# CGroup Caching @ Graalvisor

DAVID NUNES, Instituto Superior Técnico, Portugal

Cloud computing is a transformative technology that delivers a wide range of on-demand services and resources over the internet. It enables businesses and individuals to access, scale, and pay for computing capabilities as needed. This flexible and cost-effective approach supports digital transformation, innovation, and efficiency in various sectors.

Serverless computing streamlines application development by abstracting infrastructure management. Developers concentrate on code, while the cloud provider handles scaling and maintenance. It offers agility and an even better cost-efficiency, making it suitable for contemporary applications.

This paper presents a study of the utilization of control groups (`cgroups`) in serverless environments, specifically in the context of Function as a Service (FaaS). The use of `cgroups` for function invocation in FaaS has been known to have performance issues during start-up, resulting in latency and initialization difficulties. To address this problem, we propose a caching mechanism for `cgroups`, which is implemented using the GraalVM platform. We evaluate the effectiveness of this approach using four representative workloads, including Hello World, Fibonacci, File Hashing, and Video Transformation. Our evaluation results will show if our proposed caching solution has the potential to significantly improve the performance and initialization of `cgroups` in FaaS environments.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Software and its engineering → Virtual machines**.

Additional Key Words and Phrases: Serverless, GraalVM, Control Groups, Function-as-a-Service, Graalvisor

## 1 INTRODUCTION

Serverless[4] technology is a relatively recent development in the field of cloud computing, which has gained significant popularity in recent years. At its core, serverless is a paradigm shift in the way applications are built and deployed, where the focus is on breaking down applications into small, modular logic units called functions. These functions are executed in response to specific triggers or events and are fully managed by the underlying cloud platform (e.g. FaaS).

One of the major advantages of serverless is its ability to provide automated scalability and elasticity, without the need for infrastructure management on the part of the developer. With serverless, applications are able to automatically scale up or down in response to changing usage patterns, and can even scale to zero when not in use, thus reducing costs. Additionally, serverless also offers a pay-as-you-use billing model, which can significantly reduce costs and improve the economics of application deployment.

This is a stark contrast to traditional service offerings, where developers have more responsibilities and less flexibility in terms of scalability and elasticity. Serverless technology eliminates much of the operational overhead and provides developers with a more efficient, cost-effective, and easy-to-use platform for building and deploying their applications. This makes it a highly attractive option for developers who are looking for a more streamlined and efficient development experience.

Author's address: David Nunes, david.j.s.nunes@tecnico.ulisboa.pt, Instituto Superior Técnico, Lisboa, Portugal.

In a serverless architecture, the ability to quickly and efficiently allocate new execution environments, such as containers or virtual machines, is critical for keeping up with high rates of function invocations. To achieve this, the virtualization stack must have low overhead and efficient resource management, allowing for smooth and speedy scaling. Recent works have attempted to host multiple function invocations in the same language runtime to minimize resource consumption[5] and avoid runtime initialization latency.

The problem arises when executing a large number of function invocations simultaneously in the same runtime, which leads to scheduling issues, particularly in terms of creating resource isolation through Linux control groups (`cgroups`). Current `cgroup` implementations in Linux are not well-suited to handle the high volume of predominantly short invocations present in serverless platforms, resulting in a decrease in performance and scalability. Another research work proposed a mechanism for allocating CPU resources among co-located functions in a Function as a Service (FaaS) environment [8], which enables cloud computing clients to specify CPU requirements for their functions. However, this approach does not address the challenges associated with the initialization of `cgroups`, which are known to result in latency and performance issues.

The scalability limitations of current operating systems in regard to the number of concurrent function invocations and expected latency in serverless infrastructure operations have been widely acknowledged. Traditional operating systems and their associated resource isolation mechanisms, such as `cgroups`, have been optimized for a limited number of concurrently executing tasks. However, the scalability requirements of modern serverless platforms exceed those of traditional operating systems, needing further research and development of optimized resource isolation methods and operating system design.

For our solution, we study the performance of `cgroup` operations with the aim of identifying scalability bottlenecks. Based on the findings of this analysis, we will then investigate potential methods of optimizing `cgroup` management. These may include the implementation of a caching layer that reduces the need for the frequent creation and destruction of `cgroups` upon task termination, among other potential optimization techniques.

This paper makes two significant contributions to serverless computing. Firstly, it identifies and analyzes the performance bottleneck associated with `cgroup` management in serverless infrastructures, shedding light on the challenges that lead to inefficiencies and performance degradation. Secondly, it proposes and implements a caching layer designed to address this scalability bottleneck, optimizing resource management, minimizing overhead, and enhancing the efficiency of serverless functions within `cgroups`.

# 2 BACKGROUND

## 2.1 Evolution of Cloud Architectures

The evolution of cloud architecture [6] [9] represents a dynamic journey that reflects the ever-growing demands of the digital age. It encompasses a transformation from monolithic structures to the emergence of microservices and the contemporary Function as a Service (FaaS). Throughout this evolution, the service models, known as Software as a Service (SaaS), Platform as a Service (PaaS), Infrastructure as a Service (IaaS), and Function as a Service (FaaS), have played instrumental roles in shaping the cloud ecosystem.

*2.1.1 The Monolith.* The early days of cloud computing were marked by the monolithic architecture. In this model, applications were constructed as a single, unified unit that was self-contained and independent from other applications. The monolithic approach simplified development but came with challenges related to scalability and adaptability. Monolithic applications often relied on Infrastructure as a Service (IaaS) for their foundational infrastructure, with some integration of Platform as a Service (PaaS) and Software as a Service (SaaS) components.

*2.1.2 Microservices.* In response to the limitations of monolithic architecture, microservices emerged as a transformative paradigm [7]. Microservices architecture involves decomposing complex applications into smaller, independent components. Each of these microservices is designed to perform specific procedures and is developed, deployed, and maintained separately. During this phase, Platform as a Service (PaaS) played a crucial role in providing specialized environments tailored to microservices architecture. Additionally, Software as a Service (SaaS) applications began to leverage microservices to gain greater agility and scalability.

*2.1.3 Serverless and FaaS.* The relentless pursuit of efficiency and developer productivity led to the rise of Serverless Computing[2], exemplified by Function as a Service (FaaS). In the FaaS model, infrastructure management is abstracted to an unprecedented level, enabling developers to focus solely on writing code in the form of stateless functions which must also be lightweight, and executed in response to specific events or triggers. This allows the parallel run of a great number of those functions and also the resource sharing among them.

Serverless and Functions as a Service (FaaS) are often conflated with one another but the truth is that FaaS is actually a subset of serverless. FaaS is focused on the event-driven computing paradigm in which application code, or containers, only run in response to events or requests. On the other hand, serverless computing focuses on providing a wide range of services, including but not limited to computing, storage, and database services. The configuration, management, and billing of servers are invisible to the end user, providing increased scalability and cost efficiency, as well as reduced operational complexity.

Today, the cloud architecture landscape continues to evolve. FaaS and Serverless Computing remain at the forefront, reshaping how applications are developed and deployed.

## 2.2 CGroups

Control groups, commonly referred to as `cgroups`, represent a key and well-established feature within the Linux kernel architecture. This feature facilitates the systematic organization of processes into hierarchical groups, enabling precise control over their consumption and monitoring of diverse resource types. The `cgroup` filesystem, intricately incorporated within the kernel, serves as the keystone of this resource management framework. It empowers administrators and system operators with a potent toolset to effectively throttle, allocate, and oversee the utilization of critical computing resources within the Linux environment.

Subsystems, also known as resource controllers (or simply, controllers), are kernel components that modify the behavior of the processes in a group and enforce resource management. A subsystem represents a single resource, such as CPU, memory, or I/O devices. In this work, we are interested in studying the ones related to CPU.

The `cgroups` for a controller are arranged in a hierarchy. This hierarchy is defined by creating, removing, and renaming subdirectories within the `cgroup` filesystem. At each level of the hierarchy, attributes (e.g., limits) can be defined. The limits, control, and accounting provided by `cgroups` generally have an effect throughout the subhierarchy underneath the `cgroup` where the attributes are defined.

*2.2.1 CGroups Structure.* The hierarchy of `cgroups` is depicted in the form of a directory structure, with its foundational directory rooted at '`/sys/fs/cgroup/`'. Every directory, regardless of its placement within this hierarchy, constitutes a `cgroup`. It's important to note that even the root directory itself serves as a `cgroup`, meaning it is part of the grouping structure.

*2.2.2 CGroup Operations.* To gain a comprehensive understanding of `cgroup` operations, let's first emphasize the significance of administrative privileges in the management of these resource control groups, i.e., for any alterations to `cgroups`, it is imperative to have root permissions, highlighting the essential need for administrative access in managing these resource control groups. `Cgroups`, in their operations, leverages a virtual file system, entailing that their functionalities are accessed through interactions with the file system's API. The process of creating a `cgroup` is quite straightforward [1] - one simply employs the '`mkdir`' command within the hierarchy structure. Conversely, when it comes to removing a `cgroup`, a crucial requirement is that no active processes or threads should exist within it. Under these conditions, the removal can be achieved by using the '`rmdir`' command.

*2.2.3 CGroup CPU Controller.* Within each `cgroup`, you'll encounter files and, in some cases, even directories that represent other `cgroups`. These files can be categorized into two main types: core files, which consistently commence with '`cgroup.`', and controller files, which begin with the name of the respective controller they pertain to.

Inside the framework of CPU control, various files provide distinct mechanisms for management. In our particular approach, we focus on the utilization of '`cpu.max`'. This file serves as the conduit for defining the desired CPU quota, which regulates the proportion of CPU time a `cgroup` can utilize relative to a specified period. Notably,

the configuration of this file involves a specific format where the CPU quota and the associated time period are specified in the form 'quota period'. The initial value indicates the total time allowance in microseconds for all processes within a child group to execute during a single period, while the second value defines the duration of that period. For example, to allocate 10% of the CPU resources, you can represent this as '100 1000', which means using 100 us out of a 1000 us window. This representation offers multiple options for achieving fine-grained control over CPU utilization within a `cgroup`.

To enable granular control over CPU resource allocation within `cgroups`, an initial step involves the configuration of the top-level or "main" `cgroup` within the hierarchy. This configuration process entails specifying the `cgroup`'s controllers that the `cgroup` will use and associating the desired process with it. To effect these changes, the 'cpu' and 'cpuset' designations are inscribed within all the '`cgroup.subtree_control`' files up the hierarchy, while the process ID is recorded within the '`cgroup.procs`' file.

Subsequently, within the main `cgroup`, the creation of the worker `cgroups` for executing threads with specific CPU allocations can be undertaken. This involves the execution of several steps. First, a new worker `cgroup` is generated using the '`mkdir`' command within the main `cgroup` directory. Next, the 'threaded' designation is inscribed within the 'cgroup.type' file, and the desired thread is assigned to this newly created worker `cgroup`, signified by the inclusion of its thread ID within the '`cgroup.threads`' file. Lastly, the CPU allocation for the worker `cgroup` is defined by configuring the '`cpu.max`' file in accordance with the desired CPU parameters, as elaborated upon earlier.

## 3 RELATED WORK

We identified previous research that has addressed topics similar to those tackled in this project. We analyze these previous studies to illuminate their relevance in the context of our research.

### 3.1 Photons

Photons[5] identifies the inefficiencies in today's serverless platforms when invoking the same function concurrently, like the big number of cold starts due to the single concurrent invocation per container policy and the large memory usage due to containers requiring some application state.

The authors observed that the extensive number of concurrent invocations of the same function code replicates large amounts of state, including the language runtime, libraries, and shared state such as machine learning models. Therefore, they presented Photons, a framework that exploits this redundancy and allows the execution of concurrent serverless functions to be co-located in a single docker container, with the opportunity to share the application state.

### 3.2 Performance Isolation in GraalVM Native Image Isolates

In the domain of Cloud Computing and the Function-as-a-Service (FaaS) model, the common practice of co-locating functions in the same runtime to minimize startup delays and reduce memory consumption is well-recognized. Effective resource management is essential to ensure equitable treatment among co-located functions. The core objective of this project was to devise a mechanism for dynamic CPU resource management when functions share the same runtime, addressing a deficiency in existing solutions like Docker, which primarily relies on `cgroups` for CPU control.

This work[8] is the most relevant to our study, since we want to implement their suggested future work, focusing on optimizing latency overhead by implementing strategies for caching `cgroups`, thereby mitigating latency associated with their creation.

### 3.3 Graalvisor

The research presents a solution to address the challenge of virtualization stack bloat in Serverless computing. Graalvisor[3] consists of a virtualized polyglot language runtime designed for the efficient execution of lightweight and short-lived Serverless functions. Graalvisor optimizes performance by running each function in a compact execution environment, with a fast launch time of under 500 microseconds. This approach significantly reduces the redundancy in virtualization stacks, leading to lower memory consumption and fewer cold starts.

Graalvisor offers a user-friendly endpoint for registering and invoking functions. When functions are invoked, Graalvisor intelligently schedules them for execution within specific cluster nodes and lambda executors (virtual machines).

### 3.4 Analysis and Discussion

As our exploration has revealed, the extensive body of research within this field has addressed a range of issues and challenges that bear similarity to those addressed in our work. While these prior studies offer valuable insights, they present opportunities for synergistic integration with our proposed solution. However, it is essential to note that none of these existing studies effectively tackle the primary challenge we confront head-on: the mitigation of latency and initialization complexities inherent in the management operations of `cgroups` within serverless environments.

In this context, we are poised to leverage the findings and methodologies put forth in the research discussed in Section 3.2. This referenced work provides a promising baseline for delving deeper into our primary challenge. We integrated the insights and techniques from 'Performance Isolation in GraalVM Native Image Isolates' into Graalvisor, mentioned in Section 3.3, and explored innovative approaches that address the intricacies of mitigating latency and minimizing initialization complexities when managing `cgroups` within the dynamic realm of serverless environments.

## 4 SOLUTION ARCHITECTURE

This section details the design of the architecture for our solution. The organization of our solution will be discussed in detail in Section 4.1, where we will present an overview of how the different components are structured and interact with each other.

### 4.1 Overview

Our objective in this project is to improve the performance of cgroup management operations and to apply them in the context of large-scale, serverless computing environments. Therefore, we propose a pre-populated cache of cgroups with different sizes in memory. Initially, the entire cache would be populated during the system's startup process (it should be noted that the cache is nevertheless dynamic and can be expanded as necessary after initialization).

In our system, whenever a function invocation is initiated, we employ a mechanism to carefully select an empty cgroup from the cache that is capable of fulfilling the resource requirements of the incoming function invocation. This cgroup allocation algorithm is crucial as it ensures that the system is utilizing its resources efficiently and that the chosen empty cgroup is appropriate for the task at hand. Once the function invocation is completed, the cgroup is returned to the cache in an empty state, ready to be utilized again for future function invocations. This process of selecting and returning cgroups from and to the cache is ongoing and enables the system to operate at optimal performance levels while maintaining a high degree of resource utilization efficiency.

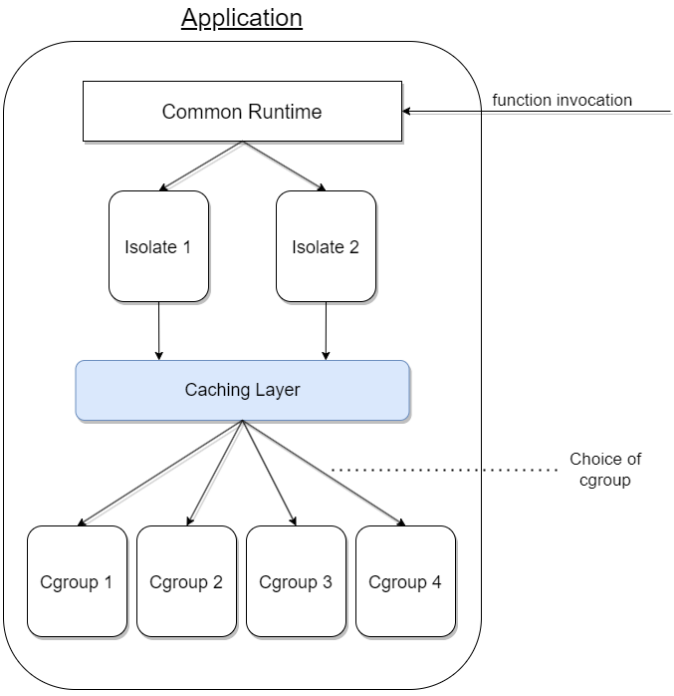



Fig. 1. Architectural diagram with cache layer.

### 4.2 Cgroup cache integration

The caching mechanism will operate within the main runtime, while individual functions will execute in isolates inside a specific control group (cgroup). Upon the arrival of a function invocation, our system performs a check to determine if there is an available cgroup in the cache with the required CPU quota. If such cgroup is present, the task is allocated to it by writing the task's thread ID into the cgroup's 'cgroup.threads' file. Once allocated, it becomes unavailable for future invocations.

The cache works lazily, populating and emptying the Java caching structure to indicate cgroup availability and unavailability. Threads are removed from the cgroup file system only when a new, different thread requires the cgroup. This way, a thread will be automatically removed from a cgroup by the operating system when it terminates or when another thread needs to be inserted into an available cgroup, that earlier had a thread running.

In the event that a cgroup of the required CPU quota is not present in the cache, a new cgroup is created, as mentioned in Section 2.2, to accommodate the function's needs. Regardless of the outcome of this check, upon the termination of the task, the cgroup is marked as available in the cache data structure, thus making it eligible for future function invocations. Figure 1 illustrates the architectural configuration of the solution with the inclusion of the newly added caching layer.

**Algorithm 1:** Cgroup Caching pseudo-code.

```
new function invocation with Q quota:
begin
    if cgroupCache.ContainsQuota(Q) then
        cgroupID ⟵ RemoveCgroupFromCache(Q)
    else
        cgroupID ⟵ createNewCgroup(Q)
    if isCacheLazy then
        if !threadInCgroup then
            RemovePreviousThreadFromCgroup
            InsertThreadInCgroup(cgroupID)
    else
        InsertThreadInCgroup(cgroupID)

function ends execution:
begin
    AddCgroupToCache()
    if !isCacheLazy then
        RemovePreviousThreadFromCgroup
```

### 4.3 Cgroup data structure

Our proposed caching solution utilizes a key-value store data structure, which allows for efficient storage and retrieval of data through the use of unique keys. Given the programming language in which the solution is implemented, Java, the specific key-value store data structure utilized is a concurrent hashmap, which offers a high level of efficiency and performance for lookups, making it well-suited for our caching requirements. The hashmap is designed such that the keys represent the quota of the cgroups, relative to a period of 100000 microseconds, and the values are a concurrent list, CopyOnWriteList in the Java language case, where each element is a cgroup Id. The use of a concurrent hashmap allows for fast and

safe access to the stored cgroups, even in a multi-threaded environment. The CopyOnWriteList guarantees that all the elements within it can be accessed simultaneously without any contention among threads. This allows for highly concurrent and scalable operations, making our solution suitable for large-scale environments. The combination of a concurrent hashmap with a concurrent list enables us to achieve high performance and efficiency, making it an ideal data structure for our solution. Figure 2 represents an example of a fully populated cache.

*4.3.1 Complexity.* In the context of the ConcurrentHashMap storing the association between CPU quotas and corresponding lists of cgroup IDs, the insertion and removal operations exhibit constant time complexity, denoted as O(1). Similarly, the lookup operation also demonstrates a constant time complexity of O(1) on average. These complexities affirm the efficiency and swiftness of all operations conducted on this data structure.

On the other hand, the CopyOnWriteList exhibits specific time complexities for various operations. Read operations are highly efficient, with a constant time complexity of O(1). This makes it well-suited for scenarios where reads significantly outnumber writes since the copy-on-write strategy ensures thread safety during updates. However, the insert and remove operations, which involve copying the list, result in potentially higher time complexity. The time complexity for these insert and remove operations can become O(n), where 'n' represents the number of elements in the list being copied. In practice, this means that while the CopyOnWriteList provides an excellent level of safety for concurrent operations, it is most efficient when the reads significantly dominate over the writes.

Nevertheless, it's worth highlighting that the CopyOnWriteList is the sole thread-safe List implementation in the Java language. By practicing caution and ensuring that the size of a specific list (representing the number of cgroups for a given quota) doesn't grow excessively, we can maintain good performance.

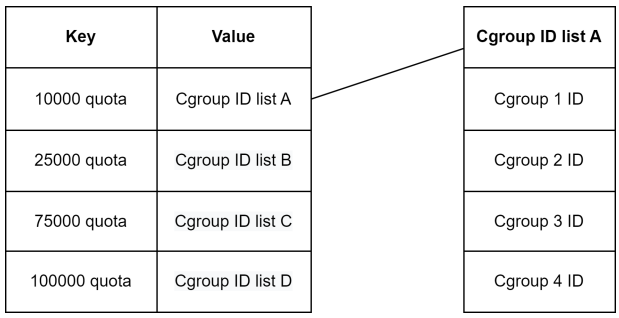



Fig. 2. Example of a populated cgroup cache.

## 5 IMPLEMENTATION

To implement our proposed architecture, we leveraged Graalvisor, described in Section 3.3, as the base of our code. We also incorporated the insights gained from "Performance Isolation in GraalVM Native Image Isolates" [8] described in Section 3.2, adapting these findings to our specific objectives. Throughout the implementation process, we encountered unique challenges and developed innovative solutions to bridge the gap between theory and practice.

### 5.1 C Modifications

The outset of our project involved the intricate process of incorporating the code pertaining to cgroup management operations, composed in the C programming language, with Graalvisor's existing codebase, which is mainly Java but is prepared to run C code. This integration necessitated more than a mere merge; it entailed thoughtful additions and alterations to ensure the seamless coexistence of these code components.

In the initial phase of the integration, we primarily replicated the code without substantial modifications. This approach was undertaken to facilitate the commencement of testing procedures, specifically focusing on the creation, updating, and deletion of cgroups. Additionally, it was essential to ascertain the precise execution of the function registered within Graalvisor, thereby confirming its confinement within the designated cgroup and exclusive utilization of the allocated CPU resources.

After encountering certain challenges during the integration process, we reinforced the integrated code to bolster resilience. This entailed a comprehensive review of the foundational C code. Rather than a superficial examination of each C instruction, we systematically introduced code segments designed to validate system calls, proactively identifying and mitigating potential issues. This rigorous approach was employed to prevent latent errors that could compromise the reliability and stability of the integrated codebase, since these errors occurred silently, without any apparent error messages to alert us.

Moreover, a significant transformation was introduced in the code responsible for allocating CPU resources to newly created cgroups. In contrast to the previous methodology, which relied on 'cpu.weight', our approach introduced the utilization of 'cpu.max'.

From a technical perspective, it would have been possible to reimplement the instructions and system calls originally written in C using Java. However, in the interest of code efficiency and pragmatic simplicity, we chose to maintain the original C code, thus preserving the majority of the existing code.

### 5.2 Java Modifications

In the context of the modifications within the main Graalvisor codebase, the initial step involved the implementation of a straightforward extension. This extension facilitated the testing of the C developments, as discussed in Section 5.1, by enabling the invocation of the new methods responsible for executing the C code. During this stage, we conducted a comprehensive verification process to ensure that both the main cgroup and the worker cgroups were successfully generated and that they possessed the desired configurations. This validation process was critical for confirming the seamless interaction between the Java code and the C code, as well as the interaction between the C code and the underlying system.

*5.2.1 Caching CGroups.* The next phase of our implementation journey involved the detailed development of cgroup cache logic, a critical component in enhancing the efficiency of cgroup operations. To meet this objective, we introduced a novel class, 'CgroupCache',

meticulously crafted to take on the multifaceted role of managing various cgroup operations while placing a special emphasis on cgroup caching mechanisms. This class contains the Java code that calls the C code which we discussed in Section 5.1. CgroupCache is responsible for managing various essential aspects of cgroup operations. It orchestrates the entire life cycle of cgroups, which encompasses the creation of the primary cgroup (as explained in Section 2.2), and the subsequent creation, updating, and removal of worker cgroups.

In addition, the constructor of this class incorporated a boolean parameter, which was made configurable through the environment variable 'use_cgroup_cache'. Graalvisor leveraged this parameter to enable or disable the cache as needed. This flexibility was instrumental in facilitating the succeeding performance comparisons between the cached and uncached versions, enabling us to discern and quantify the extent of any potential improvements achieved through caching. Graalvisor underwent a specific modification where we introduced an additional query parameter for function invocation." This parameter represents the CPU quota required for the function's execution. This change ensured that during function registration, the CPU quota information was associated and preserved for subsequent use.

To enable the deletion of cgroups following each function invocation (in the uncached version), we established a tracking mechanism to monitor which cgroups were handling specific threads. To achieve this, we used a ConcurrentHashMap, utilizing thread IDs as keys and cgroup IDs as corresponding values. This implementation ensured that as threads were inserted into cgroups, this data structure was updated accordingly. When threads were subsequently removed from cgroups, the associated key-value pairs were efficiently removed from the structure. Finally, as mentioned in Section 4.3, we established another ConcurrentHashMap in which the CPU quota of the cgroup served as the key. The corresponding value was a CopyOnWriteArray that contained multiple cgroup IDs with the same CPU quota.

## 5.3 Graalvisor Extension

As Graalvisor bootstraps, it is configured to consider the environment variable introduced in Section 5.2.1. This variable assumes binary values, where 'true' signifies the activation of cgroup caching, and 'false' designates its deactivation, thereby dictating Graalvisor's behavior accordingly.

*5.3.1 Uncached Version.* The CgroupCache class will invoke the C code responsible for creating the main cgroup structure, which envelops the worker cgroups that are going to be created later on. When a function is registered and invoked in Graalvisor, the platform reads the CPU quota query parameter, creating a corresponding cgroup, with the specified CPU quota, for that function, and adding an entry to the structure that maps thread IDs to cgroup IDs. When a function execution concludes, Graalvisor automatically deletes the corresponding cgroup, removes the entry from the map, and any new function invocations follow the same pattern, creating and removing their respective cgroup.

*5.3.2 Cached Version.* This variant, which supports cgroup caching, operates in a manner very similar to the uncached version previously discussed, but it introduces some key differences:

- After initializing and establishing the main cgroup structure during startup, the CgroupCache class preloads the cache by creating cgroups with frequently used CPU quotas. This preloading prevents on-the-fly cgroup creation when function invocations occur.
- When a function execution occurs, it will operate within a cached cgroup with a corresponding CPU quota, if such a cgroup is present. Furthermore, this cached cgroup will be removed from the cache while the function is in progress, marking it as unavailable. This approach minimizes the need to create new cgroups by reusing previously created ones.
- Regarding the deletion of the cgroups, contrary to the version lacking cached cgroups, the cached variant follows a different strategy. Instead of deleting the cgroup upon the completion of a function's execution, it is preserved and reincorporated into our caching structure, as elucidated in Section 4.3. As a result, forthcoming invocations that demand an identical CPU quota can efficiently reutilize these pre-existing cgroups, thus minimizing the need for creating new ones and consequent delay.

Our cache operates with a lazy approach. To efficiently manage cgroup utilization, we've implemented a structure that maps cgroups to thread IDs. When a function concludes its execution, the cgroup it utilized is marked as available in the cache, but the associated thread isn't immediately removed. Instead, this information is retained. When a new function execution requires a cgroup, two scenarios arise:

- If the new function is running on the same thread that previously occupied the selected cgroup, the cgroup is promptly marked as unavailable and removed from the cache.
- In cases where the new function is executed on a different thread, the old thread is removed from the cgroup at this point, allowing the new thread to be added to the cgroup filesystem.

This strategy minimizes the need for unnecessary cgroup operations, resulting in more efficient cgroup utilization.

# 6 EVALUATION

In this section, we aim to provide an evidence-based analysis of how well our system works and the extent to which it meets its intended objectives, namely, maintaining consistently low latency for cgroup management operations.

## 6.1 Benchmarks

To evaluate the performance of our proposed solution, we utilized a set of four benchmarks, present in Section 3.3, three of which were originally introduced in the Photons[5] paper. These benchmarks have been subsequently employed in the Performance Isolation in GraalVM Native Image Isolates[8] and will provide an appropriate benchmark for our proposed caching approach. These benchmarks provide a comprehensive and representative sample of the functions commonly employed in current serverless technologies. They

include Hello World, Fibonacci, File Hashing, and Video Processing, and represent both IO-intensive, mixed, and CPU-intensive benchmarks. Each of these functions is registered as an HTTP request to the Graalvisor API, which, upon completion of the function execution, returns a response to the client.

- **Hello world:** This benchmark exemplifies the most elementary and expedient type of function one can envision. In this scenario, a basic program prints the iconic 'Hello World' string, and this string is subsequently returned to the user as a response.
- **Fibonacci:** This benchmark demonstrates a CPU-bound function tailored to calculate Fibonacci numbers. It receives an integer representing the nth term of the Fibonacci sequence, specifically the 150th number, and calculates its value. It serves as a model for applications with high computational demands typically encountered in mathematical and computational contexts
- **File Hashing:** This benchmark emulates data processing tasks that often encompass file downloads and the simultaneous processing of data chunks. This benchmark is designed to replicate the operations commonly encountered in diverse applications where data is divided into smaller segments and processed concurrently. Specifically, it simulates the process of downloading a file, with the file used in this case having a size of 41KB, from a local server. Following the download, the benchmark proceeds to execute a hashing operation on the acquired file.
- **Video Processing:** This benchmark mimics video processing, simulating the process of downloading a portion of a video and subsequently reducing its resolution. It effectively represents the types of video processing and transformations that are frequently implemented using serverless technologies in contemporary applications.

## 6.2 Evaluation environment

The experiments were conducted using a virtual machine hosted on a computer running the Windows 11 Pro N operating system. The host machine is equipped with an AMD Ryzen 5 3600 processor, operating at 3.80GHz, comprising 6 cores (12 Logical Processors). The virtual machine itself runs the Linux Ubuntu 22.04.3 LTS operating system. The virtualization was achieved through the Oracle VM VirtualBox hypervisor, providing the virtual machine access to 4 out of the 12 logical CPU cores, and 12GB of memory out of the 32GB available in the host. However, it's important to note that, as our experiments were conducted within a virtualized environment, there are certain factors related to the host operating system and virtualization technology that are beyond our control and understanding. These factors may introduce overhead, variability, and limitations that, albeit vastly limited, could impact the experimental results such as CPU Fluctuations, Network Performance, Disk I/O, Memory Allocation, Hypervisor Overheads, or Clock Drift.

## 6.3 Metrics

The performance metrics that are of relevance to our research are the utilization of CPU and the function execution times, which contain the latency of `cgroup` management operations, such as the creation, deletion, and updating of `cgroups`:

- **CPU Usage** At first, we experimented with various CPU quotas to verify that the functions executed within their designated `cgroups` while utilizing only the allocated CPU resources. Subsequently, we standardized these CPU quotas to one full core for all functions in both the cached and uncached variants. Our primary objective was to evaluate whether running Graalvisor with cached `cgroups` resulted in reduced execution times compared to the uncached `cgroups` version. Note that not all of the functions need a full core of CPU, and we were able to see that only the necessary amount was used.
- **Function Execution Times** The total processing time in Graalvisor encompasses both function execution within a `cgroup` and the time required for any essential `cgroup` management operations. In the warmup phase of the Graalvisor's cached version, the cache is populated with a set of `cgroups`. Therefore, the sole additional overhead during function execution relates to the `cgroup` management operations when they are necessary. On the other hand, the uncached version dynamically generates and deletes a `cgroup` for each function while simultaneously inserting a thread into the respective `cgroup`. These factors ensure that the disparity in execution times between the two variants primarily consists of the `cgroup` management operations. Consequently, our measurements exclusively account for the function's processing time on the server side.

## 6.4 CGroup Management Costs

To gain a comprehensive understanding of the costs associated with `cgroup` management operations, we needed to measure the time required for each of the operations. In that sense, we conducted an experiment using a script that replicates and measures all the essential `cgroup` operations performed by our solution to execute a function within a `cgroup`, described in Section 2.2. The script involves the following actions:

(1) Creating a `cgroup`
(2) Updating the `cgroup`'s allowed CPU
(3) Launching a dummy thread (`"sleep 1"`)
(4) Inserting the formerly created thread into the `cgroup`
(5) Removing the `cgroup` once the thread ended executing

These steps were executed repeatedly, at various levels of concurrency, and using the collected timing data, we generated the graph depicted in Figure 3.

Based on these findings, we can infer that the major cost factors in `cgroup` management operations lie within 'Cgroup adding,' which involves adding a thread to the `cgroup`, and 'Cgroup creation,' which pertains to the creation of the `cgroup` itself. Therefore, we can expect to achieve improved function execution results by strategically mitigating or eliminating the impact of these time-consuming operations.

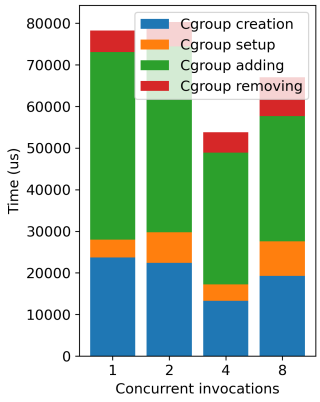
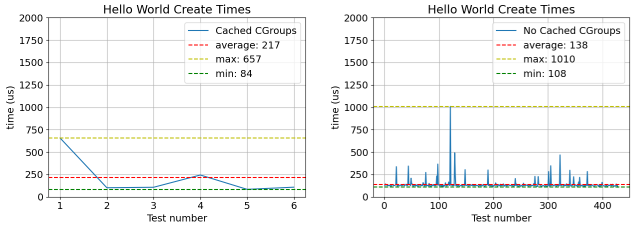



Fig. 3. CGroup Management Operations Times.

## 6.5 Lazy Reclamation Results

The different types of functions were executed consecutively for a specific number of tests, enabling the execution times to stabilize and provide reliable values. Functions that had lower resource consumption and quicker execution were subjected to a larger number of tests for accuracy, specifically 500 invocations for the 'Hello World' and 'Fibonacci' functions, and 100 invocations for 'File Hashing' and 'Video Processing' functions. To enhance the quality of the results and to evaluate the system during typical execution, we excluded the initial 10% of function executions, which can be affected by system warm-up and may yield longer times. Subsequently, we identified and removed any outliers from the dataset, resulting in the data used for the plotted results below.

It is important to highlight that the subsequent plots for the cached versions of the system are based on a lazy caching strategy. This strategy optimizes resource utilization and updates `cgroup` threads only when necessary, thereby minimizing overhead and ensuring efficient operation.

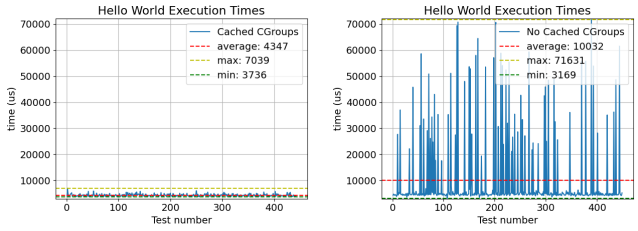



**(a)** Cached Version. **(b)** Uncached Version.

Fig. 4. Hello World CGroup Creation Times.

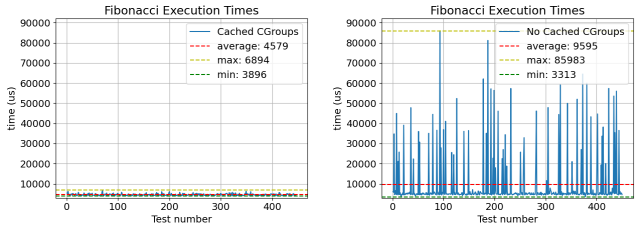
The graphs present in Figure 4 illustrate the `cgroup` creation times for the 'Hello World' function. We can see that performance doesn't change much, and that makes sense since the only thing we are doing is creating the `cgroups`. One interesting observation is that these times are considerably lower than those measured in Section 6.4. While we lack concrete evidence, it's plausible that creating `cgroups` consecutively might make the latter ones more efficient than the initial ones.

The graphs displayed in Figures 5, 6, 7 and 8 represent the processing times of each function. The Y scale was adjusted, accordingly, to contain the minimum and maximum values for each of them, allowing an enhanced visualization and comparison.




**(a)** Cached Version. **(b)** Uncached Version.

Fig. 5. Hello World Execution Times.



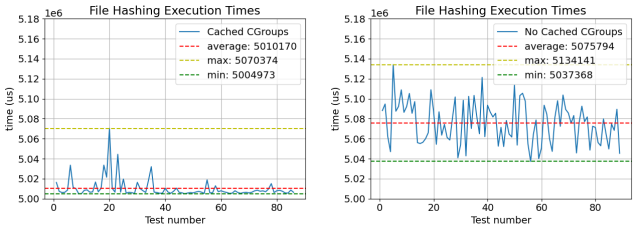

**(a)** Cached Version. **(b)** Uncached Version.

Fig. 6. Fibonacci Execution Times.

Given the inherently swift execution of the 'Hello World' and 'Fibonacci' functions, the benefits of avoiding `cgroup` creation and deletion are more significantly noticeable. As illustrated in Figures 5 and 6, functions that operate with cached `cgroups` exhibit a notably enhanced speed, experiencing, respectively, an improvement of approximately 70% and 50% on average, compared to their uncached counterparts.

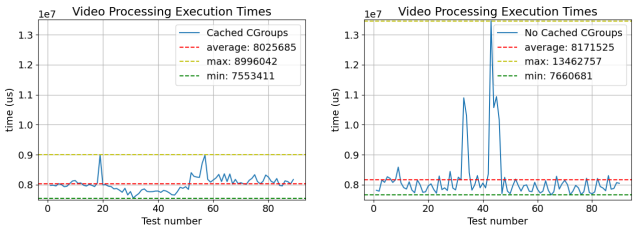
Conversely, for functions with substantially prolonged execution times, such as 'File Hashing' and 'Video Processing,' as depicted in Figures 7 and 8, the obtained benefits of bypassing `cgroup` creation and deletion are notably less apparent, culminating in a relatively minor speed increase of approximately 1.5% on average.

Despite the functions that are more time-consuming yielding a less pronounced improvement in performance, it is essential to acknowledge that they still derive advantages from utilizing cached `cgroups`. The less apparent results are predominantly attributable to the relatively minimal overhead incurred by `cgroup` management operations. In essence, the time taken for these functions to execute significantly surpasses the duration of the `cgroup` management operations. Consequently, the marginal gains in execution time are rendered less discernible.




**(a)** Cached Version. **(b)** Uncached Version.

Fig. 7. File Hashing Execution Times.




**(a)** Cached Version. **(b)** Uncached Version.

Fig. 8. Video Processing Execution Times.

## 6.6 Non-Lazy Reclamation Results

Figures 9, 10, 11 and 12 display graphs containing data from similar experiments to the previous set, running under the same conditions and for the same number of tests, but without utilizing the lazy caching reclamation. In this non-lazy approach, after each thread completes its execution, not only the associated `cgroup` with the finished thread is removed from the cache but also the thread itself is removed from the `cgroup` file, which adds two additional steps to every function execution in the cached variant: removal and re-addition of the thread to the `cgroups`.

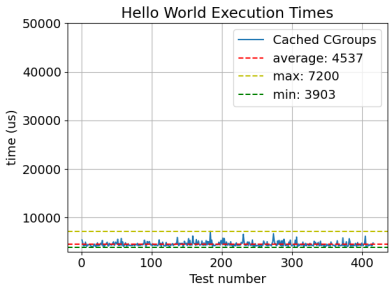



Fig. 9. Hello World Execution Times.

In this experimental scenario, we can meticulously observe, as evidenced in Figure 9, that the performance of the 'Hello World' function exhibited only marginal alterations when the lazy cache reclamation was omitted from the equation.

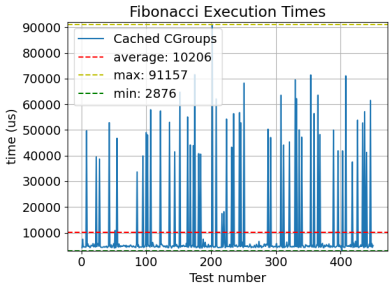



Fig. 10. Fibonacci Execution Times.

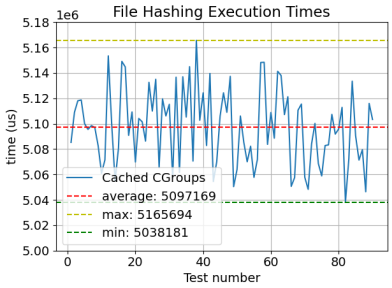



Fig. 11. File Hashing Execution Times.

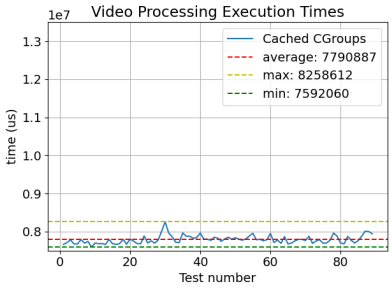



Fig. 12. Video Processing Execution Times.

In contrast, when scrutinizing the execution times of the remaining functions, specifically 'Fibonacci', 'File Hashing', and 'Video Processing', as illustrated in Figures 10, 11, and 12, a noteworthy deterioration in performance was detected when compared to the lazy caching approach. In fact, these functions exhibited performance levels similar to those of the uncached version. The precise cause for this observation remains somewhat enigmatic. However, our investigation pinpointed the issue, establishing a clear correlation between the extended execution times and the process of inserting and removing threads from the `cgroups`. Furthermore, it is plausible that the intricacies of the underlying evaluation environment, characterized by virtual machine settings detailed in Section 6.2, may have also influenced the results. To provide a comprehensive explanation, we have deffer a more detailed analysis of this aspect to forthcoming research efforts.

### 6.7 Discussion

The evaluation of our cgroup caching solution demonstrates its potential benefits, particularly the advantages of employing a lazy caching approach. The results indicate significant performance improvements, with functions that have shorter execution times (expectably the majority of invocations) exhibiting the most noticeable gains. The overall findings support the viability of cgroup caching in optimizing resource management and reducing overhead, making it a promising option for enhancing the efficiency and scalability of serverless infrastructures. The lazy approach, in particular, stands out as an effective strategy for reducing the impact of cgroup management operations and improving function execution times. However, it's important to note that in contrast, the non-lazy cache approach did not yield the same level of performance improvement. These results suggest that the lazy cache reclamation strategy plays a crucial role in achieving superior execution times. This approach presents a promising direction for future optimization efforts, allowing for more efficient and resource-conscious execution of serverless functions within cgroups.

In a nutshell, a cgroup cache appears to be a strategy worth considering as an addition to serverless infrastructure, with the potential to significantly enhance performance and resource efficiency.

## 7 CONCLUSION

In this paper, we have delved deep into the realm of serverless computing and its resource management challenges, particularly focusing on cgroups and their associated management operations. The overarching objective of this paper was to enhance the scalability and performance of serverless infrastructures, specifically in the context of cgroup management.

### 7.1 System Limitations and Future Work

It is vital to address the system's limitations and implications for future research. While the cgroup caching approach, particularly the lazy variant, showed substantial advantages, it is important to acknowledge that not all functions benefited equally. Functions with longer execution times exhibited a less significant performance boost, indicating that there is room for further optimization. The primary bottleneck appears to be related to the insertion and deletion of threads from cgroups.

In light of these limitations, several promising avenues for future research and optimization efforts present themselves. The following areas deserve special consideration:

(1) **Fine-tuning Caching Strategies**: Future work could focus on refining and tailoring the caching strategy to be more effective for functions with longer execution times. Strategies to mitigate the performance impact associated with cgroup management operations could be explored to ensure that all functions, regardless of their runtime and resource usage, derive benefits from the caching system.
(2) **Evaluation in Diverse Environments**: Our evaluations were conducted within a specific virtualized environment, which may not be entirely representative of real-world serverless infrastructure. Future research should include diverse environments, including cloud-based and on-premises setups, to better understand how our caching solution performs in various contexts.
(3) **Comprehensive Benchmarks**: Extending benchmark tests to cover a wider range of functions and use cases will provide more comprehensive insights into the effectiveness of the caching approach.
(4) **Resource Monitoring and Allocation**: Exploring resource monitoring and allocation mechanisms within the cgroup hierarchy, especially in response to varying workloads and the dynamic creation and deletion of cgroups, could help improve the overall efficiency of cgroup caching.

### 7.2 Concluding remarks

Our findings strongly suggest that a cgroup cache is a valuable strategy for enhancing the performance and resource efficiency of serverless infrastructures. While the lazy caching approach has shown substantial benefits, there remains room for optimization and further research to address the limitations observed in our study. By embracing these future directions, we can look forward to a more resource-efficient and high-performing serverless computing environment.

## ACKNOWLEDGMENTS

I want to express my deep gratitude to my parents and my brother for their unwavering support, constant encouragement, and boundless care throughout the years. Their enduring presence and belief in me have been instrumental in making this project a reality.

I'd like to extend my appreciation to my dissertation supervisors, Professor Luís Veiga and Professor Rodrigo Bruno, for their invaluable guidance, relentless support, the wealth of knowledge they've shared, and for never giving up on me, providing me with all the tools necessary for the completion of this thesis.

I'm compelled to offer my heartfelt gratitude to my girlfriend, Bea, whose relentless presence and support have been the driving force behind my ability to give my best effort in these recent months. Her encouragement and help in overcoming personal life obstacles have not only been a source of strength but have also contributed to me becoming the best version of myself. I'm profoundly thankful for her presence in my life.

Lastly, I want to convey my serious appreciation to my friends and colleagues who have played a pivotal role in my personal development and remained steadfast in their support over the past few years.

I want to express my profound appreciation to all of you for being a part of my journey as I complete this significant phase of my life. Your presence and support have meant the world to me. Thank you sincerely from the depths of my heart.

## REFERENCES

[1] [n. d.]. RedHat - Resource Management Guide. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/index

[2] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. 2017. *Serverless Computing: Current Trends and Open Problems.* Springer Singapore, 1–20. https://doi.org/10.1007/978-981-10-5026-8_1

[3] Rodrigo Bruno, Serhii Ivanenko, Sutao Wang, Jovan Stevanovic, and Vojin Jovanovic. 2022. Graalvisor: Virtualized Polyglot Runtime for Serverless Applications. arXiv:2212.10131 [cs.DC]
[4] Karim Djemame, Matthew Parker, and Daniel Datsev. 2020. Open-source Serverless Architectures: an Evaluation of Apache OpenWhisk. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 329–335. https://doi.org/10.1109/UCC48980.2020.00052
[5] V. Dukic, R. Bruno, Ankit Singla, and G. Alonso. 2020. Photons: lambdas on a diet. *Proceedings of the 11th ACM Symposium on Cloud Computing* (2020). https://doi.org/10.1145/3419111.3421297
[6] Nane Kratzke. 2018. A Brief History of Cloud Application Architectures. *Applied Sciences* 8, 8 (2018). https://doi.org/10.3390/app8081368
[7] Shanshan Li, He Zhang, Zijia Jia, Chenxing Zhong, Cheng Zhang, Zhihao Shan, Jinfeng Shen, and Muhammad Ali Babar. 2021. Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. *Information and Software Technology* 131 (2021), 106449. https://doi.org/10.1016/j.infsof.2020.106449
[8] Filipe Sousa. 2022. *Performance Isolation in GraalVM Native Image Isolates*. Master Thesis. Instituto Superior Técnico. https://rodrigo-bruno.github.io/mentoring/81120-Filipe-Sousa-dissertacao.pdf
[9] Jayachander Surbiryala and Chunming Rong. 2019. Cloud Computing: History and Overview. In *2019 IEEE Cloud Summit*. 1–7. https://doi.org/10.1109/CloudSummit47114.2019.00007

Received 31 November 2023; revised 31 November 2023; accepted 31 November 2023