Performance Isolation in GraalVM Native Image Isolates

Filipe Sousa
Instituto Superior Técnico

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.

1 INTRODUCTION

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 [10] and GraalVM Isolates [6]. 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

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, 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 [1] 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.

There exist two recent concepts in cloud computing that go hand-in-hand: *Function-as-a-Service (FaaS)* and *Serverless* [7]. 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 AWS Lambda [2] and Azure Functions [3] 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" [13]. 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 GraalVM. The GraalVM [5] 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.

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.

GraalVM Native Image. Native Image [15] 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

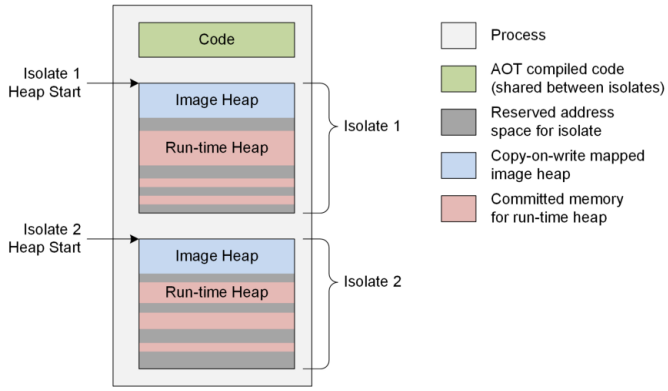

Figure 1: Isolates architecture (from [6]).

get initialized at image build time through a flag in the native-image command. The result of the initializations is called the "image heap".

GraalVM Native Image Isolates. The Native Image Java API offers Isolates [6], 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 1 shows the Isolates architecture.

2.1.3 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:

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

2.1.4 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.

2.1.5 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.

2.2 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 [8] and SONIC [12] 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 [9] and Photons [10]. 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 [4] 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 [14] is a paper that makes use of the GraalVM Isolates [6] 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 [11] 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.

3 SOLUTION ARCHITECTURE

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. Finally, we present the requirements a client's function needs to adhere to in order for the system to function correctly.

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

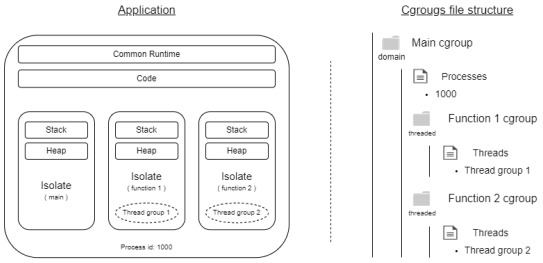

Figure 2: Cgroups file structure with two functions running in two different Isolates. The thread ids of the threads in a thread group are put in the cgroup associated to each function.

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. 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 2 shows the cgroups file structure with two functions running in two different Isolates.

In Listing 3.1 it is described the procedure when a new request arrives.

Listing 1: Application that creates an Isolate and enters it.

```
1 - Receive request
2 - if CPU is available
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.

4 EVALUATION

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 it against the other workloads but never with more than one instance of itself.

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.

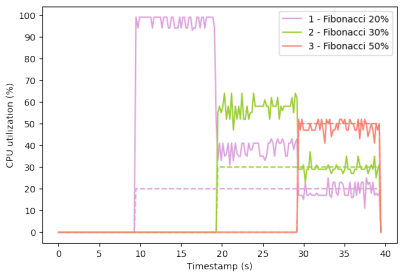

Figure 3: 3 Fibonacci functions with different CPU percentages

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.

4.4 Results

4.4.1 CPU Utilization - CPU Bounded Workloads. Figures 3 and 4 represent 3 Fibonacci functions, running concurrently, with different (Figure 3) and same (Figure 4) CPU percentages.

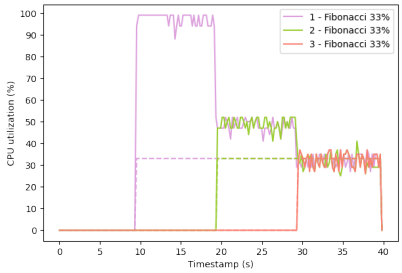

Figure 4: 3 Fibonacci functions with the same CPU percentages

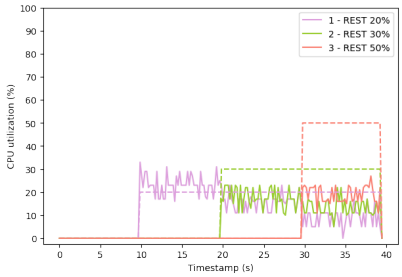

Figure 5: 3 REST functions with different CPU percentages

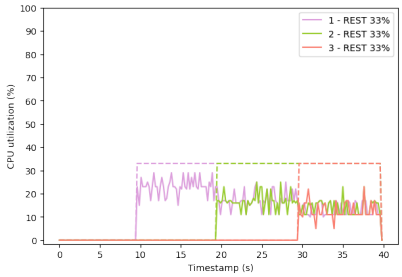

Figure 6: 3 Fibonacci functions with the same CPU percentages

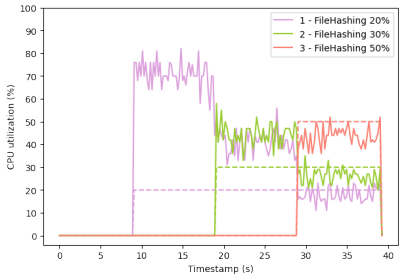

Figure 7: 3 File Hashing functions with different CPU percentages

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.

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. Figures 5 and 6 represent 3 REST functions, running concurrently, with different (Figure 5) and same (Figure 6) 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 5 and Figure 6. There does seem to be a bigger degree of separation in the first Figure where the CPU quotas are different.

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. Figures 7 and 8 represent 3 File Hashing functions, running concurrently, with different (Figure 7) and same (Figure 8) CPU percentages. Figures 9 and 10 represent 3 Video Transformation functions, running concurrently, with different (Figure 9) and same (Figure 10) CPU percentages.

These two workloads are in the middle between CPU and IO bound functions. They both have a phase where they download

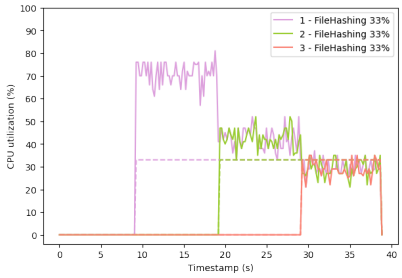

Figure 8: 3 File Hashing functions with the same CPU percentages

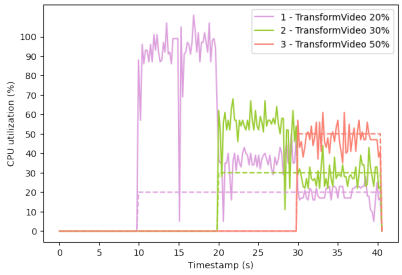

Figure 9: 3 Video Transformation functions with different CPU percentages

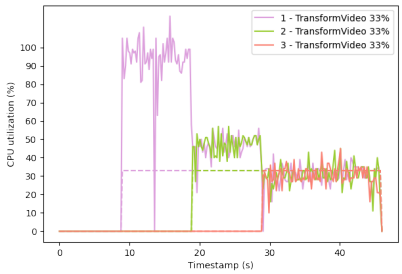

Figure 10: 3 Video Transformation functions with the same CPU percentages

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 7, 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 8, 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.

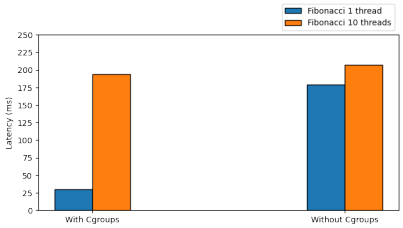

Figure 11: Two Fibonacci functions running concurrently, one function with one thread (blue), and the other function with ten threads (orange). Left: With cgroups. Right: Without cgroups.

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

In Figure 9, 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 10, 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 - 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 11 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 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

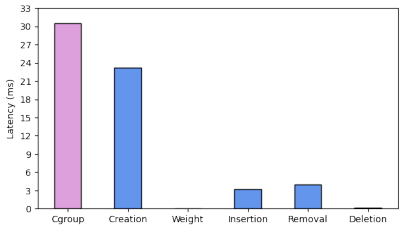

Figure 12: Fibonacci

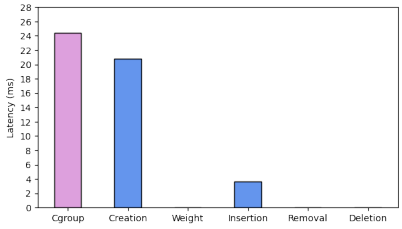

Figure 13: File Hashing

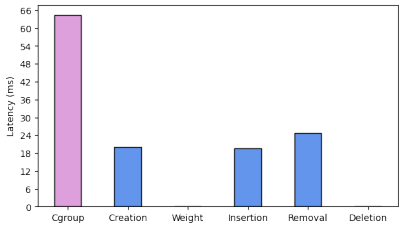

Figure 14: Image Classification

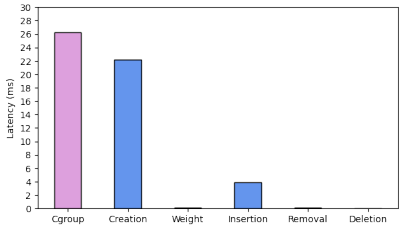

Figure 15: REST

much faster since we are dividing the CPU per function, and not per thread.

4.4.5 Latency Overhead. In the Figures 12 through 16 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

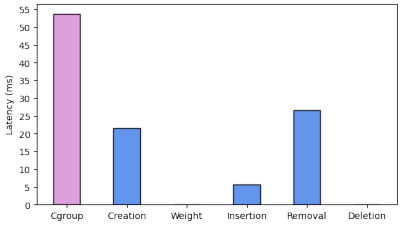

Figure 16: Video Transformation

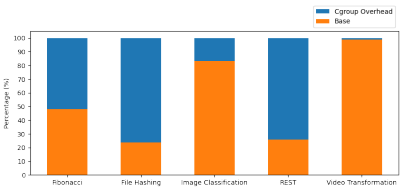

Figure 17: How much does the cgroup latency overhead weigh on the total function latency.

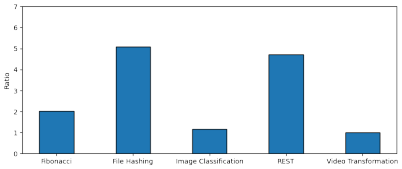

Figure 18: Ratio between the total function latency with the cgroup mechanism and without.

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.

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 17, 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 18, it shows the ratio between the total function latency with the cgroup mechanism and without.

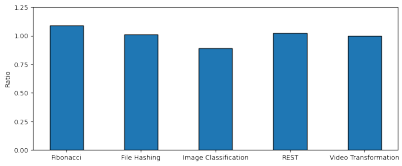

Figure 19: Ratio between the memory used by each workload with the cgroups mechanism and without.

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.

4.4.6 Memory Overhead. In Figure 19, 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.

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 [6] 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 [10] 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.

REFERENCES

[1] 2020 idg cloud computing survey. https://www.infoworld.com/article/3561269/the-2020-idg-cloud-computing-survey.html. Accessed: 2022-01-14.
[2] Amazon web services lambda. https://aws.amazon.com/pt/lambda/. Accessed: 2022-01-14.
[3] Azure functions. https://docs.microsoft.com/en-us/azure/azure-functions/. Accessed: 2022-01-14.
[4] Cloudflare: how workers works. https://developers.cloudflare.com/workers/learning/how-workers-works. Accessed: 2022-01-14.
[5] Graalvm documentation. https://www.graalvm.org/docs/introduction/. Accessed: 2022-01-14.
[6] 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-a044cc50b67e. Accessed: 2022-01-14.
[7] The state of serverless. https://www.datadoghq.com/state-of-serverless/. Accessed: 2022-01-14.
[8] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. {SAND}: Towards high-performance serverless computing. In *2018 {Usenix} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 923–935, 2018.
[9] G. Czajkowski and L. Daynes. Multitasking without compromise: a virtual machine evolution. *ACM SIGPLAN Notices*, 47(4a):60–73, 2012.
[10] V. Dukic, R. Bruno, A. Singla, and G. Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 45–59, 2020.
[11] 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*, 31(10):2289–2301, 2020.
[12] 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)*, pages 285–301, 2021.
[13] 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)*, pages 181–188. IEEE, 2018.
[14] S. Wang. Thin serverless functions with graalvm native image. Master's thesis, 2021.
[15] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger. Initialize once, start fast: application initialization at build time. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.