TÉCNICO LISBOA

CGroup Caching @ Graalvisor

David Jorge Santos Nunes

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

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

Examination Committee

Chairperson: Prof. Valentina Nisi
Supervisor: Prof. Rodrigo Fraga Barcelos Paulus Bruno
Member of the Committee: Dr. João Nuno De Oliveira e Silva

November 2023

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Declaration

I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

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.

Abstract

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 thesis 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 proposed a caching mechanism for `cgroups`, which is implemented using the GraalVM platform. We evaluated the effectiveness of this approach using four representative benchmarks, including Hello World, Fibonacci, File Hashing, and Video Transformation. Our evaluation results reveal the substantial potential of our proposed caching solution in improving the performance of `cgroups` within FaaS environments, by removing the overhead related to initialization, with particularly noteworthy gains observed in functions characterized by rapid execution.

Keywords

Serverless computing; GraalVM; Control Groups; Function-as-a-Service; Graalvisor.

Resumo

A computação em nuvem é uma tecnologia transformadora que disponibiliza uma ampla gama de serviços e recursos pela internet. Isso permite que empresas e indivíduos acedam, dimensionem e paguem por capacidades de computação conforme necessário. Esta abordagem flexível e económica oferece suporte à transformação digital, inovação e eficiência em vários setores.

A computação serverless (”sem servidor”) simplifica o desenvolvimento de aplicações ao abstrair a gestão da infraestrutura. Os desenvolvedores concentram-se no código, enquanto o provedor de nuvem cuida da escalabilidade e manutenção, o que oferece agilidade e uma eficiência de custos ainda melhor, tornando-o adequado para aplicações contemporâneas.

A presente tese estuda a utilização de grupos de controlo (`cgroups`) em ambientes serverless, especificamente no contexto de Funções como Serviço (FaaS). O uso de `cgroups` para a invocação de funções em FaaS é conhecido por apresentar problemas de desempenho durante o start-up, resultando em latência e dificuldades de inicialização. Para resolver esse problema, propomos um mecanismo de armazenamento em cache para `cgroups`, implementado através do uso da plataforma GraalVM. Avaliamos a eficácia desta proposta usando quatro benchmarks representativas, incluindo Hello World, Fibonacci, Hashing de Ficheiros e Transformação de Vídeo. Os resultados revelam o potencial da nossa solução de armazenamento em cache na melhoria do desempenho de `cgroups` em ambientes FaaS, removendo o overhead relacionado com a inicialização e mostrando ganhos particularmente significativos em funções de execução rápida.

Palavras Chave

Computação Serverless; GraalVM; Grupos de Controlo; Função como Serviço; Graalvisor.

Contents

List of Figures

List of Algorithms

1

Introduction

Contents

1.1 Motivation

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.

1.2 Shortcomings of current solutions

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 [6] [7], 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 regarding 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.

1.3 Proposed Solution

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.

1.4 Contributions

This thesis 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`. These contributions offer valuable insights and practical solutions to enhance the performance and resource utilization in evolving serverless systems.

1.5 Document Roadmap

In Chapter 2, we address the background, where we explore the fundamental concepts and technologies underpinning this thesis, such as an in-depth description of `cgroup` management operations. In Chapter 3 we present an overview of the related work, offering insights into prior work relevant to the topics discussed in this thesis. In Chapter 4 we delve into the architecture of our solution, explaining the inner workings of the `cgroup` cache and its integration into the Function as a Service (FaaS) workflow. In Chapter 5 we give a thorough description of our implementation, introducing novel ideas and some

system modifications pivotal to our research. In Chapter 6 we present our evaluation, covering the experimental environment, workloads, and metrics used for performance comparisons. Lastly, in Chapter 7 we summarize this thesis's main focus, highlight key findings, and propose directions for future research in the field.

2

Background

Contents

In this chapter, our objective is to provide readers with the fundamental concepts and technologies pivotal to a thorough comprehension of the contributions made by this research. To achieve this, we have structured these topics into four key categories: Cloud Computing, Resource Management and Scheduling, Managed Runtimes, and CGroups. By doing so, we aim to offer a comprehensive foundation for exploring the subsequent discussion of related work.

2.1 Cloud Computing Deployment and Service Models

Cloud computing has been growing a lot in the last few years, regarding active users and providers. It is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction [8]. This cloud model comprises five essential characteristics, four service models, and four deployment models.

2.1.1 Essential characteristics

Cloud computing is an on-demand self-service, meaning that each consumer can automatically provision the needed computing capabilities without requiring any human interaction with each service provider. Another characteristic is broad network access: capabilities are available over the network and easily accessed by standard mechanisms. Cloud computing also consists of resource pooling, i.e., the provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with resources assigned according to consumer demand. The rapid elasticity allows resources to be elastically provisioned and released to rapidly scale as needed and, often, it looks to the consumer as if they are unlimited. Most importantly, cloud systems control and optimize resource usage by leveraging a metering capability at some level of abstraction appropriate to the type of service. Resource usage can be monitored, controlled, and reported providing transparency for both the provider and consumer of the utilized service.

2.1.2 Deployment models

Cloud infrastructure is the collection of hardware and software that enables the characteristics of cloud computing mentioned above. It can be deployed in several ways. Private cloud deployment is a model where the infrastructure is provisioned for exclusive use by a single organization. It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises. The Community cloud deployment model is characterized by the infrastructure being provisioned for exclusive use by a specific community of consumers. It may be owned, managed, and

operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises. In the Public cloud deployment model, the cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them, and exists on the premises of the cloud provider. The composition of two or more of the previous cloud deployment models is called a Hybrid cloud deployment.

2.1.3 Service models

Cloud computing has four main service models to offer, each satisfying a unique set of business requirements. These models are known as Software as a Service (SaaS), Platform as a Service (PaaS), Infrastructure as a Service (IaaS), and Function as a Service (FaaS).

Infrastructure as a Service: The consumer is capable of provisioning storage, processing, networks, and other fundamental computing resources of the infrastructure where he will also deploy and run the software. While the consumer has no responsibility for the management or control of the underlying cloud infrastructure, they do have control over the operating systems, storage, and deployed applications.

Platform as a Service: The consumer may deploy applications created using programming languages, libraries, services, and tools supported by the provider, onto the cloud infrastructure. The consumer has no responsibility for the management or control of the underlying cloud infrastructure, including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.

Function as a Service: The newest model to appear and the most relevant one to us, allows the consumer to run code in response to events. In this model, he can be solely focused on individual functions in the application code, since the management and control of the underlying cloud infrastructure is the responsibility of the cloud provider, which has to provide the illusion of always-available resources.

Software as a Service: The consumer is allowed to use the provider's applications running on a cloud infrastructure. These applications are accessible from various client devices through either a thin client interface, such as a web browser, or a program interface. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application settings.

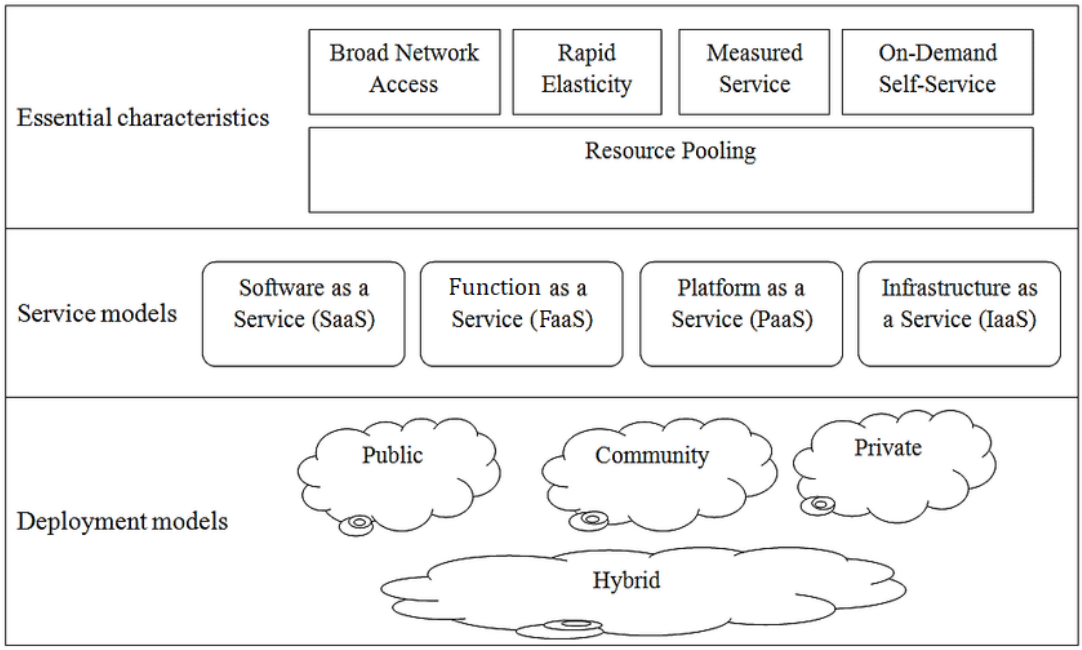


Figure 2.1: Cloud computing diagram.

2.2 Evolution of Cloud Architectures

The evolution of cloud architecture [9] [10] 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, referenced in Section 2.1.3, have played instrumental roles in shaping the cloud ecosystem.

2.2.1 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.

2.2.2 Microservices

In response to the limitations of monolithic architecture, microservices emerged as a transformative paradigm [11]. Microservices architecture involves decomposing complex applications into smaller, in-

dependent 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.

2.2.3 Serverless and FaaS

The relentless pursuit of efficiency and developer productivity led to the rise of Serverless Computing [12], 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. This evolution underscores the continuous quest for efficiency, scalability, and adaptability in the digital realm, emphasizing the integral role that cloud service models have played in shaping the modern cloud computing environment.

2.3 Resource Management and Scheduling

One of the most important things to consider when thinking about cloud computing is the management of resources, especially CPU and memory. All cloud users want the best set of resources at the lowest cost possible. In order to improve the utilization of resources such as saving energy, maximizing resource sharing, and reducing operating costs in the cloud, many strategies for automatic resource scheduling have been studied and developed. However, most scheduling strategies that currently exist were developed thinking about the more conventional cloud computing models that are characterized by longer-term resource allocation and big workloads. Scheduling is the mapping of tasks to resources. it manifests in threads to CPUs, containers to CPUs, and VMs to CPUs. The strategies of the schedulers

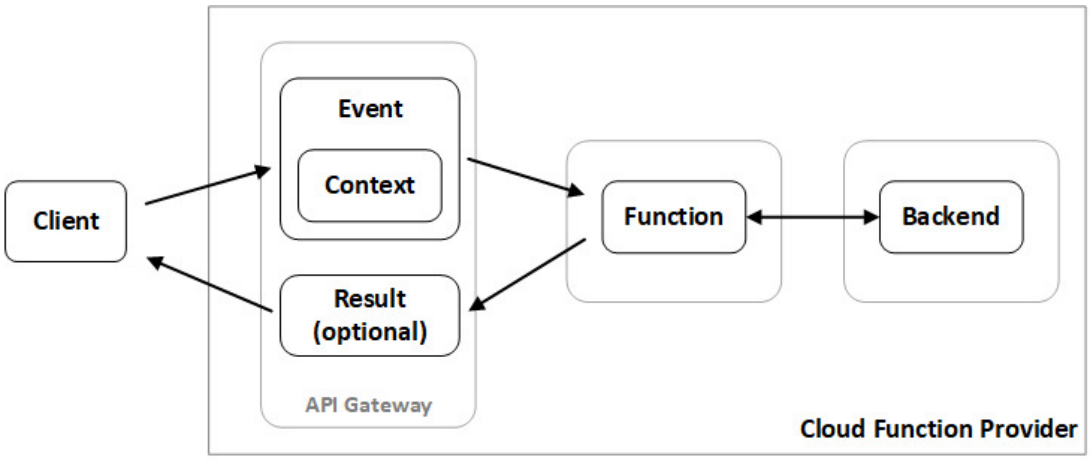


Figure 2.2: Function as a Service Architecture diagram [1]

are typically abstract and they may be applied to a variety of these domains. In this section, we present the most relevant design decisions for resource management and scheduling.

The cloud computing system maps a large number of physical resources to the virtual machines (VM). The chosen VM scheduling algorithm will assign tasks to the VMs and then deploy them to different physical machines (PM) to achieve the sharing of resources, as well as to ensure the Quality of Service and system performance.

Virtual Machine scheduling can be divided into three steps: user requests, resource management, and deployment stage. First, users send requests over the Internet to the cloud providers, generating the workloads that are processed using cloud resources. Then, the scheduling center monitors the physical and virtual nodes in real time. When a VM or PM fails, any tasks or VMs that were running on that failed machine must be migrated to other machines with similar resources. At last, the deployment stage can be divided into two levels: the matching process between tasks and VMs and the matching process between VMs and PMs. A reasonable scheduling method to choose the VMs is to find a mapping between tasks and VMs that meet a certain optimization objective. After choosing the VMs, they are deployed to PMs to ensure the implementation of tasks.

The improvements to scheduling can be oriented to different objectives, depending on the user and provider's needs. Recently, the main five aspects that the schedulers are following are QoS (Quality of Service), Energy, Resources, Cost, and Workload.

Since scheduling in cloud computing is an NP (Non-Deterministic Polynomial) problem, a heuristic algorithm is needed to solve it. Traditional batch mode heuristic scheduling algorithms (BMHA) usually aim at optimizing the execution time and load balancing of the tasks, so that the scheduling system can get a good QoS. The main examples of BMHA-based algorithms are First Come First Served (FCFS), Round Robin (RR), Min-Min, Max-Min, and Shortest Job First (SJF). These algorithms are widely used and many others were derived from them [13]. A brief description of each one is given below.

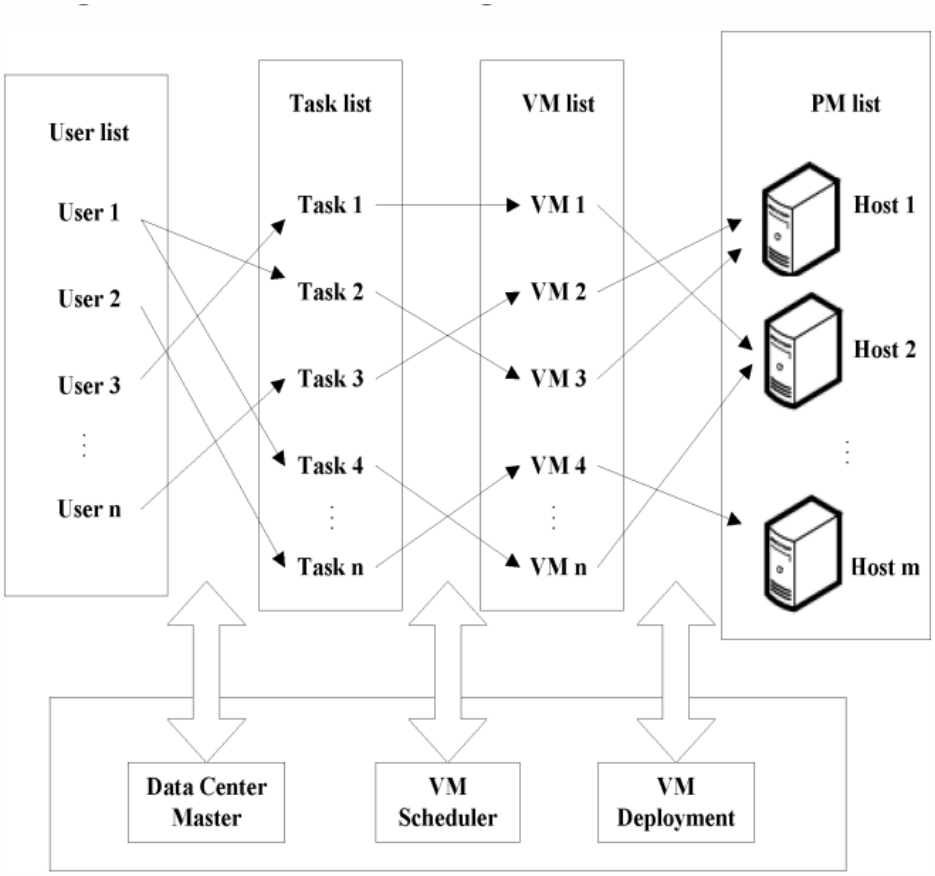


Figure 2.3: VM scheduling model [2].

First come first served This algorithm assigns each incoming task to a free virtual machine by order of arrival.

Round Robin The cloud resources are provided based on the time slices, i.e. each VM has a list of time slices for each task to be executed. The first task is randomly allocated to a VM, and the subsequent tasks are assigned in a circular order to the other available ones. Once a task is assigned, the chosen VM is moved in a circular motion to the end of the available VMs list. Moreover, if the execution of a task has not finished in the time slice given by the VM, the process is interrupted and the next task on this specific VM list will take place.

Min-Min The algorithm first computes the expected execution time of tasks on each VM, and then schedules, the task with the minimum execution time to the corresponding VM. After the scheduling process, each VM will have tasks assigned by order of execution time, from lowest to highest.

Max-Min This algorithm is similar to the Min-Min algorithm. It computes the expected execution time of each task for every VM and then schedules the task with the maximum execution time for the corresponding VMs. After the scheduling process, each VM will have tasks assigned by order of execution time, from highest to lowest.

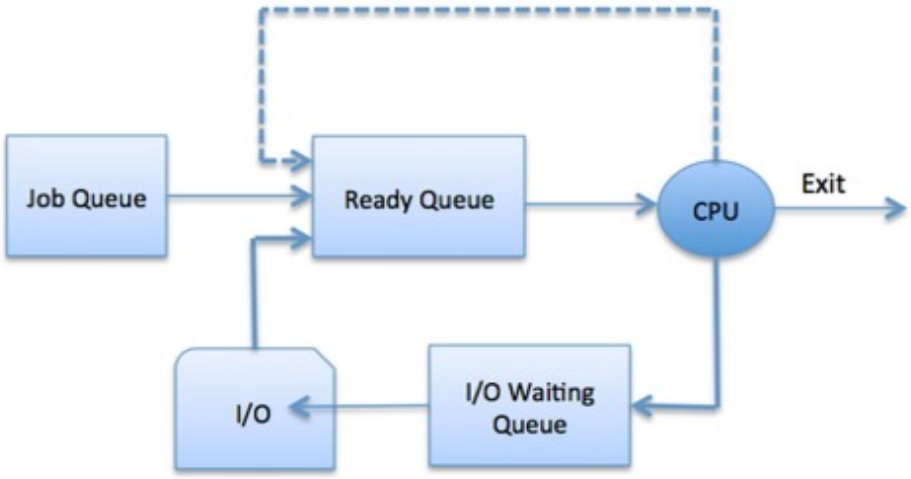


Figure 2.4: Resource scheduling in operating systems.

Shortest Job First In this algorithm, it is performed a sort of the incoming tasks based on their length and then, for the new sorted list, proceeds as the first come first served algorithm [14].

The fundamental principles previously elucidated are integral components of the contemporary Linux kernel's resource management framework. A viable approach to manage resources and interact with scheduling from user space is by manipulation of CPU `cgroups`.

2.4 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. Various subsystems have been implemented, making it possible to do things such as:

- **Resource limiting:** We can configure a `cgroup` to limit how much of a particular resource a

process can use;

- **Prioritization:** We can control how much of a resource a process can use compared to processes in another cgroup when there is resource contention;
- **Accounting:** Resource limits are monitored and reported at the cgroup level;
- **Process Control:** One can change the status (frozen, stopped, or restarted) of all processes in a cgroup with a single command.

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. Thus, for example, the limits placed on a cgroup at a higher level in the hierarchy cannot be exceeded by descendant cgroups.

2.4.1 CGroups Structure

The hierarchy of cgroups is depicted in the form of a directory structure, with its foundational directory rooted at '/sys/fs/cgroup/' illustrated in Figure 2.5. 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.

2.4.2 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 [15] - 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.

2.4.3 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

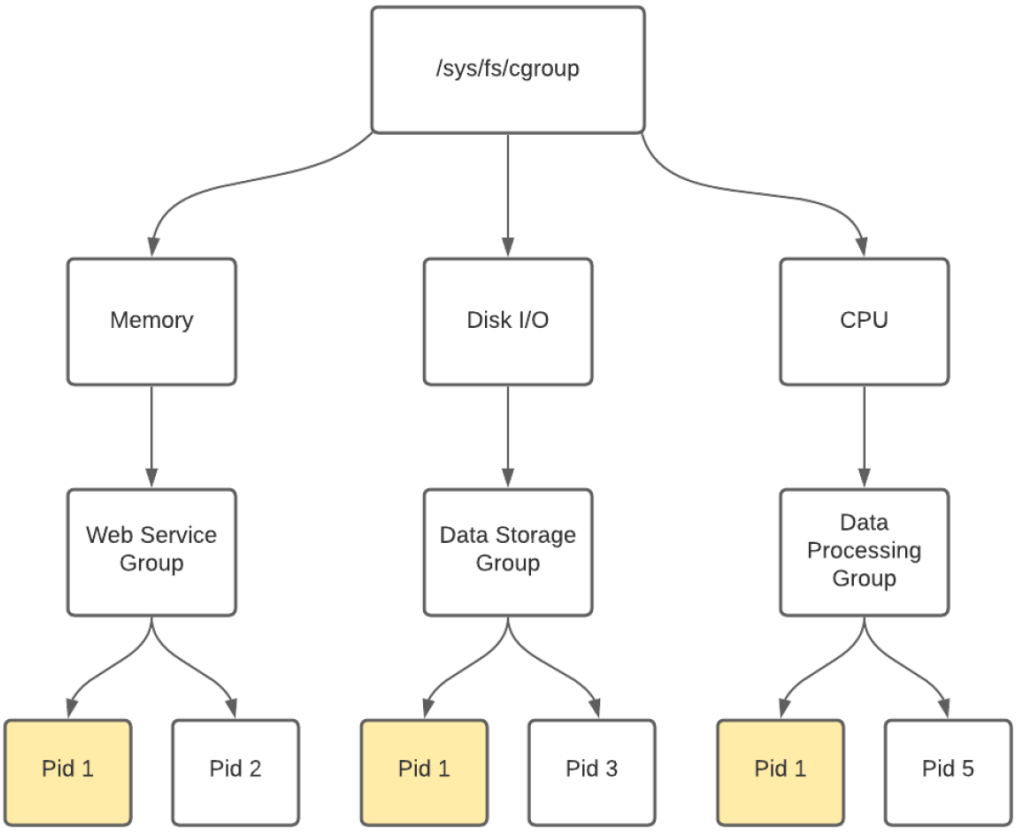


Figure 2.5: CGroup filesystem representation.

with 'cgroup.', and controller files, which begin with the name of the respective controller they pertain to. For instance, you'll find controller files with names like 'cpu.' to signify their association with the CPU controller, which is the most relevant to this work.

Inside the framework of CPU control, various files provide distinct mechanisms for management, including 'cpu.weight', 'cpu.max', and 'cpu.uclamp.max', among others. In our particular approach, we focus on the utilization of 'cpu.max'. This specific 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 total duration of that period. For example, to allocate 10% of the CPU resources, you can represent this as '100 1000' or '1000 10000', which means using 100 us out of a 1000 us window and 1000 us out of a 10000 window, respectively. 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. Figure 2.6 presents a graphical representation of a cgroup hierarchy that aligns with the aforementioned description.

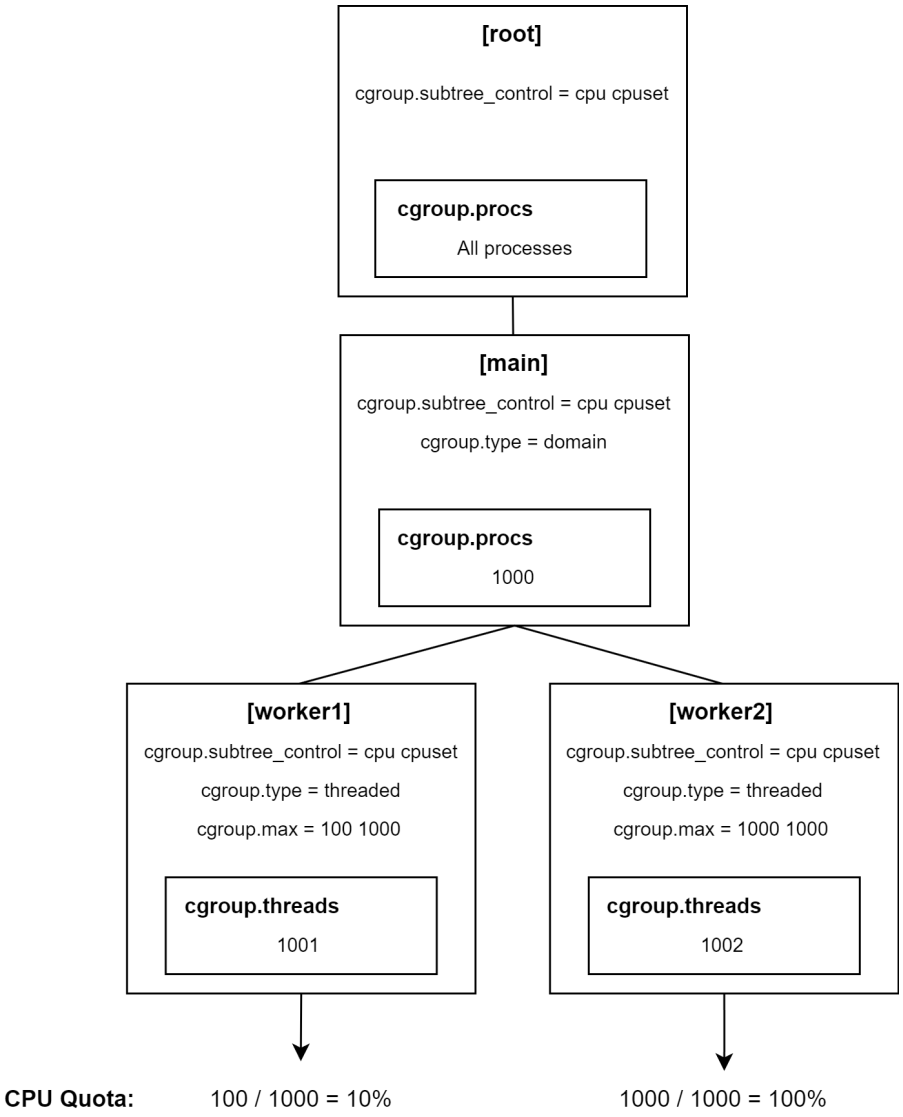


Figure 2.6: Example of a cgroup hierarchy

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.

2.5 Managed Runtimes

In today's cloud computing landscape, most applications are developed using high-level languages (such as Java, Javascript, and Python) which require a language runtime to execute. In this section, we discuss the most important aspects of such runtimes and also elaborate on some techniques used to reduce the overheads in terms of runtime initialization and runtime memory footprint.

2.5.1 The Java Virtual Machine

"The Java Virtual Machine is the cornerstone of the Java platform." This technology is responsible for Java's hardware and operating system independence, the small size of its compiled code, and its security [16].

It is an abstract machine detailed by a specification that formally describes what is required in a JVM implementation. The most popular Java Virtual Machine implementation is the OpenJDK HotSpot JVM, by Oracle. It performs interpretation and just-in-time (JIT) compilation. The Java Virtual Machine allows a computer to run programs in any language with functionality as long as they can be compiled to bytecode.

Figure 2.7 represents the components of Java Virtual Machine, which works as follows: After compiling a .java file, a .class file, containing bytecode, is generated. The Class Loader then loads this bytecode into the runtime data structures, which are subsequently used by the JVM's execution engine components, namely the interpreter, the Just-In-Time (JIT) compiler, and the Garbage Collector. The interpreter dynamically reads the instructions and executes the corresponding machine instructions. The JIT compiler is used to compile the bytecode into native code and store it so it can be used multiple times, enhancing performance. The Garbage Collector is responsible for managing the memory and automatically removing unused memory from the heap.

2.5.2 GraalVM

GraalVM is a novel Java VM implementation, based on HotSpot Java Virtual Machine, designed to achieve better performance. It adds an advanced just-in-time (JIT) optimizing compiler, which is written

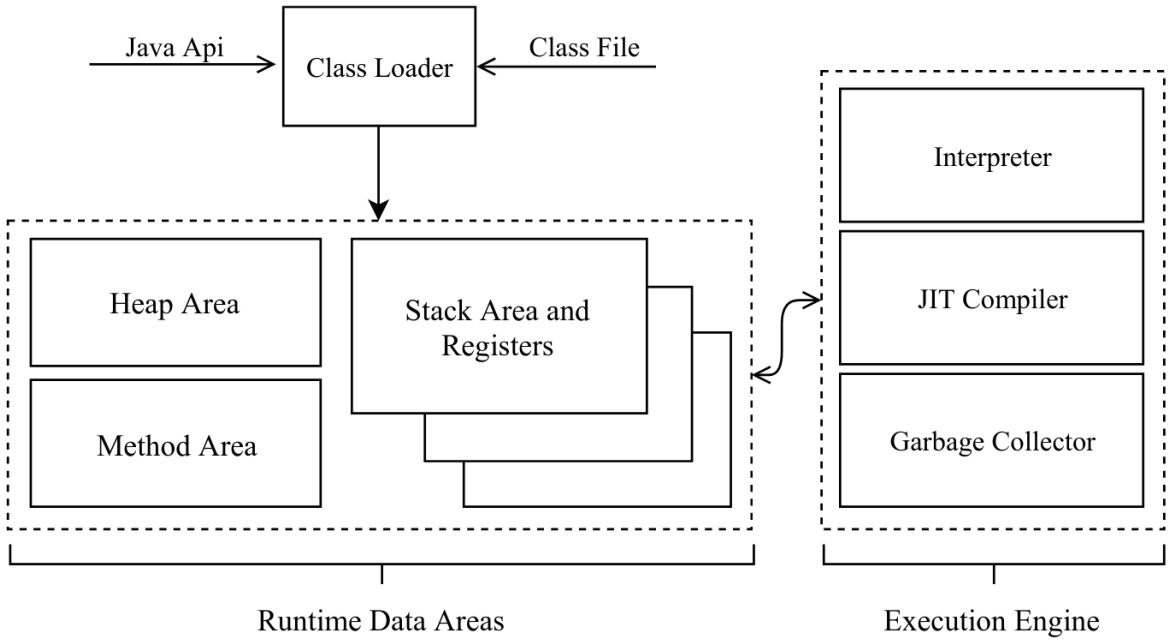


Figure 2.7: Java Virtual Machine. [3]

in Java, to the HotSpot Java Virtual Machine, for faster startup and a lower memory footprint. The first production-ready version, GraalVM 19.0, was released in May 2019 [17].

In addition to running Java and JVM-based languages, GraalVM's language implementation framework (Truffle) makes it possible to run JavaScript, Ruby, Python, and several other popular languages on the JVM. With GraalVM Truffle, Java and other supported languages can directly interoperate with each other and share data and resources [18].

JVM Runtime Mode When running programs on the HotSpot JVM, GraalVM defaults to the GraalVM compiler as the top-tier JIT compiler. At runtime, an application is loaded and executed normally on the JVM. The JVM passes bytecodes for Java or any other JVM-native language to the compiler, which compiles that to the machine code and returns it to the JVM. Interpreters for supported languages, written on top of the Truffle framework, are themselves Java programs that run on the JVM.

Java on Truffle Is an implementation of the Java Virtual Machine Specification, built upon GraalVM as a Truffle interpreter. It is a minified Java VM that includes all core components, implements the same API as the Java Runtime Environment library (`libjvm.so`), and reuses all JARs and native libraries from GraalVM.

Native Image Native Image is an innovative technology that compiles, ahead-of-time, Java code into a standalone native executable or a native shared library. The Java bytecode that is processed during the build of a native executable includes all application classes, classes from its dependencies, third-party dependent libraries, and any classes that are required. A generated self-contained native executable is specific to each individual operating system and machine architecture that does not

require a JVM. It does not run on the Java VM, but includes necessary components like memory management, thread scheduling, and so on from a different runtime system, called “Substrate VM”. The resulting program has a faster startup time and lowers runtime memory overhead compared to an entire JVM.

Isolates The GraalVM provides yet another kind of virtualization we call “language-level” virtualization, allowing a library written in one language to be called directly from another without performance penalties. GraalVM provides a way to use hardware resources even more efficiently by allowing multiple applications to share the same language runtime.

A GraalVM isolate is a disjoint heap that allows multiple tasks in the same virtual machine instance to run independently. In a traditional Java application server, all of the functions share the same memory heap, and if one task uses a lot of memory it can trigger garbage collection (GC), slowing down the other functions sharing that heap. Since isolates are disjoint, each isolate can be garbage-collected independently (or just destroyed before any GC is needed). Isolates are a great tool for managing application multitenancy, or just breaking down a single monolithic application into manageable microservices.

3

Related Work

Contents

In this chapter, our aim is to provide an insightful overview of previous research that has addressed topics similar to those tackled in this thesis. We systematically analyze these previous studies to illuminate their relevance in the context of our research. While acknowledging the merits of these prior works, we also delve into the nuanced aspects that differentiate them from our specific objectives, shedding light on the unique contributions and focus of our project.

3.1 SAND

SAND [19] presents a novel serverless computing platform that introduces two key strategies to provide lower latency, better resource efficiency, and more elasticity than other serverless platforms: 1) application-level sandboxing, and 2) a hierarchical message bus. The authors presented SAND's design and implementation, as well as their experience in building and deploying serverless applications on it.

In their new sandboxing approach, they use tools such as containers to isolate different applications and lighter Operating Systems concepts, such as processes, to isolate functions of the same application. This approach allows the allocation and release of resources for function executions to be much faster and more resource-efficient. The authors combine this with their hierarchical message bus, where each host runs a local message bus to enable quick triggering of functions executing on the same system, to reduce significantly the latency of the interaction between function instances.

3.2 SONIC

In order to reduce data passing latency and cost, SONIC [20] proposes a mechanism that chooses dynamically between three data passing methods, which they call VM-Storage, Direct-Passing, and state-of-practice Remote-Storage. The authors show that no single data-passing method prevails under all scenarios and the optimal choice depends on application-specific parameters such as the size of input data, the size of intermediate data, the application's degree of parallelism, and network bandwidth.

SONIC is a system that jointly optimizes the inter-lambda data exchange method and lambda placement, monitors application parameters, and uses simple regression models to adapt its hybrid data passing accordingly. Finally, it uses an online Viterbi algorithm [21], to globally minimize the application's end-to-end latency, normalized by cost. SONIC is also able to adjust to the best data-passing method considering spontaneous infrastructure changes such as network bandwidth fluctuations.

3.3 Multitasking Virtual Machine

The design of the Multitasking Virtual Machine [22] (called from now on simply MVM) is an adaptation of the Java HotSpot virtual machine that extends it on several novel techniques: an in-runtime design of lightweight isolation, an extension of a garbage collector to provide best-effort management of a portion of the heap space, and a transparent and automated mechanism for safe execution of user-level native code. It enables safe, secure, and scalable multitasking.

Safety is accomplished by strict isolation of applications from one another, enhanced security is guaranteed by resource control mechanisms that prevent some denial-of-service attacks, and the improved scalability results from the application of MVM's main design principle: sharing as much of the runtime as possible among different applications and replicating everything else. The authors describe the system as a 'no compromise' approach since every known API and mechanism of the Java programming language is available to applications.

3.4 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.

3.5 Thin Serverless Functions with GraalVM Native Image

Thin Serverless Functions with GraalVM Native Image [23] is a paper where the authors designed, implemented, and tested a serverless proxy runtime using GraalVM Native Image Isolate, leveraging runtime sharing for concurrent invocations of the same function executions inside the same runtime, which was proposed by Photons. It introduces a solution with object caching and state sharing, by making use of isolates pooling and sharing.

In their design, each invocation is executed in a different isolate with independent heap space. Since isolates split the private states of each invocation automatically, it ensures the correctness of their executions. Using Native Image allows the isolate proxy to achieve a faster start-up and a lower memory

footprint.

3.6 Automated Fine-Grained CPU Cap Control in Serverless Computing Platform

The article Automated Fine-Grained CPU Cap Control in Serverless Computing Platform [24] tries to solve the problem of resource allocation for multi-tenant serverless computing platforms explicitly taking into account workload fluctuations.

Therefore, the authors present a platform-aware technique for managing CPU resources in a serverless computing platform: a resource manager that dynamically adapts the CPU cap (or CPU usage limit) concerning applications with similar performance requirements, which are organized in `cgroups`. This paper, experimentally, confirms that the proposed resource manager can effectively eliminate the burden of explicit reservation of computing capacity, and reduce skewness and average response time, while not overusing CPU resources.

3.7 SOCK: Rapid Task Provisioning with Serverless-Optimized Containers

The authors of the next paper analyze Linux container primitives, identifying container initialization and package dependencies as common causes of slow lambda startup. Thus, they propose SOCK [25], a streamlined container system optimized for serverless workloads, which avoids major kernel bottlenecks and has two goals: 1) low-latency invocation for Python handlers that import libraries and 2) efficient sandbox initialization.

SOCK uses Zygote provisioning, where new processes are forked from the main process, the Zygote, which already has imported the library dependencies that are needed to run the application and has done some initialization work, reducing the initialization work that the child processes have to do. This means that the system must maintain a set of Zygote processes with different sets of preinstalled packages (SOCK's package cache), which is difficult and could be large, in environments executing many different types of applications.

3.8 CNTR: Lightweight OS Containers

CNTR [26] introduces a system for building and deploying lightweight OS containers, that provides the performance advantages of lightweight containers and the functionality of large containers by splitting

the traditional container image into two parts: the “fat” image, and the “slim” image - containing the tools and the main application respectively.

CNTR transparently combines the two container images using a new nested namespace, without any modification to the application, the container manager, or the operating system. It also enables the application container to efficiently and dynamically expand with additional tools in an on-demand fashion at runtime.

3.9 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.

The solution leveraged GraalVM Native Image Isolates for memory isolation and `cgroups` for CPU management. At its core, an HTTP server managed function execution requests with predefined CPU quotas, creating, adjusting, and populating `cgroups` as needed. The evaluation encompassed an analysis of CPU quota efficiency, latency, and memory overhead, revealing varying effectiveness based on function I/O characteristics, with notable efficiency for CPU-bound tasks. Overhead was measured at 20ms to 70ms in latency with minimal memory impact.

This work [6] 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.

3.10 Graalvisor

Graalvisor [7] introduces a high-performance Serverless platform, leveraging GraalVM’s advanced technology, including Native Image, Isolate, and Truffle. This innovative approach co-locates function invocations at scale, reducing latency and memory footprint compared to traditional Serverless platforms.

Graalvisor is driven by three key observations:

- The growing popularity of Function-as-a-Service (FaaS) as a programming paradigm, with an estimated market value of USD 7.7 billion by 2021.

- FaaS operates within a Serverless framework, where users have limited control over the execution environment, with the runtime as the new virtualization boundary.
- Existing virtualization technologies often suffer from latency and memory issues, which Argo mitigates by utilizing a lightweight Native Image Unikernel-based virtual machine.

The research presents a solution to address the challenge of virtualization stack bloat in Serverless computing. Graalvisor 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).

In an evaluation of Graalvisor's performance, the study demonstrates substantial improvements. For Java functions, throughput per memory increases by an average of 170×, while JavaScript functions see a boost of 26.6×, and Python functions experience a 2.07× improvement. When applied to a real-world Serverless trace, Graalvisor reduces overall memory usage by 83% and trims tail latency (99th percentile) by 68%.

3.11 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 3.9. 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 3.10, 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

Contents

This chapter 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. In Section 4.2, we will delve into the specifics of the data structure that has been selected for implementation. This will include an analysis of the data structure's properties, advantages, and limitations. Finally, in Section 4.3, we will explain how the new data structure is integrated into the isolate, and how it fits within the function invocation workflow, i.e. our proposal for the `cgroup` allocation algorithm. We will also provide an overview of the modifications made to the isolate and function invocation workflow to accommodate the new data structure.

4.1 Overview

Our objective in this thesis 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.

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 oper-

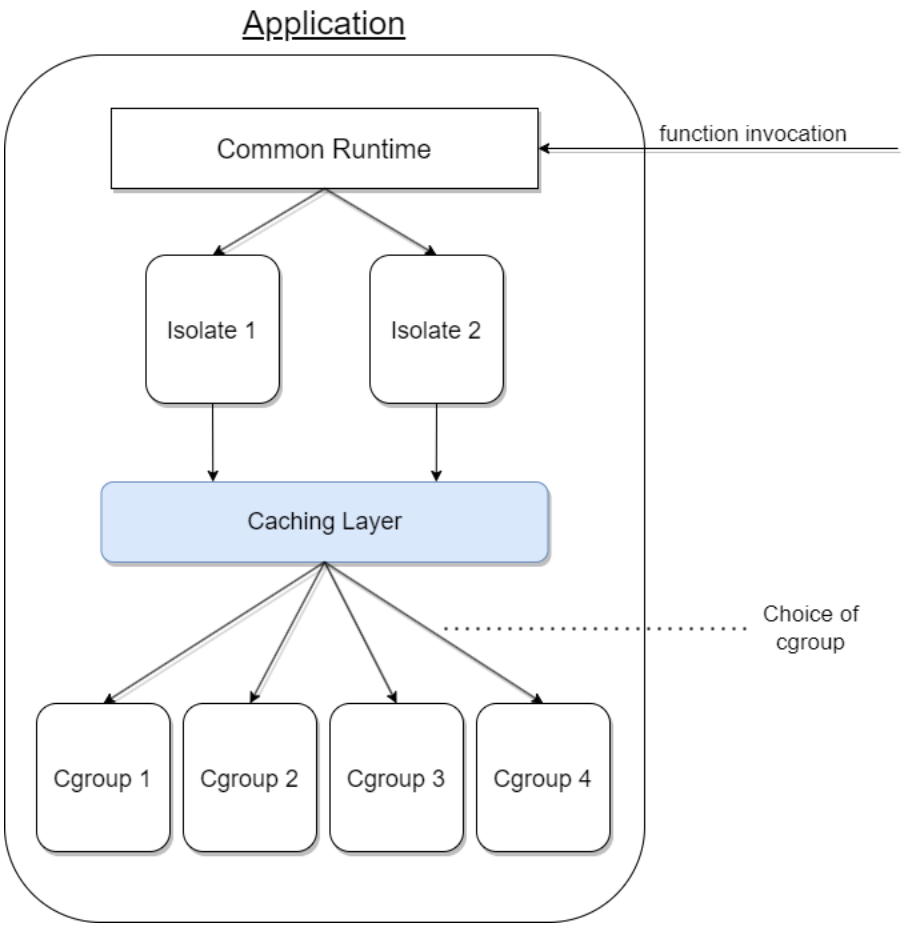


Figure 4.1: Architectural diagram with cache layer.

ating 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 2.4, 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 4.1 illustrates the architectural configuration of the solution with the inclusion of the newly added caching layer.

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

Algorithm 4.1: 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
```

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 4.2 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.

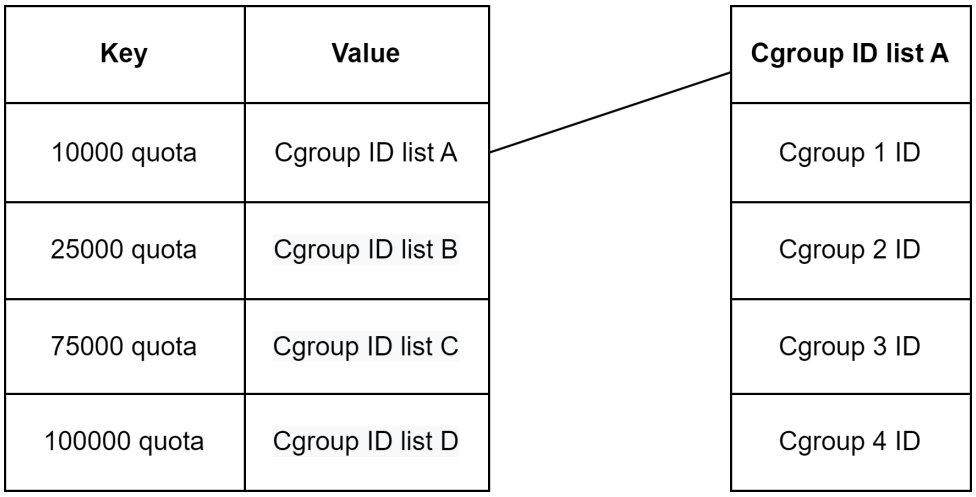


Figure 4.2: Example of a populated `cgroup` cache.

5

Implementation

Contents

In this chapter, we transition from the theoretical architecture to the practical realm of implementation. We will detail the step-by-step process of bringing our proposed solution to life, explaining the technical procedures involved. Furthermore, we will address the challenges and obstacles encountered during implementation, providing valuable insights into the complexities of realizing our project. By the end of this chapter, readers will have a comprehensive view of how our conceptual framework was transformed into a functional system.

To implement our proposed architecture, we leveraged Graalvisor, described in Section 3.10, as the base of our code. We also incorporated the insights gained from "Performance Isolation in GraalVM Native Image Isolates" [6] described in Section 3.9, 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 re-implement 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 2.4), 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

Contents

In this chapter, we shift our focus to the evaluation of our solution's performance and effectiveness. 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. To achieve this, we begin by introducing the benchmarks that will be used to invoke the functions executed in the cloud environment. We provide a detailed description of their functionality and relevance. Subsequently, we present the performance metrics used to measure the effectiveness of our proposed mechanism and the methods used to collect them. This is followed by describing the experimental setup, including the infrastructure used and the specific tests performed. Finally, we present the results from the tests conducted on our solution and draw conclusions regarding the feasibility and utility of a `cgroup` caching algorithm.

6.1 Benchmarks

To evaluate the performance of our proposed solution, we utilized a set of four benchmarks, present in Section 3.10, 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 [6] 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**: The host machine's CPU may be shared with other processes or virtual machines, leading to unpredictable fluctuations in available CPU resources for our experiments.
- **Network Performance**: Network performance can be influenced by the host's network usage and virtualization settings, potentially affecting data transfer times and network-related measurements.
- **Disk I/O**: Disk I/O operations may be influenced by the host's disk usage and virtualization settings, potentially leading to variations in data read and write times.
- **Memory Allocation**: The host's memory usage and virtualization settings can affect the available memory for the virtual machine, potentially leading to variations in memory-related experiments.
- **Hypervisor Overheads**: The virtualization layer (hypervisor) can introduce overhead in terms of resource management, which may impact the precise control of resources within the virtual machine.
- **Clock Drift**: Virtualized environments can experience clock drift, where the virtual machine's system clock may not be perfectly synchronized with the host machine's clock. This can affect time-related measurements.

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 2.4. 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 6.1.

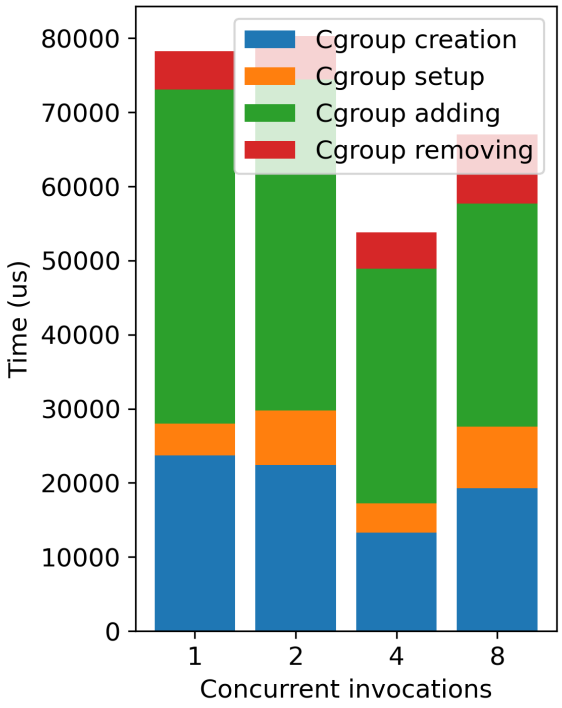


Figure 6.1: CGroup Management Operations Times.

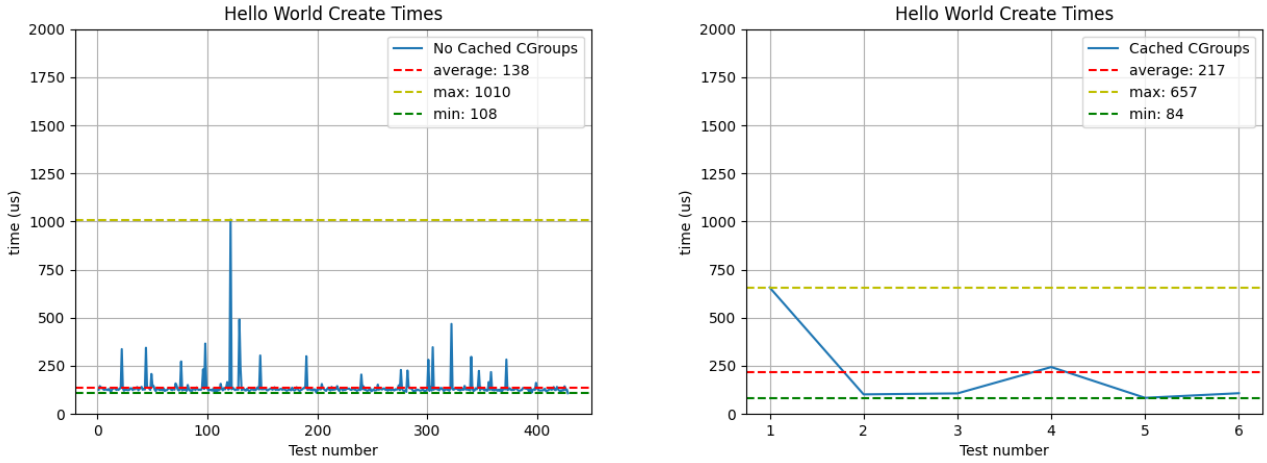
Based on these findings, we can infer that the major cost factors in `cgroup` management operations lie within 'Cgroup adding,' which involves adding a thread to the `cgroup`, and 'Cgroup creation,' which pertains to the creation of the `cgroup` itself. Therefore, we can expect to achieve improved function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

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.



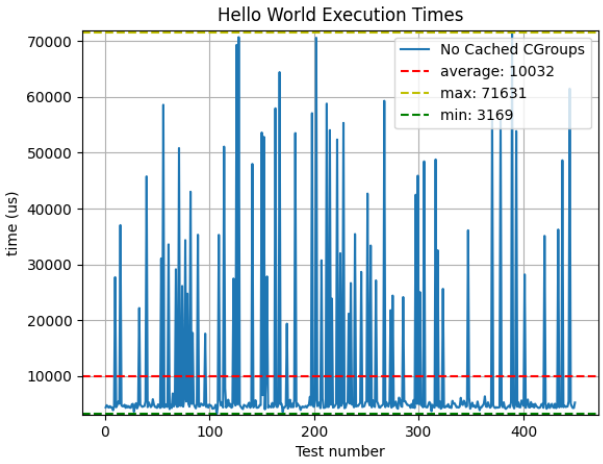
(a) Uncached Version.

(b) Cached Version.

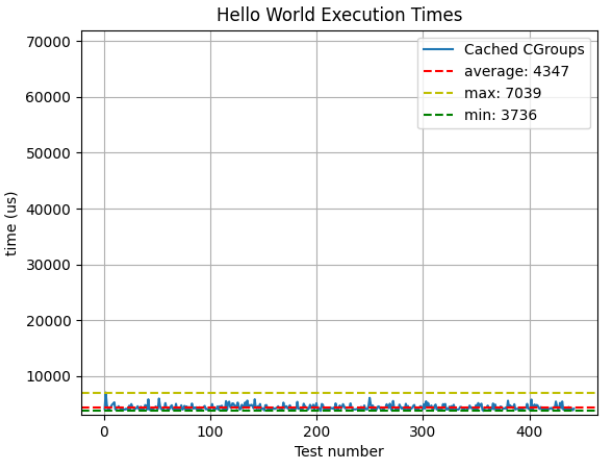
Figure 6.2: Hello World CGroup Creation Times.

The graphs present in Figure 6.2 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 6.3, 6.4, 6.5 and 6.6 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.

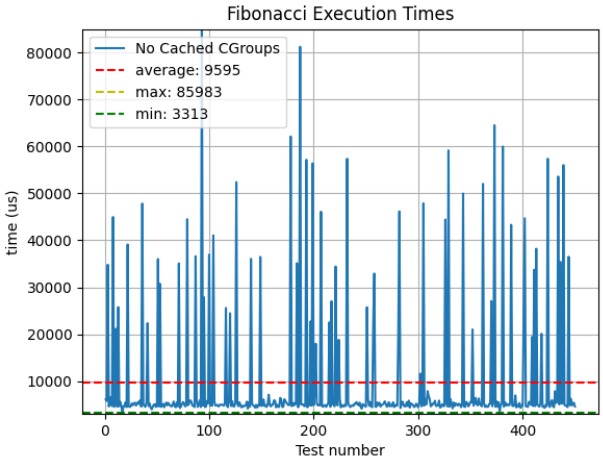


(a) Uncached Version.

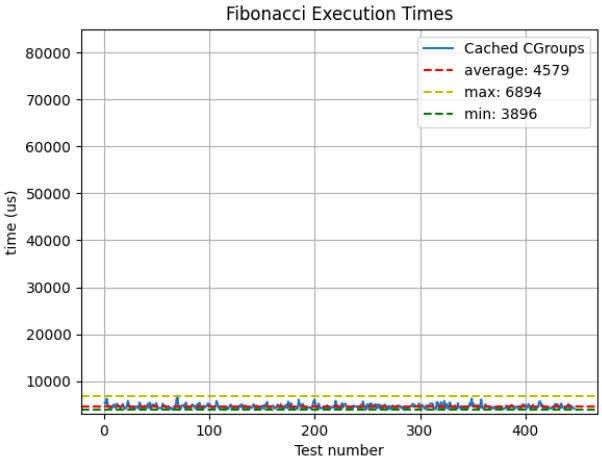


(b) Cached Version.

Figure 6.3: Hello World Execution Times.



(a) Uncached Version.



(b) Cached Version.

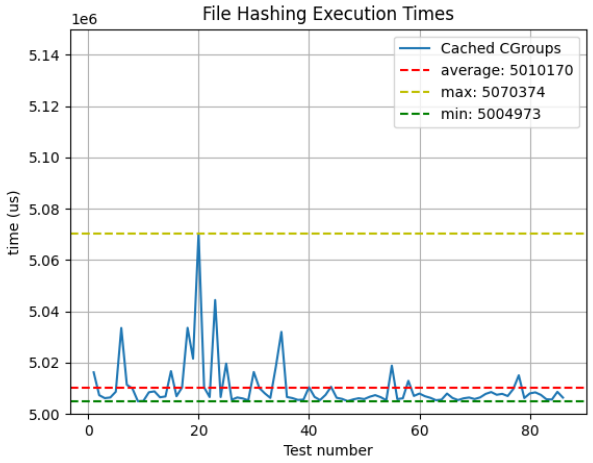
Figure 6.4: Fibonacci Execution Times.

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 6.3 and 6.4, 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

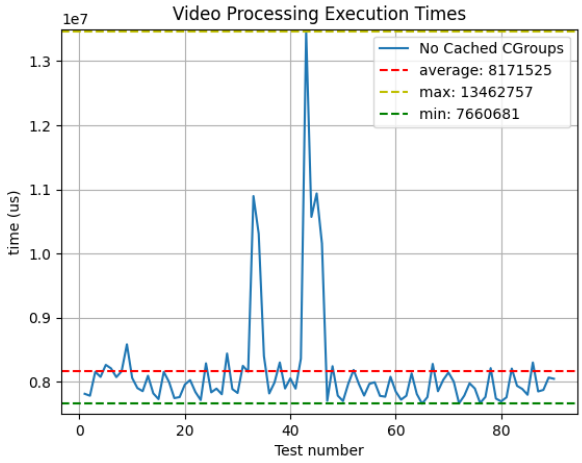
'Video Processing,' as depicted in Figures 6.5 and 6.6, the obtained benefits of bypassing `cgroup` creation and deletion are notably less apparent, culminating in a relatively minor speed increase of approximately 1.5% on average.

(a) Uncached Version.

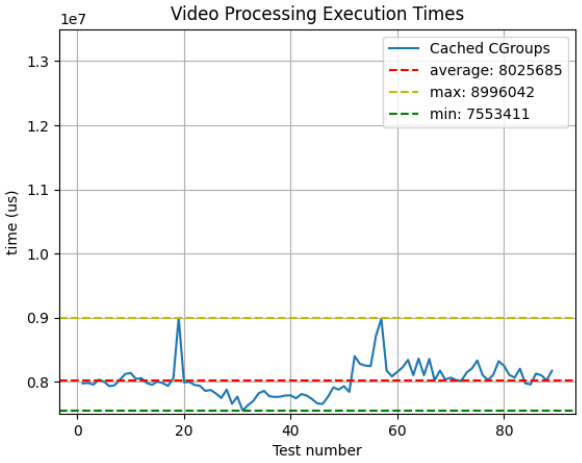


(b) Cached Version.

Figure 6.5: File Hashing Execution Times.



(a) Uncached Version.



(b) Cached Version.

Figure 6.6: Video Processing Execution Times.

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.

6.6 Non-Lazy Reclamation Results

Figures 6.7, 6.8, 6.9 and 6.10 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.

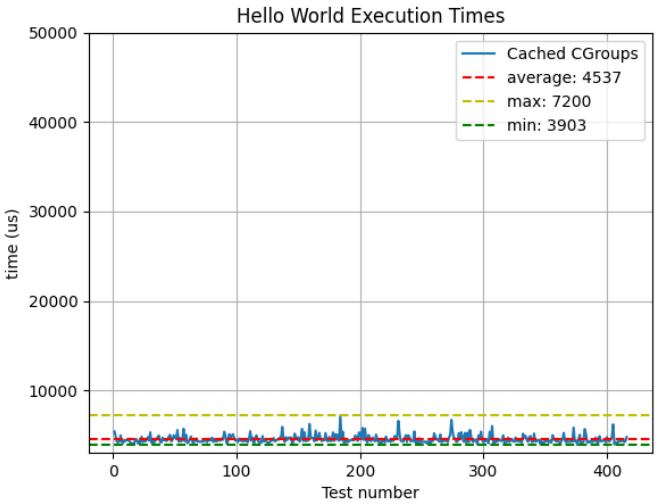


Figure 6.7: Hello World Execution Times.

In this experimental scenario, we can meticulously observe, as evidenced in Figure 6.7, that the performance of the 'Hello World' function exhibited only marginal alterations when the lazy cache reclamation was omitted from the equation.

In contrast, when scrutinizing the execution times of the remaining functions, specifically 'Fibonacci', 'File Hashing', and 'Video Processing', as illustrated in Figures 6.8, 6.9, and 6.10, 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

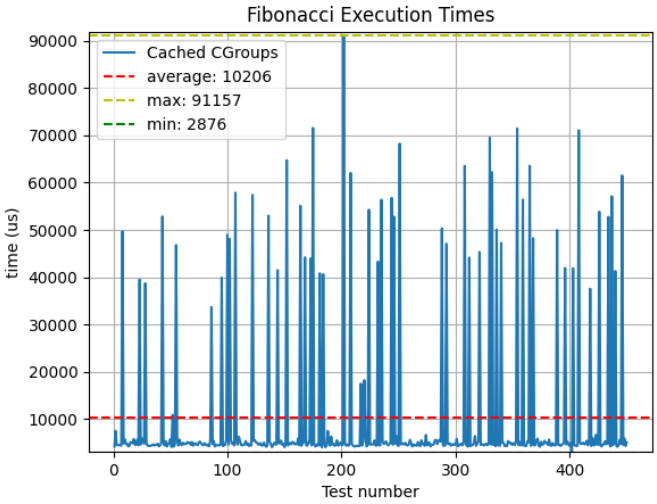


Figure 6.8: Fibonacci Execution Times.

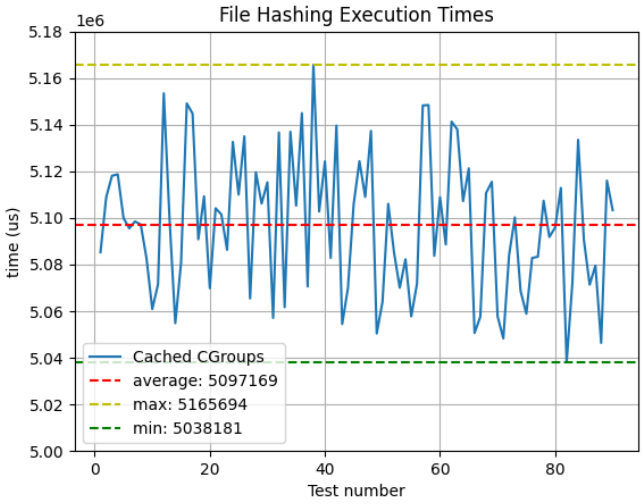


Figure 6.9: File Hashing Execution Times.

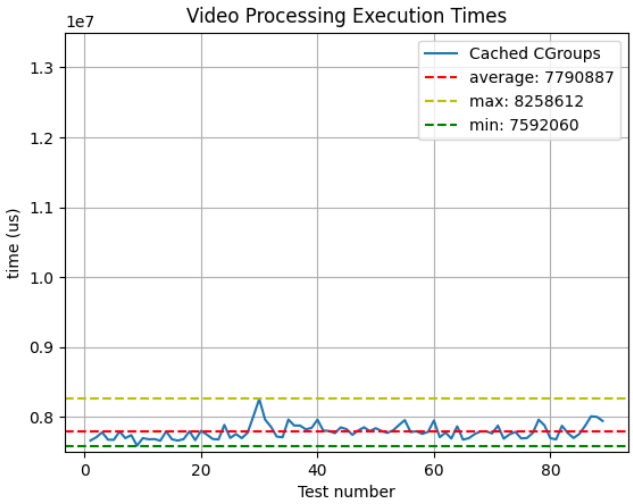


Figure 6.10: Video Processing Execution Times.

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 deffered 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

Contents

In this thesis, 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 thesis was to enhance the scalability and performance of serverless infrastructures, specifically in the context of cgroup management.

7.1 Key Findings and Contributions

Our exploration of the landscape commenced with an in-depth examination of serverless computing in Chapter 2. We dissected the intricate web of cloud computing and its evolution into serverless architectures, shedding light on the pivotal role of Function-as-a-service (FaaS) platforms and the compelling benefits they offer in terms of scalability, elasticity, and cost-efficiency. Additionally, we explored the underlying mechanisms and concepts of serverless computing, such as event-driven architectures and the fundamental tenets of statelessness.

In Chapter 3, we embarked on a comprehensive review of related work in the field. We discovered the abundance of research focusing on serverless computing, its challenges, and the numerous optimization strategies proposed. Notably, our investigation highlighted the growing interest in resource management, with a particular emphasis on cgroups, making it clear that our research contributes to an active and evolving domain.

Chapter 4 introduced the architecture of our Graalvisor-integrated solution, which aimed to mitigate the performance bottlenecks associated with cgroup management operations in serverless environments. We provided a detailed account of how this solution works, its design principles, and the integration of a cgroup cache to improve resource management in FaaS platforms. This architectural blueprint served as the foundation for our subsequent implementation.

Our journey took a significant leap forward in Chapter 5, where we unveiled the intricate details of our Graalvisor-integrated solution's implementation. We elucidated our unique approach to resource management in FaaS by integrating a cgroup cache with a focus on a lazy caching strategy. This innovation aimed to optimize resource usage, minimize overhead, and enhance the efficiency of serverless functions executing within cgroups.

Chapter 6 marked a pivotal moment in our research as we embarked on a rigorous evaluation of our cgroup caching solution integrated into Graalvisor. Here, we provided a comprehensive analysis of the environment, workload, and metrics used to gauge the performance of Graalvisor with our cgroup cache integration. The experimental results showcased the advantages of a cgroup cache, particularly the impact of a lazy caching approach. Our findings demonstrated significant improvements in function execution times, with the most pronounced gains observed in functions with shorter execution times. These results underscore the potential of cgroup caching to enhance performance and resource

efficiency in serverless systems.

7.2 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.

5. **Security and Isolation**: As serverless infrastructure adoption grows, ensuring security and isolation within `cgroup` caching becomes paramount. Future work should explore techniques for enhancing security and ensuring that cached `cgroups` are appropriately isolated.

7.3 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.

Bibliography

[1] R. Pellegrini, I. Ivkic, and M. Tauber, "Towards a security-aware benchmarking framework for function-as-a-service," 01 2018, pp. 666–669.

[2] S. Singh and I. Chana, "A survey on resource scheduling in cloud computing: Issues and challenges," *Journal of Grid Computing*, vol. 14, no. 2, pp. 217–264, Jun 2016.

[3] "Java Virtual Machine." [Online]. Available: https://en.wikipedia.org/wiki/Java_virtual_machine

[4] K. Djemame, M. Parker, and D. Datsev, "Open-source serverless architectures: an evaluation of apache openwhisk," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, 2020, pp. 329–335.

[5] V. Dukic, R. Bruno, A. Singla, and G. Alonso, "Photons: lambdas on a diet," *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020.

[6] F. Sousa, "Performance isolation in graalvm native image isolates," Master Thesis, Instituto Superior Técnico, 2022. [Online]. Available: https://rodrigo-bruno.github.io/mentoring/81120-Filipe-Sousa-dissertacao.pdf

[7] R. Bruno, S. Ivanenko, S. Wang, J. Stevanovic, and V. Jovanovic, "Graalvisor: Virtualized polyglot runtime for serverless applications," 2022.

[8] P. Mell, T. Grance *et al.*, "The nist definition of cloud computing," 2011. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf

[9] N. Kratzke, "A brief history of cloud application architectures," *Applied Sciences*, vol. 8, no. 8, 2018. [Online]. Available: https://www.mdpi.com/2076-3417/8/8/1368

[10] J. Surbiryala and C. Rong, "Cloud computing: History and overview," in *2019 IEEE Cloud Summit*, 2019, pp. 1–7.

[11] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, and M. A. Babar, "Understanding and addressing quality attributes of microservices architecture: A systematic

literature review," *Information and Software Technology*, vol. 131, p. 106449, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920301993

[12] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, *Serverless Computing: Current Trends and Open Problems*. Springer Singapore, 2017, pp. 1–20.

[13] E. Meriam and N. Tabbane, "A survey on cloud computing scheduling algorithms," in *2016 Global Summit on Computer Information Technology (GSCIT)*, 2016, pp. 42–47.

[14] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 55–67, 1994.

[15] "RedHat - Resource Management Guide." [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/index

[16] "The Java Virtual Machine specification." [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se19/jvms19.pdf

[17] "GraalVM wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/GraalVM

[18] "GraalVM architecture." [Online]. Available: https://www.graalvm.org/22.0/docs/introduction/#graalvm-architecture

[19] 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 *USENIX Annual Technical Conference*, 2018. [Online]. Available: https://www.usenix.org/system/files/conference/atc18/atc18-akkus.pdf

[20] A. Y. Mahgoub, K. Shankar, S. Mitra, A. Klimovic, S. Chaterji, and S. Bagchi, "Sonic: Application-aware data passing for chained serverless applications," in *USENIX Annual Technical Conference*, 2021. [Online]. Available: https://www.usenix.org/system/files/atc21-mahgoub.pdf

[21] G. Forney, "The viterbi algorithm," *Proceedings of the IEEE*, vol. 61, no. 3, pp. 268–278, 1973.

[22] G. Czajkowski and L. Daynès, "Multitasking without compromise: A virtual machine evolution," *SIGPLAN Not.*, vol. 47, no. 4a, p. 60–73, mar 2012. [Online]. Available: https://doi.org/10.1145/2442776.2442785

[23] S. Wang, "Thin serverless functions with graalvm native image," Master Thesis, ETH Zurich, Zurich, 2021-04-22.

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

[25] E. Oakes, L. Yang, D. Zhou, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Sock: Rapid task provisioning with serverless-optimized containers," in *USENIX Annual Technical Conference*, 2018. [Online]. Available: https://www.usenix.org/system/files/conference/atc18/atc18-oakes.pdf

[26] J. Thalheim, P. Bhatotia, P. Fonseca, and B. Kasikci, "Cntr: Lightweight os containers," in *USENIX Annual Technical Conference*, 2018. [Online]. Available: https://www.usenix.org/system/files/conference/atc18/atc18-thalheim.pdf