Performance Isolation in GraalVM Native Image Isolates

Filipe Gomes Crispim de Sousa

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Rodrigo Fraga Barcelos Paulus Bruno
Prof. Luis Manuel Antunes Veiga

Examination Committee

Chairperson: Prof. Nuno Miguel Carvalho dos Santos
Supervisor: Prof. Rodrigo Fraga Barcelos Paulus Bruno
Member of the Committee: Prof. João Nuno De Oliveira e Silva

November 2022

Abstract

In the cloud computing service model Function-as-a-Service (FaaS), small, stateless, and event-driven functions, are invoked countless times in parallel. For each of these function invocations, a new container and runtime will have to be started, resulting in non-negligible latency. One solution to this problem is by co-locating functions in the same runtime, which reduces the amount of runtime start ups and improves the memory footprint, since there are less runtimes.

Since function invocations will share the runtime, there is no control over how much CPU a function gets in relation to others. If a function has multiple threads it may grab a bigger slice of the CPU. In this work, we design and implement a mechanism for managing the CPU shares between co-located functions in a FaaS environment, in the form of an http server that receives requests to run client's functions. The main purpose is to allow cloud computing clients to set CPU requirements for their functions and making sure these are met to the best of the system's abilities.

To implement the proposed solution, we use GraalVM Native Image. GraalVM is a technology that works on top of the HotSpot JVM and offers Isolates, which already provide memory isolation between the different function instances running in parallel. We enhance the isolation to also offer CPU isolation. To evaluate the solution, we use 5 different workloads: Fibonacci, REST API, File Hashing, Image Classification, and Video Transformation. The main metric we study is CPU utilization to confirm that the solution fulfills its goals. Additionally, we evaluate how much latency and memory overhead the mechanism adds to the system in order to check that there is no substantial performance degradation.

Keywords

Function-as-a-Service; Function Co-location; GraalVM Native Image Isolates; CPU Management.

Resumo

No modelo de serviço de Computação em Cloud Function-as-a-Service, funções pequenas, sem estado, que são accionadas por eventos, são invocadas inúmeras vezes. Por cada uma destas invocações, um novo container e runtime têm de ser iniciados, resultando em latência não desprezável. Um solução para este problema é colocar múltiplas funções no mesmo runtime, o que reduz a quantidade de começos de runtimes e a memória utilizada, visto que existem menos runtimes.

Visto que invocações de funções vão partilhar o runtime, não existe controlo sobre quanto CPU uma função recebe por comparação com as outras. Se uma função tiver múltiplas threads, ela pode utilizar mais CPU. Neste trabalho, nós propomos um método para gerir as shares de CPU entre funções co-localizadas num ambiente de Function-as-a-Service, através de um servidor http que recebe invocações de funções de clientes. O principal propósito será permitir a clientes da cloud fixarem um mínimo de CPU para as suas funções e garantir que este é cumprido o melhor possível.

Para implementar a solução proposta, nós iremos utilizar GraalVM Native Image. GraalVM é uma tecnologia que existe por cima da HotSpot JVM e oferece Isolates que já garante isolação de memória entre diferentes instâncias de funcções a correr em paralelo. Nós iremos, por cima disto, garantir isolação ao nível do CPU. Para avaliar a solução, nós usamos 5 workloads diferentes: Fibonacci, REST API, File Hashing, Image Classification, e Video Transformation. A métrica principal que estudamos é a utilização de CPU para confirmar que a solução garante o requesitado. Adicionalmente, nós avaliamos quanta sobrecarga em termos de latência e memória o mecanismo acrescentea ao sistema de maneira a verificar que não existe uma degradação de performance substancial.

Palavras Chave

Função como serviço; Colocação de funções; GraalVM Native Image Isolates; Gestão de CPU.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Contents

Function-as-a-Service (FaaS) is a paradigm of cloud computing where clients create small functions that get triggered by events, and is usually coupled with Serverless, which leaves the server management and scaling to the cloud provider. This makes the client's life much easier when compared to its Infrastucture-as-a-Service (IaaS) counterpart because the client does not interact with a virtual machine to get a server running, the client simply uploads the code to the cloud provider. With Serverless, users have automatic scalability, whereas, with IaaS, users would have the extra work of setting up an autoscaling policy. It also has the advantage that the client only pays for the time that the code is really running, which is something that can't be said for IaaS, where the client pays for the time that the server is up and idle.

1.1 Motivation and Current Shortcomings

Despite the advantages, FaaS creates a new problem, which is referred to as ”cold start”. Each time a function gets triggered, a container and the function's runtime have to be started, which is a big source of latency. Some solutions to this problem include runtime recycling, restoring the runtime from a snapshot, and co-locating functions in the same runtime. With co-location, since multiple functions are sharing a runtime, the start up latency only occurs for the first instance. It also improves the memory footprint by having less runtimes running in parallel. As an extra improvement, if we co-locate different invocations of the same function it is possible to improve the memory footprint even more by sharing the common data between the functions instead of replicating it.

Co-location of functions in the same runtime results in a lack of isolation between the different functions, which now have to share resources, such as memory and CPU. It is essential for different functions to not be able to access each other's memories, and there already are solutions to this, like Photons [3] and GraalVM Isolates [2]. However, CPU isolation remains a challenge. Without proper management, the lack of CPU isolation can result in different users getting varying shares of CPU depending on the number of threads the functions have. An example of this is when a function has three threads and another has only one. In Linux, all of these 4 threads will have the same priority and will get the same share of CPU, which means that, the first function will get 75% of the CPU, while the other one will only get 25%.

In summary, if we have complete isolation by putting each function in a new container, we incur high start up latency, but if we relax the isolation and allow co-location of functions, we potentially incur unfair resource utilization.

1.2 Goals

Here are described the main goals of this paper:

- **Related Work:** Study the current state of the art on the topics discussed and directly related to this project. This includes: Function-as-a-Service(FaaS), Serverless, Function Co-location, and CPU management.
- **Design:** Mechanism in a FaaS server that manages the amount of CPU a function gets.
- **Implementation:** Implement the mechanism on top of Graal VM Native Image Isolates in order to get memory isolation between functions.
- **Evaluation:** Evaluate the mechanism against a set of relevant benchmarks. It should keep the CPU constraints as well as not add significant overhead.

1.3 Document Roadmap

In Chapter 2, we address the related work, which includes the background and the state of the art. In the background section, we introduce the technologies that we will use, like GraalVM Native Image Isolates and cgroups. In the state of the art section, we talk about other papers that touch topics related to this paper. In Chapter 3, we present our solution to create a mechanism that enables the management of CPU shares between co-located functions. We start by presenting the solution architecture, and follow it up by presenting the implementation details. In Chapter 4, we present the evaluation methodology, which includes the workloads used to test our solution, the metrics we captured, the setup where the tests were performed and what experiments were done, and finally, the results. In Chapter 5, we conclude by summing up the main points discussed in this work, the results, and future work.

2 Related Work

Contents

This section is divided into Background and State of the Art. In the Background, we introduce the technologies and concepts that are relevant to the implementation of our solution. We explain what they are and, when relevant, how to use them. These technologies include Cloud Computing and Serverless, Java Virtual Machine (JVM), GraalVM Native Image Isolates, CPU Scheduling, and Cgroups. In the State of the Art, we mention the most recent and relevant works that attempt to solve similar and adjacent topics to the ones discussed in the project. More specifically the topics are Serverless functions co-location and CPU management. We conclude the State of the Art by comparing the different papers and analyzing what they lack in regard to our goals.

2.1 Background

2.1.1 Cloud Computing and Serverless

Cloud computing [4] is a paradigm where a cloud service provider hosts client applications and takes the responsibility of managing the servers and data storage. The commonly used method of payment is ”pay-as-you-go”, which means the client only pays when using the resources. Cloud computing makes it so that the client does not have to deal with the responsibility of setting up and maintaining the infrastructure and, as a result, also allows a company to go faster into market.

The service provided by cloud computing can be divided into three main service models: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). IaaS refers to services that offer a high-level API to manage low-level details such as computing resources, servers' location, security, auto-scaling policies, etc. The clients have access to virtual machines where they will run their applications. Amazon Web Service's EC2 [5] and OpenStack's Nova [6] are examples of IaaS services. In PaaS, the provider also offers a development environment and tools, which include an operating system, programming-language execution environment, database, and web server, that software developers use to develop their applications. An example of a PaaS service is the Google App Engine [7]. SaaS is when the provider already offers a full application with which the user interacts to simply set configuration settings (e.g. through a web browser). Google Docs and Word Online are examples of SaaS services. In Figure 2.1 we can see the client and provider's responsibilities for each of the three cloud service models.

Apart from the three categories previously mentioned, there are two more recent concepts in cloud computing that go hand-in-hand: *Function-as-a-Service (FaaS)* (Figure 2.2) and *Serverless* [8]. FaaS consists in breaking an application in a set of small event-driven functions that do not keep state in between invocations. Serverless means that the client does not manage the underlying server where the application will run. The provider is responsible for the operating system, containers, runtime and scaling the servers up and down, whereas the client only writes the code. Cloud technologies like

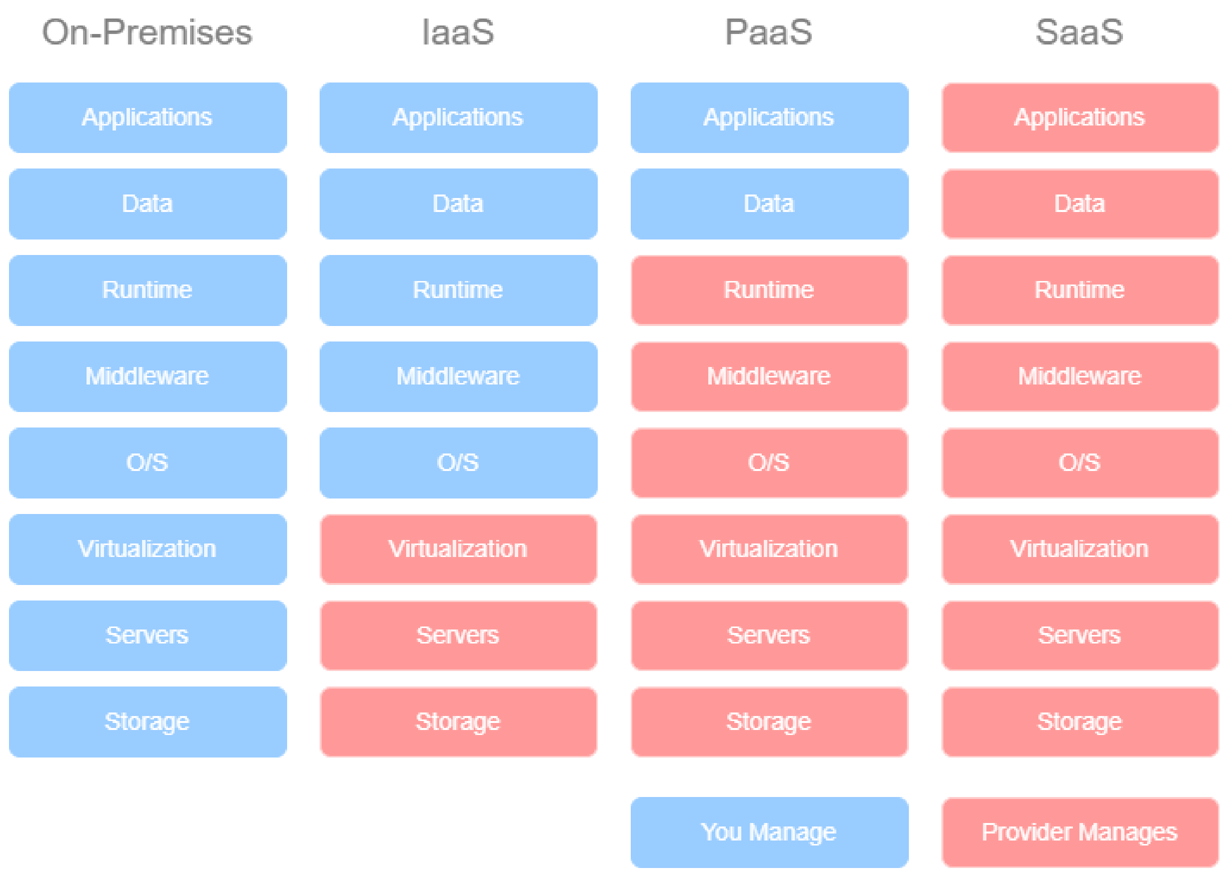


Figure 2.1: Cloud service models comparison.

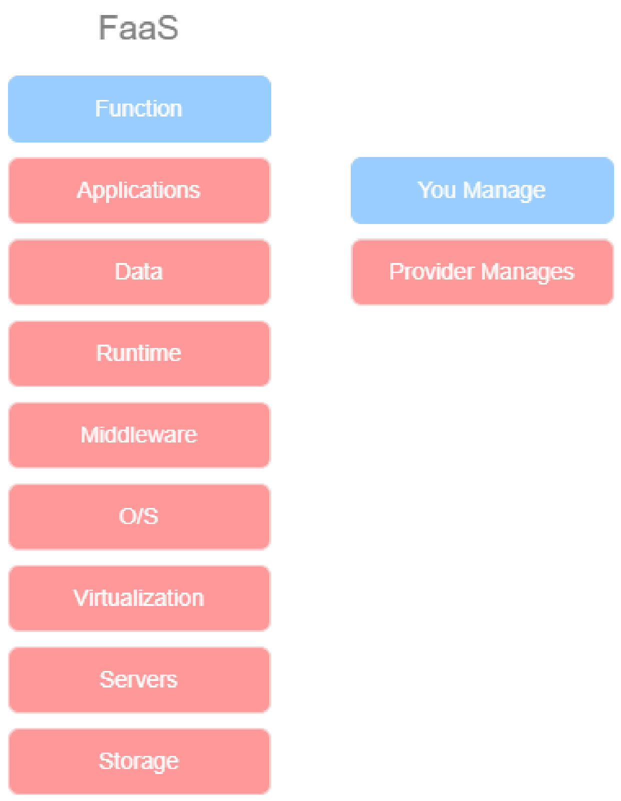


Figure 2.2: Function-as-a-Service.

AWS Lambda [9] and Azure Functions [10] offer these two cloud computing models together. With FaaS combined with Serverless, the client only pays when the application is running instead of paying for a server even when it is idle. It also makes it easier for developers as the provider automatically implements scaling policies, freeing the developer from figuring out when and how to scale.

But this is not always optimal. With Serverless, each time a new function is triggered, a container and runtime need to be started, resulting in higher latency than if the application had always been running. This is known as ”cold start” [11]. Following there are some solutions to this problem:

- **Restoring from snapshot:** Store a snapshot of a runtime after it finishes initializing and recreate new runtimes fromm that snapshot thereby eliminating the repetition of the runtime initialization.

- **Forking hot runtime:** This is a similar idea to restoring from a snapshot. If there is a function executing when another arrives, that runtime can be forked to run the new function thereby, again, eliminating the repetition of the runtime initialization.

- **Runtime recycling:** When a function finishes executing keep the runtime warm for another function. By doing this there is only one start up for multiple functions.

- **Co-locating functions:** With co-location we go a level deeper than runtime recycling, by running multiple functions in the same runtime.

Regarding language runtimes in a FaaS context, there are some observations that can be made. The large memory footprint of a language runtime poses problems since there are a lot of functions running concurrently on the same machine. Since the functions are small and end quickly, and there is a long JIT warm-up time, the runtime can not make the best use of JIT-compiled code. Also, starting a new runtime causes considerable latency, which is exacerbated if there are new runtimes constantly being started. For all these reasons, it is clear that language runtimes were not made with a FaaS context in mind, which makes the techniques mentioned previously necessary.

2.1.2 Java Virtual Machine

The most popular languages for Serverless applications include Python, Javascript, and Java. All of these run on top of a language runtime. In this project we will be using Java, and an important part of the Java runtime is the Java Virtual Machine (JVM), which is why will introduce its main concepts here. We are using the JVM, but any other runtime would be valid for our goals.

The JVM is an abstract computer, defined by a specification, that executes Java bytecode. The most widely used implementation of the JVM, distributed by Oracle, is called OpenJDK HotSpot JVM. It performs interpretation and just-in-time (JIT) compilation. The JVM does not know anything about the Java language, only Java bytecode, which means that any language that can be compiled to Java bytecode in a valid class file format, can run on the JVM.

Depicted in Figure 2.3 are the main components of the JVM [12] which include the class loader, the execution engine, and the native method interface. The class loader is responsible for loading a class file, linking it (which involves verification of the code, memory allocation, and resolution of references), and initialization. The class loader works on demand, it only loads a class when it is needed. The execution engine is composed of the interpreter, the just-in-time (JIT) compiler, and the garbage collector. The interpreter dynamically reads the instructions and translates them to machine code. The JIT compiler compiles and stores frequently used code at runtime for efficiency. The garbage collector automatically manages the heap by freeing up space that is no longer referenced from anywhere in the program. The native method interface is a framework that provides an interface to communicate with an application written in another language.

The JVM defines the following data areas: method area, heap, call stack, program counter (PC) register, and the native method stacks. The method area stores code, constants, and other class data.

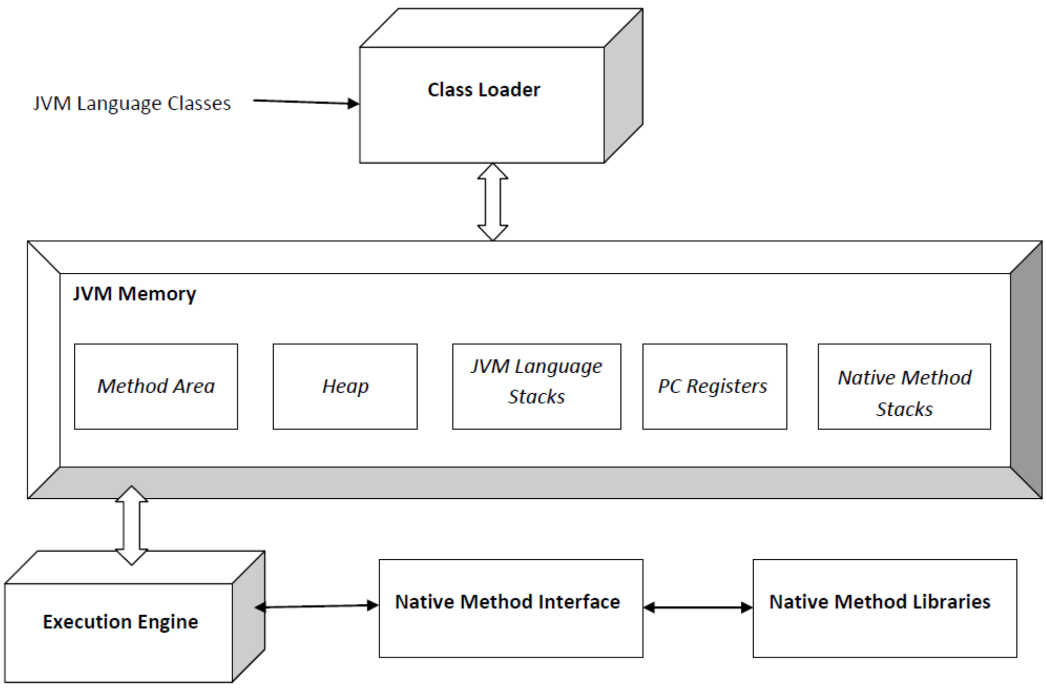


Figure 2.3: Java Virtual Machine architecture [1].

It is logically part of the heap but it may not be garbage collected depending on the implementation. The heap stores objects and arrays. The call stack stores frames for method invocations. Each new method invocation is a new frame, and when a method returns, the frame gets destroyed. While the heap and method area are global and every thread can access them, each thread has its own call stack. The PC (program counter) register contains the address of the JVM instruction currently being executed. The native method stacks are the same as the normal stacks but are for methods written in other languages.

2.1.3 GraalVM

The GraalVM [13] is a Java Virtual Machine (JVM) build, which adds to the HotSpot JVM its own JIT compiler and tools like the Truffle framework and Native Image.

2.1.3.A Graal JIT Compiler

The just-in-time compiler is integrated with the Java HotSpot VM. It provides extensibility by allowing truffle-implemented languages to run in the same Java Virtual Machine (JVM). For example, Java code (host), can be integrated with Python code (guest), and pass data back and forth in the same memory space, if there is a Python truffle implementation. The compiler also offers performance advantages through optimizations such as aggressive inlining and polymorphic inlining.

2.1.3.B GraalVM Native Image

Native Image [14] is a technology that performs ahead-of-time compilation of Java bytecode to create an executable for a specific architecture. In order to not have to compile every class, the Native Image first does a step where it finds all the classes, methods and fields that are reachable at run time. This is done through iteratively performing points-to analysis and heap snapshotting until a fixed point is reached. This tool is based on a closed-world assumption, i.e., all Java classes must be known and available at build time.

The Native Image also makes it possible to run class static initializations at image build time instead of at run time, which improves the start up latency of the application. ”Image build time” here means the ahead-of-time compilation of the bytecode, and is used to differentiate from ”build time”, which means the compilation of the source code to bytecode. It is possible to decide which classes get initialized at image build time through a flag in the native-image command. The result of the initializations is called the ”image heap”.

2.1.3.C GraalVM Native Image Isolates

The Native Image Java API offers Isolates [2], which can only be used when compiling the code with the Native Image. An Isolate is a heap, which means there can be multiple separate heaps each running a separate task. All Isolates use the same AOT compiled code and have access to the ”image heap”, which has the static initializations performed at image build time. The ”image heap” uses copy-on-write instead of replicating it for each Isolate, which improves the memory footprint. An Isolate does not have access to another Isolate’s heap by design. Isolates improve the memory footprint, since when a block of memory is no longer needed it can simply be released, whereas if there were multiple tasks running in the same heap it would accumulate until eventually the data got garbage-collected. Figure 2.4 shows the Isolates architecture.

Isolates can be used to host Serverless functions on the same runtime. Each function would run on a separate Isolate, and the Isolates already provides memory isolation. But, as mentioned before, there is no mechanism in place to control the CPU utilization. The amount of CPU each function gets can vary wildly depending on the number of threads each function creates.

To work with Isolates there exist two opaque pointer types, `Isolate` and `IsolateThread`, as well as a utility class, `Isolates`, that has all the methods to work with Isolates. An Isolate may have multiple threads attached to it so, the type `IsolateThread` represents a thread that is attached to an Isolate, while the type `Isolate` represents the Isolate as a whole. When the application starts it is already in an Isolate.

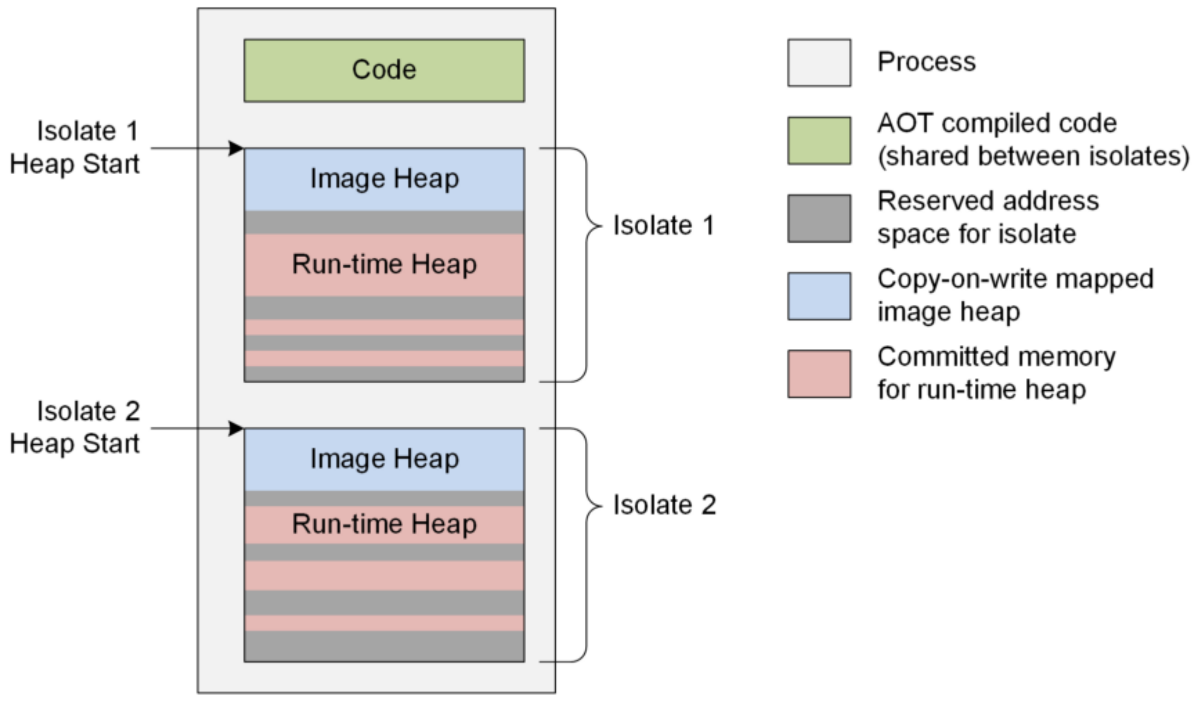

Code
Isolate 1 Heap Start
Image Heap
Run-time Heap
Isolate 1
Isolate 2 Heap Start
Image Heap
Run-time Heap
Isolate 2
Process
AOT compiled code (shared between isolates)
Reserved address space for isolate
Copy-on-write mapped image heap
Committed memory for run-time heap

Figure 2.4: Isolates architecture (from [2]).

Listing 2.1: Application that creates an Isolate and enters it.

```
1  private static int launchInNewIsolate(int argument) {
2      IsolateThread processContext = Isolates.createIsolate(
3              Isolates.CreateIsolateParameters.getDefault());
4      int result = launchFunction(processContext, defaultContext, argument);
5      Isolates.tearDownIsolate(processContext);
6      return result;
7  }
8
9  @CEntryPoint
10 private static int launchFunction(@CEntryPoint.IsolateThreadContext
11             IsolateThread processContext, int argument) {
12     return computation(argument);
13 }
```

In Listing 2.1 we have an example of a simple application that creates an Isolate to run a task and destroys the Isolate afterwards. In line 2 there is the creation of the Isolate with the method `createIsolate`. It automatically attaches the current thread to the new Isolate. Being attached does not mean it is running in the new Isolate, it simply means it can run in the new Isolate. A thread can be attached to multiple Isolates.

In line 3 the function `launchFunction` is called, which makes a transition into the new Isolate. For a function to be a transition into a new Isolate it needs to be annotated with the annotation `@CEntryPoint`, and have exactly one argument which either is annotated with `@CEntryPoint.IsolateThreadContext` and is an IsolateThread, or is annotated with `@CEntryPoint.IsolateContext` and is an Isolate. That argument represents the Isolate it is entering. In line 4, the function `tearDownIsolate` destroys the Isolate which frees the memory back to the operating system.

2.1.4 CPU Scheduling

In the context of this work, we are interested in scheduling jobs as regular operating systems. It can be defined as the task of deciding which process gets the CPU at each moment and how much.

There are many ways of performing scheduling depending on the specific goals. The most relevant examples are the following:

- **Maximize throughput:** The amount of work that is done by unit of time.

- **Minimize wait time:** The amount of time a process waits since it is ready to execute until it gets the CPU.

- **Minimize latency:** The amount of time a process takes to finish since it got the CPU.

- **Maximize fairness:** Make sure that every process gets an equal amount of time with the CPU, unless processes have different priority levels in which case their time share would reflect that.

- **Minimize resource starvation:** Make sure there is not a process that is never able to get the CPU.

In multitasking, there are two important mechanisms: scheduling and dispatching. Scheduling refers to the algorithms used to select the next process to get the CPU, while dispatching refers to how the next process in line gets the CPU.

There are two modalities in dispatching: cooperative and preemptive. With cooperative dispatching, the context switch only happens through the cooperation of the currently running process. That might be by yielding control periodically, or by being blocked waiting for some I/O. Preemptive scheduling does the opposite, the operating system interrupts the currently running process without assistance from it.

Scheduling algorithms can be logically divided in terms of using priorities or not. We will introduce some algorithms from both categories. Two very simple and well-known scheduling algorithms that do not use priorities are First-In-First-Out (FIFO) and Round-Robin. With FIFO, as the name implies, the processes are executed by order of arrival, and once a process starts executing, it executes until the end. With Round-Robin, there exists the concept of time quanta. Time quanta is a fixed quantity of time after which the scheduler evaluates which is the next process to execute. With this concept out of the way, in round-robin, each time a new process arrives, it goes to the end of the queue, and, at each time quanta, the current process running is stopped and put at the end of the queue. With both of these algorithms there is no starvation problem, that is, every process will eventually be executed.

Priorities are a mechanism through which more importance can be given to critical processes, thereby allowing them to finish quicker. This way, the new process to be executed is the one with the highest priority. To this can be added preemption, where if a new process arrives that has higher priority than the one executing currently, it gets the CPU. Priorities can be fixed, or not. One example of an algorithm that uses fixed priorities is Shortest-Job-First (SJF), where the process with the lowest execution duration has the highest priority. A variant of SJF, that does not use fixed priorities, is called Shortest-Remaining-Time-First (SRTF). As the name indicates, the process with the lowest remaining time has the highest priority and it is preemptive.

There is one problem that arrives with priorities, which is starvation. Processes with low priorities may never get the CPU. To combat this, there exists the concept of aging. With aging, the wait time for the CPU is incorporated into the priority. The higher the wait time, the higher the priority. An example of an algorithm that uses this concept of aging is Highest-Response-Ratio, where the priority is given by:

$$\textit{priority} = \frac{\textit{waiting time of a process so far} + \textit{estimated run time}}{\textit{estimated run time}}$$

2.1.5 CPU Scheduling in Linux

Linux offers a few ways for the user to influence the scheduling of processes. One possibility are the two commands: `systemd-run` and `cpulimit`. They both allow placing a limit on the CPU quota that a process is allowed to get. Another, less direct way, is by changing the priority of a process. This is done with the `nice` command, which allows giving a process a nice value between -20 and +19, which gets summed to the priority. The default nice value is 0 and lower values represent more priority.

Lastly there are cgroups, which allow organizing processes hierarchically and distributing resources through the hierarchy, such as the amount of CPU. Child cgroups share the CPU shares of the parent cgroup, and that division can be configured, possibly giving more shares to certain children. The processes in the same cgroup share the CPU shares of the cgroup equally.

Virtualization and container technology have some mechanisms to manage CPU. VirtualBox is a hypervisor (runs virtual machines) and allows setting the number of virtual CPUs of a virtual machine. Firecracker [15] is an open source virtualization technology that offers lightweight virtual machines, called microVMs, and offers the same functionality as VirtualBox. Docker [16] is a container technology that makes use of cgroups to offer the client control over the amount resources each container gets in much more detail. We intend to offer the same capabilities as Docker but for each function inside a container.

2.1.6 Cgroups

When compared to the other CPU scheduling techniques, cgroups are more direct than nice values, and more powerful than systemd-run and cpulimit, since it allows dividing resources by groups of processes instead of only one by one.

Since we will be using them for our solution, in this section we will do an overview of how to work with them, specifically where it pertains to the CPU. As mentioned before, cgroups allow organizing processes hierarchically and distributing resources through the hierarchy, such as CPU, memory, etc. These resources are called ’Controllers’.

Cgroups also allow more fine-grained control through threads, i.e., it is possible to group resources per groups of threads. To do this, the processes that have the threads in question are put in a specific cgroup, and then other cgroups are created inside it. Then, the latter cgroup types are changed to ’threaded’, and the threads are divided through them. This is important because in our work, each function will actually be a separate thread (or group of threads), not a process.

2.1.6.A Cgroups Representation

Cgroups’ hierarchy is represented as a directory with its root located at ’/sys/fs/cgroup/’. Each directory created inside the root, no matter at what level, is a cgroup. The root is also a cgroup.

2.1.6.B Cgroup Creation and Destruction

Cgroups uses a virtual file system so its operations are invoked through file system API interactions. To create a cgroup, one only needs to use the command ’mkdir’ somewhere in the hierarchy. To remove a cgroup, it can not have any live processes in it, and in that case one only needs to use the command ’rmdir’. Root permissions are necessary to alter cgroups.

2.1.6.C Cgroup Files

Inside each cgroup there are files and, possibly, directories that are other cgroups. The files can be divided in core and controller files. The core files’ names all start with ’cgroup’. and the controller files’ names start with the name of each controller, like for example, ’cpu.’.

Some of the most relevant core files are cgroup.type, cgroup.procs, and cgroup.threads. To make a cgroup 'threaded' one would write to the file cgroup.type as follows:

```
echo "threaded" > cgroup.type
```

To put the process with process id 500 in a cgroup one would do as follows:

```
echo "500" > cgroup.procs
```

And lastly, to put the thread with thread id 500 in a cgroup one would do as follows:

```
echo "500" > cgroup.threads
```

2.1.6.D CPU Controller

To manipulate the amount of CPU a cgroup gets, one uses the controller files that start with 'cpu.'. There are various files that represent different ways of controlling the CPU, like: cpu.weight, cpu.max, cpu.uclamp.max, etc. In our solution, we will utilize cpu.weight, which is a read-write single value file with a default value of 100.

To understand how weight works, it is good to think of an example. Let's imagine we have one cgroup that has one process which is our system, and inside that cgroup there are three threaded cgroups, each with a single thread that represents a different function invocation.

By default, each of those three threaded cgroups will have a weight value of 100. That value dictates how much of the CPU that cgroup gets in relation to the other cgroups. Threads inside the same cgroup share the CPU equally. The amount of CPU the threaded cgroups get is given by:

```
100 / (100 + 100 + 100) = 1 / 3
```

If we wanted to give 50% to one of the cgroups, we could write 200 in the file cpu.weight:

```
echo "200" > cpu.weight
```

Then, the amount of CPU that cgroup would get would be:

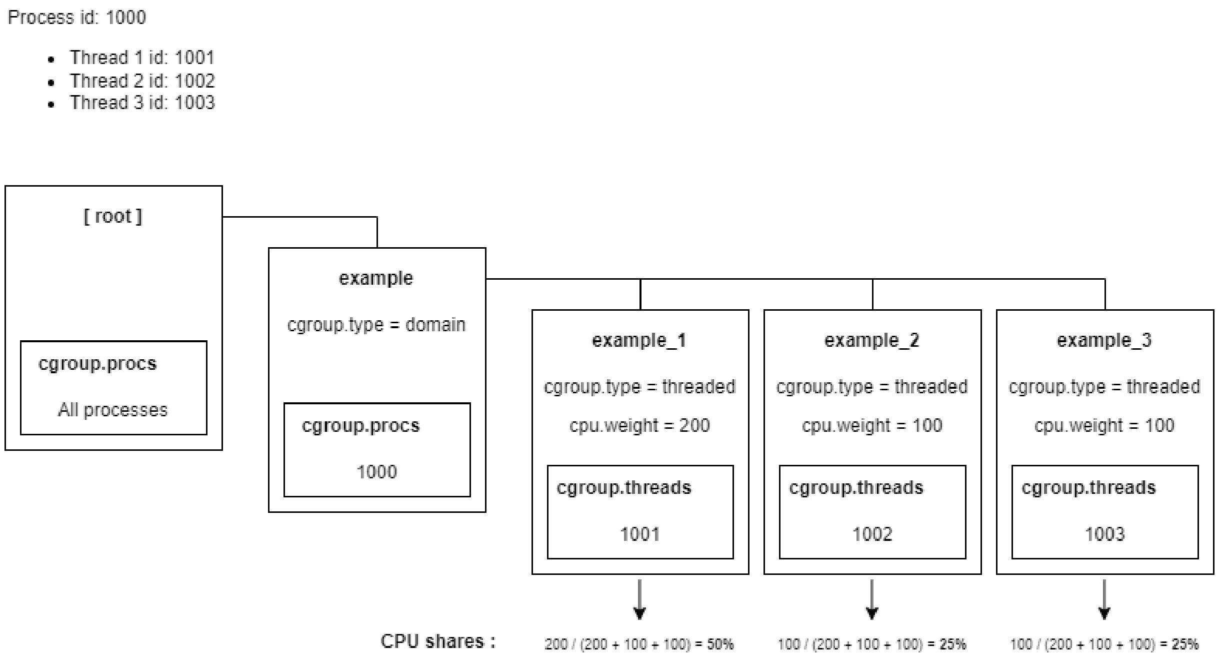


Figure 2.5: Example of cgroup hierarchy with a base cgroup and three threaded cgroups inside it.

```
200 / (200 + 100 + 100) = 1 / 2
```

This case is illustrated in Figure 2.5. The others would then both get 25%. It is important to keep in mind that these percentages are in relation to what the parent cgroup gets, which might already not be 100%.

2.2 State of the Art

In this project there are two relevant topics: co-location of functions, and CPU management of co-located functions. In this section we present some works and technologies that tackle at least one of these topics.

SAND [17] and SONIC [18] co-locate Serverless functions in the same container. SAND intends to improve the start-up delay and communication latency of applications that are divided in multiple functions that communicate between each other. In order to do this, functions that belong to the same application run in the same container as separate processes. They also implement a message bus between functions in the same container/application. SONIC, similarly to SAND, is also focused on message passing between Serverless functions. It proposes a mechanism that chooses between three message passing options dynamically to minimize data passing latency and cost. One of the methods is called VM-Storage, and consists in saving the local state of the sending function in the container's storage and scheduling the receiving function to execute on the same container. These two papers are the most distant from this project as they only perform container co-location, not runtime co-location, and there is no CPU managament between the tasks.

For running multiple functions/applications in the same JVM runtime, there exist the Multitasking Virtual Machine [19] and Photons [3]. The MVM is a modification of the JVM that allows sharing as much of the runtime between two different applications and replicating everything else. The applications are completely isolated and have their own heap. Photons are directed at Serverless functions, and consist in running concurrent executions of the same function in the same JVM. A photon represents an individual lightweight function invocation. All photons within the same execution environment, share the same object heap and the application runtime code cache, meaning that all the optimized code produced during the code warm up phase (including code interpretation, profiling, and compilation to native code) benefits all photons, resulting in faster execution. To provide data separation among multiple function executions within the same runtime they implement a function loader that intercepts and instruments the user bytecode. The function loader automatically inserts appropriate operations and modifies access to global static program elements. These two papers are closer to this project than the previous ones due to performing runtime co-location, but there is still no CPU management between co-located tasks.

An important concept we use in this work is the Isolate, which consists in a separate heap for a function, which allows running multiple functions in the same runtime. Cloudflare [20] is an example of a Serverless computing service that offers Isolates. It works with Javascript and for each function invocation it launches a new Isolate, since starting a new Isolate is orders of magnitude faster than starting a new runtime. Thin Serverless Functions with GraalVM Native Image [21] is a paper that makes use of the GraalVM Isolates [2] to run Serverless functions. It keeps a pool of Isolates and re-utilizes them for each new function invocation. It proposes caching database connections in the Isolates so as to not have to create them for every new function, as well as storing shareable data, like a machine learning model, in a specific Isolate for sharing. The latter is done because machine learning models use native code which means the models can not be put in the image heap. We keep getting closer to our project with this technology and paper, since they perform runtime co-location with separate heaps (Isolates) for each task, just as we intend to do.

The paper Automated Fine-Grained CPU Cap Control in Serverless Computing Platform [22] proposes a resource manager that dynamically adjusts the allocation of CPU capacity of different applications in a distributed Serverless computing platform with the goal of minimizing the response time skewness as experienced by the end-user. To do this it uses cgroups. We intend to do something similar but with runtime co-location and Isolates, whereas this paper does it between different containers.

In Table 2.1, we examine which goals of this project the papers/technologies mentioned in the state of the art tackle. The goals are: container co-location, runtime co-location, and CPU management.

Table 2.1: Goals of this project that each state of the art paper/technology solves.

State of the art / Goals	Container Co-location	Runtime Co-location	CPU management
SAND	Yes	No	No
SONIC	Yes	No	No
MVM	Yes	Yes	No
Photons	Yes	Yes	No
Cloudflare	Yes	Yes	No
Thin Serverless Functions	Yes	Yes	No
CPU Cap Control	No	No	Yes

Runtime co-location implies container co-location but we subdivided to show the level of co-location each work has. As can be seen, none tackle all goals.

3 Solution Architecture

Contents

The goal of this project is to create a CPU management mechanism for functions co-located in the same runtime, thereby allowing a Serverless cloud provider to set a minimum CPU boundary for the functions. When a request is sent to a cloud provider, it is first received by the Load Balancer. The Load Balancer's purpose is to select a container to run the function. Once a function reaches the container, this is where the mechanism we implemented enters into action. In a container there will be an application running which is a server waiting for function requests. We implemented this server and integrated the mechanism into it.

In the next sections, we will start by explaining how the server and mechanism work. Next, we present some implementation details. This is followed by the requirements a client's function needs to adhere to in order for the system to function correctly. And finally, despite the load balancing not being part of what we aimed to implement, we discuss some possible algorithms.

3.1 Server and CPU Management Mechanism

The basic concept is that each function that is invoked will have its own cgroup. Associating a function to a cgroup corresponds to writing the thread ids of the threads that belong to the function to a specific file in the cgroup.

In order to give a function a specific CPU quota, that quota needs to be transformed into a weight, and written to the file *cpu.weight* in the function cgroup. This concept was explained in the chapter dedicated to Related Work. Each cgroup has a weight. To get the quota of a cgroup, one divides its weight by the sum of all the weights of the cgroups at the same directory level. To ensure a minimum quota for a function, we write exactly that quota as the weight. If all the weights sum to 100, it means the application is full, there is no more CPU available, and the function will get exactly the CPU that was set for it. If the sum gives less than 100, then the function will get more CPU than it had requested.

This is an important detail, we are not going to force a function to stay at a specific quota. We are setting it as a minimum. So, if it can get more CPU and wants it, it will get it. We aim to be work conserving, i.e., we do not waste CPU if there are functions that can take advantage of it, we just enforce each one gets their minimum quota.

When the application first starts, it creates the main cgroup and writes its process id to the file *cgroup.procs* in that cgroup (Figure 3.1). It then starts the server. The server has an integer variable with the amount of available CPU which is updated with each request that arrives and leaves. It also has a list of current requests being computed where each request stores the ideal CPU and the current CPU. Figure 3.2 shows the cgroups file structure with two functions running in two different Isolates.

In Algorithm 1 it is described the procedure when a new request arrives.

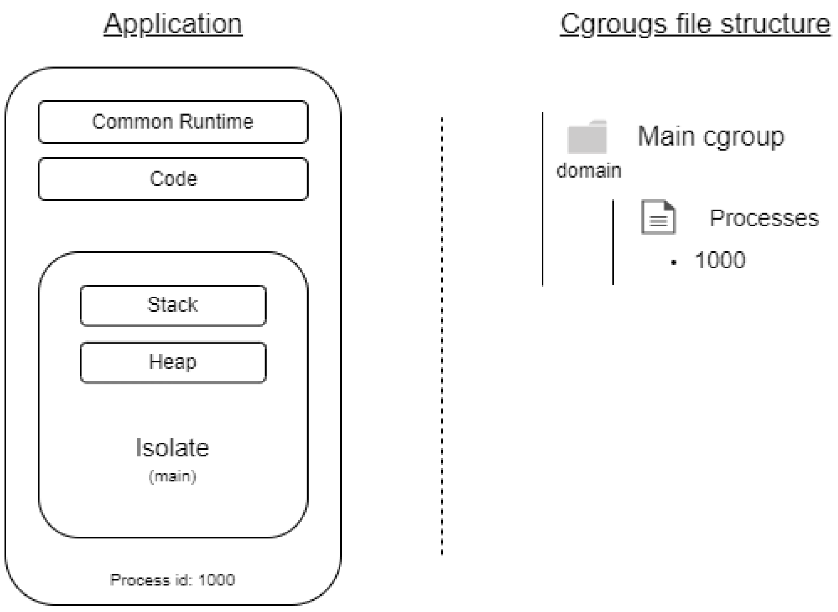


Figure 3.1: Cgroups file structure upon launch of application. The process id of the process is put inside the new cgroup.

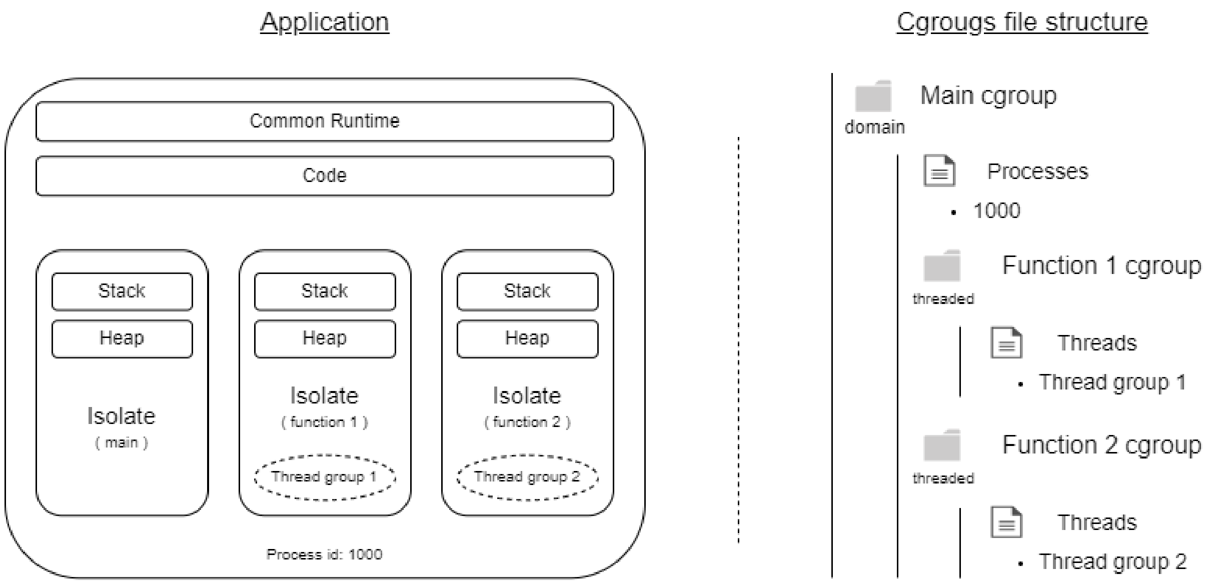


Figure 3.2: Cgroups file structure with two functions running in two different Isolates. The thread ids of the threads in a thread group are put in the cgroup associated to each function.

Algorithm 1 High level pseudo-code of the server

```
 1: Receive request
 2: if CPU is available then
 3:     Get thread id
 4:     Create isolate
 5:     Get CPU
 6:     Store request in list of current requests
 7:     Create function cgroup
 8:     Set cgroup weight
 9:     Insert thread in cgroup
10:     Enter isolate
11:     Perform computation
12:     Receive request
13:     Exit isolate
14:     Remove thread from cgroup
15:     Delete function cgroup
16:     Return CPU
17:     Iterate active requests and update CPU if possible
```

A request is in JSON format and has fields for the name of the function, an argument to the function (possibly empty), and the CPU the function wants.

Getting CPU (Step 5) implies checking if there is enough CPU to satisfy what the function requires. If there is not, we start the function with less than what it wants and when another function ends and returns its share of the CPU, we update the functions that have less than they should (Step 17).

Removing a thread from the function cgroup (Step 14) is done by writing the thread id to the main cgroup of the application, which is the parent directory of all the functions' cgroups. This automatically removes the threads from the function cgroup.

3.2 Implementation Details

This project was developed in Java SE11, on Ubuntu 20.04.5 LTS. The project is compiled by the GraalVM ahead-of-time compiler into an executable. This executable needs to be run with superuser priveleges in order to be able to manage cgroups. The version of GraalVM used was built from the source, which is available at github, https://github.com/oracle/graal. The version used corresponds to commit 05f6853ec39e69ccb0cfa420853530903f2cbf67.

While managing the cgroups, a very important piece of information are the thread ids. These thread ids are the ones at operating system level, not the ones at application level. With Java it was not possible to get the real thread id. So, in order to get the thread ids, we use system calls in C++ code, and we call that code through the Java Native Interface (JNI), which is a programming framework that enables Java code to call native applications.

After doing this, we also moved all the management of cgroups (directory creation/deletion and writing to files) to C++ code.

We do not destroy the isolates after they are used. This means we are just leaving the memory occupied. This is wrong, obviously. What should be done is the caching of the isolates. The reason we do not destroy them, is because it is a big source of latency, but since caching the isolates was not essential to take the results and there were some time constraints we decided to simply not delete them. It stays for future work.

3.3 Model of Execution for Functions

A function running in this application needs to follow these restrictions:

- It has to be stateless.
- It has to end.
- It cannot launch daemon threads that do not die.
- All resources need to be released upon function termination.

The first two restrictions are inherited from FaaS. The two last ones are important so that we can perform a clean release of resources such as the isolates and the cgroups.

3.4 Load Balancing

The load balancer is responsible for receiving a request to launch a function and deciding which runtime instance will get that request. Given that every function has a minimum CPU requirement, there are two situations to consider:

- If there is at least one runtime with enough CPU shares for the incoming function.
- If there are no runtimes with enough CPU shares for the incoming function.

The first situation is easy to deal with. We propose simply iterating through the available runtimes and choosing the first one with space for the new function. For the second situation, we propose choosing the best runtime according to a specific metric. This means the function will start with less CPU shares then required and we will dynamically adjust that as other co-located functions end.

Regarding the metric to decide the best runtime we propose two different ones.

3.4.0.A Least total CPU shares

We sum the CPU shares of every concurrently running function in the runtime and choose the runtime that minimizes that value. This naturally promotes consolidation and maximizes resource utilization. In Algorithm 2 it is represented the full load balancing algorithm with the least total CPU shares metric.

Algorithm 2 Load balancing by least total CPU shares

```
1:  Receive request
2:  CPU $\leftarrow$ CPU shares required
3:  R $\leftarrow$ Runtimes
4:  minShares $\leftarrow \infty$
5:  bestRuntime $\leftarrow$ null
6:  for r in R do
7:      S $\leftarrow$ 0
8:      F $\leftarrow$ Running functions of r
9:      for f in F do
10:         S $\leftarrow$ S + CPU shares(f)
11:     end for
12:     if $CPU \leq 100 - S$ then                 ▷ Choose the first runtime where the function fits.
13:         Send request to r
14:         Return
15:     end if
16:     if $S < minShares$ then                    ▷ Best runtime has the most available CPU shares
17:         $minShares \leftarrow S$
18:         $bestRuntime \leftarrow r$
19:     end if
20: end for
21: if $minShares \geq 100$ then
22:     Try again later
23: else
24:     Send request to bestRuntime
25: end if
26: Return =0
```

3.4.0.B Least time to ideal CPU shares

For each runtime, we order the functions in ascending order of remaining time to end the computation. Then, in that order, we find the functions that, once they finish, will leave enough CPU for the new function and sum their remaining times. We choose the runtime that minimizes that value. In Algorithm 3 is represented the full load balancing algorithm with the least time to ideal CPU shares metric. This method will require that some statistics be taken on the performance of the functions for different input sizes in order to have a good approximation of the remaining time of a function.

Algorithm 3 Load balancing by least time to ideal CPU shares

```
1: Receive request
2: CPU ← CPU shares required
3: R ← Runtimes
4: minTime ← $\infty$
5: bestRuntime ← null
6: for r in R do
7:     S ← 0
8:     F ← Running functions of r
9:     for f in F do
10:        S ← S + CPU shares(f)
11:    end for
12:    if $CPU \leq 100 - S$ then                ▷ Choose the first runtime where the function fits.
13:        Send request to r
14:        Return
15:    end if
16:    Order functions in F by remaining computation time
17:    time ← $0$
18:    sum ← $0$
19:    for f in F do
20:        time ← time + remaining time(f)
21:        sum ← sum + CPU shares(f)
22:        if $CPU \leq 100 - S + sum$ then
23:            break
24:        end if
25:    end for
26:    if $time < minTime$ then                  ▷ Calculates the least time to ideal CPU shares
27:        $minTime \leftarrow time$
28:        $bestRuntime \leftarrow r$
29:    end if
30: end for
31: if $bestRuntime = null$ then
32:    Try again later
33: else
34:    Send request to bestRuntime
35: end if
36: Return
```

4 Evaluation Methodology

Contents

In this chapter, we will start by presenting the workloads implemented, which represent the functions running in the cloud. We explain what they do and why they are relevant. Next, we present the metrics used to capture the performance of our mechanism, and how they are captured. This is followed by the setup, in which we explain on what infrastructure the tests were performed and exactly what tests were done. Lastly, we present the results of tests and comment on them.

4.1 Workloads

In our evaluation we have 4 workloads that were used in the Photons paper and represent a good and encompassing sample of the functions in use in the current Serverless technologies. These are: REST, File Hashing, Image Classification, and Video Transformation. To these workloads we also joined a Fibonacci workload.

- **REST API:** It is representative of a simple data request which is ubiquitous in the Internet. When triggered it reads a field in a database. The database utilized was mongodb.
- **File Hashing:** It is representative of data processing pipelines that divide data in chunks and process them in parallel. It downloads a file and hashes it. The file is downloaded from a local server implemented in python. The hashing is performed with the MessageDigest class in Java.
- **Image Classification:** It is representative of machine learning inference. It loads a machine-learning model and an image, and classifies the image. It downloads the data from a local Minio server and the classification is done with the TensorFlow library.
- **Video Transformation:** Recent work has proposed using Serverless functions for implementing video transformations. It downloads a portion of a video and diminishes its resolution. It downloads the data from a local Minio server and the transformation is done with the Ffmpeg library.
- **Fibonacci:** Represents a pure CPU bound function in contrast with the rest of the functions that all have an IO component. It receives an integer representing the n^{th} term of the fibonacci sequence and calculates its value.

The workload Image Classification does not follow the restrictions in the model of execution of the Solution Architecture chapter. It uses the TensorFlow library which launches daemon threads that do the work and never die. Even when new requests arrive, it is always the same threads doing the work. Even when there are concurrent Classification Image functions running concurrently, those threads are being shared.

This is obviously bad and does not work with this system. This means we cannot have multiple Image Classification workloads running concurrently. We anyway used it to compare against the other workloads but never with more than one instance of the Image Classification workload.

4.2 Metrics

The metrics we captured were the following:

- **CPU Utilization:** In order to evaluate the mechanism created, we are interested in capturing how well the CPU quotas are enforced, when running functions concurrently. For this, percentage of CPU utilization is captured through the use of the *top* command in Linux, which displays multiple real-time metrics for all the processes and threads in the system.
- **Function Latency:** Another relevant aspect is how much overhead our mechanism adds to the project. Since in a serverless model, functions are by norm small, the mechanism cannot too much overhead so as to not overshadow the function execution itself. With this in mind, we capture the duration of: full request since receiving until sending response back; execution of the function; creation of the cgroup; setting the weight of the cgroup; insertion of the thread in the cgroup; removal of the thread from the cgroup; and deletion of the cgroup.
- **Process Memory:** This is another metric to capture how much overhead our mechanism creates, which works more as a sanity check since the only extra memory is that of the list to store the information of what requests are running and on queue as well as some extra variables for bookkeeping. The memory the process is using is given by the resident set size (RSS) which is captured, again, from the *top* command. Resident set size consists in the amount of memory a process has in main memory.

4.3 Setup

The experiments were performed on a virtual machine where the host computer has a Windows 11 Home operating system and an 11th generation i7 intel processor with 2.80GHz and 4 cores. The virtual machine is Linux Ubuntu 20.04.5 LTS, the virtualization is done through the Oracle VirtualBox hypervisor, and the virtual machine has access to all the 4 cores.

In order to simplify the project and the visualization of the results we pinned the application to a single CPU core in order for the calculations to be done in relation to 100%. With n cores, the maximum percentages would be n x 100%. To pin the application to one CPU core we used cgroups again, by writing 0 to the file cpuset.cpus in the main cgroup, where 0 identifies one of the CPUs.

4.3.1 CPU Utilization

The first experiment has the goal of observing if the CPU quotas set for the functions are kept. In order to better capture this, we modified the code, so that every function invocation actually starts an infinite loop, always repeating the same original function. Every code modification is indicated in the description of the experiment. If it doesn't indicate anything, nothing was changed.

Each function is associated with a cgroup. Given this information, in order to capture the CPU utilization data, we have a different process running in parallel to the application, that every 100ms will go through the directories that represent the cgroups, and read the files *cgroup.threads* and *cpu.weight*. This will give us the ids of all the threads in the cgroup and the percentage of cpu the function running in this cgroup wants. Then, it will run the top command in thread mode, sum the percentages of all the threads in the cgroup and store it associated to the current timestamp. In order to be able to associate each cgroup to a function, we name the cgroups based on the function name and the id of the isolate (every isolate has a unique id).

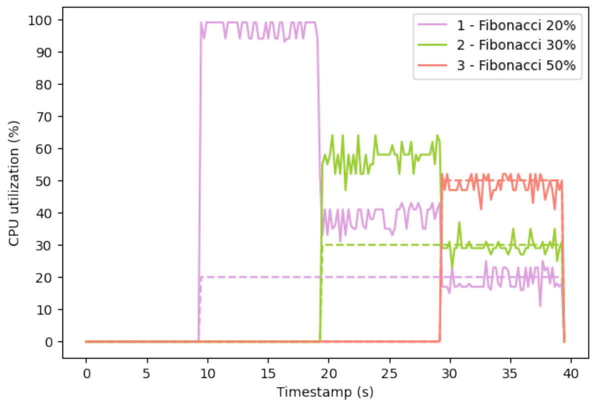
In this experiment we run 3 invocations of the same function concurrently, each starting 10 seconds after the other. This was done for every workload except for Image Classification, because all invocations of this workload share the same cgroup. For each workload two runs were made, one where the CPU quotas were 50%, 30%, and 20%, and another where they were all 33%. After this, the same was done for 2 concurrent functions, but this time, using different workloads, and select combinations. For each combination, again, two runs were made where the CPU quotas were 70% and 30%, followed by 50% and 50%.

For the second experiment, the code was also altered (in a different way). This time, it was done in order to observe the impact of threads on CPU utilization. A function invocation comes with an argument that chooses the amount of threads that will perform the operation. This way, a Fibonacci invocation with 10 threads, would have 10 threads all doing exactly the same thing and the function only ends once all threads finish.

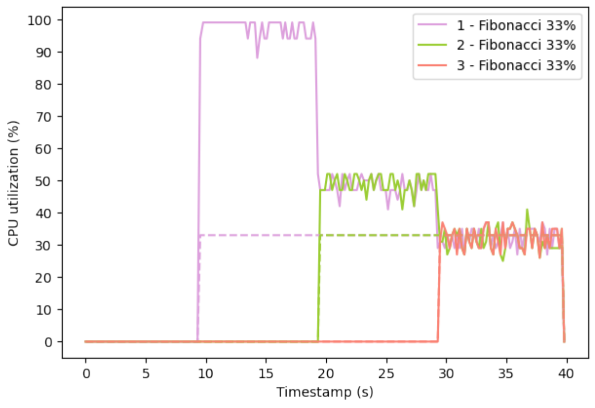
The experiment consists in running two Fibonacci functions concurrently where one uses only 1 thread, and the other uses 10 threads. This is done for two different setups: With the mechanism in place and both functions getting 50% of the CPU; without the mechanism. For each setup the experiment was performed 10 times and we performed the mean.

4.3.2 Latency Overhead

The third experiment was to run each workload 10 times and take the mean of the cgroup latency overhead. This includes the creation of the cgroup, setting of the weight, insertion of threads, removal of threads, and deletion of cgroup.



(a) 3 Fibonacci functions with different CPU percentages



(b) 3 Fibonacci functions with the same CPU percentages

Figure 4.1: CPU Utilization of 3 concurrently running Fibonacci functions which represent CPU bound functions. The dashed lines represent the CPU quota set for the function.

In the fourth experiment, we run each workload 10 times with and without the mechanism, and take the total function execution time in order to compare them.

4.3.3 Memory Overhead

The fifth and last experiment again makes the functions enter an infinite loop. We run each workload once for 10 seconds with and without the mechanism, and capture the resident set size (physical memory) of the process for those 10 seconds. Then we take the mean of all those datapoints.

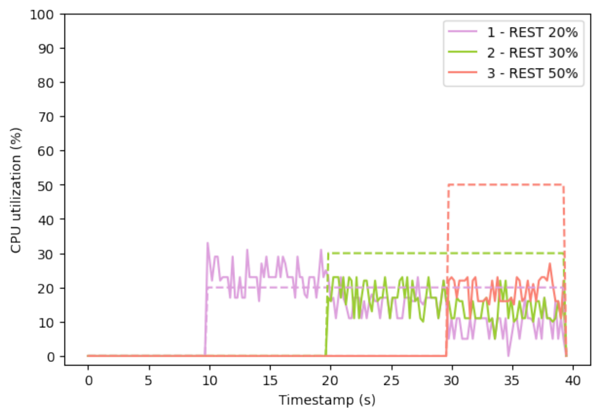
4.4 Results

4.4.1 CPU Utilization - CPU Bounded Workloads

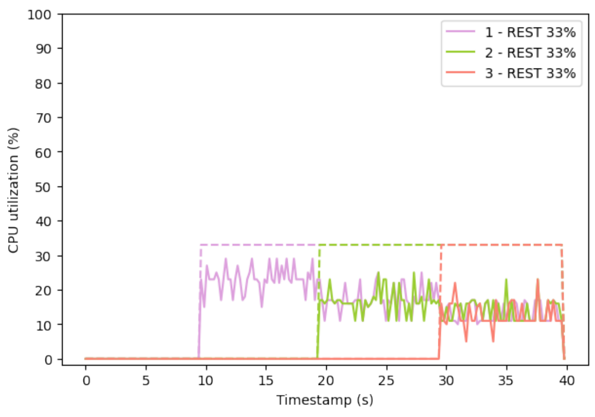
The two graphs in Figure 4.1 represent 3 Fibonacci functions, running concurrently, with different (Figure 4.1a) and same (Figure 4.1b) CPU percentages.

In the first graph, first starts a function which requires 20% CPU. While running alone it got a mean of ~97% with a standard deviation of ~2. This makes sense since, because it is running alone and is not sharing the CPU with any other function, it should get 100%. The reason it doesn't get up to exactly 100% and only to 97%, is because the application is still sharing the CPU with other processes in the machine.

At ~20 seconds a new function starts up that requires 30%. This means that they should get, respectively, 40% and 60% CPU. The first function gets a mean of ~38% with a standard deviation of ~3, and the second function gets a mean of ~57% with a standard deviation of ~4.



(a) 3 REST functions with different CPU percentages



(b) 3 Fibonacci functions with the same CPU percentages

Figure 4.2: CPU Utilization of 3 concurrently running REST functions which represent IO bound functions. The dashed lines represent the CPU quota set for the function.

At ∼30 seconds a new function starts up that requires 50%. Now all three functions add up to 100%, which means each function should get exactly what they asked for (respectively 20%, 30%, and 50%). The first function gets a mean of ∼19% with a standard deviation of ∼3%, the second function gets a mean of ∼29% with a standard deviation of ∼2, and the third function gets a mean of ∼49% with a standard deviation of ∼3.

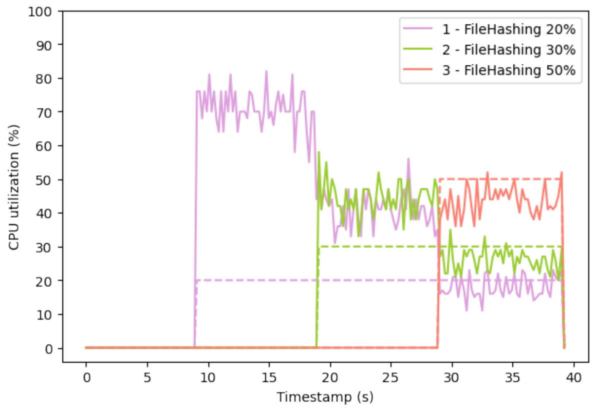
Regarding the second graph, as each function comes in, the first function gets ∼97%, ∼48%, and ∼32% (should get 100%, 50% and ∼33%), the second function gets ∼49% and ∼32% (should get 50%, and ∼33%), and the third function gets ∼32% (should get ∼33%). The standard deviations are all between 2 and 4.

For this first experiment, the CPU quotas were kept very close to the requirement, which is an indication that the mechanism is working. We will now show the results for the IO bound function, REST.

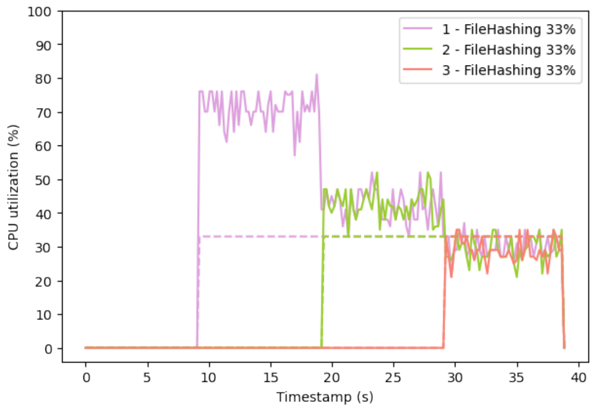
4.4.2 CPU Utilization - IO Bounded Workloads

The two graphs in Figure 4.2 represent 3 REST functions, running concurrently, with different (Figure 4.2a) and same (Figure 4.2b) CPU percentages.

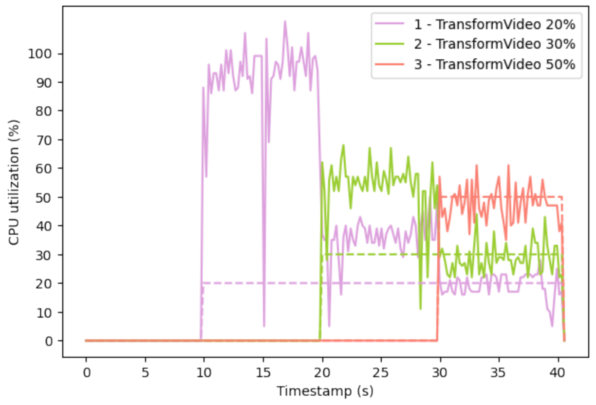
This result is the reason this experiment is divided on if a workload is CPU or IO bound. As you can observe through the dashed lines, the CPU requirements were not kept at all. In all sections of both graphs where functions are running concurrently, irrespective of if they have different CPU quotas, they get more or less the same amount of CPU. And when the first function is running alone, where it should get 100%, it gets a mean of ∼23% on both Figure 4.2a and Figure 4.2b. There does seem to be a bigger degree of separation in the first Figure where the CPU quotas are different.



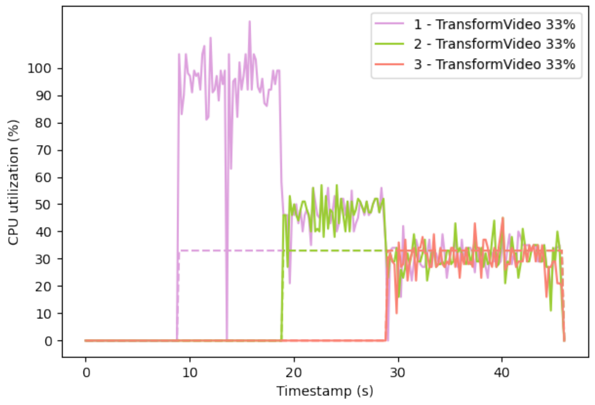
(a) 3 File Hashing functions with different CPU percentages



(b) 3 File Hashing functions with the same CPU percentages



(c) 3 Video Transformation functions with different CPU percentages



(d) 3 Video Transformation functions with the same CPU percentages

Figure 4.3: CPU Utilization of 3 concurrently running File Hashing (above) and Video Transformation (below) functions which represent functions with a mix of CPU and IO. The dashed lines represent the CPU quota set for the function.

This result shows that the CPU quotas set through cgroups aren't absolute. A function will only use the CPU if it needs it. This means that what this mechanism actually guarantees, is not that a function always has a minimum CPU quota. (because the function may have periods of IO blocking, longer or shorter, and therefore not using CPU at all). In fact, it guarantees that if a function is in a step of its computation that is CPU bounded, it will get the minimum CPU requirement.

4.4.3 CPU Utilization - CPU and IO Workloads

In Figure 4.3, the two graphs in the first row represent 3 File Hashing functions, running concurrently, with different (Figure 4.3a) and same (Figure 4.3b) CPU percentages. The two graphs in the second row represent 3 Video Transformation functions, running concurrently, with different (Figure 4.3c) and same (Figure 4.3d) CPU percentages.

These two workloads are in the middle between CPU and IO bound functions. They both have a phase where they download either a text file or a video, followed by a computation phase. This is visible in the Figures by the bigger deviation from the CPU set due to the IO phase. But apart from the bigger deviation, when compared with a pure CPU bound function, they seem to adhere to CPU requirements set. This shows that the mechanism is working.

In Figure 4.3a, the first function gets ∼71%, ∼41%, and ∼18% (should get 100%, 40% and 20%), the second function gets ∼44% and ∼26% (should get 60%, and 30%), and the third function gets ∼43% (should get 50%). The standard deviations are all between 3 and 6.

In Figure 4.3b, the first function gets ∼71%, ∼42%, and ∼30% (should get 100%, 50% and ∼33%), the second function gets ∼43% and ∼29% (should get 50%, and ∼33%), and the third function gets ∼29% (should get ∼33%). The standard deviations are all between 3 and 5.

File Hashing gets particularly bad results when compared with Video Transformation, which shows that its IO component has a bigger impact.

In Figure 4.3c, the first function gets ∼92%, ∼36%, and ∼19% (should get 100%, 40% and 20%), the second function gets ∼54% and ∼29% (should get 60%, and 30%), and the third function gets ∼48% (should get 50%). The standard deviations are all between 4 and 16.

In Figure 4.3d, the first function gets ∼93%, ∼47%, and ∼31% (should get 100%, 50% and ∼33%), the second function gets ∼47% and ∼31% (should get 50%, and ∼33%), and the third function gets ∼30% (should get ∼33%). The standard deviations are all between 5 and 17.

Video Transformation gets very good results with particularly high deviation, in part also due to the big spike that happens, curiously (and probably coincidentally), on both graphs when the first function is running alone.

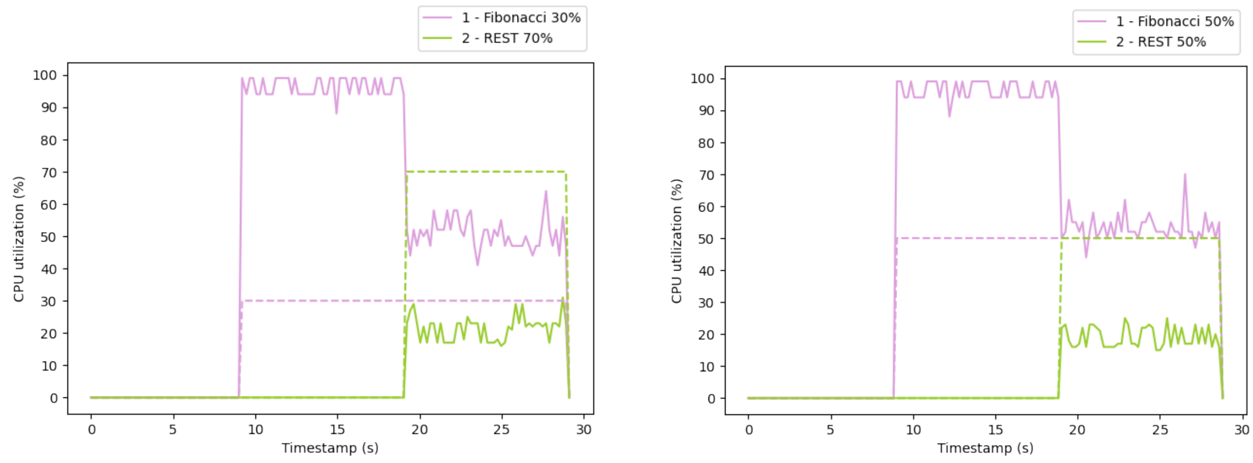
4.4.4 CPU Utilization - Different Workloads Combined

The two graphs in Figure 4.4 represent a Fibonacci and a REST function, running concurrently, with different (Figure 4.4a) and same (Figure 4.4a) CPU percentages.

In Figure 4.4a, the Fibonacci function starts first and gets ∼96%, and ∼51% (should get 100% and 30%). When the REST function comes in, it gets ∼21% (should get 70%). The standard deviations are all between 2 and 5.

In Figure 4.4a, the Fibonacci function again starts first and gets ∼96%, and ∼53% (should get 100% and 50%). When the REST function comes in, it gets ∼19% (should get 50%). The standard deviations are all between 2 and 5.

Once again it's observable that: once an IO bound function is involved, the CPU requirements aren't kept. Despite the percentages of CPU in both graphs being different, it results in a very similar result since the REST function never wants to use more than 20%.



(a) Fibonacci and REST functions running concurrently with different CPU percentages

(b) Fibonacci and REST functions running concurrently with same CPU percentages

Figure 4.4: CPU Utilization of 2 concurrently running Fibonacci and REST functions which represent CPU bound and IO bound functions respectively. The dashed lines represent the CPU quota set for the function.

Another interesting detail is that, in both figures, the functions do not add up to 100%, instead going only to 70%. Why does the Fibonacci function not grab 80%? In order to gain more information, we tried the same experiment of Figure 4.4a but inverted the CPU quotas, thereby giving 70% to the Fibonacci function and 30% to the REST function. The result is in Figure 4.5.

Changing the percentages made a wild difference. Now the Fibonacci function gets almost 100% and the REST function get below 10%. This is one more demonstration that you cannot predict the system behaviour when there are IO functions running. Despite this, what the system does guarantee is that:

When a function is steadily using the CPU in a specific time interval, its minimum CPU quota will be guaranteed.

The two graphs in Figure 4.6 represent a Fibonacci and an Image Classification function, running concurrently, with different (Figure 4.6a) and same (Figure 4.6b) CPU percentages.

In Figure 4.6a, the Fibonacci function starts first and gets ∼97%, and ∼32% (should get 100% and 30%). When the Image Classification function comes in, it gets ∼64% (should get 70%). The standard deviations are all between 3 and 11.

In Figure 4.6b, the Fibonacci function again starts first and gets ∼97%, and ∼49% (should get 100% and 50%). When the Image Classification function comes in, it gets ∼47% (should get 50%). The standard deviations are all between 3 and 6.

This represents a pure CPU bound function and a mixed one (both CPU and IO). The mechanism works decently well, but it is noticeable (more so in Figure 4.6a), that there is much more standard deviation for the function with an IO component.

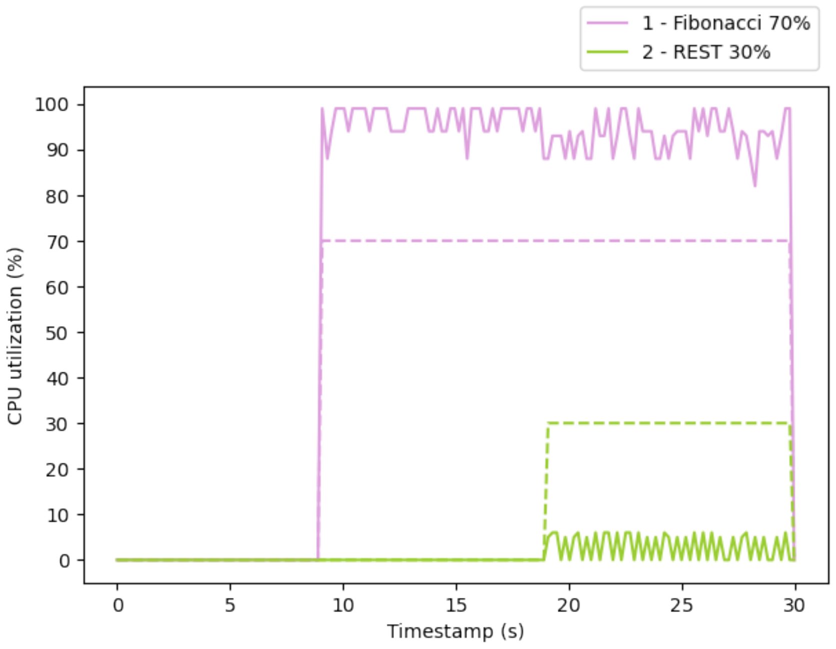
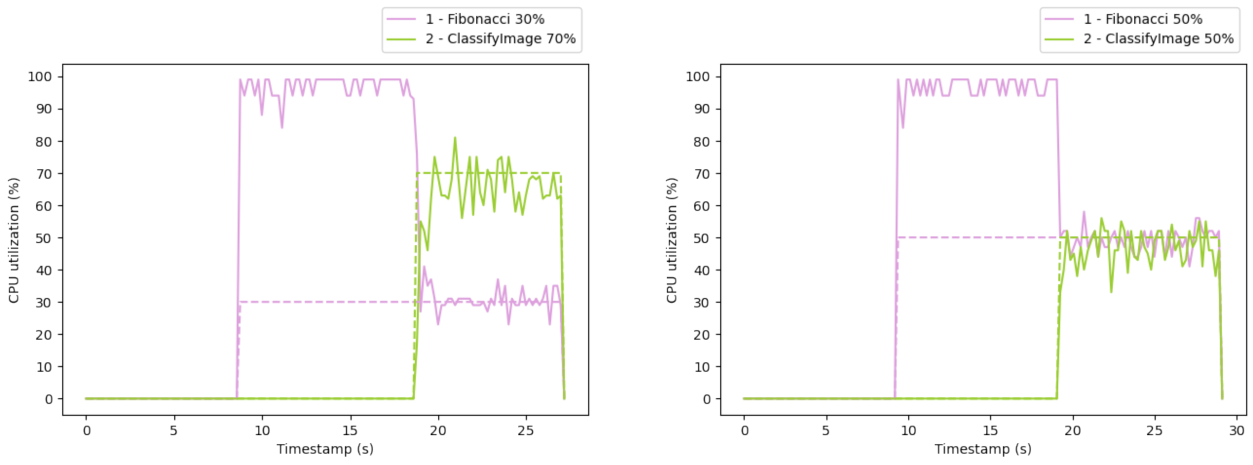


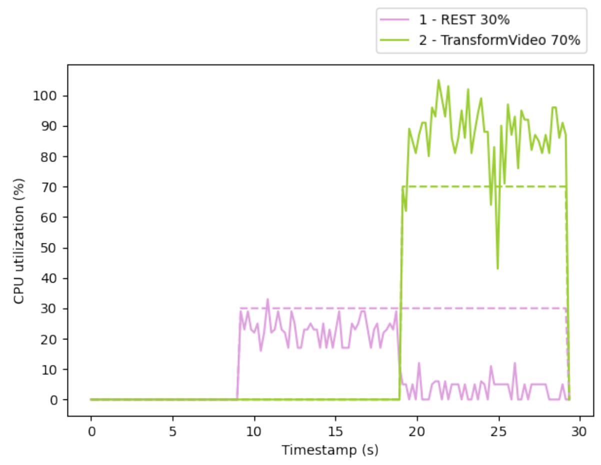
Figure 4.5: Fibonacci and REST functions running concurrently with different CPU percentages. Fibonacci has 70% and REST has 30%.



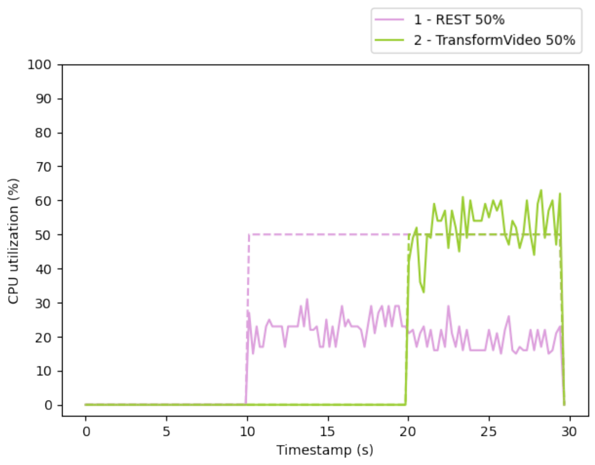
(a) Fibonacci and Image Classification functions running concurrently with different CPU percentages

(b) Fibonacci and Image Classification functions running concurrently with same CPU percentages

Figure 4.6: CPU Utilization of 2 concurrently running Fibonacci and Image Classification functions which represent a CPU bound function and a function with both IO and CPU respectively. The dashed lines represent the CPU quota set for the function.



(a) REST and Video Transformation functions running concurrently with different CPU percentages



(b) REST and Video Transformation functions running concurrently with same CPU percentages

Figure 4.7: CPU Utilization of 2 concurrently running REST and Video Transformation functions which represent a IO bound function and a function with both IO and CPU respectively. The dashed lines represent the CPU quota set for the function.

The two graphs in Figure 4.7 represent a REST and a Video Transformation function, running concurrently, with different (Figure 4.7a) and same (Figure 4.7b) CPU percentages.

In Figure 4.7a, the REST function starts first and gets ∼23%, and ∼3% (should get 100% and 30%). When the Video Transformation function comes in, it gets ∼87% (should get 70%). The standard deviations are all between 3 and 12.

In Figure 4.7b, the REST function again starts first and gets ∼23%, and ∼19% (should get 100% and 50%). When the Video Transformation function comes in, it gets ∼52% (should get 50%). The standard deviations are all between 3 and 7.

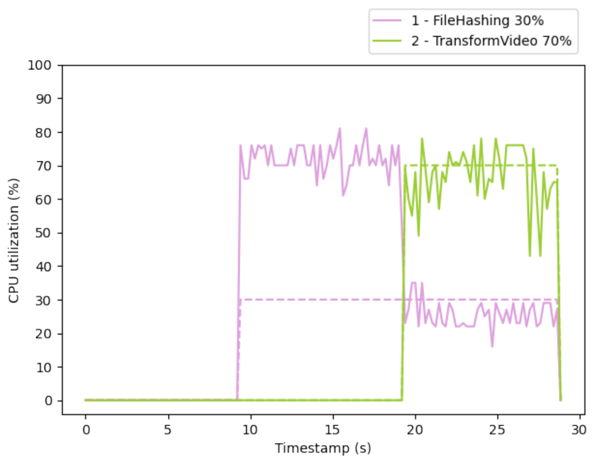
This represents a pure IO bound function and a mixed one (both CPU and IO). The results are identical to running a pure IO with a pure CPU bound function as seen Figures 4.4b and 4.5. The only difference is that there is more deviation in the line of the Video Transformation function when compared to the Fibonacci function.

The two graphs in Figure 4.8 represent a File Hashing and a Video Transformation function, running concurrently, with different (Figure 4.8a) and same (Figure 4.8b) CPU percentages.

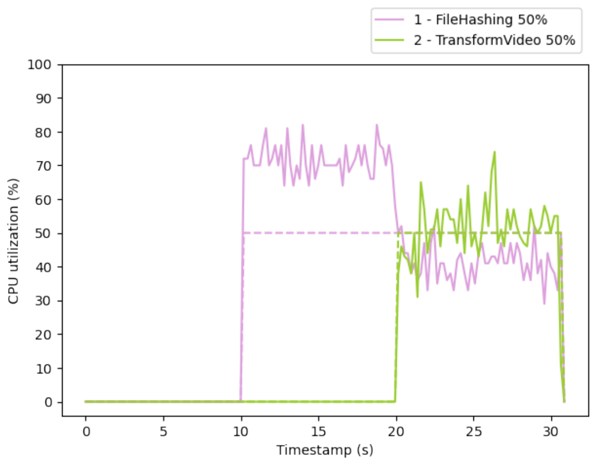
In Figure 4.8a, the File Hashing function starts first and gets ∼72%, and ∼26% (should get 100% and 30%). When the Video Transformation function comes in, it gets ∼67% (should get 70%). The standard deviations are all between 3 and 9.

In Figure 4.8b, the File Hashing function again starts first and gets ∼72%, and ∼41% (should get 100% and 50%). When the Video Transformation function comes in, it gets ∼52% (should get 50%). The standard deviations are all between 4 and 8.

This represents two mixed functions (both CPU and IO). When both functions are running concur-



(a) File Hashing and Video Transformation functions running concurrently with different CPU percentages



(b) File Hashing and Video Transformation functions running concurrently with same CPU percentages

Figure 4.8: CPU Utilization of 2 concurrently running File Hashing and Video Transformation functions which represent functions with both CPU and IO. The dashed lines represent the CPU quota set for the function.

rently the quotas are kept to a decent degree. When the File Hashing is running alone, it behaves in a similar manner to the REST function: it does not use all the CPU, which means it has a big IO component (downloading a file). This is also noticeable when both functions are running concurrently in Figure 4.8b, where the File Hashing function gets slightly less CPU.

4.4.5 CPU Utilization - Multiple Threads in a Function

By setting minimum CPU quotas, it is possible to make sure that some functions do not end up taking more CPU than they should. This happens when a function uses threads. CPU scheduling is done at thread level, so if a function has multiple threads, it will get more CPU than a function that uses less threads (assuming they are both trying to use the CPU). In the following experiment we attempt to verify if our mechanism regulates this situation.

In Figure 4.9 there are represented two experiments. The two bars to the left represent two Fibonacci functions that ran concurrently with the cgroups mechanism where each function got 50% CPU. The two bars to the right represent two Fibonacci functions that ran concurrently without the cgroups mechanism. On both experiments the blue function had only 1 thread, and the orange function had 10 threads all doing exactly the same computation.

Here the absolute values between the experiments are not very relevant because there is always a little variance. The important thing, is to compare the blue bar to the orange bar on both experiments.

If we look first to the bar to the right, what is happening is that 11 threads, all performing the same computation (Fibonacci), are fighting for the CPU, and they all get the same amount. This means that the orange function will get 10 times more CPU than the blue function and the blue function will not be

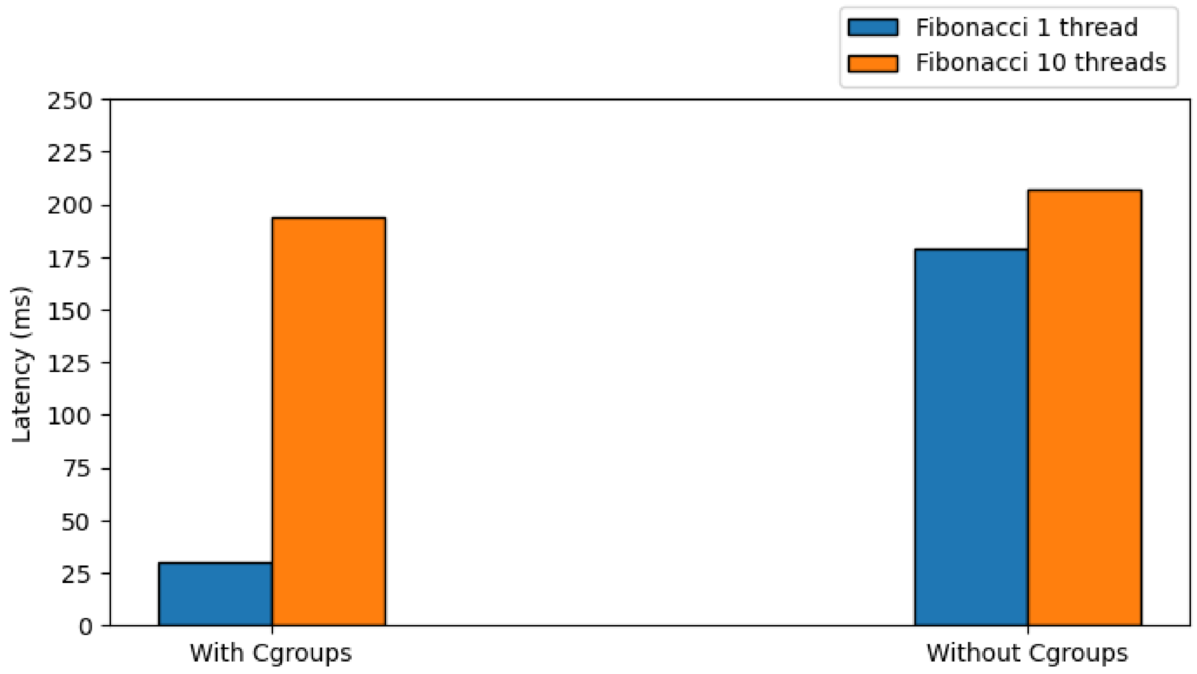


Figure 4.9: Two Fibonacci functions running concurrently, one function with one thread (blue), and the other function with ten threads (orange). Left: With cgroups. Right: Without cgroups.

able to run as fast.

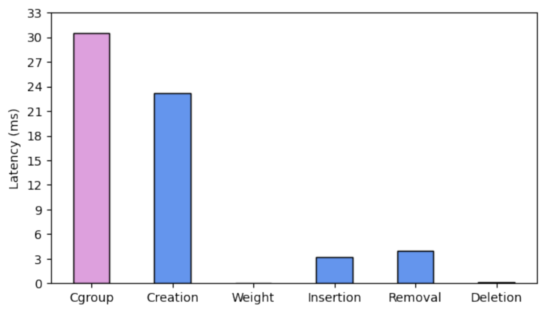
Now looking to the bar to the left. By using the cgroup mechanism we can observe that the function with only 1 thread runs much faster since we are dividing the CPU per function, and not per thread.

4.4.6 Latency Overhead

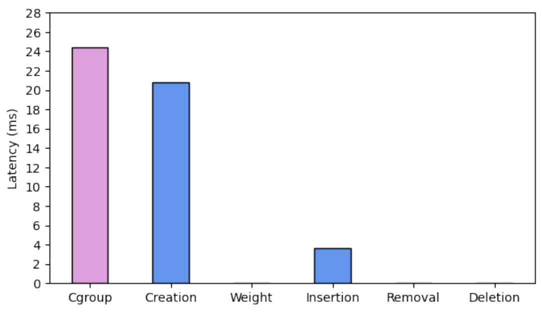
In the graphs in Figure 4.10 the pink bar to the left represents the total cgroup overhead. It is the sum of all the bars to the right.

Setting the the cgroup's weight and deleting a cgroup have negligible latency for all the workloads. The creation of the cgroup is consistent throughout the workloads at ∼20ms. Insertion of threads in the cgroup is between 3ms and 5ms on all workloads except Image Classification. Removal of threads from cgroup is 3ms for Fibonacci, negligible for File Hashing and REST, and ∼25ms for Image Classification and Video Transformation.

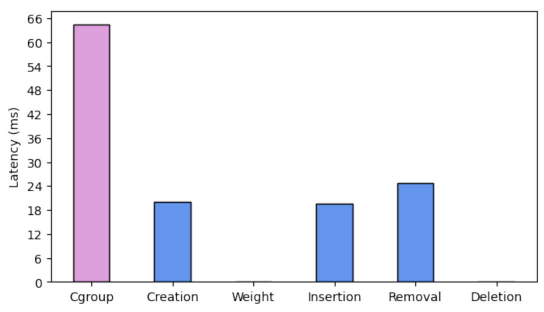
In the case of insertion, what is measured is just the insertion of the main threads that start the computation. Any threads that start during the computation are automatically inserted in the cgroup, but this is not measured. The reason Image Classification has a higher insertion latency is because it is treated specially, since it behaves badly, as mentioned before. And so, in truth, multiple threads are inserted at the beginning, resulting in more latency.



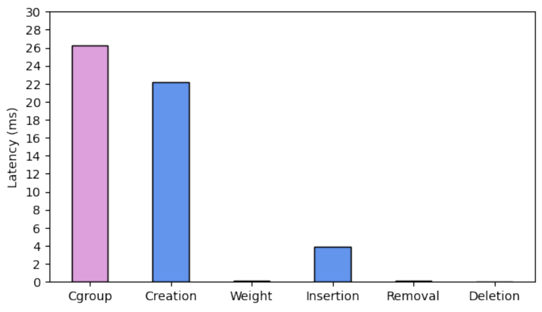
(a) Fibonacci



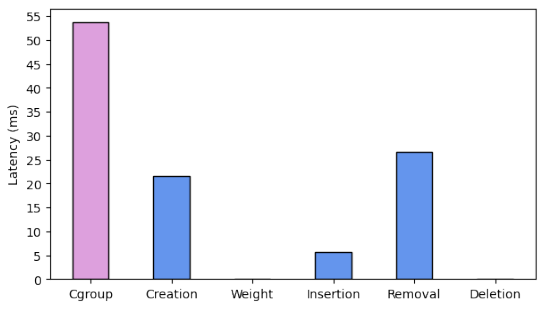
(b) File Hashing



(c) Image Classification



(d) REST



(e) Video Transformation

Figure 4.10: Decomposition of the cgroup latency overhead for every workload.

For removals, the explanation for the bigger latency in the Image Classification workload is the same as for the insertion. For the Video Transformation, more investigation needs to be done to understand, since only the removal of the first thread is being accounted for.

Fibonacci, File Hashing, and REST have a total cgroup latency below 30ms. Image Classification and Video Transformation have a total cgroup latency below 60ms. In order for this mechanism to make sense, the function should have significantly bigger latency than the cgroup overhead. We verify if that is the case in the following Figures.

In Figure 4.11a, it shows how big is the cgroup overhead (blue) when compared to the base latency of the function (orange). The smaller the blue section the better. In Figure 4.11b, it shows the ratio between the total function latency with the cgroup mechanism and without.

This shows that for faster functions like Fibonacci, File Hashing, and REST, where the cgroup overhead is a big part of the total function latency, it will worsen the total latency between 2 and 5 times. This is very bad, obviously.

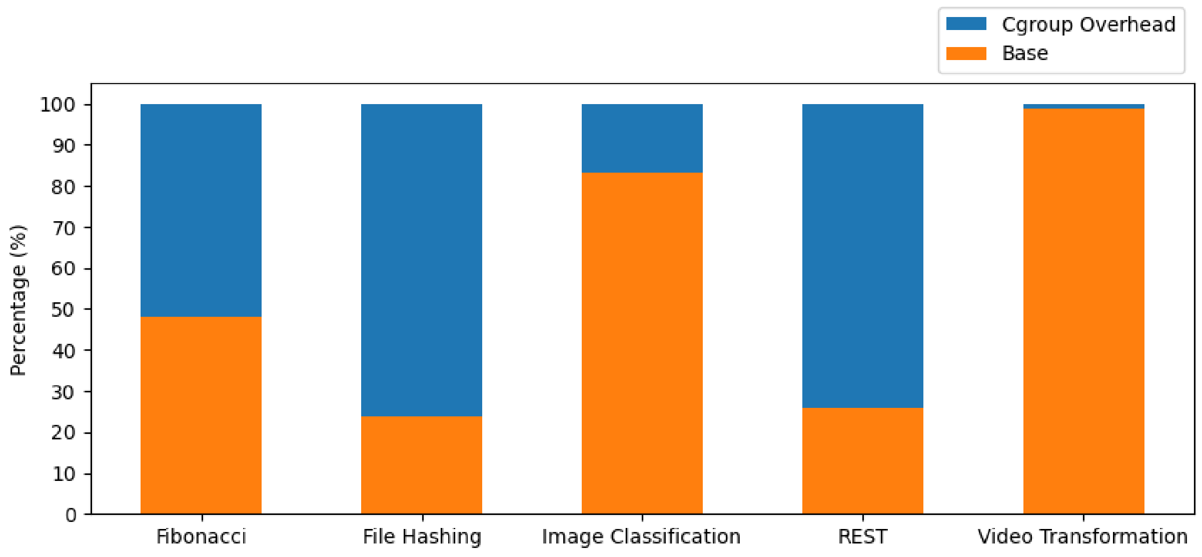
For slower, more computationally intensive functions, like Image Classification and Video Transformation, the cgroup overhead represents very little of the total function latency, and so, it makes sense to implement the mechanism.

4.4.7 Memory Overhead

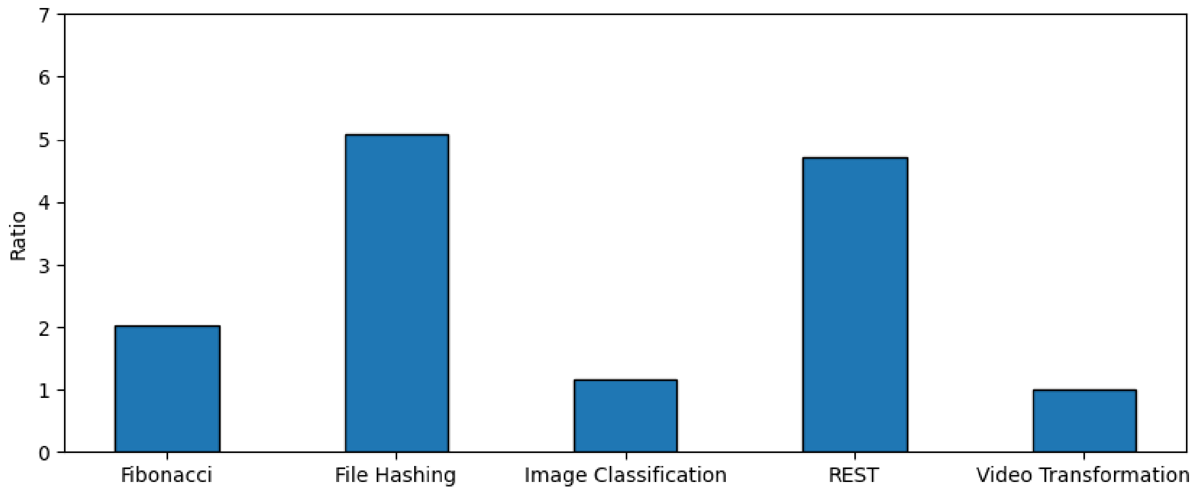
In Figure 4.12, is the ratio between the memory used by each workload with the cgroups mechanism and without. This memory is only in relation with the process of the application, so it does not include the cgroup directories.

This Figure shows that the application is not using exceedingly more memory that it could become a problem. The reason some values are a bit over 1 and other a bit under is due to the different conditions in the system when the workloads were run, which cause a little variability. The important takeaway is that it is close to 1.

Regarding the cgroup directories, their size is negligible. They just contain a series of very small files, each just containing either one or a considerably small amount of lines with a number or small string. They grow linearly with the function invocations, not with the number of threads.



(a) How much does the cgroup latency overhead weigh on the total function latency.



(b) Ratio between the total function latency with the cgroup mechanism and without.

Figure 4.11: Impact of the cgroup latency overhead on each of the five workloads.

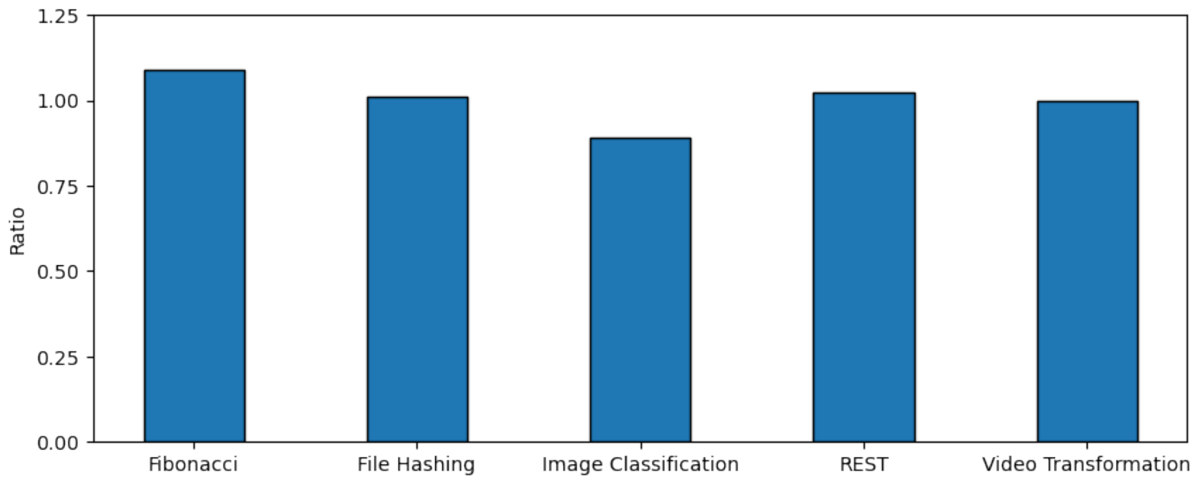


Figure 4.12: Ratio between the memory used by each workload with the cgroups mechanism and without.

5

Conclusion

In the Cloud Computing service model Function-as-a-Service, it makes a lot of sense to run functions in the same runtime in order to reduce start-up latency and memory footprint. This is known as co-location. By doing this, functions are now sharing resources and it is important to have a way of managing the resources to make sure no function is treated unfairly.

The goal of this project is to create a mechanism that manages the amount of CPU that a co-located function has access to, without adding significant overhead. When running functions in separate containers, there already exist products that offer some form of CPU management. A good example of this is Docker, that uses the Linux technology, cgroups, to manage each container's resources separately. But there is no mechanism to do this dynamically, when co-locating functions in the same runtime.

In our solution we use the GraalVM Native Image Isolates [2] technology to run each separate function in a different Isolate, which already provides memory isolation. To manage the CPU we use cgroups. The solution consists in an http server that receives requests to run functions with a specific minimum CPU quota that needs to be guaranteed, and dynamically creates cgroups, adjusts its weights and inserts threads into them.

We test this solution in how well the CPU quotas are guaranteed for the functions, and how much latency and memory overhead the mechanism adds. We use 5 workloads which are based in a previous work [3] and are representative of the functions used in current Serverless technologies.

We found that, how well the CPU quotas are guaranteed, depends on how much IO the function has. If the function is very IO bound it will not use the CPU a lot, thereby getting less than it had been given. With CPU bound functions the mechanism works very well. Regarding the overhead, the cgroup management adds between 20ms and 70ms of latency overhead and insignificant memory overhead. A function needs to be significantly longer than this latency overhead in order for it not to be relevant.

As future work, a way to improve the problem of the latency overhead is to cache the cgroups and reuse them for different functions instead of just deleting them. The creation of cgroups costs 20ms so it would be a great cut. In the same logic, the isolates should also be cached, since destroying them is too expensive in terms of latency.

Bibliography

[1] "Java virtual machine," https://en.wikipedia.org/wiki/Java_virtual_machine, accessed: 2022-01-14.

[2] "Isolates and compressed references: More flexible and efficient memory management via graalvm," https://medium.com/graalvm/isolates-and-compressed-references-more-flexible-and-efficient-memory-management-for-graalvm-a044cc50b67 accessed: 2022-01-14.

[3] V. Dukic, R. Bruno, A. Singla, and G. Alonso, "Photons: Lambdas on a diet," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 45–59.

[4] "2020 idg cloud computing survey," https://www.infoworld.com/article/3561269/the-2020-idg-cloud-computing-survey.html, accessed: 2022-01-14.

[5] "Amazon web services ec2," https://aws.amazon.com/pt/ec2/, accessed: 2022-01-14.

[6] "Open stack nova," https://docs.openstack.org/nova/latest/, accessed: 2022-01-14.

[7] "Google app engine," https://cloud.google.com/appengine, accessed: 2022-01-14.

[8] "The state of serverless," https://www.datadoghq.com/state-of-serverless/, accessed: 2022-01-14.

[9] "Amazon web services lambda," https://aws.amazon.com/pt/lambda/, accessed: 2022-01-14.

[10] "Azure functions," https://docs.microsoft.com/en-us/azure/azure-functions/, accessed: 2022-01-14.

[11] J. Manner, M. Endreß, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, 2018, pp. 181–188.

[12] "Java virtual machine specification," https://docs.oracle.com/javase/specs/jvms/se17/jvms17.pdf, accessed: 2022-01-14.

[13] "Graalvm documentation," https://www.graalvm.org/docs/introduction/, accessed: 2022-01-14.

[14] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger, "Initialize once, start fast: application initialization at build time," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[15] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)*, 2020, pp. 419–434.

[16] D. Merkel *et al.*, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[17] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt, "{SAND}: Towards high-performance serverless computing," in *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 923–935.

[18] A. Mahgoub, L. Wang, K. Shankar, Y. Zhang, H. Tian, S. Mitra, Y. Peng, H. Wang, A. Klimovic, H. Yang *et al.*, "{SONIC}: Application-aware data passing for chained serverless applications," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 285–301.

[19] G. Czajkowski and L. Daynes, "Multitasking without compromise: a virtual machine evolution," *ACM SIGPLAN Notices*, vol. 47, no. 4a, pp. 60–73, 2012.

[20] "Cloudflare: how workers works," https://developers.cloudflare.com/workers/learning/how-workers-works, accessed: 2022-01-14.

[21] S. Wang, "Thin serverless functions with graalvm native image," Master's thesis, 2021.

[22] Y. K. Kim, M. R. HoseinyFarahabady, Y. C. Lee, and A. Y. Zomaya, "Automated fine-grained cpu cap control in serverless computing platform," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 10, pp. 2289–2301, 2020.