run internet-facing services. The second reason, and arguably the biggest, is to save cost. Many case studies show that migrating from on-premise infrastructure to cloud environments can have huge cost benefits [32] [1] [14]. Cloud services are also well equipped to deal with increasing demand. It is very fast and easy to allocate additional resources in the cloud, rather than buying and integrating additional hardware. Many cloud services even offer the ability to auto-scale. Auto-scaling means (de-)allocation resources according to the demand, without any manual actions required.

One of the more recent developments in Cloud computing is serverless. One of the first serverless services was introduced in 2014 with AWS Lambda [12]. Serverless, and Function-as-a-Service (FaaS) more specifically, improve the advantages of cloud computing even more. Serverless removes the requirement to allocate hardware and is built to automatically scale with each request.

Since the demand for computing resources is steadily growing, the hardware also needs to be more powerful to meet those demands. However, the growth of single-core performance for processors is declining across processor generations, while the amount of cores on a single processor is growing [50]. Therefore, a single program is only able to utilize the full capacity of a processor, if it is parallel. Since this trend will most likely continue, it is useful to research the efficiency of running parallel programs for different use cases. Furthermore, with the introduction of cloud computing, hardware with many cores is available to anyone, which allows developers to run parallel programs.

## 1.1 Problem

FaaS is already widely used across many different industries, such as web applications, service integrations, or in the field of Internet of Things (IoT) [22]. From a developer's perspective, FaaS works similarly to a black box. The developer uploads a code package to the provider and observes a certain behavior from the function once it is deployed. In most cases, the developer doesn't care what happens in between. That includes scheduling, invocations, and the underlying hardware. It is the responsibility of the

provider that the function runs without issues [7]. Providers usually don't give a lot of insights in terms of the architecture and hardware behind their serverless services. However, not knowing about specific characteristics of the serverless environment can lead to inefficient code. For example, depending on the underlying processor, it can be worth it to vectorize the function code. Vector instructions, also called Single Instruction Multiple Data (SIMD), combine multiple instructions into one, which processes several data elements at once. These SIMD instructions can be executed by specialized hardware on the processor in a single instruction cycle.

FaaS is often used for small, fine grained-tasks. However, providers like AWS Lambda and Google Cloud Functions (GCF)<sup>1</sup> allow the developer to allocate a lot of memory and computing resources to a single function. In both cases, the developer can only configure the function memory and the processing power is scaled by the service provider accordingly. This can be quite intransparent and if the developer isn't careful, he will pay for more resources than he effectively uses.

## 1.2 Motivation

Having a better understanding of the underlying hardware and architecture of serverless environments has several advantages. Not only can it make the functions more efficient, but the developer can also make more informed decisions when configuring his functions. Especially for higher memory function configurations, it is very useful to know more about the hardware resources that are allocated. For example, if the developer knows how many physical cores are allocated to a function, he can make better decisions on how to split the workload among them. Therefore, we want to look at the following 3 points in this thesis:

1. **Architectural and environmental research:** We want to give insights into the underlying infrastructure for each service. Additionally, we want to research the resource allocations in a serverless environment and see how they align with the respective costs that the providers charge. The focus will be on what resources are available for parallel executions within a function.
2. **Cost savings by parallelization:** Given a certain function configuration, we want to research how much cost can be saved by parallelizing the program. Sometimes a function needs a high amount of memory to run a program. These functions are quite costly to run over longer periods, so if the developer can reduce execution times by parallelizing, the cost benefits are also great.
3. **Cost savings by increasing the function configuration:** The last point we want to research is the following scenario: The developer deployed an already parallel program. Can he decrease his cost by increasing the function configuration and therefore decrease the overall execution time?

---

<sup>1</sup><https://cloud.google.com/functions/>



For both 2. and 3. we want to research the conditions that need to be met to experience benefits. For example, for what function configurations can we save costs by parallelizing. Furthermore, we want to formulate conclusions on what the developer has to keep in mind when running parallel programs within a single function.

## 1.3 Outline

In chapter 2, we will introduce the necessary background to understand the evaluated services. In chapter 3, we will present the existing research that has been done in regards to parallelization and performance analysis in serverless environments. In chapter 4, we will describe the services we evaluate and their architecture. Our methodology to evaluate the services is described in chapter 5. Finally, we present the results of our evaluations in chapter 6 and discuss our findings in chapter 7. In chapter 8 we summarize our work and what future work could be done to acquire further insights in this area.



## 2 Background

In this chapter, we introduce the necessary background that is needed to understand serverless services and how we evaluate them. First, we describe in more detail what serverless services are and what their usual characteristics are. Since we run parallel programs within the functions, we also give an overview of parallel computing and the differences between shared and distributed memory parallelism.

### 2.1 Serverless Services

Over the last couple of years, several serverless services have emerged. They all share the following characteristics:

**No server allocation:** As the name serverless suggests, there is no manual allocation of servers by the developer. All underlying servers and infrastructure is managed by the providers. The developer has some options to configure the service, but those options are often limited.

**Automatic scaling:** All serverless services have the option of automatic scaling in one way or another. FaaS usually scales by creating a new function instance for each request that can not be handled by an already existing instance. Other services scale according to the resource utilization of active instances. If certain thresholds are met, instances are started or shut down.

**Pay-per-use:** One of the biggest advantages of serverless is that the developer only pays for what is effectively used. Traditionally in cloud computing, VMs have to be allocated in anticipation of incoming requests. That also means that usually more resources are allocated than needed to deal with sudden spikes in workload. In FaaS, the developer doesn't pay for idle CPU time. If an instance is still running, but not currently handling requests, the provider bears the full cost. Since the provider doesn't want to allocate resources that are not utilized, serverless services are therefore usually scaled down to 0.

**Cold starts:** Since instances can scale down to 0, serverless services increase the phenomenon of cold starts. A cold start is when a request is sent to a serverless endpoint, but there are no active instances that can handle the request. In this case, a delay is introduced to startup a new instance with the function code of the developer.

**Stateless:** Serverless services are inherently built to be stateless. State in this context means persisting data across invocations, for example by writing it to disk. The provider makes no guarantees that consecutive invocations are routed to the same instance, hence there is no guarantee consecutive invocations will be able to see the persisted data.

Although most serverless services share these characteristics, there are still some differences between them. In this thesis, we focus on FaaS and Container-as-a-Service (CaaS), which was introduced more recently.

### 2.1.1 Function-as-a-Service

FaaS realizes the serverless characteristics completely. The developer only writes the function code, that executes the desired functionality. The developer then has some additional options to configure the function. For example, the language of the function and the resource allocation. Everything else, from scheduling to setting up the function environment, is handled by the provider. All major cloud providers support several different languages for their FaaS implementations. Popular languages are Python, Java, and Go. The developer can implement the function in one of the supported languages and then choose one of the provided runtimes. As already mentioned, FaaS provides automatic scaling behavior. Each concurrent request is handled by its own function instance so that no request needs to wait for another request to finish. Providers do offer some options to modify this behavior to some extent, for example, limit the number of concurrent active instances.

### 2.1.2 Container-as-a-Service

CaaS is one of the more recent trends in serverless. Both Amazon and Google have their respective offerings with AWS Fargate<sup>1</sup> and Google Cloud Run (GCR)<sup>2</sup>. Both function very similar to FaaS, but instead of deploying functions, the developer deploys custom Docker images<sup>3</sup>. This has the advantage that the developer has the full flexibility of custom containers. The developer can set up a custom environment inside the Docker container, even for languages that are not supported for the respective FaaS services. Additionally, all external dependencies and libraries can simply be installed in the container image. The developer has also full control over all compilation steps of the program, for example, what compiler to use and with which options. The code running in these containers differs in some ways from the code that would be deployed in a function. The application executed in the container needs to start an HTTP server to actively listen to incoming requests.

---

<sup>1</sup><https://aws.amazon.com/fargate/>

<sup>2</sup><https://cloud.google.com/run/>

<sup>3</sup><https://www.docker.com/>

```
1 /* ... */
2 func main() {
3     // Set function handler for path
4     http.HandleFunc("/", handler)
5     // Determine port for HTTP service.
6     port := os.Getenv("PORT")
7     // Start HTTP server.
8     if err := http.ListenAndServe(":"+port, nil); err != nil {
9         log.Fatal(err)
10    }
11 }
12
13 func handler(w http.ResponseWriter, r *http.Request) {
14     // Execute workload
15 }
16 /* ... */
```

Listing 2.1: Minimal application code for CaaS in Go

Listing 2.1 shows a minimum example of a containerized application in CaaS. In the `main()` function the HTTP server is established, which listens to incoming requests. The developer can then define endpoints that are handled by respective handlers, which the developer also defines. In FaaS, the developer would only need to define the handler function, while the service provider makes sure that the request is delivered to the function handler.

In this thesis, we will focus on GCR as CaaS, since AWS Fargate currently does not automatically scale down to 0, while GCR does. This is a very important factor since if there is always at least 1 instance running, we pay for idle CPU time, which is what the developer usually wants to avoid when going serverless.

## 2.2 Parallel Computing

Since a huge part of this thesis is about the parallel execution of programs, we want to briefly introduce some concepts of parallel programs. Since all our evaluations are done with shared memory parallelism, we also want to explain the concept of it and what the key differences are to distributed memory parallelism.

### 2.2.1 Processing Units

In parallel computing, we divide the program into multiple sub-problems, that can be executed by different processing units concurrently. A multi-core processor has a physical processing unit for each core. Each core provides at least 1 hardware thread that the Operating System (OS) can use to schedule and execute processes. Modern processor

utilize the simultaneous multithreading (SMT) technology to provide 2 hardware threads per physical core. SMT uses context-switching between the 2 threads to increase the overall throughput of the system. For example, if a process, that is executed by a hardware thread, waits for a memory call, the physical core can switch to another hardware-thread and therefore increase the overall utilization of the physical resources. An example for an SMT implementation is Intel Hyper threading [36]. To the OS, a single physical core appears as 2 independent processing units.

### 2.2.2 Threads

Threads can be defined as imperative programs that run concurrently and share a memory space [41]. Although threads share the same memory space, each thread needs some private data, so that it can run independently. For example, each thread has its own program counter or private stack pointers and register values. Many languages offer standardized Application Programming Interfaces (APIs) to create threads. The OS will then schedule and interrupt threads according to available resources. A scheduling algorithm is used by the OS to maintain fairness between threads.

### 2.2.3 Shared Memory Parallelism

Since threads share the underlying memory space, they can access the same variables, except for thread-local, or thread-private, variables. That means, if the program consists of a problem that needs to work on one large array and the array was initialized in the shared memory space, it is available to all threads. This saves a lot of time since the data doesn't have to be replicated.

The workload of a program can also be split with distributed memory parallelism. In this case, the program is executed by independent processes, that may even execute on different physical machines. These processes don't share the same memory space, which is why the data needs to be either: initialized in a distributed fashion; or initialized from the main process and sent to the respective child processes. Both shared and distributed memory parallelism have their respective advantages and disadvantages.

In this thesis, we purely use shared memory parallelism, since we utilize the hardware resources within a given function instance to parallelize the program.

## 3 Related Work

Serverless has not only grown in popularity among developers, but also among the scientific community. In this chapter we present the research that has been done in regards to parallelization and performance evaluations of serverless services.

### 3.1 Distributed Parallelism using FaaS

The majority of previous work on parallelizing applications via FaaS has focused on splitting the workload via separate function instances [52], [17], [18], [37], [48]. Shankar et al. [52] show that the elasticity provided by serverless computing can be used to efficiently execute linear algebra algorithms, which are inherently parallel. They implemented a system to split the algorithm into tasks, which are then executed by AWS Lambda functions. Data is shared between functions via a persistent object store. The performance of their system performance "is within a factor of 2 of optimized server-centric MPI implementations, and has up to 15% greater compute efficiency (total CPU-hours), while providing fault tolerance" [52].

Barcelona-Pons et al. [17] analyze the performance of fork/join workflows using existing services like AWS Step Functions, Azure Durable Functions and OpenWhisk Composer. They found that, while Composer is the best solution as of to date, none of the function orchestration solutions offer suitable performances to execute parallel workloads.

Barcelona-Pons et al. "present Crucial, a system to program highly-concurrent stateful applications with serverless architectures" [18]. The system ports multi-threaded applications to a serverless environment, by leveraging a distributed shared memory layer. Cloud functions are coordinated via shared objects. Their performance results are similar to that of an equivalent Spark cluster.

Jiang et al. [37] developed their own platform LambdaML to evaluate the performance of distributed Machine Learning algorithms between Infrastructure-as-a-service (IaaS) and FaaS. The functions use a communication channel, which consists of persistent storage such as AWS Simple Storage Service (S3) or Elastic Search, to read and write intermediate results. Their results indicate that the FaaS implementation is only comparable in terms of performance, when there is low communication overhead. Although it can be much faster due to high elasticity, it is never significantly cheaper than IaaS.

Pons et al. [48] evaluate the performance of parallel function executions from a platform perspective. They use concurrent function invocations to evaluate AWS Lambda, GCF, Azure Cloud Functions (ACF) and IBM Cloud functions. Their main focus is on

the evaluation of platform scheduling, possible interference between concurrent function invocations and if concurrent invocations are executed by different instances. These are all important factors, when considering implementing a highly parallel application with a FaaS platform. Their results show that AWS's and IBM's respective FaaS platforms perform well, due to proactive scheduling, finer resource allocation and faster elasticity, while services like Azure did not reach the same parallelism degree.

While distributing workloads across function invocations can be useful due to the high elasticity, many results show that there is still significant synchronization and communication overhead involved. In contrast to previous work, we analyze the performance of intra-function parallelism, where we run parallel programs within a single function instance to test parallel performance.

## 3.2 Performance evaluations of FaaS platforms

A lot of other work focuses on the evaluating the general performance of FaaS platforms and the performance improvements made to isolated invocations [21], [20], [49], [44].

Cordingly et al. [21] present their framework Serverless Application Analytics Framework (SAAF), to improve the observability, among other things, on performance of workloads on FaaS platforms. Their framework supports 5 different platforms and several different programming languages. They also used their framework to conduct some performance evaluations on AWS Lambda between different memory configurations and their respective speedups between highest and lowest memory configuration. In their results they achieved a speedup of about 69.2x between 128MB and 10240MB allocated memory to a function.

Chadha et al. [20] examine the underlying processor architecture for GCF and evaluate the performance of compute-intensive tasks written in Python. They optimise their programs with a Just-in-Time (JIT) compiler based on LLVM and present their results on performance, memory consumption and cost. Their results show that the optimisation of FaaS functions can improve performance by 18.2x and save costs by 76.8%.

Quaresma et al. [49] evaluate the negative impact of automatic garbage collection on function performance. They also introduce the Garbage Collector Control Interceptor (GCI), to counteract the impact of garbage collection. The GCI decides based on incoming requests to the function and the memory used by the runtime to run the Garbage Collector (GC), instead of optimising the GC itself. Their experiments on AWS Lambda show that with a more efficient usage of the GC, the response time can be improved up to 10.86% and the cost reduced by 7.22%.

Pawlik et al. [44] evaluate the performance of scientific workflows on the major FaaS platforms AWS Lambda, GCF and IBM Cloud Functions. They used bag-of-tasks type workflows to conduct their performance evaluations and studied parameters like raw performance, efficiency of infrastructure provisioning and overheads, which are for example introduced by the API and network layers. Their results on raw computing performance show that it increases with higher memory configuration for the functions,



excluding IBM Cloud Functions.

All these previous approaches test the pure performance of FaaS instances, some even taking parallel executions into account. In contrast to this, our focus is more on performance evaluations of the parallelization of functions. We also want to look at a broader range of function configurations and evaluate the parallel efficiency for each. We also put a strong emphasis on potential cost savings between sequential and parallel executions.



## 4 Evaluated Services

In this chapter, we want to present and describe the evaluated services. For each service, we will give an overview of the underlying architecture, how functions are invoked and created and how they are implemented. Furthermore, we present what is known about the infrastructure for each of them. Finally, we give an overview of the important terms that are used in a serverless environment and how they differ for each service provider. For the evaluation, we chose 2 of the most popular FaaS platforms and 1 of the most popular CaaS platform.

### 4.1 AWS Lambda

We start with AWS Lambda, which is the FaaS offering of Amazon. The developer defines fine-grained functions, which are then executed by the provider. When a customer defines a function, either through the cloud console or the aws Command Line Interface (CLI), the code is stored in an S3 bucket. Afterward, the function is ready to be invoked. The user can invoke the function manually or define certain events, which are also called triggers. For example, by using an API Gateway to call the function via an URL, or use events from other services, like a database upload. Functions can be invoked both synchronously, where the response from the function is awaited, and asynchronously. We will focus on synchronous invocations in this thesis since we will also use them for our evaluations.

#### 4.1.1 Invocation Architecture

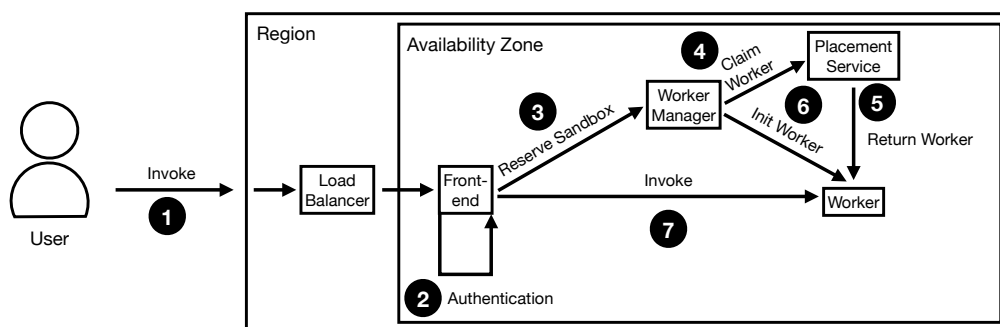


Figure 4.1: Behind the scenes of a Lambda function invocation

Figure 4.1 shows what happens under the hood when an AWS Lambda function is invoked[6]. In this scenario, there is no active instance that can handle the incoming invocation and a new instance has to be started. After the request reaches AWS ❶, it is first received by a Load Balancer (LB) for a specific region. The LB routes the request to an available Front End component in an availability zone. The difference between a region and an availability zone is that a region is a physical location with a cluster of data centers, while an availability zone is one or more data centers that are close to each other with redundant power, network, and connectivity [3]. There are several availability zones in a single region. The Front End Invoke component of AWS will authenticate the request, retrieve metadata about the function and enforce limits, such as the current concurrency limit of a function and region ❷. The Front End component then asks the Worker Manager which worker the request can be sent to ❸. The Worker Manager tracks the status of all Workers and Sandboxes and schedules invocations to them. If the Worker Manager currently has a Worker, which can handle the incoming request, it will either reuse an existing Sandbox or create a new one on that Worker. If not, the Worker Manager claims a Worker from the Placement Service ❹ ❺. The Placement Service manages all available Workers. The Worker Manager sets up the environment on the Worker, downloads the function code, and initializes the function ❻. Afterward, the Front End component is notified, and it can then directly invoke the function within the Worker ❼.

#### 4.1.2 Worker Component

The Workers are the core component of AWS Lambda functions. They execute the code of the function and return the response. In figure 4.2 we take a closer look at the

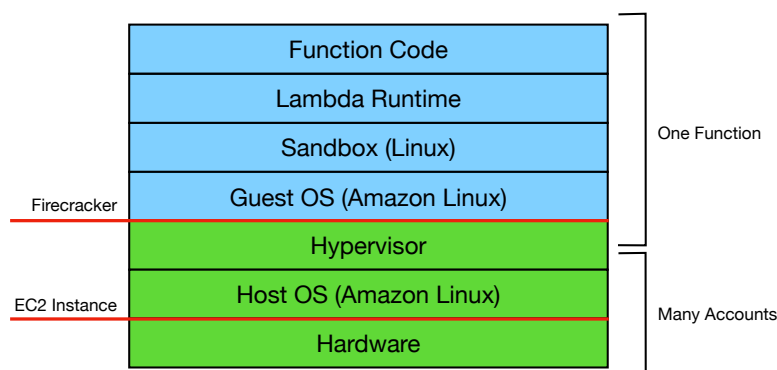


Figure 4.2: Architectural structure of an AWS Worker instance

structure of the Worker instance. At the bottom of the stack is the hardware, which is the same as for a regular Elastic Compute Cloud (EC2)<sup>1</sup> instance. The Host OS of the instance is Amazon Linux. On top of the host OS, a hypervisor is installed. Amazon

<sup>1</sup><https://aws.amazon.com/ec2/>

implemented their own open-source Virtual Machine Monitor (VMM), called Firecracker. Firecracker is used to create Micro Virtual Machines (MVM). Up to the hypervisor, everything is shared among multiple different customer accounts. Lambda functions of several different accounts will be placed on the same EC2 instance, on the same host OS. The hypervisor will create an MVM for each Lambda function with a guest OS installed. There are up to hundreds of MVMs running concurrently on the same Worker [6]. Each of them might belong to a different customer. The Sandbox has the same contents that a Linux OS would have and is isolated by the same technologies like containers, for example by cgroups or namespaces. The Lambda Runtime implements the necessary environment for the function to run in. This could be for example Node.js or Java. Finally, we have the function code of the customer. Workers have a maximum lease lifetime of 14 hours. After that time no new invocations are routed to that specific Worker and all MVMs are gracefully terminated. Once the Worker is shut down, it is returned to the Placement Manager.

### 4.1.3 Firecracker

Running several customers on the same physical machine allows for more flexible scheduling of functions and higher overall resource utilization. However, there are many security and isolation concerns, which is why Amazon built its own VMM. Both AWS Lambda and AWS Fargate are based on Firecracker. The open-source VMM was first launched in 2018 and is written in Rust [9]. Firecracker uses the Linux Kernel-based Virtual Machine (KVM) to create MVMs. To reduce startup times, a Firecracker MVM only includes the most necessary tools for a guest OS to run. Firecracker uses the `virtio`

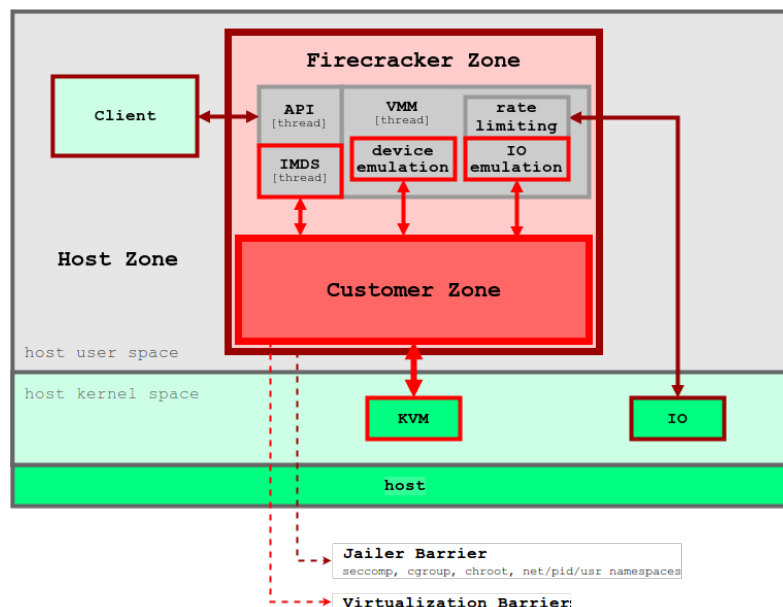


Figure 4.3: Architecture of a Firecracker MVM [11].

protocol to emulate devices so that the guest OS sees for example a hard drive or a network card. Additionally, the MVM provides a client with a restful API that allows the user to startup the MVM or limit its resources. The Firecracker hypervisor can start several Firecracker processes. Each Firecracker processor encapsulates one MVM. Figure 4.3 shows the architecture of one of these processes. It includes the client API, which runs in its own thread as an in-process HTTP server, and the VMM with the device and IO emulations, which interact with the guest OS. The memory overhead for the VMM threads is supposed to be very low and should not exceed 5MiB, according to the specification [10]. The figure also shows the several layers of isolation. Should a process ever escape the virtualization barrier of the guest OS, there is still another isolation layer, called the Jailer Barrier, which uses seccomp, cgroups, and more as isolation techniques.

#### 4.1.4 Instance Life-cycle

After the MVM is started on the host OS, the Lambda execution environment has to be initialised. Figure 4.4 shows the life cycle of an execution environment. There are

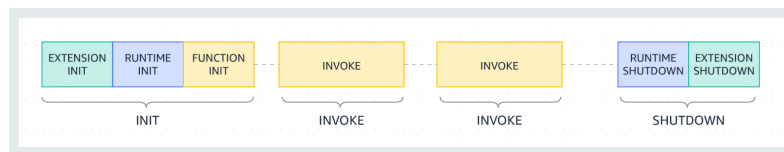


Figure 4.4: Life cycle of a Lambda execution environment [13].

3 different initialization phases. In the first initialization phase extensions are started. Extensions are augmentations to the Lambda function, which run independently from the function code and can be used to integrate external monitoring or observability tools [16]. The next phase bootstraps the function runtime. Afterward, the function’s initialization method is executed. From now on the instance is ready to execute function invocations. To reduce the number of cold starts, an instance is kept warm for a certain amount of time, in which it can receive consecutive invocations without needing to execute the initialization phase again. After a certain amount of time has passed without an invocation, the runtime is shut down and all extensions that are run in the background are also terminated. If an invocation crashes, the runtime also shuts down but immediately initiates again.

#### 4.1.5 Function Configuration

AWS Lambda functions can be configured in a lot of different ways [8]. In this thesis we will configure the following aspects of a function:

**Memory:** We start with the most important attribute we can configure, the function memory. The function memory determines the performance capabilities of our function. We can configure functions in 1MB increments, starting at 128MB with a

maximum of 10240MB. The amount of CPU resources allocated to a function is directly tied to the memory. At 1769MB the equivalent of a full virtual CPU (vCPU) is allocated. A virtual CPU is a virtualized version of the actual resources, allowing the provider for more fine-grained allocations. The actual mapping between vCPU and physical cores or hardware threads is not defined by the provider. If only a fraction of a vCPU is allocated, the CPU time the function instance is allowed to use is limited.

**Function Runtime:** As already mentioned, the user can configure the language for the function runtime. There are several available out of the box, for example, Java, Python, and Go. The user can also write custom runtimes that fulfill his requirements.

**Concurrency:** One of the principles of serverless is the auto-scaling behavior. If all active instances are currently busy, we can still handle an incoming request by starting new instances concurrently, which can handle new requests. This concurrency can be limited so that only a certain number of functions is allowed to run at the same time.

## 4.2 Google Cloud Functions

The next service we look at is GCF. GCF is also a FaaS platform and therefore the service equivalent to AWS Lambda. The developer writes small-grained functions to be executed by the provider. The developer can upload the function for example via the `gcloud` CLI or the cloud console. When using the cloud console, the code is uploaded, built, and packaged by the Google Cloud Platform (GCP). This is in contrast to AWS, where the developer can build and package the function locally before uploading it. Functions can be triggered either via HTTP or event triggers. With an HTTP trigger, the function is executed when the URL attached to the function is called. With event triggers, on the other hand, the function is executed in response to certain events of the cloud project of the developer. These events can be induced by other services from the GCP. The function evaluations conducted in this thesis are invoked with HTTP requests.

### 4.2.1 Google Cloud Platform Cluster Architecture

As already mentioned, FaaS function a lot like black boxes, where the developer doesn't know what happens when a function is invoked. The major cloud providers usually don't disclose their full infrastructure architecture. Google showed some of their basic concepts at the Cloud Next Event in 2018 [47]. Figure 4.5 shows the architecture behind the GCP, which is the basis for all major cloud services that Google offers. On the bottom is the hardware, grouped into data centers. On top of the hardware, Google runs their custom cluster manager called Borg [54]. The cluster manager consists of a BorgMaster, managing several Borglets. A BorgMaster manages a whole data center and

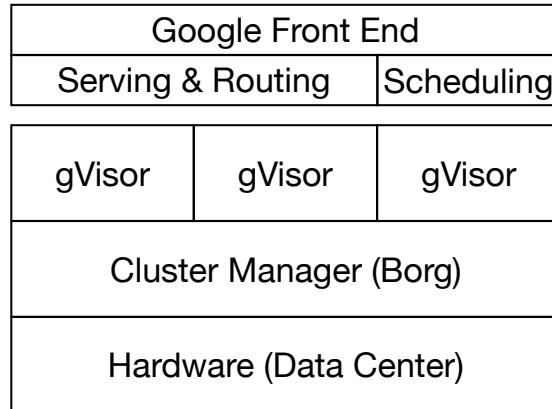


Figure 4.5: Google Cloud Platform cluster architecture

is replicated up to 5 times. Borglets are local agents that are present on every machine in a data center. The BorgMaster receives tasks with constraints, like memory or CPU requirements, and queues them for execution. Borglets stop and start tasks. A cluster consists of many thousand machines and can handle rates above 10000 tasks per minute. Most of Google's technologies are built on top of Borg.

The ingress point for all incoming traffic to Google's networks is the Google Front End component. It takes care of routing and has some built-in security measures like denial of service protection. Afterward, there is an LB, which is depicted as Serving & Routing in figure 4.5. It receives and dispatches requests. The Scheduling component is specific to serverless services. It decides when instances are created, reused, or evicted. We could not find more publicly available information on the internal scheduling of GCF.

#### 4.2.2 gVisor

The functions themselves run in gVisor, which is an application kernel running in user space of the host OS and written in Go [33]. gVisor is different from a VM since it does not use virtualized hardware. It rather provides a virtualized environment to sandbox containers. System calls by the application are intercepted and handled by the gVisor kernel without a hardware virtualization layer in between. The amount of system calls between gVisor and the host kernel is highly reduced, which also reduces the attack surface. The upside is that the fixed cost of hardware virtualization overhead is lowered, however, the trade-off is less application compatibility and higher per-system call overhead. gVisor implements most, but not all, of the Linux system interface calls. gVisor is built to be very lightweight, with startup times around 150ms and a memory footprint of about 15MB [55]. Figure 4.6 gives an overview of the isolation barriers between the host, gVisor, and the application. The application is a Linux binary, which



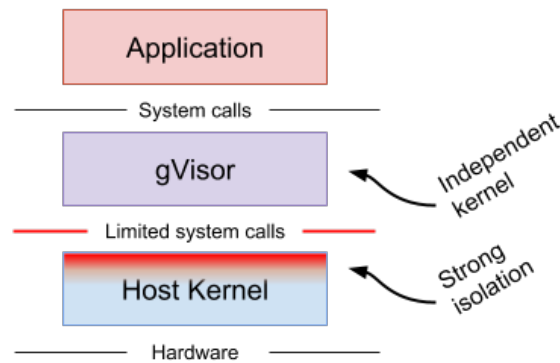


Figure 4.6: gVisor layers overview [29]

implements the runtime specifications of the Open Container Initiative (OCI)<sup>2</sup>. This also indicates that all GCF are built and deployed in Docker images.

### 4.2.3 Function Configuration

GCF can be configured in the same way as AWS Lambda functions, with the main options being memory, function runtime, and concurrency. There are some differences:

**Memory:** The developer can choose between 128MB, 256MB, 512MB, 1GiB, 2GiB, 4GiB and 8GiB. Note that these are fixed options, which means the developer can not modify his functions in 1MB increments. Furthermore, the upper limit is lower than with AWS Lambda. In contrast to AWS Lambda, the number of vCPUs does not increase with the function memory. However, functions do get more powerful with increased function memory. Instead of more vCPUs, the processors assigned get an increased clock frequency, according to the documentation [24].

**Function Runtime:** The developer can choose from a list of supported runtimes [23]. Note that there are fewer supported languages than for AWS Lambda and the developer has no option to bring his own runtime or deploy custom container images.

## 4.3 Google Cloud Run

Finally, we present the last service we are evaluating in this thesis with Google Cloud Run. As described in subsection 2.1.2, GCR is a CaaS. This means that there is more developer effort involved since the developer has to provide not only the code but also the container images. On the other hand, CaaS offers greater flexibility, since the developer can bring his own container images, with pre-installed custom dependencies

<sup>2</sup><https://opencontainers.org/>

and runtimes that would not be supported by GCF. Container images can be built and deployed via the `gcloud` CLI.

### 4.3.1 Knative

GCR is a managed Knative<sup>3</sup> service. Knative is built on top of Kubernetes, an open-source tool for automating deployment, scaling, and management of containerized applications. Knative is the serverless extension to Kubernetes and extends it with components in regards to building and serving containers, as well as an additional component dealing with events. It also allows containers to scale down to 0, one of the essential principles of serverless.

Apart from GCR being a managed Knative service, there is not a lot of information publicly available on how GCR is built under the hood. The GCP's documentation about their Google Kubernetes Engine service [27] states that the Kubernetes cluster consists of multiple Compute Engine VMs [28]. The Knative platform can then automatically deploy container images, according to the resource requirements, on available VMs in the cluster. However, to run multiple containers from different customers on a single compute engine, there needs to be another isolation layer. This isolation layer is gVisor. Since gVisor implements the OCI runtime specifications, it can be used interchangeably with the default Docker runtime, effectively sandboxing the containers[40]. Multi-tenancy is a requirement to effectively utilize underlying resources for serverless services. However, that requires strong isolation methods, so containers of different customer accounts cannot interfere with each other. Google also confirmed that they built gVisor into the infrastructure that runs all their serverless services [19].

### 4.3.2 Container Configuration

Since GCR is a CaaS platform, the configuration options are different from those of FaaS platforms. We will again focus on the most important characteristics:

**Memory:** The developer can configure the memory that is available to the container. The options are the same as with GCF, with the same steps and upper limit.

**CPUs:** The first difference to FaaS is that the processing power can be configured independently. The developer can allocate either 1, 2, or 4 vCPUs to the container.

**Concurrency:** While concurrency in FaaS is the number of concurrent function instances, in CaaS it means the maximum amount of simultaneous requests a single container can receive. If the concurrency limit is reached, consecutive requests will be routed to other container instances.

**Instance number:** The developer can also configure a minimum and a maximum number of container instances that can be used for a given GCR service. Note

---

<sup>3</sup><https://knative.dev>

that if the minimum number is higher than 0, the service cannot scale down to 0 anymore.

Since the developer deploys a Docker image, a runtime does not have to be specified.

### 4.3.3 Pricing Model

The pricing model is different from FaaS since the developer has also more control over what is executed. For example, the code that starts the container is also provided by the developer and can be shorter or longer, depending on the application and the container image. GCR charges for:

**Container startup:** This can be compared to the initialization time of a function, which for example with AWS Lambda is also billed. However, the initialization also includes not only the startup time of the container but also the establishment of the HTTP server, as shown in the main function in listing 2.1

**Container shutdown:** Google bills the duration from the moment the container receives the SIGTERM signal until it is gracefully shut down.

**Active request:** The customer is billed, as long as one active request is handled on the container. The customer is not billed, when the container is running but no request is handled. This principle can be seen in figure 4.7. However, this is only the case as long as the number of minimum instances is set to 0.

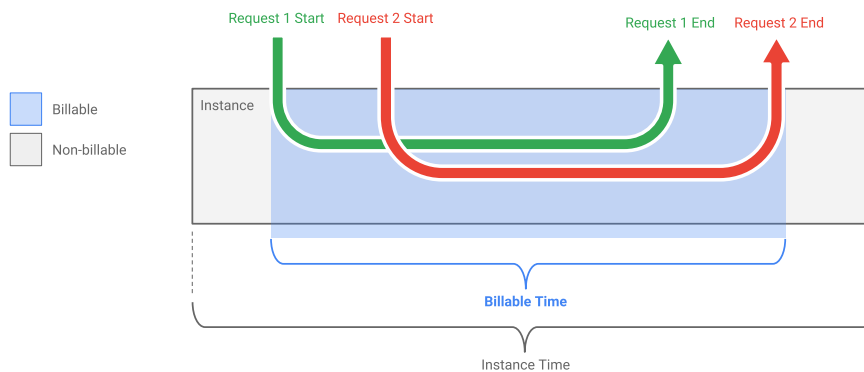


Figure 4.7: Billable time of a Google Cloud Run instance [26]



## 5 Evaluation Setup

In this chapter, we describe our full evaluation setup. We start with the runtimes and languages used to evaluate the services. We then list all benchmarks that were used and with what inputs they were configured and executed. We then give an overview of how the evaluated services were configured. Finally, we explain our testing process and how we collected which metrics.

### 5.1 Runtimes

In this section, we describe the languages we used to evaluate the services and what runtimes were used for each. Since there are use-cases for every programming language, it is important to use more than one language to evaluate these services. This not only allows us to cover a higher scope of use-cases but also increases the insights about the different platforms, which we would not otherwise get.

#### 5.1.1 C++

The first language we chose for the evaluations is C++. We chose this language since it is widely used for HPC applications and is often used to parallelize programs. C++ is also known for making efficient use of the resources it is run on, which is crucial in serverless environments since longer execution times also result in higher costs. Additionally, C++ provides all tools necessary to implement and execute the chosen benchmarks.

To parallelize the programs, we chose OpenMP<sup>1</sup>, since it is a well-established and documented API. OpenMP is used for shared memory parallel programming. The OpenMP API is implemented in most C/C++ compilers. To parallelize a program with OpenMP, the developer can use pragmas in the code, which function as entry points to the compiler. The compiler then generates the threaded code based on the clauses used in the pragma. OpenMP is mainly used to parallelize for-loops, but can also perform task-based parallelism, which we will not use in our implementations. Listing 5.1 shows a small code example of how loops are annotated to parallelize them. The sequential parts of the program are executed by a master thread. As soon as a master thread encounters a parallel section, the workload of the loop is split and the program is forked into several threads. At the end of a parallel region, a barrier is inserted to synchronize all threads. Since OpenMP is going by the principle of shared memory, all variables defined outside of the loop are by default accessible by all threads. The developer

---

<sup>1</sup><https://www.openmp.org/>

has some additional options. For example, he can define private variables for each thread, or perform reductions on certain variables. OpenMP also provides environment variables and methods to get the number of available processors and set the number of threads the program is supposed to use. We use these to get more information about the environment, but also limit the available threads to test single-threaded execution.

```
1 /* ... */
2 #pragma omp parallel for
3 for (int i = 0; i < upperLimit; i++) {
4     // workload
5 }
6 /* ... */
```

Listing 5.1: Example of loop parallelization using OpenMP

AWS Lambda provides certain runtime images out of the box, where user can simply upload the compiled code or even the source code itself and doesn't have to configure anything else. C++ is not one of those languages. However, in 2018 AWS introduced the Lambda Runtime API [46], which makes it possible to write functions in any programming language. The Runtime API gives the possibility to interact from any given language with the Lambda environment. AWS also implemented an open-source C++ runtime [4], which we will use for our evaluations. To build an AWS Lambda C++ function, we create an EC2 instance with Amazon Linux installed. On the instance we compile the runtime and create and build our functions using `gcc`<sup>2</sup> and `cmake`<sup>3</sup>. The executable of the function is then packaged together with all needed dependencies into a zip file, which is then used to create the Lambda function. GCR deploys container images, which gives us the possibility to configure custom runtimes. To build the executable we use the docker image `alpine:edge` and install `gcc`<sup>4</sup> and `cmake` for the compilation process. We then copy the binary into the deployment image, which is based on `debian:buster-slim`. GCF unfortunately does not provide a C++ runtime or the ability to develop and deploy custom runtimes. For both AWS Lambda and GCR we use the compiler flag `-O3` to maximize performance. AWS also offers the possibility to use vectorized code, using Advanced Vector Extensions 2, which is a vectorization extension to the Intel x86 instruction set to perform SIMD instructions [15]. We verified that our code got vectorized by looking at the produced assembly. We also verified the vectorization for code compiled in the Docker image, which we used for GCR.

### 5.1.2 Go

For the second language to evaluate the services, we chose Go. Go was released in 2012 [30] and has since then received regular updates, with the latest version 1.17

---

<sup>2</sup>Version: `g++ 7.3.1 20180712 (Red Hat 7.3.1-13)`

<sup>3</sup><https://cmake.org/>

<sup>4</sup>Version `g++ (Alpine 10.3.1_git20210625) 10.3.1 20210625`

released in August 2021 [45]. We chose this language for a couple of reasons. Firstly, Go is widely used and all major cloud providers support Go in their serverless offerings. Furthermore, Go was built with concurrency in mind, making it very easy to parallelize our functions.

To parallelize our code we use goroutines, which is the native implementation for lightweight threads. Goroutines can be used to execute functions concurrently. These functions can also be anonymous, which means they don't have to be defined in advance.

```

1 /* ... */
2 for i:= 0; i < threads; i++ {
3     go func (index int) {
4         lowerBound := index * (bound / threads)
5         upperBound := (index + 1) * (bound / threads)
6         for j := lowerBound; j < upperBound; j++ {
7             // workload
8         }
9     }(i)
10 }
11 /* ... */

```

Listing 5.2: Example of loop parallelization using goroutines

5.2 shows a short example of how we use goroutines to parallelize our code. The keyword `go` creates the goroutine. Contrary to OpenMP, we have to manually split the workload, by computing the upper and lower bound of the loop according to the number of threads. We use an anonymous function, which receives the index as a parameter, to compute the respective bounds of the loop. All variables defined outside of the goroutine are shared by default. Variables defined inside of the goroutine are private and only available to the respective routine. Go has its own scheduler to schedule goroutines on the hardware. It uses the following 4 concepts [38] [31]:

**Goroutines G:** As already mentioned, G are lightweight threads and very similar to OS threads. For example, they each have their own stack and can be context switched.

**Worker Threads M:** M are threads that are still managed by the OS.

**Processor P:** P is a resource that is required to execute Go code. For each logical processor on the executing machine, one P is available to the Go runtime. M must have an associated P to execute Go code.

**Run queues:** There are 2 types of queues. A local queue, which is attached to a P and contains all Gs to be executed in the context of P. All Gs in the local queue can be context switched on and off the attached M. There is also a global queue, which contains all Gs which have no assigned P.

The scheduler not only decides which goroutines are attached to which P, but it also decides when to context switch between goroutines. After spawning a goroutine, the main program continues its execution. This is why we need a mechanism to wait for all spawned goroutines to finish their executions. The Go runtime provides this construct called a `WaitGroup`. When spawning a routine, we increase the counter of a `WaitGroup` and decrement the counter when the routine is finished. The `WaitGroup` has a blocking method, which waits until the counter is 0. All methods that are part of the `WaitGroup` construct, utilize atomic instructions to prevent race conditions.

Go is rising in popularity ever since its first release. All major commercial FaaS services provide out-of-the-box Go runtimes. The versions we used are 1.16.5 for AWS Lambda, 1.16.7 for GCR, and 1.13 for GCF. The docker image used for GCR is based on `golang:1.16-buster`, while the deployment image is again `debian:buster-slim`. Go currently has 2 compiler implementations. Google's self-hosting compiler `gc` and `gofrontend`, which is a frontend to other compilers, for example, `gcc` or `llvm`. The front end to `gcc` is also called `gccgo`. There are some advantages to using `gccgo`, for example, the ability to vectorize the code, however, we still decided to use the `gc` compiler. We made this decision since it is the standard compiler, which is not only more widely used but `gccgo` is also not available for GCF. The consequence is that none of the Go implementations utilize vector instructions.

### 5.1.3 Java

Finally, we use Java as the third evaluation language. Java is consistently one of the most popular programming languages. We chose this language not only for its widespread usage, but also to get a contrast to the other two previously mentioned languages. Even though Java applications are also compiled, they are not compiled to binaries but bytecode. The bytecode is then run by a Java Virtual Machine (JVM). The upside is that the compiled code can be executed in any JVM, regardless of the underlying architecture. The bytecode is interpreted by the JVM and translated to machine code at execution time. The JVM also includes a Just In Time (JIT) compiler, which can translate certain parts of the bytecode to native machine language. This is usually done for code that is executed exceedingly often. The concept of a JIT compiler entails that the code usually needs time to warm up, which means that the execution time of the application can go down for consecutive runs. The downside for serverless environments is, that the JVM usually has more overhead and it takes longer to spin up new instances. The JVM also needs to warm up again on new instances.

Java is not as famous for concurrency as the other 2 languages, but it also provides tools and libraries for shared-memory programming. We use the `ExecutorService` Class to create a thread pool and submit tasks to be executed by these threads. The structure of the code shown in 5.3 is very similar to creating goroutines in Go. We manually split the workload by computing upper and lower bounds for the loop. All variables defined outside of the runnables are shared. Each runnable is put in a task queue, which is then scheduled for execution. The benefit of using a fixed thread pool is, that the threads



are created once and can be reused for several parallel regions within a program. As with goroutines, the main program continues execution and we need to use a construct to synchronize the program again. For our applications, we use a vector. Each thread adds a number to the vector after it finishes its workload. The main thread checks the size of the vector and continues execution once the vector size is equal to the number of threads.

```
1 /* ... */
2 ExecutorService executorService = Executors.newFixedThreadPool(threads);
3 for (int i = 0; i < threads; i++) {
4     final int lowerBound = i * (bound / threads);
5     final int upperBound = (i + 1) * (bound / threads);
6     final int index = i;
7     executorService.submit(new Runnable() {
8         @Override
9         public void run() {
10             for (int j = lowerBound; j < upperBound; j++) {
11                 // workload
12             }
13             return;
14         }
15     });
16 }
17 /* ... */
```

Listing 5.3: Example of loop parallelization using Java ExecutorService

Similar to Go, all major FaaS services provide Java runtimes. For AWS Lambda, the runtime version we used is Java Coretto 11 [2]. Java Coretto is a custom Java distribution based on OpenJDK and patched by AWS. The Java runtime for GCF is also based on OpenJDK version 11. The GCR docker image is based on `maven:3.8-jdk-11`. The deployment image, in this case, is the same since the JVM is needed to execute the bytecode. The JIT compiler also can vectorize the code. We verified that our code got vectorized by looking at the produced assembly of the JIT compiler.

## 5.2 Benchmarks

In this section, we want to present the benchmarks we chose to evaluate the services. In total, we are testing with 5 benchmarks. Each of these benchmarks is implemented in all of the languages described in the previous section.

### 5.2.1 Microbenchmarks

Microbenchmarks are often used to test sub-sets of systems or very specific hardware. In our case, FaaS oftentimes function as subsystems and only perform small-grained functionality. Additionally, functions tend to not run very long, which makes a good use-case for microbenchmarks. The microbenchmarks we chose were all taken from the NPbench Github repository<sup>5</sup>. All of the benchmarks in the repository are implemented in Python, which we then translated to our chosen languages. We chose specific benchmarks, which have no loop dependencies so that an efficient parallelization would be possible. All microbenchmarks consist of an initialization phase and a computational phase. In the initialization phase, we initialize the arrays and fill them with data. In the computational phase, the actual results are computed. The workload of both phases is split among the available threads.

#### Atax

The first microbenchmark is called Atax. The benchmarks perform a matrix transpose and vector multiplication. First, we initialize 2 arrays with the size of N as the vectors. One of the vectors is used for the intermediary result. We also initialize an array with the size of M times N as the matrix. We fill one of the vector arrays and the matrix array with values. All values are computed by a formula based on the index of the value in the array, and the overall length of the arrays. We then perform the following computations:

$$\left[ \begin{pmatrix} a_{1,1} & \cdots & a_{1,N} \\ \vdots & \ddots & \vdots \\ a_{M,1} & \cdots & a_{M,N} \end{pmatrix} \times \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix} \right] \times \begin{pmatrix} a_{1,1} & \cdots & a_{1,N} \\ \vdots & \ddots & \vdots \\ a_{M,1} & \cdots & a_{M,N} \end{pmatrix}$$

Since the computational part is split into 2 parts, our programs also contain 2 parallel sections to compute the results.

#### Go Fast

The next microbenchmark is called Go fast. The benchmark performs a trace computation on a matrix and a scalar addition of the computed trace to the matrix. In the initialization phase, we create an array of the size N times N and fill the array with randomized values. Generating random numbers in a parallel context is often not trivial [51]. Since we have to do the same in subsection 5.2.2 with the Monte Carlo application, we will discuss how we deal with this problem in more detail in that section.

We compute the trace of the matrix, by summing up the tanh values of all diagonal matrix elements.

$$trace = \sum_{i=0}^N \tanh(a_{i,i})$$

---

<sup>5</sup><https://github.com/spcl/npbench>

Afterward, we add this sum to every matrix element. This means that we split our computational part into 2 separate parallel sections. We also have to perform a reduction on the trace, since each thread keeps its own value of the trace, to prevent race conditions.

## MVT

The final microbenchmark we used for the evaluations is called MVT. It performs a matrix-vector product and transpose. In the initialization phase we create 4 vectors as arrays with the size of N, and 1 matrix as an array with the size of N times N. Similar to *Atax*, they are all filled with values based on the index of the value and the size of the array. The computations performed on the vectors and matrix are shown below.

$$x1 = \begin{pmatrix} x1_1 \\ \vdots \\ x1_N \end{pmatrix} + \left[ \begin{pmatrix} a_{1,1} & \cdots & a_{1,N} \\ \vdots & \ddots & \vdots \\ a_{N,1} & \cdots & a_{N,N} \end{pmatrix} \times \begin{pmatrix} y1_1 \\ \vdots \\ y1_N \end{pmatrix} \right]$$

$$x2 = \begin{pmatrix} x2_1 \\ \vdots \\ x2_N \end{pmatrix} + \left[ \begin{pmatrix} y2_1 \\ \vdots \\ y2_N \end{pmatrix} \times \begin{pmatrix} a_{1,1} & \cdots & a_{1,N} \\ \vdots & \ddots & \vdots \\ a_{N,1} & \cdots & a_{N,N} \end{pmatrix} \right]$$

We can combine both computations into a single nested loop.

### 5.2.2 Applications

As previously mentioned, microbenchmarks are good to test subsystems. However, it is advised to also use real applications for benchmarking. Applications are better at mapping the conditions of a real system, which consequently can produce results that are closer to reality than microbenchmarks. For our evaluation, we chose 2 applications.

#### Heat

The first application we chose is called Heat, which computes the heat distribution in a 2d solid body. We can model the heat distribution by following the Poisson equation:

$$-\frac{\delta^2 u}{\delta x^2} - \frac{\delta^2 u}{\delta y^2} = k * \frac{\delta u}{\delta t}$$

$u(x, y, t)$  is the temperature in point  $p(x, y)$  at time  $t$ . In a stable state, the temperature won't change anymore for a given time, which gives us the following equation:

$$-\frac{\delta^2 u}{\delta x^2} - \frac{\delta^2 u}{\delta y^2} = 0$$

To solve this equation we can apply the Dirichlet boundary conditions, which means that we extend our grid and set fixed temperatures at the boundaries. We initialize a

matrix with the size of N times N. The matrix represents all points in the body and contains the temperature for each. We extend the matrix by 1 on every boundary, so we can insert our fixed temperature heat sources. We then iteratively compute the new temperature for each point. We use the Jacobi method to compute the new temperature, by calculating the mean of the 4 neighbors of a point. The values of the neighbors are taken from the previous iteration. To avoid loop dependencies within an iteration step, we initialize 2 arrays, which are then swapped after each iteration. The application finishes either after the change over all points within an iteration, also called residual, is below a threshold, or an upper limit of iterations is reached.

The parallel workload of the application is the initialization of the arrays and the computations within an iteration. Since we use the values of a previous iteration to calculate the new temperatures, there is a loop dependency between iterations. However, we can split the calculations of the new temperature values within an iteration. Similar to the Go fast benchmark, each thread keeps its own residual value. We have to perform a reduction on the residual between each iteration.

### Monte Carlo

The last application we use for the evaluation is the Monte Carlo simulation. The Monte Carlo simulation is, in essence, the generation of random objects. These random objects can be used to estimate certain numerical quantities related to a simulation model [39]. In our case, we use the Monte Carlo simulation to estimate digits of  $\pi$ . To do this we

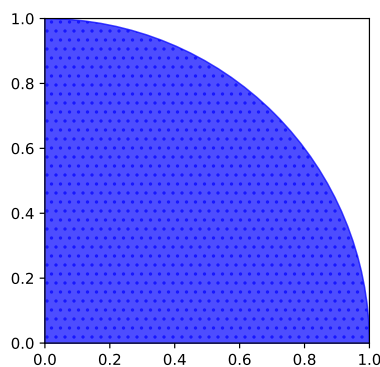


Figure 5.1: Monte Carlo simulation to estimate digits of  $\pi$ . The simulation generates points in the square randomly and uses the points that fall into the blue marked area for the estimation

draw a circle with a radius of 1, as seen in figure 5.1. The area of each quadrant of the circle, marked in blue, is  $\frac{\pi}{4}$ . To approximate  $\pi$ , we generate 2 random numbers between 0 and 1 for each iteration. 1 for the x and 1 for the y coordinate. We then calculate if the point is in the blue marked area. We count all points that fall into the area. We calculate the ratio between points inside the circle and the total amount of points and multiply

that number by 4 to get the approximation. These calculations are shown again below:

$$PointIn(x, y) = x^2 + y^2 \leq 1$$

$$Estimation_{\pi} = \frac{PointsIn}{TotalPoints} * 4$$

$x$  and  $y$  are randomly generated and defined with  $0 \leq x, y \leq 1$ . The total amount of points is also equal to the number of iterations of the simulation. The higher the number of iterations, the more accurate the estimation will be.

As mentioned in the previous section, the generation of pseudo-random numbers in a parallel context is often not trivial. The reason for this is that most random number generators are implemented in a way that they behave deterministic based on an initial seed. This determinism is useful since we can reproduce behavior given that we know the initial seed. However, the downside is that for these generators to be deterministic they need to keep an internal state, which is modified each time a number is generated. If we call the same generator from different threads, the application slows down due to false sharing of the generator internals. We can resolve this conflict, by initializing a pseudo-random number generator for each thread. Each generator keeps its own internal state, which is not shared among threads. To improve the results, each thread is initialized with a different seed. This way we can generate random numbers inside of a thread, without slowing down the application. Each thread also keeps its own counter for the points that are inside the circle. After all iterations are done, we sum up the counters of all threads and calculate the estimation of  $\pi$ .

### 5.2.3 Input Overview

After we described all benchmarks and how they work, we want to give an overview of the input configurations for the respective benchmarks. We execute all benchmarks with 3 types of inputs. Small (S), Medium (M), and Large (L) input sizes. The memory constraints for S and M are that they should be able to run on every configuration. That means that the M input for every benchmark needs to be below 512MB. The L input should be executable by the function configurations with 4096MB and above. Table 5.1 shows the input sizes of the benchmarks and a rough estimation of how much memory is needed to allocate all arrays. For Monte Carlo, the memory footprint does not change by a significant margin and was therefore omitted. We tried to define the inputs in a way that, for the same input, the benchmarks have a similar memory footprint but also execution times.

We execute S and M for functions configured with 2048MB and below, while we execute M and L for functions with 4096MB and above. We test different input sizes to gather more information about the serverless environments and see if there are any differences in the functions for handling small and large amounts of data. To have a consistent data point across all configurations, input M is executed on all configurations.

	<b>S</b>	<b>M</b>	<b>L</b>
<b>Atax</b> M, N Memory	2000, 2500 40MB	4000, 5000 160MB	20000, 22000 3520MB
<b>Go fast</b> N Memory	2000 32MB	6000 288MB	22000 3872MB
<b>MVT</b> N Memory	2000 32MB	5500 242MB	18000 2592MB
<b>Heat</b> N Iterations Memory	2200 10 77MB	5200 10 432MB	15200 10 3696MB
<b>Monte Carlo</b> Iterations	$10^7$	$10^8$	$10^9$

Table 5.1: Benchmark configurations for small, medium, and large input size

### 5.3 Service Configurations

In this section, we want to describe how we configured our evaluated services. We tested several different configurations for each service, to get a better idea of how the resource allocations for a function changes between configurations and what performance improvements can be observed between them.

For AWS Lambda and GCF, the function memory configurations are 512MB, 2048MB, 4096MB, 8192MB 10240MB. For GCF the highest configuration is 8192MB since that is the upper limit set by Google, while AWS Lambda allows a memory allocation up to 10240MB. The memory configuration of GCR is the same as GCF, with one small exception. Instead of 2048MB, we allocate 2148MB, since that is the minimum required memory to allocate 4 vCPUs. We allocate 4 vCPUs for every memory allocation but for 512MB, where we allocate 2 vCPUs. We chose these configurations to test the maximum resources that can be allocated. With 512MB we also include a lower configuration, to see how efficient parallelizing is on higher and lower configurations.

To make sure consecutive configurations are handled by the same instance, we set the concurrency of functions to 1 for both AWS Lambda and GCF. This approach lets us measure the effects of parallelization more consistently since we reduce the variation which is caused by cold starts and warm-up times. For GCR, we set the maximum amount of containers to 1 and the maximum of concurrent requests that can be handled to 1 for the same reasons. Limiting the container instances makes sure the requests are all sent to the same container and limiting the number of concurrent requests makes sure the container doesn't handle 2 or more requests at the same time, which would

decrease the performance of one single request. We verify that always the same instance is invoked by keeping a global counter. That counter is incremented for every function invocation and included in the response of the function. If the counter is 1, we know that a new instance was initiated. All functions and containers using services by Google were deployed in us-central1. For AWS Lambda the region we used is us-east1.

## 5.4 Evaluation Process

In the next section we describe our evaluation infrastructure. Before we do any executions, we deploy our functions manually. Some of them functions were deployed via the cloud console on the browser, the others were deployed using the respective CLIs of AWS and Google.

The main evaluation infrastructure is handled by mostly Python scripts, which can also be seen in figure 5.2. The main execution script is responsible for the following

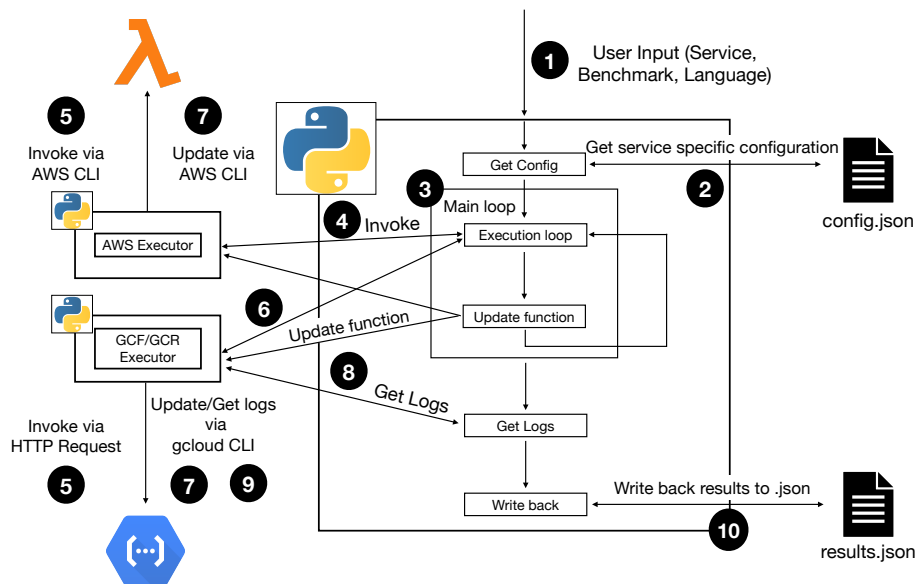


Figure 5.2: Execution script

things:

**Retrieve Configuration:** When running the script, we decide what function we want to run in what language and for which service. We pass those 3 as parameters to the script **1**. The script then retrieves necessary configuration information for the specific function that it is supposed to invoke **2**. This is for example the URL to the container, or the exact function name, which is also used for the invocation.

**Invoke Function:** In the main loop of the program **3**, the function is repeatedly invoked **4**. For each function configuration, we run 2 different inputs, in total 20

times. 10 invocations with single-threaded, 10 times with multi-threaded execution. That sums up to a total of 40 executions per function configuration. Since we use synchronous invocations, the script awaits the function response before it triggers another invocation. We collect all responses from the function invocations in a dictionary.

**Update Function:** After all executions for 1 specific function configuration are finished, we increase the memory for the function ⑥.

**Write Back Results:** After all executions are finished, we write back the dictionary, which holds the responses of all invocations ⑩. For GCF and GCR, we additionally retrieve the function/container logs and also write them to a JSON file ⑧.

To each function invocation, we pass a JSON as a parameter. The JSON contains 2 keys: "Input" and "Threads". The values for the input key are either "S", "M", or "L". The "Threads" key is set to either 1 or 6. At the start of an invocation, the function initializes the data based on the input and according to the table 5.1. If the value of "Threads" is 1, the amount of threads in the function is also set to 1, to test single-threaded execution. If the value is 6, we retrieve the maximum amount of hardware threads available to the runtime and set the number of threads to that number. We chose the number 6 since this is also the maximum amount of hardware threads we observed in our evaluations. With this input, we test the parallel execution time of the benchmark. Parallel execution is always conducted with the maximum amount of hardware threads available.

For each service, we use a separate executor script, which contains its own implementations of invoking and updating a function. The AWS executor uses the aws CLI to invoke and update functions. The GCF and GCR executors use python requests to invoke function via HTTP requests and use the gcloud CLI to update function and container configurations.

### 5.5 Metrics

In this section, we describe the metrics we measure for each service. Since this thesis focuses on the exploration of parallel execution in a serverless environment, we primarily look at the execution times of the functions. We also look at the memory consumption of the functions to see if any differences are noticeable between runtimes and services. We will describe both in more detail in the following subsections.

We use different methods to retrieve these metrics for the different services. For AWS we can use the CLI to invoke functions and get the most necessary information directly as a response, when including the parameters `-log-type Tail -query "LogResult" -output text` for an invocation. In the response we get the following data:

**Init Duration:** The initialization duration of the function. It is only sent if a new instance is invoked.



**Duration:** The duration of the actual execution of the function.

**Billed Duration:** What is effectively billed to the customer. If the invocation starts a new instance, the billed duration is the sum of the initialization duration and the duration of the function execution. We use this value for all our results.

**Max Memory Used:** The maximum memory used of a function. This value is kept over the lifetime of an instance. This means that if the first invocation of an instance has the highest memory consumption, this value will not change for consecutive invocations to the same instance.

For GCF and GCR, there is no option to include these data points in the response to a request. This is why retrieve the function runtime, which is equal to the billed time, from the logs. We do this via two separate steps. For GCF, we retrieve the function execution ID, which is assigned by the GCP. The execution ID is part of the HTTP header sent to the function. We include this ID in the function response so that we can save it for later use. After all invocations are finished, we retrieve all logs for that function via the `gcloud` CLI. The logs return a list of JSON objects. For each function invocation, there is a JSON object, which contains the function execution time. We can search and match this log entry with the execution ID, which we saved earlier. The function execution time is equal to the billed time and will be used in our results.

Our approach to retrieve GCR logs is relatively similar, however, the GCP doesn't assign an execution ID for a request. Google stores only the URL for a given request. Because of this reason, we write the request handlers of our containers in a way that they expect an identifier as part of the request URL. In our Python script, we generate a unique ID which we add to the request URL. We also save this ID for later use. After all requests are finished we again retrieve the logs via the `gcloud` CLI and match the request IDs with the log entries. We then retrieve the request latencies for each request. Afterward, we write all our results back to an InfluxDB<sup>6</sup> bucket. Each entry in the bucket contains the the data on all attributes, which are listed in table 5.2. For our results,

Attribute	Description
Benchmark	-
Language	-
Service	-
Input	-
Memory	The memory the function was configured with
Threads	The amount of threads that were used to execute the function
Max Memory Used	Only for AWS since GCF and GCR don't directly return it
Value	The execution time for a function

Table 5.2: InfluxDB entry structure

<sup>6</sup><https://www.influxdata.com/>

we query the InfluxDB either for all recorded values or the mean values for a specific function configuration.

### 5.5.1 Execution Time

As mentioned previously the main focus of this thesis is the parallelization of serverless functions, and therefore the focus is on the execution times of the function. We mostly look at the billed time, which is provided by the cloud platforms themselves, which we retrieve as mentioned previously. We additionally measure the execution time of only the computational part of the benchmark. This means that we add a timer to each program that starts after memory initialization and finishes just before we return our response. We use this timer to make better distinctions between function or memory initialization and the actual computations of the benchmark. For example, if our custom timer has an expected speedup between single- and multi-threaded execution, but the billed time doesn't, we can deduct that either the memory or function initialization slowed down the function. For our results analysis, we primarily used the billed time, if not mentioned otherwise.

### 5.5.2 Memory Usage

Collecting memory usage is not as straightforward as measuring the execution time. Unfortunately, every cloud provider differentiates in the way they provide memory usage. As previously mentioned, we can collect the maximum memory used for AWS Lambda functions. But since we repeatedly call the same instance, we cannot retrieve the current memory usage for every single invocation. However, we still save these values, which gives us a general idea of the memory consumption and allows us to identify patterns across runtimes.

For GCF we cannot retrieve the memory from the logs, but Google provides a metric explorer, which also contains values for memory usage. Similar to AWS, we can't retrieve the memory usage for a single invocation. The metric explorer groups values in interval buckets. For example, if a function consumed 240MB, Google will not store the value directly, but increase the counter of the bucket [128, 256). We can also retrieve the mean value over all buckets, which we also used for our results analysis.

GCR does not provide memory usage, but memory utilization. This value works similarly to the memory usage metric provided by the Docker runtime. According to the documentation, this metric is polled every 60 seconds[25], which means to get a decently accurate result for a benchmark, requests need to execute consistently across the full 60 seconds. When executing requests with the medium input, we could not sample a measurable difference in memory utilization. However, when invoking requests repeatedly with the large input, we could measure expected memory utilization, which is also what we used in our results.

## 6 Results

In this chapter, we present the results of our evaluations and general observations.

### 6.1 Observations

While evaluating the three different commercial cloud platforms, we encountered varying behavior in regards to threads, memory, and hardware information, which we present in this section.

#### 6.1.1 Threads and Virtual CPUs

Firstly, we look at the number of threads that are available for each configuration. Since we allocate as many threads as there are hardware threads or logical processors available to the environment, the terms are equal in this context. In chapter 4 we explained how vCPUs are allocated for each service. As seen in figure 6.1, we could observe at least

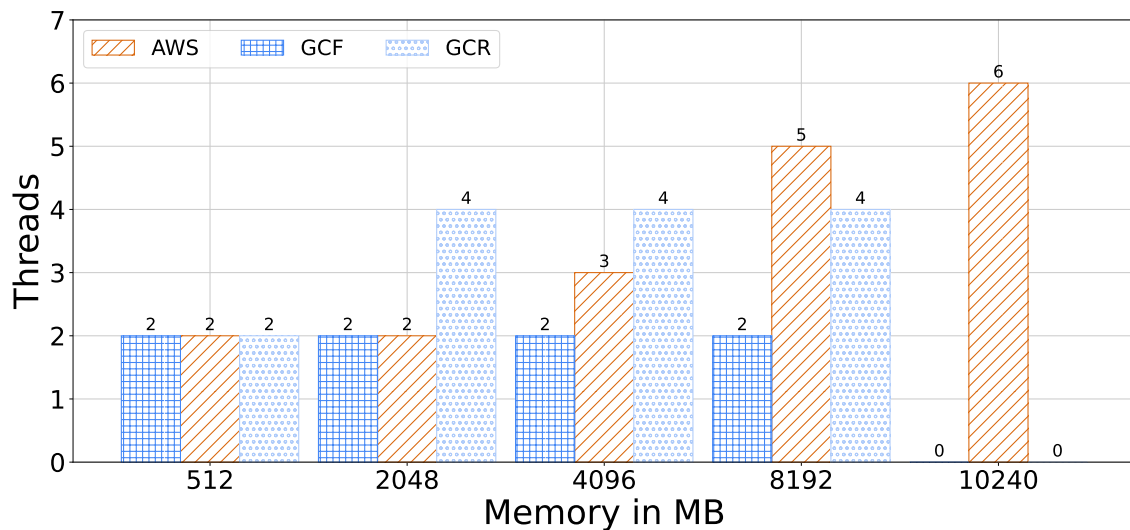


Figure 6.1: Thread allocation for all service configurations

2 threads for every configuration. This means that the amount of allocated vCPU is not equal to the number of threads. For example, in the lowest AWS configuration, we observe 2 threads, while not even getting allocated a full vCPU, according to the documentation of AWS [8]. Additionally, if we assume that we get a full vCPU per 1769MB allocated, we can conclude that the amount of threads is always rounded

up. For example, 4096MB translates to 2.3 vCPUs, but we observe 3 threads for that specific configuration. This behavior is the same for the other configurations. GCF always allocates 2 threads, regardless of the memory allocated to the function. As described in section 4.3, GCR allows the user to specify the amount of vCPUs directly, contrary to the other services. For the performance evaluations, we chose the maximum amount of vCPUs available for each configuration, as described in section 5.3. Since the lowest number of vCPUs allocated is 2, the number of threads allocated should not be surprising. However, when doing test runs with 1 vCPU, we could always observe 2 threads, which is consistent with the behavior seen in the other services. The only exception to this is the Java container, which for 1 vCPU only allocates 1 thread available to the user.

Next, we look at the processor models the functions ran on, which we collected by reading out `/proc/cpuinfo`. For AWS, we always observed the same processor model,

Platform	CPU Family	CPU Model	GHz	Microarchitecture
AWS	6	63	2.5	Haswell
GCF	6	79/85	2.3-2.8	Broadwell/Skylake
GCR	6	85	2.3-2.8	Skylake

Table 6.1: Processor information

which is specified in table 6.1. GCF had both Broadwell and Skylake processors with varying degrees of clock speed specified. For GCR we could only observe one Processor model, which also had varying degrees of clock speed in the same range as the GCF invocations.

### 6.1.2 Memory

Next, we look at the memory consumption of the different services and runtimes. Generally, it can be said that the maximum memory usage is relatively similar across services and runtimes with small differences. Figure 6.2 shows the memory consumption for the MVT benchmark with the input L. The functions and container were configured with 4096MB memory allocated. GCR consumes more memory than the other services, which can be attributed to the fact that a full container image was used. The container image we used may have more libraries installed than the functions. Additionally, the Java runtime consumes more memory than the other runtimes. Another example of Java having a higher memory consumption is, that at 512MB in AWS the Heat benchmark did not successfully execute due to out-of-memory errors, while it did execute in Go and C++.

Another behavior we could observe is related to the garbage collection of Go. We could reproduce an issue, where a function instance running the Go runtime would crash after consecutive invocations to the same instance. This behavior could be eliminated by manually running the Go GC before returning the result of the function. We assume that

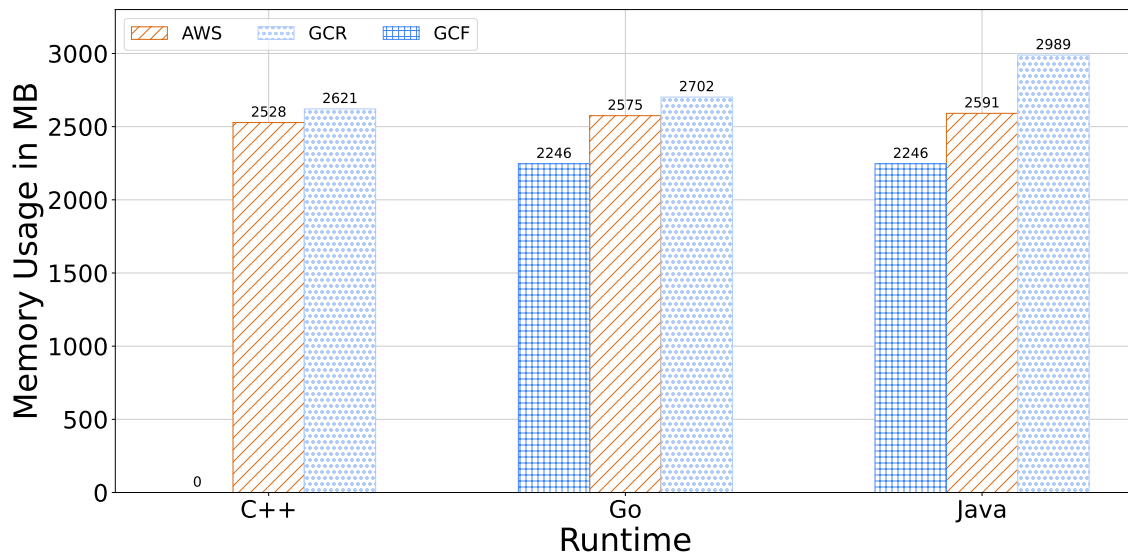


Figure 6.2: Memory usage of the function MVT for the 3 evaluated runtimes (Input L)

Go is not freeing the function memory after a function invocation. To confirm this, we did a couple of consecutive invocations and called `runtime.ReadMemStats()` for each. This method returns a bunch of memory statistics of the Golang runtime, including how many bytes are currently allocated. We called this method at the beginning and the end of an invocation. Tables 6.3 and 6.2 show our results for reading out the memory

Invocation	Memory Start	Memory End	GC Start	GC End
1	0	152	0	1
2	152	152	1	2
3	152	152	2	3

Table 6.2: Memory stats when not manually calling the GC

Invocation	Memory Start	Memory End	GC Start	GC End
1	0	0	0	2
2	0	0	2	4
3	0	0	4	6

Table 6.3: Memory stats when manually calling the GC at the end of a function invocation

stats. Memory Start is the allocated memory in MiB at the start of a function invocation, while Memory End is the same value at the end of an invocation. GC is the number of times the GC has been called within the lifecycle of an instance. Table 6.2 shows that the allocated memory for a function is kept across invocations, which can lead to problems for consecutive invocations. This works as long as the function memory is big

enough to allocate twice the memory of a single invocation. However, usually functions memory is chosen sparingly since the user wants to save cost as much as possible. In our evaluation runs, we had to manually run the GC, since for many configurations consecutive invocations would have crashed.

## 6.2 Performance Evaluations

In this section, we will present our performance evaluations for each service and runtime. We tested every benchmark on every function configuration, that is defined in section 5.3, with only a single thread, and with multiple threads available. That means we can compare single and multi-threaded performance and observe how those behave with increasing memory configuration.

### 6.2.1 Sequential Speedup

First we look at sequential speedups. In this case we look at the function execution times, when only using 1 thread. We then compare how much speedup is acquired by increasing the memory configuration of the function. 512MB is our baseline value, which we compare with the increased function configurations.

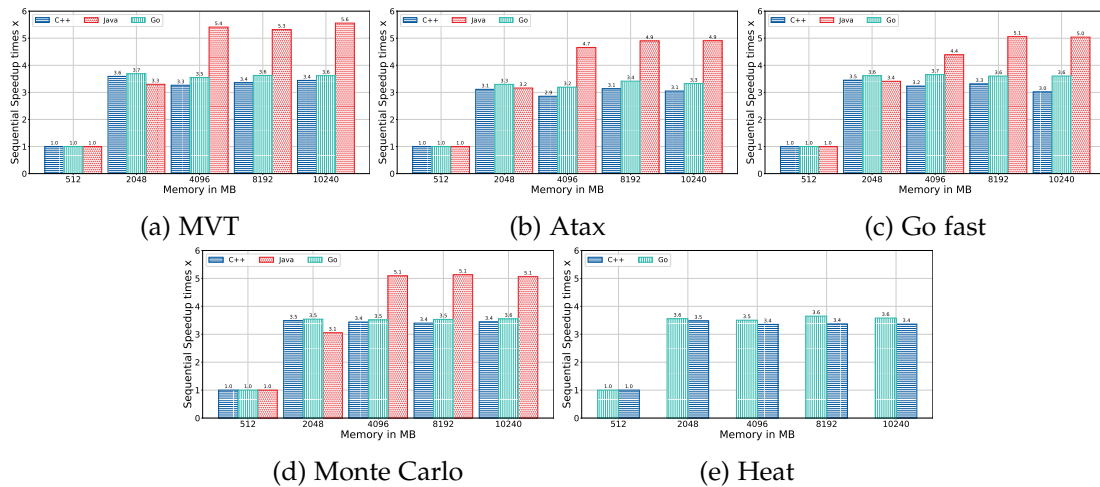


Figure 6.3: Sequential speedups obtained from increasing the function configuration for AWS. Comparing all function configurations with 512MB to see how much faster they run by only increasing the memory configuration.

Figure 6.3 shows the sequential speedups for all benchmarks in AWS. Each graph of a benchmark contains the speedups for the 3 runtimes from the lowest to the highest memory configuration. Since we compare higher configurations with 512MB, it appears as a baseline. For the Heat benchmark, only Go and C++ are depicted since the lowest configuration did not execute successfully for Java due to memory issues. As can be

seen for all the graphs, the speedups for Go and C++ are consistently at about 3.5. That means that by increasing the function memory from 512 to for example 2048MB, the program runs 3.5 times as fast. This is easily explained by the fact that at 512MB we get allocated a little less than a third of a full vCPU. All other evaluated configurations have at least 1 full vCPU allocated. The only exception to this rule is Java, where we get even higher speedups than on the other runtimes. This is possibly due to the automatic garbage collection having an impact on performance for lower heap sizes [35]. For low heap sizes, the JVM runs automatic garbage collection between loop iterations, which has a big impact on performance.

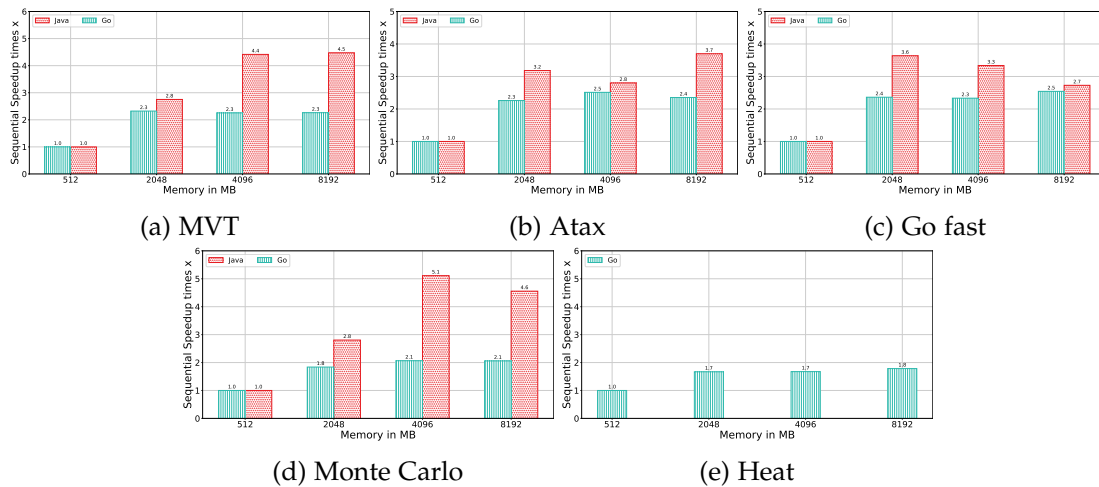


Figure 6.4: Sequential speedups for GCF

Figure 6.4 shows the sequential speedup for GCF. The graphs are structured in the same way as the ones showing sequential speedups for AWS, with the difference that we did not run C++ on GCF. According to the GCF documentation [24], processor clock speed should also increase by increasing the function memory. The configuration for 2048MB has 3 times the clock speed as 512MB, while 4096MB and 8192MB have 6 times the clock speed of 512MB. This can be observed to some extent for the Java runtime. However, this might partially be related to an increased heap size, similar to what was observed on AWS. The speedup obtained by the Go runtime is pretty consistently between 2 and 3. This implies that either the benchmarks consistently get no benefit of an increased clock speed from 2.4GHz to 4.8GHz, or that the resources for a single thread are not increased.

Lastly, we look at the sequential speedups obtained for GCR, by increasing function memory and the amount of vCPUs. Figure 6.5 shows the sequential speedups for GCR, again structured in the same way as the other 2 services. For Go and C++ no consistent sequential speedup can be observed, by increasing the container memory or the amount of vCPUs. This implies that the type of CPU never changes across the different configurations. Furthermore, there is always at least one full vCPU assigned to the container, which means that there is no speedup between the lowest and higher

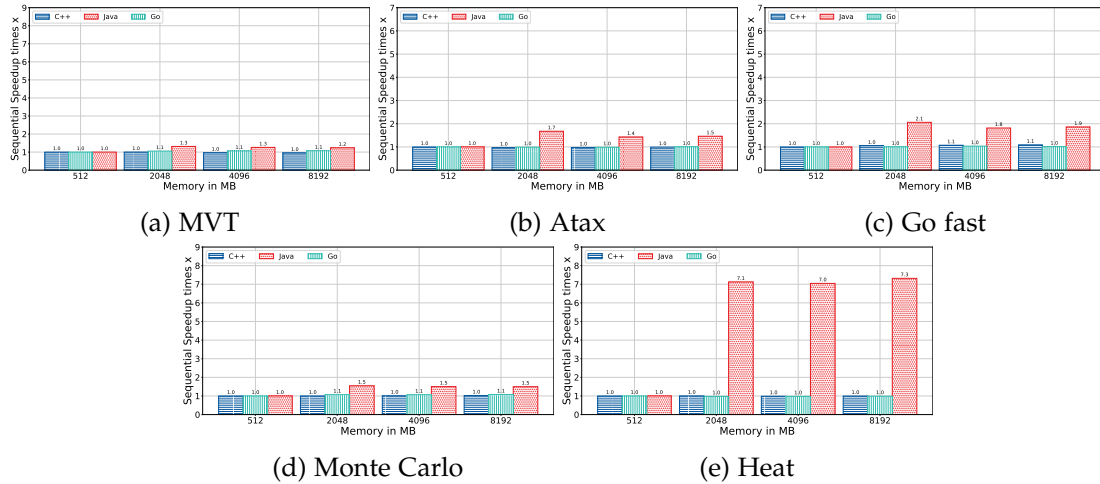


Figure 6.5: Sequential speedups for GCR

configurations. Since we always use only 1 thread, the amount of vCPUs has no impact on the performance. For every Java implementation, we obtain varying degrees of speedup. This also strongly supports the argument that the Java implementations benefit from purely increasing the allocated memory if the memory is low enough at the starting point. Although the results in figure 6.2 show that the memory usage is the highest for GCR and Java, it is the only service that was able to successfully run the Heat benchmark with medium-sized input and a memory allocation of 512MB. However, our results show that the invocation count is reset to 1 for each consecutive invocation, which means that a new instance was used. This would also explain high sequential speedups for the higher memory configurations, where there was not a cold start for every invocation.

In conclusion for the sequential speedups, it can be said that the Go and C++ runtimes generally don't benefit from increased memory, but from increased CPU time and clock speed. The Java runtime additionally benefits from increased memory, if the heap size was low enough to trigger the garbage collection between loop iterations.

## 6.2.2 Parallel Speedup

Next, we look at the speedup that is obtained by parallelizing the programs, which means we now take full advantage of the hardware that is available to the developer. In the following graphs, we compare single- and multi-threaded executions of the same function or container on the same configuration. We always use as many threads, as are available for a specific configuration. We compare the acquired speedups across benchmarks, runtimes, and configurations.

We start again with AWS and look at all benchmarks across configurations and runtimes, which can be seen in figure 6.6. The first observation we can make is that even though at 512MB there are 2 threads available, using both threads doesn't yield



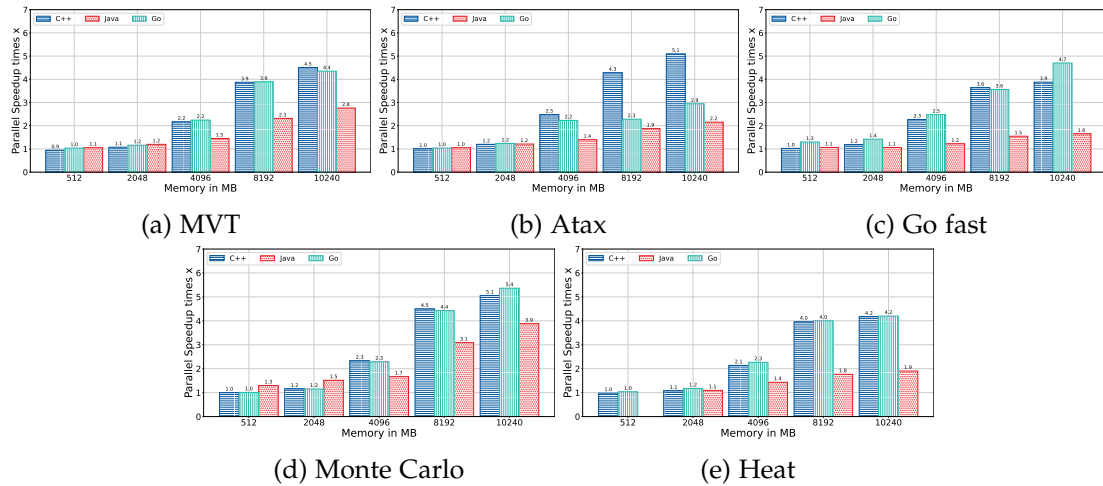


Figure 6.6: Speedup between single- and multi-threaded execution of a specific function configuration for AWS.

any significant speedup. This is because there is no full vCPU allocated for this configuration. At 2048MB a small speedup is obtained. However, since there are still not 2 full vCPUs allocated, both threads are not fully utilized. For the configurations 4096MB and upwards, the speedup obtained is similar to the threads available for the specific configuration. Especially the Go and C++ implementations yield the expected speedups across most benchmarks. The Java implementation, however, has worse speedups consistently across all benchmarks, when compared to the other 2 runtimes.

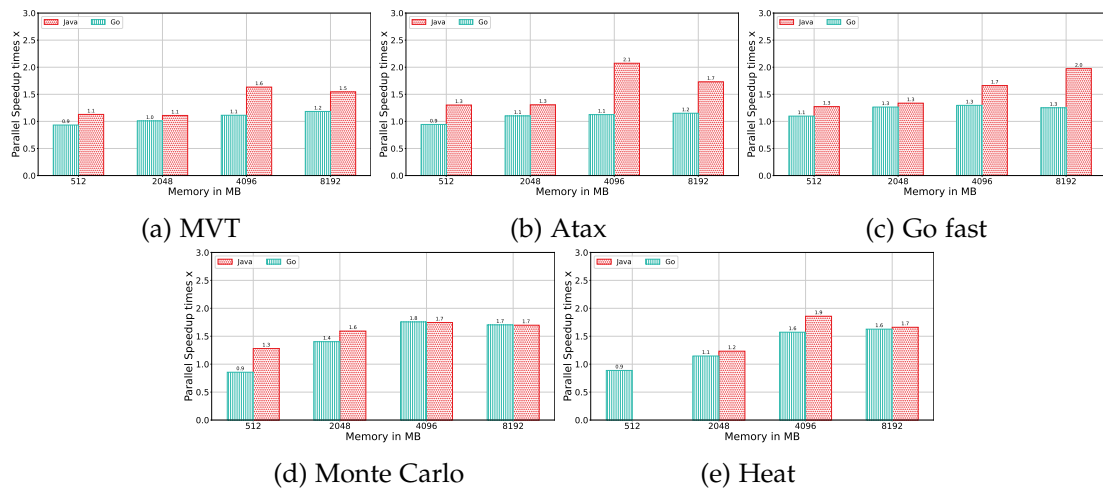


Figure 6.7: Speedup between single- and multi-threaded execution of a specific function configuration for GCF.

Next, we look at the speedups obtained in GCF, which can be seen in figure 6.7. From the graphs, we can obtain an upper limit of 2 in terms of speedup, which correlates to

the number of allocated threads, which are always 2. However, there are still differences between the different configurations. More specifically, we can observe that there is usually a small jump from 2048MB to 4096MB and above. This implies, since no additional threads are allocated, that one of the threads is more powerful or has a higher share in clock time. This is also in line with the observations made in subsection 6.2.1, because according to the documentation the clock speed increases from 2048MB to 4096MB, but no sequential speedup could be observed. If we assume that the clock time is spread across both threads, and only at full capacity from 4096MB and above, the full speedup of 2 can also only be obtained from 4096 and higher. Another point that stands out, is that the Java implementations generally yield more speedup, than the Go counterparts. Since GCF uses an older runtime version of Go than AWS Lambda and GCR, we tried to reproduce these bad results on GCR. We executed some runs with the container image `golang:1.13-buster` and observed similar issues as on GCF.

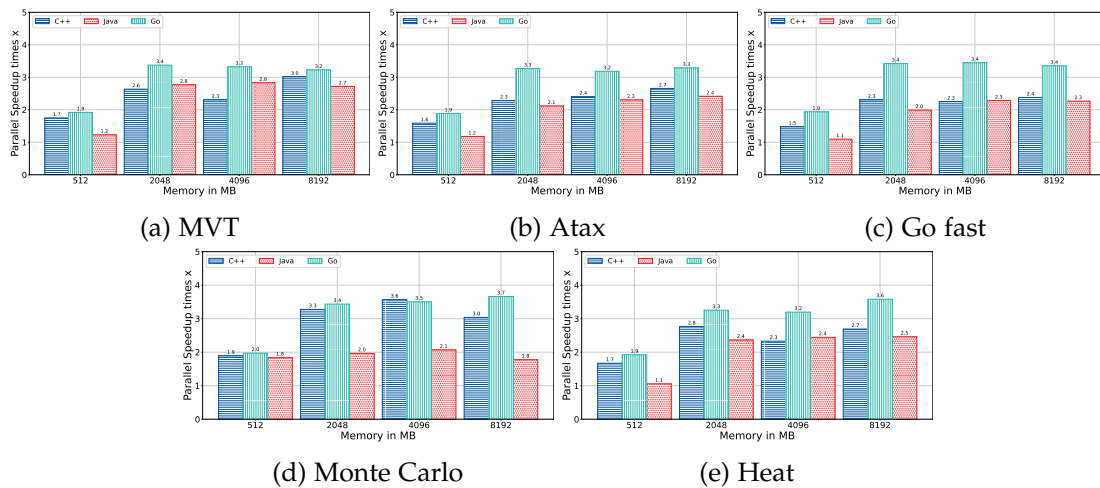


Figure 6.8: Speedup between single- and multi-threaded execution of a specific function configuration for GCR.

Lastly, we look at parallel speedups obtained with GCR. Since this time the highest vCPU configuration is 4, the speedup is also capped out at 4. Only the lowest configuration at 512MB has 2 vCPUs allocated. In figure 6.8 we can see that the speedups for the lowest configuration are around 2 for C++ and Go while being around 1 for Java. This can be explained through Java being bottlenecked by the garbage collection for lower memory configurations. Similar to the parallel speedup observed on the AWS platform, the Java implementation lags a little behind the other 2 runtimes. For most benchmarks, most notably seen in 6.8b and 6.8c, the speedup obtained by the C++ implementation is a little bit less than expected. Since this can be observed across all benchmarks it is reasonable to assume that the overhead of invoking a request via CaaS is higher than via FaaS, at least for the C++ runtime.

Generally, the speedups obtained are in line with what is expected, with some small exceptions and differences. Across all services, the Java implementations obtain a little

less parallel speedup.

### 6.2.3 Function Execution Times

Before we look at the total costs of the executions, we look at the execution times of the benchmarks across the services. This gives us a better impression of what services and runtimes perform better overall and we can take that into account when looking at the costs. Furthermore, the total execution time has to be taken into account, when looking at the cost. This is especially important for the services of the GCP, which we will explain in subsection 6.2.4.

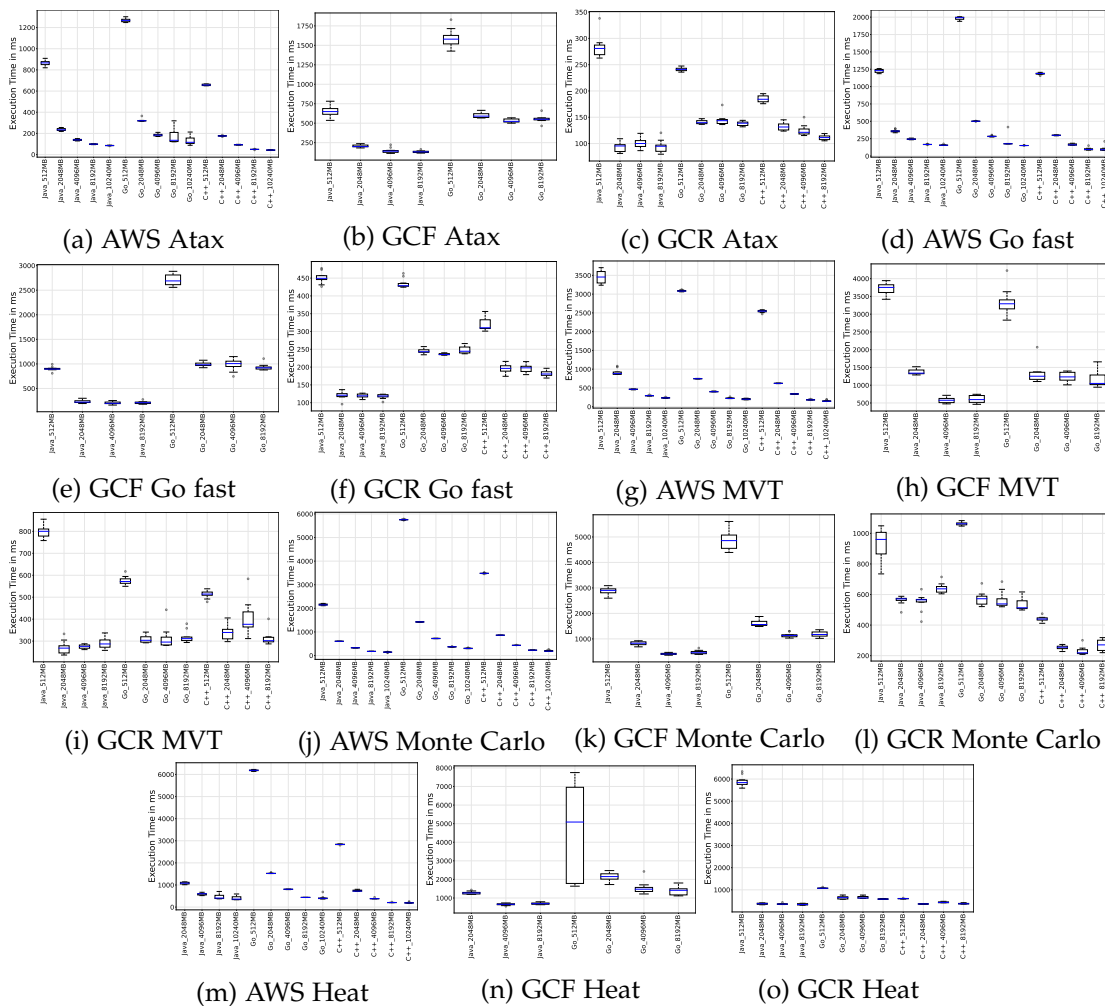


Figure 6.9: Multi-threaded execution times for all benchmarks (Input M) across services. Each sub-graph is grouped by runtime and then sorted by memory configuration.

In figure 6.9 the execution times of all benchmarks are depicted across services and

runtimes. The Java implementations often perform the worst on the lowest configurations, while they perform similar or better in higher configurations to their counterparts. As mentioned before, this is attributed to garbage collection having an impact on the performance of lower memory configurations. However, for higher configurations the performance is similar. Only for higher thread counts, the Java implementations fall behind again. If we compare both FaaS services, the differences are small for the lower memory configurations. Since GCF doesn't allocate more threads for higher memory configurations, the execution time stagnates at a certain point, while the execution time for AWS continues to go down.

If we compare GCR with the other services, we can observe that the total execution time is a lot lower for 512MB. This is due to the configuration since we allocate 2 vCPUs. On the higher configurations, GCR is close to AWS in terms of execution time.

#### 6.2.4 Cost Comparison

After evaluating the performance, we now present a cost comparison between the different services and runtimes. We take the mean function execution time and calculate the costs for 1 million function invocations. In the calculation we assume there is no free tier for all services. In figure 6.10 all formulas to calculate the costs are listed. Additionally to the compute time, each service provider charges a fixed cost per 1

$$\begin{aligned}
 \text{AWS}(t) &= 1000000 * t * \text{price\_per\_ms} + 0.2 \\
 \text{GCF}(t) &= \left( \frac{m}{1024} * t * 1000000 * \frac{0.0000025}{1000} \right) + \left( \frac{p}{1000} * t * 1000000 * \frac{0.0000100}{1000} \right) + 0.4 \\
 \text{GCR}(t) &= \left( \frac{m}{1024} * t * 1000000 * \frac{0.0000025}{1000} \right) + \left( c * t * 1000000 * \frac{0.00002400}{1000} \right) + 0.4
 \end{aligned}$$

Figure 6.10: Cost formula for all, with  $t$  = time in ms,  $m$  = function memory in MB,  $p$  = processor MHz,  $c$  = number of vCPUs

million function invocations. AWS charges \$0.2, while Google charges \$0.4 for their respective services [5] [24]. The price-per-ms in the AWS formula is dependent on the function memory. All functions were executed in U.S. East (N. Virginia). The cost formula for GCF is split into GB-seconds on the left side and GHz-seconds on the right side. Similar to GCF, the GCR formula is split into GB-seconds and vCPU-seconds. Both the number of vCPUs and memory allocated influence the final price [26]. Since we ran all our functions in `us-central1`, we calculate the cost with the values for Tier 1 pricing for both GCF and GCR. While AWS rounds the time to the nearest increment of 1ms, Google rounds the time up to the nearest increment of 100ms.

In figure 6.11 the costs for all benchmarks in AWS are shown. The red bars represent the cost for single-threaded execution, while the blue bars show the costs for multi-threaded execution. In the beginning, the prices for the single and multi-threaded executions are about the same, which is in line with the observed speedups. The cost

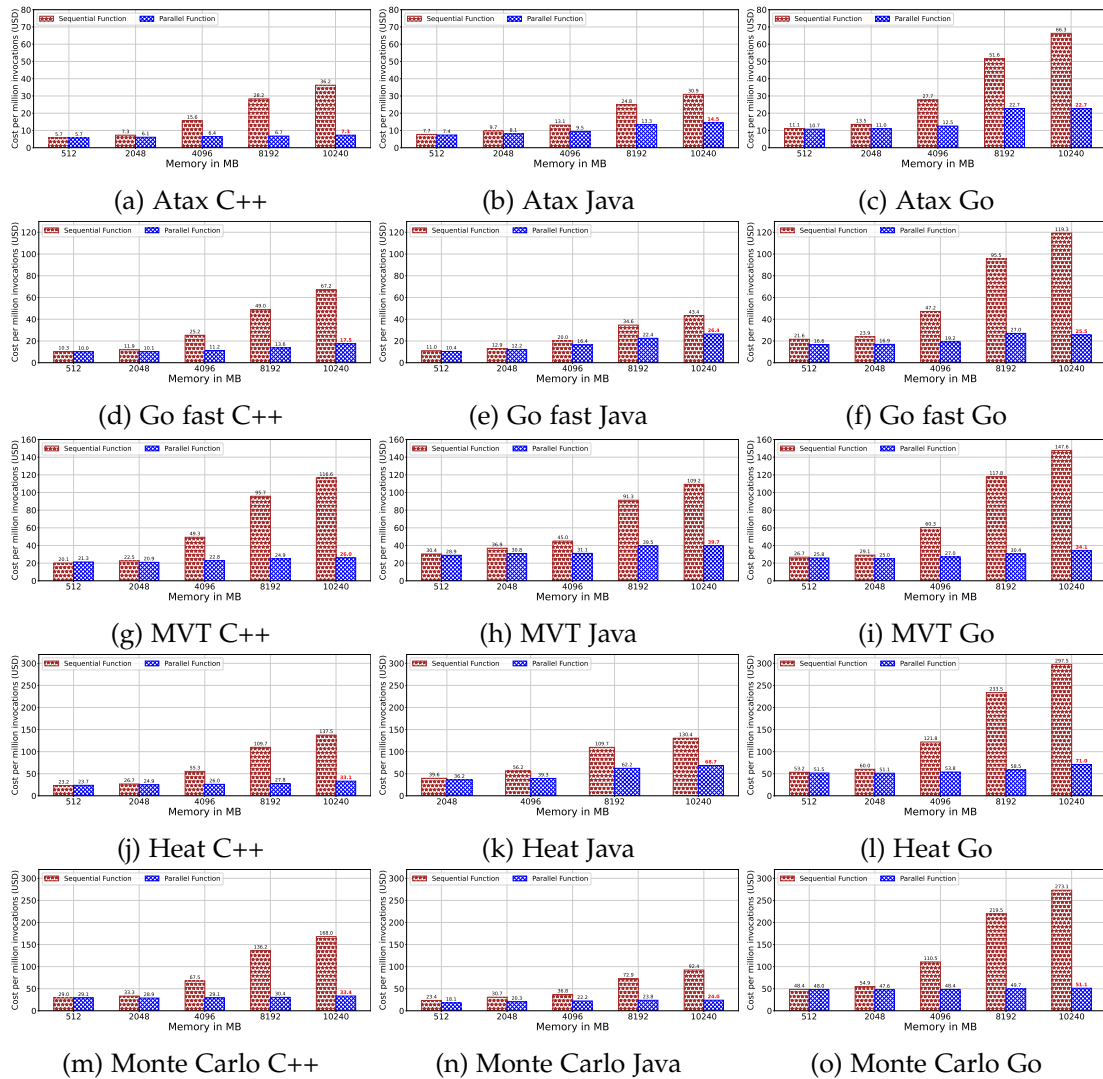


Figure 6.11: Cost comparison between single- and multi-threaded execution for AWS. Parallel functions with the highest percentage in cost savings are marked in red.

strongly increases for sequential execution, even with the sequential speedup obtained. The cost for the parallelized version also increases with higher configurations, but much slower. For example, the parallel C++ implementations increase by about 30% from the lowest to the highest configuration, while the sequential version increases by over 500%. Overall C++ is the cheapest when parallelizing, while Java is the cheapest on sequential versions. There are some exceptions to this. For example, if we compare figure 6.11h and 6.11i, we can see that for low memory the Java implementation is more costly. The sequential functions of Go are, on average, more expensive than C++ and Java. For the higher configurations, small differences in execution time can make a huge difference.

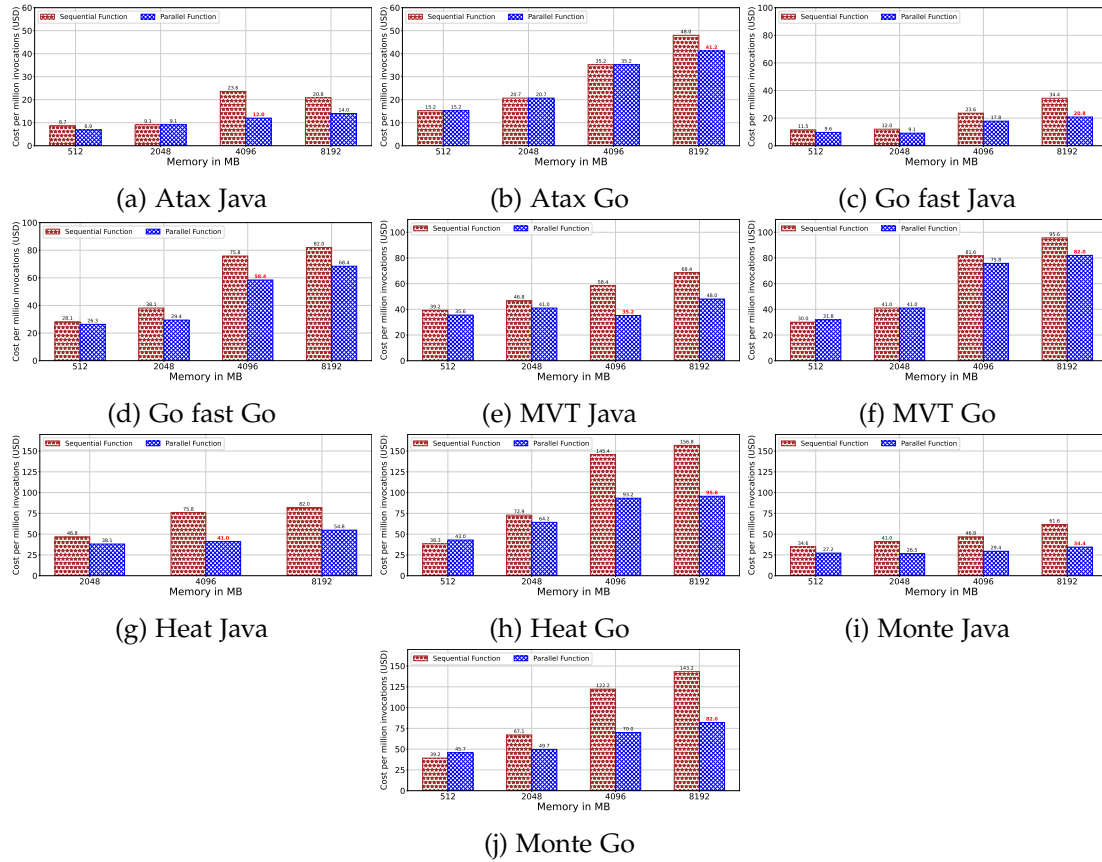


Figure 6.12: Cost comparison between single- and multi-threaded execution for GCF. Parallel functions with the highest percentage in cost savings are marked in red.

Even though we could see in figure 6.9g that the execution times are very close to each other, the differences in cost are quite high. The difference between C++ and Java for the parallel versions of 10240MB is about 50-100ms in some benchmarks but the price increases by 50%. Overall the highest cost saving between the sequential and parallel versions is 81%.

Next, we look at the costs of GCF in figure 6.12. On the lower configurations, the costs are again very close. However, due to the limited speedup of GCF, the costs savings are limited as well. Lower total execution times additionally reduce the effect that parallelizing the program has since the execution time is rounded to the nearest increment of 100ms, which can make a huge difference, especially in higher configurations where the cost per 100ms is so high. If we compare figures 6.12a, 6.12b (low overall execution time) with 6.12i, 6.12j (higher overall execution time), this difference becomes clear. Even though both benchmarks have similar speedups, Monte Carlo has a lot better cost savings, when parallelizing the program both for Java and Go. Overall the Java implementation is cheaper than the Go implementation. For GCF,

the maximum amount of cost savings we observed between the sequential and parallel versions is 49%.

Lastly, we look at the costs of GCR. Since we allocate a minimum of 2 vCPUs, we can see in figure 6.13 that the parallel implementation has always at least the same or lower cost. For configurations with the best speedups, we achieve cost savings up to 71% between single and multi-threaded runs. However, this percentage is rarely achieved for lower execution times, since Google rounds up to the nearest increment of 100ms. Another interesting fact is that for Java implementations the cost can be reduced when increasing the configuration from 512MB to 2048MB. This seems to again indicate, that

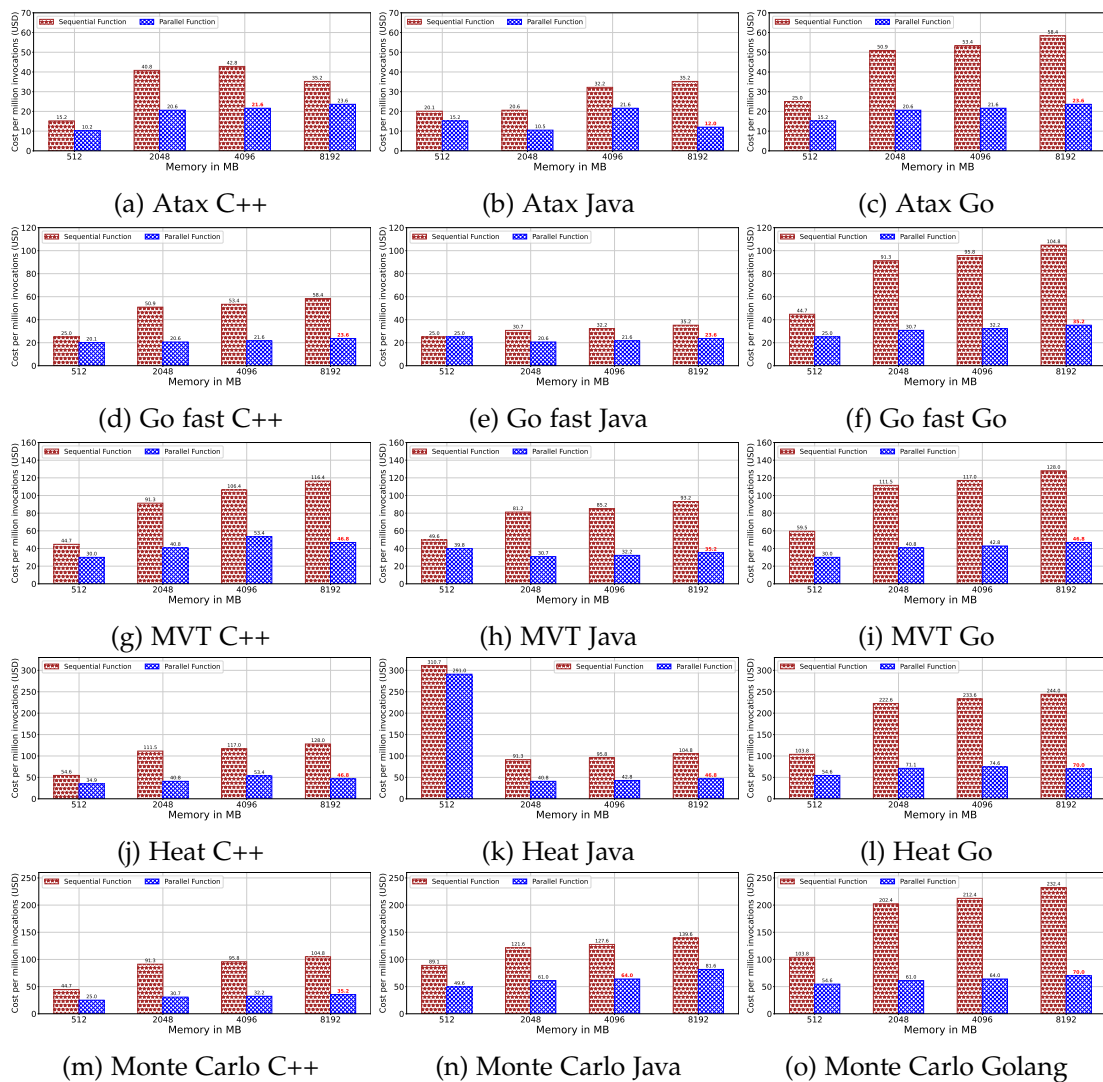


Figure 6.13: Costs comparison between single- and multi-threaded Execution for GCR. Parallel functions with the highest percentage in cost savings are marked in red.



the Java implementation is severely bottlenecked by memory for lower configurations. Apart from that, the cost rises consistently with higher memory configuration. There are some exceptions to this, for example in 6.13g the cost decreases in the parallel version from 4096MB to 8192MB. Looking at the results, the fastest execution times are relatively the same, but the higher memory configuration performs a little bit more consistently, which reduces the overall mean execution time. The sequential implementations of Go cost, on average, the most.

### 6.2.5 Impact of Cold starts

Finally, we want to discuss the impact of cold starts in our experiments. In general, cold starts will not be one of the main topics of this thesis, since the focus is more on the effects and efficiency of parallelization. Additionally, our results show that cold starts are equally long with sequential and parallel implementations. That means that the cold starts decrease the fraction of the function that can be run in parallel. If this fraction is large enough, it also has an impact on the obtained speedup. As described in chapter 5, consecutive invocations for a single configuration were on the same instance, which means that at most 1 of 10 runs is executed on a cold instance. Figure 6.14 shows the

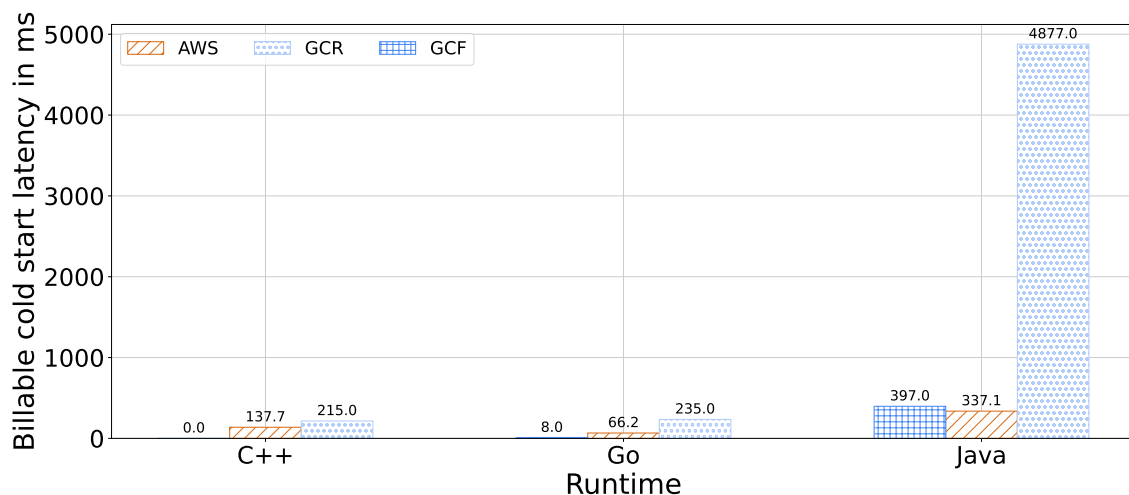


Figure 6.14: Impact of billable cold start latency

billable time in milliseconds that is caused by a cold start for each service and runtime. We focus on the difference in billable time between the invocation on a cold and a warm instance and completely neglect latencies, which are not documented by the service provider. This is because only the difference in billed time has an impact on the cost of execution of our functions. This is also why some of the values seem very low, for example, the Go runtime on GCF. However, when comparing the execution time of a cold and a warm function, the difference was barely noticeable. The Java runtime on GCR stands out with very high latency. CaaS provides the container as a server, where the creation of the former seems to take more time in Java, than with the other containers.



For Java, some of the time difference of a cold and warm instance might be also due to the JVM warming up. This also might impact the time difference between the first and second invocation of a particular instance. In our results, we could not observe a significant difference between benchmarks or function memory configurations.



## 7 Discussion

In this chapter, we want to discuss the results shown in the previous chapter. Since one of the biggest reasons for users to use serverless services is cost, we want to discuss in which cases it makes sense to parallelize functions.

### 7.1 Costs

A lot of the presented results in chapter 6 show that by parallelizing a program the cost can be reduced by a significant margin. To achieve similar results, a few conditions have to be met.

**vCPUs:** To achieve a noticeable speedup, the function needs to have more than 1 full vCPU allocated. For AWS Lambda this means a function has to have more than 1769MB allocated. GCF gets small speedups for lower memory configurations, but the full speedup of 2 can only be achieved from 4096MB and above. In GCR, vCPUs are allocated independently from memory and for every memory configuration, at least 2 full vCPUs can be allocated.

**Memory:** As previously mentioned, GCR allows to allocate 2 vCPUs even in low memory configurations, however for certain runtimes the memory can lead to performance degradations. If both threads are bottlenecked due to memory or garbage collection issues, parallelization has little to no impact.

**Parallel execution:** When all hardware conditions are met, the cost savings depend on the fraction of the program that can be executed in parallel. The higher this fraction is, the more cost can be saved.

**Cold starts:** This ties into the previous point since cold starts are a fraction of the function, which doesn't get any benefit from parallelizing the program. The more cold starts there are and the longer they take, relative to the total function execution time, the fewer costs can be saved.

Even if not all of these points are met, parallelizing parts of a program rarely has any downsides. In our results, there were almost no runs where the single-threaded execution was faster, even if there was less than 1 full vCPU allocated.

Another interesting question is if there are situations where the user can save cost by increasing the function memory or allocate more vCPUs, given that a program is already parallelized. First, we look at AWS purely from a Cost perspective. We take the

cost formula from equation 6.10 with the respective cost per millisecond values for our configurations. Figure 7.1a shows the cost for a given function memory configuration,

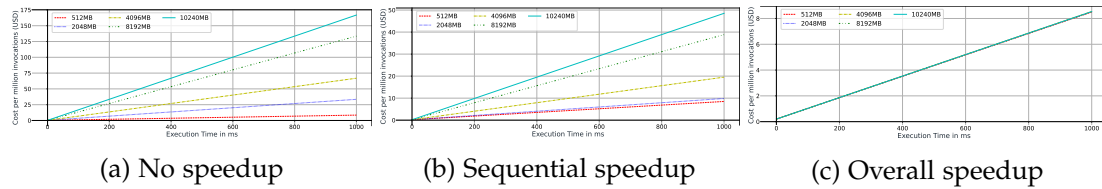


Figure 7.1: Linear cost function for each function configuration. Cost increases linearly with increasing execution time for a fixed function configuration.

assuming the execution time stays always the same. However, we already know that this is not the case, since 512MB has less than a full vCPU allocated. Our results in section 6.2.1 show that the speedup from 512MB to the other configurations is roughly 3.5. If we calculate the exact fraction of 1769 and 512, we get 3.45, which will be used going forward for this calculation. Figure 7.1b shows the costs again, but with the time divided by 3.45 for all configurations from 2048MB and above. We can see that the lines got closer to each other, but the higher configurations still are more expensive. For the last graph in 7.1c we assume that 100% of the function can be executed in parallel. We divide the time again, by the amount of speedup that can be achieved in an optimal case when compared to 512MB. For example  $\frac{10240}{1769} * \frac{1769}{512} = 20$ . If we apply this to all configurations, the prices are the same. This is an ideal case, which is almost impossible to achieve. In some of our results, we got close, but the costs still rose for increasing memory. This also means that, without memory bottlenecks, it isn't possible to save costs by increasing the function configuration.

Since the cost formulas for both GCF and GCR are also linear, we can compute the required speedups, which would be necessary to not increase the cost for a function, while increasing the memory. For example for GCF, if we compare 512MB with 4096MB, the function would have to execute about 6.25 times as fast to have the same cost. Since in GCF we have a maximum of 2 threads, that would mean the sequential speedup would need to be higher than 3. GCR is a little bit different since we can configure the amount of vCPUs and memory independently. In the configurations we chose to evaluate GCR, we always increased memory as well, which is why the costs mostly increase for higher configurations. However, if we take the same memory configuration and allocate more vCPUs, the cost can potentially decrease. This is because the cost for memory decreases, while the cost for computing time stays the same if 100% of the program runs in parallel.

In conclusion, it can be said that for FaaS increasing the function configuration, in most cases increases cost, and in optimal cases breaks even. For CaaS it might be worth allocating more vCPUs, depending on the program and runtime.

## 7.2 Runtimes

In this section, we look at the differences between the 3 evaluated runtimes. If we look purely at cost of the parallel runs, C++ overall outperforms Java and Go. Using C++ makes sense if the parallel fraction of the program is huge and the runtime by the provider includes the necessary tools to interact with the serverless environment. Setting up the C++ functions in Lambda was more complex than with the other languages, but the overall reduction in cost can be worth it. However, the user has to keep in mind that not all cloud providers support C++, which increases vendor lock-in to specific providers and services.

Go on the other hand is widely supported and with out-of-the-box runtimes, even for different versions. Furthermore, the speedups obtained with the parallel version are pretty high in most cases. Consequently, the potential cost savings by parallelizing the function are also high. The language was designed with concurrency in mind. Splitting the workload into goroutines is very easy and does not require any external libraries or dependencies.

Java is also available supported on all main serverless platforms. The overall performance in terms of execution time is pretty good for Java. The Java implementations are close to the results of the other runtimes for higher memory configurations and frequently outperform them for the medium memory configurations. Especially in GCF, Java is faster than Go in most cases. However, Java has 2 problems, which are not present in the other two languages. Firstly, if the memory is configured too low, the performance implications are quite high. It is reasonable to plan with a higher memory budget in Java. The second problem comes with the JVM. The JVM usually needs to warm up and the performance often still increases after the 2nd or 3rd consecutive invocation. In a serverless environment, where invocations often spawn new instances, the performance impact can be huge. Additionally, cold starts are higher than the other runtimes, which ties into the same problem.

Overall there are use cases for all the 3 languages. And if the user is already tied to a given language, there are scenarios where it makes sense to parallelize the program for all of them.

## 7.3 Tradeoff Cost vs. Speedup

So far we have discussed mostly the implied cost savings when considering the parallelization of a program. In most cases, this is already enough, since most use cases for serverless are to save cost. However, there might be some instances, where the user values reducing the execution time over saving costs up until a certain point. For example, if the invocation of a serverless function is tied to a user-facing website, where the response times are often very important. If the user has to wait for the response of the function and parallel execution of the function can reduce the response time, the developer might prefer having a higher cost to reduce the overall response time.

For cases like these it can be interesting to look at the ratio of cost increase to reduction of overall execution time. Lets define the ratio as cost efficiency with the following formula:

$$Efficiency(i) = \frac{Time_{512MB}}{Time_i} \div \frac{Cost_i}{Cost_{512MB}}$$

We use the function configured with 512MB as a baseline and compare it with higher-configured functions. We divide the function execution time of 512MB by the time of a given function configuration to calculate the speedup obtained between the two configurations. On the other side of the division, we calculate the cost increase from 512MB to a given function configuration. We then divide the speedup by the cost increase. For example, if the speedup between 512MB and 2048MB is 2, but the cost is also doubled, the result is 1. One possible flaw of this formula is that it scales linearly, which in reality is often not the case. The jump from from 1 to  $\frac{1}{2}$  in execution time is often more impactful than the jump from  $\frac{1}{2}$  to  $\frac{1}{4}$ . This is not factored in in this formula, but it can still give a rough idea of how much more money the user has to spend to save function execution time.

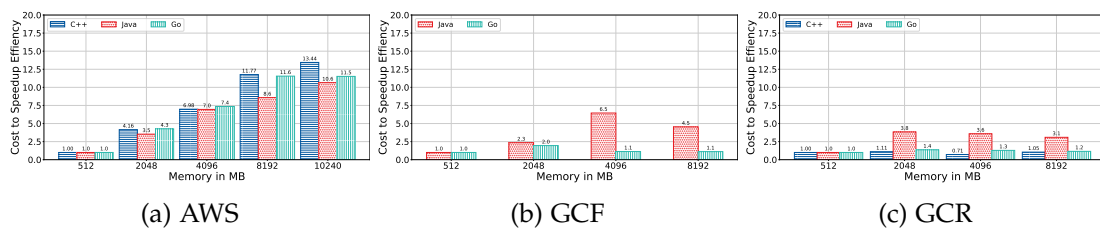


Figure 7.2: Cost efficiency for MVT of all service providers

Figure 7.2 shows the calculated cost efficiency using the above formula for all services. What stands out immediately, is that AWS performs the best in regards to this metric. This is because the speedups obtained are very high, while the costs stayed relatively the same and only increased by a small amount. The other 2 services are in most cases above 1, but lag behind AWS. Only Java has a decent efficiency. The reason for Java being higher than the others for GCF and GCR is that Java is a lot slower for the 512MB configuration, but the overall speedup for the following configurations is great. For Go, it is often costly to reduce execution time in GCF and GCR, while still being quite cheap on AWS. In general, GCR is quite low, since we already start with 2 vCPUs. This means, that if a user is looking for the cheapest way to speed up his functions, AWS is the best choice out of the 3 evaluated services. When looking to speed up a Java function, it is often efficient to increase function memory for all services.

## 8 Conclusion

In this thesis, we analyzed 2 major cloud providers and 3 of their serverless offerings. We ran 5 different benchmarks and applications single- and multi-threaded to evaluate the parallel performance of these services. We also collected information on the hardware that the functions and containers were running on. We showed that the number of hardware threads available to the developer does not always equal the number of vCPUs allocated to the function or container. Our results also show that it is highly likely that GCF doesn't allocate 2 full vCPUs until 4096MB memory configuration.

Furthermore, we show that by parallelizing the programs, the developer can save a significant amount of cost. The more vCPUs allocated to the function, the bigger the effect. For AWS Lambda we could observe cost savings up to 81%, for GCF up to 49%, and for GCR up to 71%. For AWS Lambda and GCF, we gave arguments on why the cost cannot be reduced by increasing the function configuration if the application is not bottlenecked. Since we can independently configure vCPUs and memory for GCR, there are situations where we can save costs by increasing the amount of vCPUs allocated to a container, given that the application achieves good speedups.

### 8.1 Shortcomings

Although our results answer the research questions we set out in section 1.2, not all of the results were as expected. We observed some outliers we were not fully able to explain. The first example is the results for Go on GCF. The microbenchmarks performed badly, with the highest speedups achieved being around 1.3. The runtime for GCF is 1.13, while for AWS Lambda and GCR we used 1.16. We were able to reproduce the bad results by changing the Docker image to `golang:1.13-buster` for GCR, but we did not investigate the reason for this behavior further. Furthermore, the speedups achieved for the Java implementations failed to meet our expectations across all services. This behavior was worse for applications with a higher amount of separate parallel regions, which indicates synchronization overheads. Since the standard library of Java doesn't provide synchronization methods themselves, we implemented it ourselves. Our synchronization implementation is possibly not as good as it could be. Additionally, the results of the JVM are inconsistent, which makes it harder to narrow down the underlying problem. It can also be argued that our benchmarks don't represent real scenarios close enough. We only chose 2 applications for our evaluations and both of these applications still include scientific calculations, which are not always present in everyday applications.

## 8.2 Future Work

In this section, we present how our work could be improved and extended to gain more insights into parallel execution in serverless environments.

### 8.2.1 Services

The evaluations could be extended to additional cloud providers, for example, Azure Cloud Functions. For their standard consumption plan, only the memory is configured [42]. It would be interesting to see the number of threads available for each configuration and how much the benefit is for parallel executions. Furthermore, Microsoft offers custom handlers to support custom language runtimes [43]. It could be interesting to see how efficient they are in terms of parallel execution and potential cost savings. Another service that could be evaluated is IBM Cloud Functions. For both the services of IBM and Microsoft, the limits for the maximum amount of memory that can be allocated are rather low, compared to AWS Lambda and GCF. That leads us to believe that the expected benefits are lower than what our results show us, but it would be interesting to confirm this and gain additional insights into the underlying hardware.

### 8.2.2 Runtimes

To extend the scope of the evaluation, we could add additional runtimes. It would be particularly interesting to research the difference between compiled and interpreted languages, such as Nodejs, Python, or Ruby, all of which are widely supported across cloud providers. For all runtimes, it could be interesting to gain more insights about the runtimes in the serverless environments and how they are configured. Runtime-specific implementations, like garbage collection, could be sampled more for their performance impact and the differences across runtimes.

### 8.2.3 Benchmarks

As previously mentioned, the number and variety of benchmarks in this thesis are not ideal and could be improved. It would be useful to add benchmarks that utilize different parts of the system. For example, applications that heavily interact with the file system or make a lot of network calls. Because of the differences in underlying architecture, the performance for the different service providers could heavily vary. Another approach could be to evaluate applications that use threads to process independent tasks concurrently, instead of splitting one big workload into smaller pieces.

### 8.2.4 Inter- and Intrafunction parallelism

As described in section 3.1, many existing approaches split the workload across functions to execute it in a distributed fashion. For many of these approaches, the overall synchronization overhead could be reduced by using fewer more powerful functions,



which execute their part of the workload in parallel. It could be interesting to see if any of these solutions can be improved by using parallel executions within a function.



# List of Figures

4.1	Behind the scenes of a Lambda function invocation . . . . .	15
4.2	Architectural structure of an AWS Worker instance . . . . .	16
4.3	Architecture of a Firecracker MVM . . . . .	17
4.4	Life cycle of an AWS Lambda execution environment . . . . .	18
4.5	Google Cloud Platform cluster architecture . . . . .	20
4.6	gVisor Layers Overview . . . . .	21
4.7	Billable time of a Google Cloud Run instance . . . . .	23
5.1	Monte Carlo simulation to estimate digits of $\pi$ . . . . .	32
5.2	Execution script . . . . .	35
6.1	Thread allocation for all service configurations . . . . .	39
6.2	Memory usage of the function MVT for the 3 evaluated runtimes (Input L)	41
6.3	Sequential speedups obtained from increasing the function configuration for AWS. Comparing all function configurations with 512MB to see how much faster they run by only increasing the memory configuration. . . .	42
6.4	Sequential speedups for GCF . . . . .	43
6.5	Sequential speedups for GCR . . . . .	44
6.6	Speedup between single- and multi-threaded execution of a specific func- tion configuration for AWS. . . . .	45
6.7	Speedup between single- and multi-threaded execution of a specific func- tion configuration for GCF. . . . .	45
6.8	Speedup between single- and multi-threaded execution of a specific func- tion configuration for GCR. . . . .	46
6.9	Multi-threaded execution times for all benchmarks (Input M) across services. Each sub-graph is grouped by runtime and then sorted by memory configuration. . . . .	47
6.10	Cost formula for all, with $t$ = time in ms, $m$ = function memory in MB, $p$ = processor MHz, $c$ = number of vCPUs . . . . .	48
6.11	Cost comparison between single- and multi-threaded execution for AWS. Parallel functions with the highest percentage in cost savings are marked in red. . . . .	49
6.12	Cost comparison between single- and multi-threaded execution for GCF. Parallel functions with the highest percentage in cost savings are marked in red. . . . .	50

6.13	Costs comparison between single- and multi-threaded Execution for GCR. Parallel functions with the highest percentage in cost savings are marked in red. . . . .	51
6.14	Impact of billable cold start latency . . . . .	52
7.1	Linear cost function for each function configuration. Cost increases linearly with increasing execution time for a fixed function configuration.	56
7.2	Cost efficiency for MVT of all service providers . . . . .	58

## List of Tables

5.1	Benchmark configurations for small, medium, and large input size . . . .	34
5.2	InfluxDB entry structure . . . . .	37
6.1	Processor information . . . . .	40
6.2	Memory stats when not manually calling the GC . . . . .	41
6.3	Memory stats when manually calling the GC at the end of a function invocation . . . . .	41



# Listings

2.1	Minimal application code for CaaS in Go . . . . .	9
5.1	Example of loop parallelization using OpenMP . . . . .	26
5.2	Example of loop parallelization using goroutines . . . . .	27
5.3	Example of loop parallelization using Java ExecutorService . . . . .	29





# Bibliography

- [1] Adzic, Gojko and Chatley, Robert. "Serverless Computing: Economic and Architectural Impact." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, 884–889. ISBN: 978-1-4503-5105-8. DOI: {10.1145/3106237.3117767}.
- [2] AWS. *Amazon Coretto*. <https://aws.amazon.com/corretto/>. 2021. (accessed: 10.09.2021).
- [3] AWS. *AWS Global Infrastructure*. [https://aws.amazon.com/about-aws/global-infrastructure/regions\\_az/](https://aws.amazon.com/about-aws/global-infrastructure/regions_az/). 2021. (accessed: 12.09.2021).
- [4] AWS. *AWS Lambda C++ runtime Github repository*. <https://github.com/aws-labs/aws-lambda-cpp>. 2021. (accessed: 10.09.2021).
- [5] AWS. *AWS Lambda Pricing*. <https://aws.amazon.com/lambda/pricing/>. 2021. (accessed: 10.09.2021).
- [6] AWS. *AWS re:Invent 2018: A Serverless Journey: AWS Lambda Under the Hood (SRV409-R1)*. [https://www.youtube.com/watch?v=QdzV04T\\_kec](https://www.youtube.com/watch?v=QdzV04T_kec). 2018. (accessed: 10.09.2021).
- [7] AWS. *AWS Shared Responsibility Model*. <https://docs.aws.amazon.com/whitepapers/latest/security-overview-aws-lambda/the-shared-responsibility-model.html>. (accessed: 10.09.2021).
- [8] AWS. *Configuring Lambda Functions*. <https://docs.aws.amazon.com/lambda/latest/dg/configuration-function-common.html>. 2021. (accessed: 10.09.2021).
- [9] AWS. *Firecracker Documentation*. <https://firecracker-microvm.github.io/2018-2021>. (accessed: 10.09.2021).
- [10] AWS. *Firecracker Documentation Github*. <https://github.com/firecracker-microvm/firecracker/blob/main/docs/>. 2018.
- [11] AWS. *Firecracker Threat Containment Graphic*. [https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/docs/images/firecracker\\_threat\\_containment.png](https://raw.githubusercontent.com/firecracker-microvm/firecracker/main/docs/images/firecracker_threat_containment.png). 2021. (accessed: 10.09.2021).
- [12] AWS. *Introducing AWS Lambda*. <https://aws.amazon.com/about-aws/whats-new/2014/11/13/introducing-aws-lambda/>. (accessed: 31.08.2021).
- [13] AWS. *Lambda Successful Invokes Graphic*. <https://docs.aws.amazon.com/lambda/latest/dg/images/Overview-Successful-Invokes.png>. 2021. (accessed: 10.09.2021).

- [14] AWS. *Square Enix Case Study*. [https://aws.amazon.com/solutions/case-studies/square-enix/?did=cr\\_card&trk=cr\\_card](https://aws.amazon.com/solutions/case-studies/square-enix/?did=cr_card&trk=cr_card). (accessed: 31.08.2021).
- [15] AWS. *Using AVX2 vectorization in Lambda*. <https://docs.aws.amazon.com/lambda/latest/dg/runtimes-avx2.html>. (accessed: 25.08.2021).
- [16] AWS. *Using Lambda Extensions*. <https://docs.aws.amazon.com/lambda/latest/dg/using-extensions.html>. 2021. (accessed: 10.09.2021).
- [17] D. Barcelona-Pons, P. García-López, Á. Ruiz, A. Gómez-Gómez, G. París, and M. Sánchez-Artigas. "FaaS Orchestration of Parallel Workloads." In: *Proceedings of the 5th International Workshop on Serverless Computing*. WOSC '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 25–30. ISBN: 9781450370387. DOI: 10.1145/3366623.3368137.
- [18] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López. "On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures." In: *Proceedings of the 20th International Middleware Conference*. Middleware '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 41–54. ISBN: 9781450370097. DOI: 10.1145/3361525.3361535.
- [19] E. Brewer. *gVisor: Protecting GKE and serverless users in the real world*. <https://cloud.google.com/blog/products/containers-kubernetes/how-gvisor-protects-google-cloud-services-from-cve-2020-14386>. 2020. (accessed: 10.09.2021).
- [20] M. Chadha, A. Jindal, and M. Gerndt. "Architecture-Specific Performance Optimization of Compute-Intensive FaaS Functions." In: *CoRR abs/2107.10008* (2021). arXiv: 2107.10008.
- [21] R. Cordingly, N. Heydari, H. Yu, V. Hoang, Z. Sadeghi, and W. Lloyd. "Enhancing Observability of Serverless Computing with the Serverless Application Analytics Framework." In: *Companion of the ACM/SPEC International Conference on Performance Engineering*. ICPE '21. Virtual Event, France: Association for Computing Machinery, 2021, pp. 161–164. ISBN: 9781450383318. DOI: 10.1145/3447545.3451173.
- [22] G. C. Fox, V. Ishakian, V. Muthusamy, and A. Slominski. "Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research." In: *CoRR abs/1708.08028* (2017). arXiv: 1708.08028.
- [23] GCP. *Google Cloud Functions Execution Environment*. <https://cloud.google.com/functions/docs/concepts/exec>. 2021. (accessed: 10.09.2021).
- [24] GCP. *Google Cloud Functions Pricing*. <https://cloud.google.com/functions/pricing>. 2021. (accessed: 10.09.2021).
- [25] GCP. *Google Cloud metrics*. [https://cloud.google.com/monitoring/api/metrics\\_gcp](https://cloud.google.com/monitoring/api/metrics_gcp). (accessed: 10.09.2021).
- [26] GCP. *Google Cloud Run Pricing*. <https://cloud.google.com/run/pricing>. 2021. (accessed: 10.09.2021).

- 
- [27] GCP. *Google Kubernetes Engine Documentation*. <https://cloud.google.com/kubernetes-engine/docs/>. 2021. (accessed: 10.09.2021).
- [28] GCP. *Google Virtual Machine instances*. <https://cloud.google.com/compute/docs/instances>. 2021. (accessed: 10.09.2021).
- [29] GCP. *gVisor Architecture Layers Overview*. <https://gvisor.dev/docs/Layers.png>. 2021. (accessed: 10.09.2021).
- [30] A. Gerrand. *Go version 1 is released*. <https://go.dev/blog/go1>. (accessed: 25.08.2021).
- [31] Go. *Go scheduler source file*. <https://github.com/golang/go/blob/master/src/runtime/proc.go>. 2021. (accessed: 10.09.2021).
- [32] Google. *Home Away Case Study*. [https://firebase.google.com/downloads/Firebase\\_HomeAway\\_Case\\_Study.pdf](https://firebase.google.com/downloads/Firebase_HomeAway_Case_Study.pdf). (accessed: 31.08.2021).
- [33] Google. *What is gVisor?* <https://gvisor.dev/docs/>. (accessed: 10.09.2021).
- [34] S. Heng. "Cloud Computing: Clear Skies Ahead." In: (Mar. 2012).
- [35] V. Horký, P. Libič, A. Steinhauser, and P. Tůma. "DOs and DON'Ts of Conducting Performance Measurements in Java." In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ICPE '15. Austin, Texas, USA: Association for Computing Machinery, 2015, pp. 337–340. ISBN: 9781450332484. DOI: 10.1145/2668930.2688820.
- [36] Intel. *Hyper Threading Technology*. <https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html>. 2021. (accessed: 10.09.2021).
- [37] J. Jiang, S. Gan, Y. Liu, F. Wang, G. Alonso, A. Klimovic, A. Singla, W. Wu, and C. Zhang. "Towards Demystifying Serverless Machine Learning Training." In: *Proceedings of the 2021 International Conference on Management of Data* (June 2021). DOI: 10.1145/3448016.3459240.
- [38] W. Kennedy. *Scheduling in Go Part 2*. <https://www.ardanlabs.com/blog/2018/08/scheduling-in-go-part2.html>. 2018. (accessed: 10.09.2021).
- [39] D. Kroese, T. Brereton, T. Taimre, and Z. Botev. "Why the Monte Carlo method is so important today." In: *Wiley Interdisciplinary Reviews: Computational Statistics* 6 (Nov. 2014). DOI: 10.1002/wics.1314.
- [40] N. Lacasse. *Open-sourcing gVisor, a sandboxed container runtime*. <https://cloud.google.com/blog/products/identity-security/open-sourcing-gvisor-a-sandboxed-container-runtime>. 2018. (accessed: 10.09.2021).
- [41] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems — A Cyber-Physical Systems Approach*. Second Edition. MIT Press, 2017, p. 298. ISBN: 978-0-262-53381-2.
- [42] Microsoft. *Azure Cloud Functions Pricing*. <https://azure.microsoft.com/en-us/pricing/details/functions/>. 2021. (accessed: 10.09.2021).

- [43] Microsoft. *Azure Custom Function Handlers*. <https://docs.microsoft.com/en-us/azure/azure-functions/functions-custom-handlers>. 2021. (accessed: 10.09.2021).
- [44] M. Pawlik, K. Figiela, and M. Malawski. "Performance considerations on execution of large scale workflow applications on cloud functions." In: *CoRR abs/1909.03555* (2019). arXiv: 1909.03555.
- [45] M. Pearring and A. Rakoczy. *Go 1.17 is released*. <https://go.dev/blog/go1.17>. (accessed: 25.08.2021).
- [46] D. Poccia. *New for AWS Lambda – Use Any Programming Language and Share Common Components*. <https://aws.amazon.com/blogs/aws/new-for-aws-lambda-use-any-programming-language-and-share-common-components/>. (accessed: 25.08.2021).
- [47] J. Polites. *Life of a Serverless Event: Under the Hood of Serverless on Google Cloud Platform (Cloud Next '18)*. <https://www.youtube.com/watch?v=MBBQ6P3GauY>. 2018. (accessed: 10.09.2021).
- [48] D. B. Pons and P. G. López. "Benchmarking Parallelism in FaaS Platforms." In: *CoRR abs/2010.15032* (2020). arXiv: 2010.15032.
- [49] D. Quaresma, D. Fireman, and T. E. Pereira. "Controlling Garbage Collection and Request Admission to Improve Performance of FaaS Applications." In: *2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2020, pp. 175–182. DOI: 10.1109/SBAC-PAD49847.2020.00033.
- [50] K. Rupp. *Microprocessor Trend Data*. <https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>. (accessed: 14.02.2021).
- [51] J. Salmon, M. A. Moraes, R. Dror, and D. Shaw. "2. COUNTER-BASED RANDOM NUMBER GENERATION." In: 2011.
- [52] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman. "Serverless Linear Algebra." In: *Proceedings of the 11th ACM Symposium on Cloud Computing*. SoCC '20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 281–295. ISBN: 9781450381376. DOI: 10.1145/3419111.3421287.
- [53] Statista. *Statista: Cloud Revenue over the years*. <https://www.statista.com/statistics/233725/development-of-amazon-web-services-revenue/>. (accessed: 13.02.2021).
- [54] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. "Large-Scale Cluster Management at Google with Borg." In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: Association for Computing Machinery, 2015. ISBN: 9781450332385. DOI: 10.1145/2741948.2741964.

- [55] I. L. Yoshi Tamura. *Sandboxing your containers with gVisor (Cloud Next '18)*. <https://www.youtube.com/watch?v=kxUZ41VFuVo>. 2018. (accessed: 10.09.2021).