Java File System Virtualization

Gonçalo da Silva Duarte

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Rodrigo Fraga Barcelos Paulus Bruno

Examination Committee

Chairperson: TBD
Supervisor: Prof. Rodrigo Fraga Barcelos Paulus Bruno
Member of the Committee: TBD

October 2024

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 would like to thank my advisor, Prof. Rodrigo Bruno for his patience, availability and guidance throughout this thesis.

Additionally, I would like to thank my parents and my sister for their endless support throughout my academic journey. Working on my thesis while working abroad was a significant challenge, and I couldn't have done it without them.

Finally, I want to thank all my friends for their friendship and unwavering support during good and bad times.

Abstract

As Cloud Computing moves toward Function-as-a-Service, applications are deployed as small, stateless functions running inside a container. Typically, this requires a new container runtime for concurrent invocations, however, runtime isolation imposes great latency and memory costs.

One solution to address this is enabling functions to share the same runtime, reducing the number of cold starts, and, improving the memory footprint. However, in order to completely share the runtime, the File System must be virtualized to ensure that multiple invocations running at the same time don't interfere with each other's executions.

Thus, we propose a solution to this problem by leveraging Seccomp to intercept File System related calls and modify them in order to ensure isolation between the different function executions.

In order to evaluate the implemented solution, we used multiple workloads with different degrees of File System usage to assess how File System Virtualization affects different workloads. The evaluation focuses on latency to assess whether we can introduce the File System Virtualization mechanism without impacting the performance of the system. In addition to this, we measured memory utilization to determine if the proposed solution impacts memory footprint.

Keywords

Function-as-a-Service; Runtime Sharing; Seccomp; File System Virtualization;

Resumo

À medida que Cloud Computing se aproxima de *Function-as-a-Service*, as aplicações são colocadas em produção como pequenas funções sem estado a correr dentro de um container. Tipicamente, isto requer um novo runtime para cada container, contudo, isolamento ao nível do runtime implica grandes custos em termos de latência e utilização de memória.

Uma forma de fazer frente a este problema é permitir que as funções partilhem o mesmo runtime, reduzindo o número de cold starts, e reduzindo a utilização de memória. Contudo, para partilhar completamente o runtime, há que virtualizar o sistema de ficheiros, de forma a garantir que funções concorrentes não interferem umas com as outras, corrompendo o estado do sistema de ficheiros.

Desta forma, propomos uma solução para este problema ao utilizar Seccomp para intercetar chamadas de sistema que interfiram com o sistema de ficheiros e modificá-las para garantir isolamento entre funções concorrentes.

Para avaliar a solução proposta, usamos várias workloads com diferentes graus de utilização do sisema de ficheiros para perceber como a introdução de virtualização do sistema de ficheiros afeta diferentes funções. A avaliação foca-se na latência por forma a avaliar se a introdução do mecanismo de virtualização impacta o desempenho do sistema de forma significativa. Adicionalmente, medimos a utilização de memória para avaliar o impacto da solução no consumo de memória do sistema.

Palavras Chave

Function-as-a-Service; Partilha de *Runtime*; *Seccomp*; Virtualização de sistema de ficheiros;

Contents

1 Introduction **1**

1.1 Context 1

1.2 Problem 2

1.3 Goal 3

1.4 Proposed Solution 4

1.5 Document Outline 4

2 Background **5**

2.1 Cloud Computing 6

2.1.1 Infrastructure-as-a-Service 6

2.1.2 Platform-as-a-Service 6

2.1.3 Serverless and FaaS 7

2.2 Virtualization Technology 8

2.2.1 Virtual Machines and Containers [1] [2] 8

2.2.2 Process-level virtualization(fine-grained virtualization) 9

2.2.3 Syscall interception 10

2.2.3.A ptrace 12

2.2.3.B Seccomp 12

2.3 Summary 13

3 Related Work **15**

3.1 Photons 16

3.2 FAASM [3] 16

3.3 GroundHog: Efficient Request Isolation in FaaS [4] 17

3.4 Pushing serverless to the Edge with WebAssembly runtimes [5] 18

3.5 RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing [6] 18

3.6 Container Hardening Through Automated Seccomp Profiling [7] 19

3.7 Discussion 19

4 Solution Architecture 21
4.1 System Architecture . . . 22
4.2 File System Isolation . . . 23
4.3 Performance Discussion . . . 24
4.4 Seccomp Interface . . . 24
4.4.1 Requirements . . . 25
4.4.2 Solution Design . . . 25
4.5 System call interception and emulation . . . 26
4.5.1 Requirements . . . 27
4.5.2 Implementation . . . 27
4.6 Local Copy Mechanism . . . 27
4.6.1 Requirements . . . 27
4.6.2 Solution Design . . . 28
4.7 Post execution Cleanup . . . 29
5 Evaluation 31
5.1 Evaluation Plan . . . 31
5.2 Evaluation Hypotheses and Expected Outcomes . . . 32
5.3 Evaluation environment . . . 33
5.4 Experiments . . . 33
5.5 Seccomp installation overhead . . . 33
5.6 Benchmarks . . . 34
5.6.1 Hello World . . . 34
5.6.2 Read Only . . . 36
5.6.3 Multiple Reads and Writes . . . 38
5.6.4 Fibonacci . . . 40
5.6.5 File Hashing . . . 41
5.6.6 Misc . . . 42
5.7 Memory Consumption . . . 44
6 Conclusion 47
6.1 Conclusions . . . 47
6.2 Future Work . . . 48
Bibliography 49

List of Figures

2.1 Different cloud computing models [8]. 8
2.2 Two isolates in the same process. 10
2.3 Kernel Responsibilities [9]. 11

4.1 Architecture Model for our system. 22

5.1 Average time to execute Hello World across different execution modes. 35
5.2 Average time to execute consecutive Hello World with and without FS Virtualization. . . . 36
5.3 Average time to execute Read Only Benchmark with and without the filter. 37
5.4 Average time to execute multiple read and write operations across execution modes. . . . 38
5.5 Average time to execute Reads and Writes and without FS Isolation. 39
5.6 Average time to execute the Fibonacci benchmark across different modes of execution. . 40
5.7 Average time to execute the Fibonacci benchmark with and without FS Virtualization. . . 41
5.8 Average time to execute the File Hashing benchmark with and without FS Virtualization. . 42
5.9 Average time to execute the Misc benchmark across execution modes. 43
5.10 Average time to execute the Misc benchmark with and without FS Virtualization. 44
5.11 Memory consumption across different execution modes. 45

List of Algorithms

4.1 Seccomp Handler Pseudo-Code 26

1

Introduction

Contents

1.1 Context . . . 1
1.2 Problem . . . 2
1.3 Goal . . . 3
1.4 Proposed Solution . . . 4
1.5 Document Outline . . . 4

1.1 Context

Cloud computing has revolutionized the way enterprises and developers approach computing infrastructure, offering flexibility, scalability, and cost-efficiency. The Serverless model takes Cloud Platforms one step further and relieves developers from infrastructure management, providing auto-scaling applications and bringing computing closer to being a utility with the pay-as-you-use billing model. As Cloud Computing moves towards platforms that offer fine-grained virtualization like Serverless and FaaS, applications are deployed as tiny functions running inside a container or microVM [10]. Since the dawn of the internet economy, internet companies need to handle their own Datacenters and infrastructure

to deal with fluctuations in demand, and it is easy to see how this might lead to inefficiencies when a company provisions for a peak event that vastly surpasses regular usage. Serverless computing addresses this challenge by automatically scaling up during peak traffic and scaling down when demand drops, approaching optimal resource utilization at all times. Typically, Serverless platforms allocate different VM instances for different invocations. However, the start time is frequently a lot higher than the invocations' executing times. Thus, memory footprint and startup latency are the biggest challenges in Serverless. The overhead of running each function in a separate virtualized environment (like a container or microVM) can lead to inefficient use of memory resources. Since Serverless platforms often allocate new VM instances for new invocations, memory consumption can increase quickly, especially in scenarios where functions are invoked frequently or for high-throughput applications. This increase in memory consumption can lead to higher costs and worse performance, particularly when dealing with large-scale or high-performance applications. Thus, cold starts are one of the key issues in Serverless architectures. When a function is invoked in a Serverless environment, the platform often needs to provision a new VM or container instance to execute the function. This provisioning introduces a delay before the function can start executing. According to an analysis of production AWS lambda workloads [11], cold starts can go from under to 100ms to 1 second. If a cold start occurs, it can become a bottleneck for real-time applications or other latency-sensitive systems, such as user interfaces, or financial transactions. In those cases, the startup time can exceed the execution time of the function itself, undermining the performance advantages of the Serverless model. By tackling memory footprint, startup latency, and increasing support for stateful functions Serverless computing can expand its utility beyond simple stateless functions and extend into more complex, real-time, and resource-intensive applications. As cloud providers continue to innovate in these areas, the Serverless model will increasingly become a foundation for scalable, cost-effective, and high-performance computing.

1.2 Problem

With this project, we aim to address some of the most critical challenges faced by Serverless platforms like, memory footprint, startup latency, and support for stateful applications. By allowing multiple function invocations to share the same File System, we seek to significantly reduce both memory usage and cold start latency, thus improving the efficiency of Serverless architectures. In addition to this, File System virtualization is a significant step towards better support for stateful applications, which are currently difficult to handle efficiently in Serverless environments.

Existing virtualization techniques are still very expensive for such fine-granularity and thus, running multiple function invocations inside the same VM/Container still has several limitations, like the lack of external resource virtualization. Broadly, the options for isolating functions on Linux can be separated

into three categories. Containers, in which all functions share a kernel and some combination of kernel mechanisms are used to isolate them. While this ensures that one function cannot interfere with another, it leads to high memory consumption and frequent cold start delays. The second approach is Virtualization, in which functions run in their own VMs under a hypervisor. Finally, language VM isolation, in which the language VM is responsible for isolating functions from each other or from the operating system.

Virtual machines are too expensive in terms of memory and startup time and containers provide an improvement but are still too slow compared to a solution that allows us to run multiple invocations inside a single process within a VM, sharing resources like the File System, and this is what we want to enable in this thesis.

Providers have to handle the trade-off between performance and provisioned resources for their applications. Significant advancements have been made in runtime sharing such as Photons and GraalVM Isolates [12], however, File System Virtualization remains to be addressed. Virtualizing the File System is a key component runtime sharing. In most Serverless platforms, the stateless nature of functions leads to frequent calls to external data sources to retrieve application context. These calls lead to increased latency and can create a significant bottleneck.

By virtualizing the File System, multiple function invocations can access the same resources and share state information within the same VM or container, or even inside the same process. This reduces the dependency on external data sources, reducing the number of network calls and thereby improving the performance of stateful applications. For instance, a function that processes large datasets or libraries stored on disk could benefit greatly from a shared File System, as each invocation would no longer need to independently load the entire context at startup. Instead, they could read and write to a shared File System, reducing context loading and data access times and improving throughput.

In conclusion, enabling File System Virtualization represents a critical step towards Runtime Sharing, improving the efficiency of Serverless platforms by enabling multiple function invocations to share the same File System. This approach has the potential to significantly reduce memory footprint and startup latency, while also improving the performance of stateful functions, addressing one of the key limitations of current Serverless architectures.

1.3 Goal

As a step towards virtualizing Language Runtimes, the File System needs to be virtualized, enabling shared utilization of its resources without losing consistency. This thesis presents a system that provides lightweight File System Virtualization, for multiple function invocations running inside different threads within the same process. It does so by intercepting and patching Native Image methods related to the File System in order to ensure isolation when we have multiple applications running inside the same

language runtime. The goal of this thesis is to present a system that, enables File System Virtualization in Serverless platforms without impacting performance aspects such as memory consumption, request latency, and throughput. In doing this, we take a step towards addressing the inefficiencies in current Serverless platforms and improving support for Stateful functions.

1.4 Proposed Solution

For this project we will build upon previous work done on Runtime Sharing, adding File System Virtualization through the use of Seccomp and a copy-on-write mechanism.

Our approach leverages Seccomp to create a Sandbox for thread applications making modifications to the File System. All the write calls as well as read calls made to previously modified files will be redirected to a thread-specific sandbox cleaned after each function execution.

1.5 Document Outline

Throughout this document we will outline our approach to this problem, in Section 1 we have introduced the problem identified and our goals. In the second section, we will provide context for our problem, and introduce concepts relevant to the problem we intend to solve such as Cloud Computing and its different offerings as well as Virtualization technologies and techniques. In the third section, we will discuss research done into Runtime Sharing and High-Performance Serverless Computing, diving into the similarities with existing projects as well as the challenges that still need to be addressed. We present our Solution Architecture in the fourth section of this document, presenting an overview of our design in the first subsection. Following this, we explain how we will leverage Seccomp in our project. The fifth section describes the evaluation process and it's where we define the metrics we will use to assess whether our project was successful. To wrap up the document we have a final section where we summarize the document and the results as well as, suggest future opportunities for improvements that may arise from our solution.

2

Background

Contents

2.1 Cloud Computing 6

2.2 Virtualization Technology 8

2.3 Summary 13

In this section, we introduce key technologies and concepts related to the problem we aim to address in this thesis, as well as the technical components of the solution. It starts with an overview of Cloud Computing and its different modalities and then presents existing Virtualization technologies like Virtual Machines and Containers as well as Process-level virtualization mechanisms.

2.1 Cloud Computing

Cloud computing describes computing resources and available on-demand allowing for cost reduction and reduced capital expenditures turning computing and infrastructure management into a utility for companies, providing the illusion of infinite computing resources on demand and the ability to pay for the use of computing resources as needed. This business model is built on economies of scale that allow Cloud Providers to operate data centers at a significantly lower price per unit of computing, making it extremely appealing to customers who want to deploy applications fast and without having to handle all the complexity inherent to an Internet Business, as we will see below, different models of Cloud Computing allow the customer to choose how much control or responsibility he will handle or delegate to the Cloud Provider.

2.1.1 Infrastructure-as-a-Service

Infrastructure-as-a-Service [13] allows for increased productivity by enabling engineers to focus on business and application logic as the platforms handle infrastructure and scaling. There are also security advantages due to data centralization. Cloud computing arose out of the demand from companies to expand their computing infrastructure to face the needs of their customers in a world of increasing digitization. It started with Infrastructure-as-a-Service, providing high-level interfaces to abstract details of the underlying infrastructure, typically using a hypervisor that runs the Virtual Machines as guests. The client does not control the underlying physical infrastructure but manages the Operating System, storage, and deployed applications. However, this model does not address significant challenges for the developers such as load balancing, monitoring, and auto-scaling.

2.1.2 Platform-as-a-Service

As the next step in abstraction came Platform as a Service (PaaS) [13] which provided a development environment to application developers where the Cloud Providers deliver a computing platform, including an operating system, programming language executing environment, and a web server. Developers build and run their software directly in the platform instead of managing the computing resources and underlying infrastructure. With this improvement, came the capability to scale resources to match traffic

and adapt to the customer demands. In this model the user only manages the application and the data used by it, everything else is handled by the cloud provider.

With PaaS, computing came closer to a utility, in the sense that the customer only pays for what it uses, leaving the infrastructure management and distribution to the provider. We will see that this notion is taken even further in Software as a Service(SaaS) and Function as a Service (FaaS).

In the SaaS model users access applications and databases whose infrastructure and platform are managed and deployed by the Cloud Providers. Typically the end users access a single entry point, a load balancer that distributes the work over a set of virtual machines.

2.1.3 Serverless and FaaS

In Serverless platforms, developers write functions in high-level languages, pick the events that should trigger their invocation, and let the platform manage everything else. Serverless decouples computation from storage and allows developers to abstract away resource allocation while paying only for useful work performed by the system. Serverless computing has become a mainstream cloud computing model by abstracting away infrastructure management and allowing developers to write functions that auto-scale, while only paying for the used compute time. This is appealing for several reasons, including reduced work in operating servers and managing capacity, auto-scaling as well as being able to configure only the events that trigger the function invocations and data sources providing the argument to those invocations.

The economics and scale of Serverless applications enable workloads from multiple customers to run on the same hardware with minimal cost while preserving strong security and performance isolation. FaaS delivers computing as a utility by providing fine-grained control over the work performed by a given system down to the function invocation, allowing customers to pay strictly for the computational work being done and abstracting away everything else.

In the FaaS model, code is deployed in the unit of functions whose resource management is handled by the cloud provider. Function invocations are triggered in response to events which can be requests, scheduled interruptions or even events to be consumed from a queue. This event-driven invocation can be synchronous or asynchronous. For synchronous invocation, the service that generates the event waits for the response from the function. On asynchronous invocation, the event is queued before being consumed [14]. In this thesis, we aim to ensure isolation between invocations at the File System level while allowing multiple invocations to use the same File System simultaneously and independently.

Currently, Serverless platforms rely on Virtual Machine [1] isolation, however, this can cause a great performance penalty since instance startup time is significant, particularly when compared to function invocation time. There are performance and isolation trade-offs between virtualization with strong isolation and high overhead and container technologies that provide better performance sacrificing isolation. In this thesis, we aim to improve the performance of Serverless Environments while ensuring File System

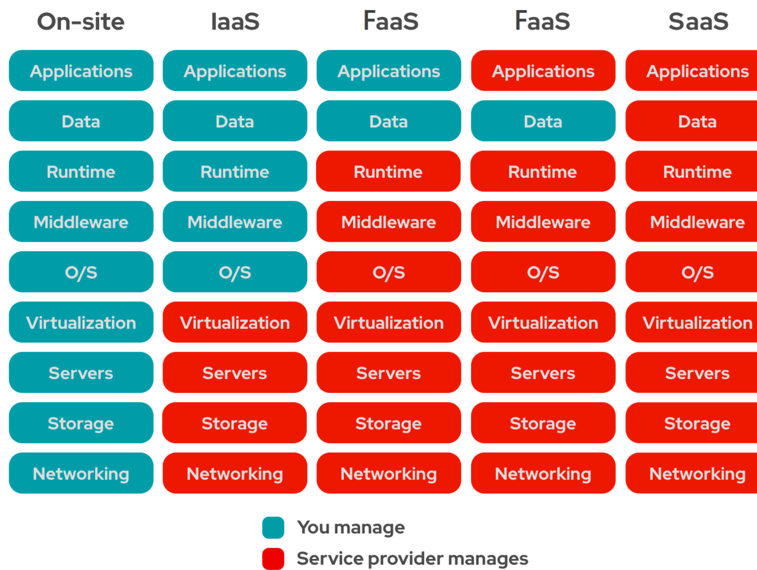

Figure 2.1: Different cloud computing models [8].

Virtualization. In the Function-as-a-Service model, tenants upload functions to be executed by a cloud provider. A function is usually written in a high-level language and accepts input arguments to return the result. The platform exposes endpoints where applications can send requests. This model allows users to create short and stateless functions that can be invoked concurrently to handle large-scale workloads.

2.2 Virtualization Technology

2.2.1 Virtual Machines and Containers [1] [2]

Virtual machines (VMs) are an abstraction of a computer system providing the functionality, of a physical computer. This abstraction enables mapping a single physical server into many virtual servers. Each VM includes a full copy of the operating system as well as necessary dependencies for a given application, even when running on the same host. This technology is leveraged by cloud services to provide virtual computing and storage resources to multiple users, allowing for more cost-efficient, flexible, and, elastic computing. However, this comes at a performance cost since virtual resources are typically less efficient and run slower than a traditional computer.

On the other hand, a container is a standalone package of software that includes the application code and all its dependencies. Containers share the underlying operating system and therefore do not require a different operating system for each application, reducing duplication and increasing efficiency. This portability allows for faster development and deployment at scale.

Containers isolate software from its environment, usually relying on isolation mechanisms built into

the Linux kernel, such as cgroups, that provide process grouping, resource throttling, and accounting, namespaces, which separate Linux kernel resources such as process IDs, and seccomp to control access to syscalls. Multiple containers usually run on isolated partitions of a single Kernel running directly on the hardware while cgroups and namespaces are also used at a higher level to separate the containers from each other. By sharing the operating system, containers eliminate the need for a hypervisor, decreasing the coordination costs aside from the hardware requirements of the solution, since each VM would require a Guest Operating System and the system would require a hypervisor to coordinate the different VMs. Thus, containers have less overhead, boot faster, and make more efficient use of resources, however, they don't provide the same level of isolation.

2.2.2 Process-level virtualization(fine-grained virtualization)

Process-level virtualization uses the resources to give an illusion of multiple processes and provide isolation by leveraging constructs such as namespaces, cgroups to isolate threads from each other, allowing them to share context and resources such as the File System and the Network. This technique enables a shared state without using external storage and with less data redundancy, allowing us to improve significantly on container and VM startup times as well as memory overhead.

Pushing virtualization into the runtime allows for reduced overhead and avoids cold starts. This can be achieved by leveraging Native Image Isolates.

Native Images refers to technologies that enable to ahead-of-time compilation of Java code to a standalone executable, the native image. This executable does not run on the JVM but instead leverages the components of a Substrate VM that provides a runtime that enables faster startup time, and lower memory overhead. GraalVM is an open/source JVM and JDK maintained and developed by OracleLabs, through the Native Image, GraalVM provides a platform to improve the overall performance of Serverless platforms, both in terms of memory footprint and startup time.

An isolate is a disjoint heap, allowing multiple tasks in the same VM to run independently. All isolates then share the same ahead-of-time compiled code and access to the Image Heap, containing the static variables which is particularly important in Serverless platforms since it allows to compile once and run several times and in parallel while aspects such as garbage collection are handled independently. This technology reduces the memory footprint since an isolate can be released when it is not needed, for instance, if a function invocation is finished. Otherwise, if we had multiple tasks running on the same heap it would be in use until the data was cleaned by the garbage collector. Isolates were designed with fast creation and low memory overhead in mind as well as low impact on performance and thus are ideal for Serverless applications while ensuring isolation. This feature is designed for runtime sharing, but we will extend this to enable File System sharing among different invocations.

With isolates, the temporary objects allocated during function invocations taking place in isolates are

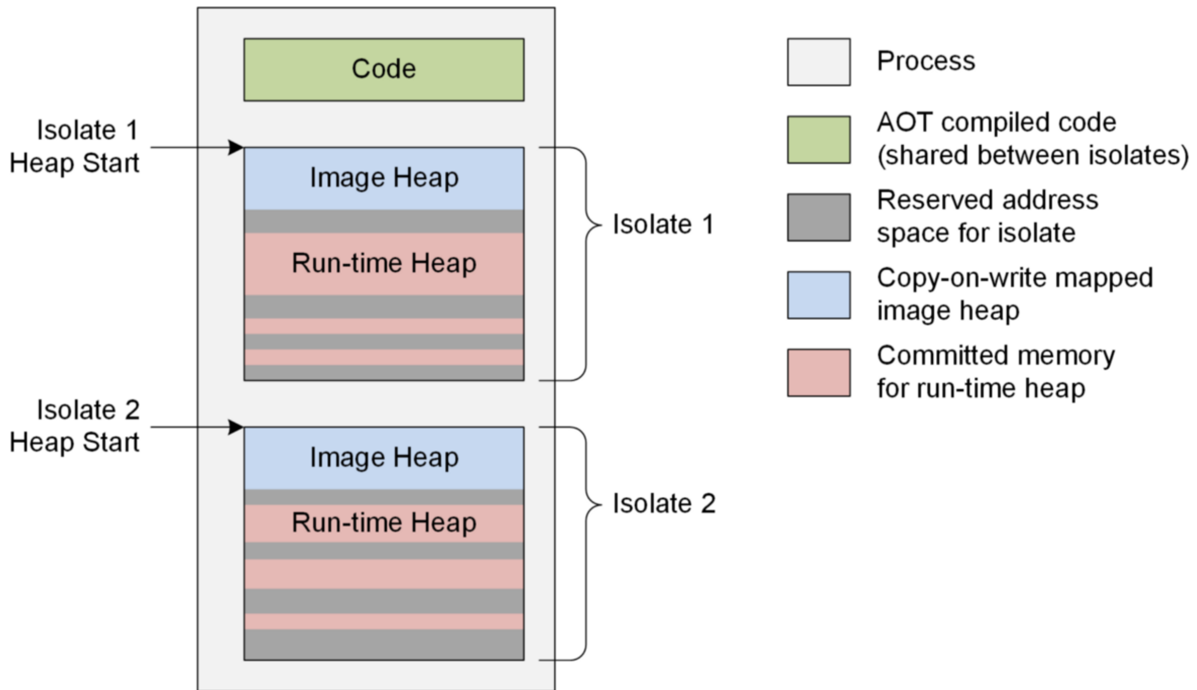

Figure 2.2: Two isolates in the same process.

freed immediately when the isolate is deleted, without the need for garbage collection. This allows for lower memory usage without the overhead introduced by garbage collection. In addition to that, the fact that the isolate's image heap is initialized during image generation allows for further optimization by pre-initializing static variables and objects to be used at runtime.

While memory isolation gives security guarantees in the sense that objects from one isolate cannot be accessed by another, Native Image Isolates do not enforce isolation at the file system level as they execute under the same file system namespace.

A Linux Namespace provides a mechanism for isolating groups of resources within the kernel such that different sets of processes have access to different sets of resources as specified by the programmer. File System namespaces are usually referred to as mount namespaces and are used to isolate mount points such that processes in different namespaces cannot view each other's files. The namespace behaves as if it is at the root of the file system and cannot access other portions of the file system unless they are mounted into the namespace.

2.2.3 Syscall interception

Linux Syscalls represent a well-defined interface between user applications, running outside of the Operating System kernel, and the OS kernel. Applications generally do not directly interact with hardware or networking, instead, they call the kernel through system calls in order to execute those tasks. The

kernel then provides capabilities like process management, memory management, and, input/output calls. In addition to this, this separation of concerns allows the kernel to manage resources for different applications as well as enforce security policies.

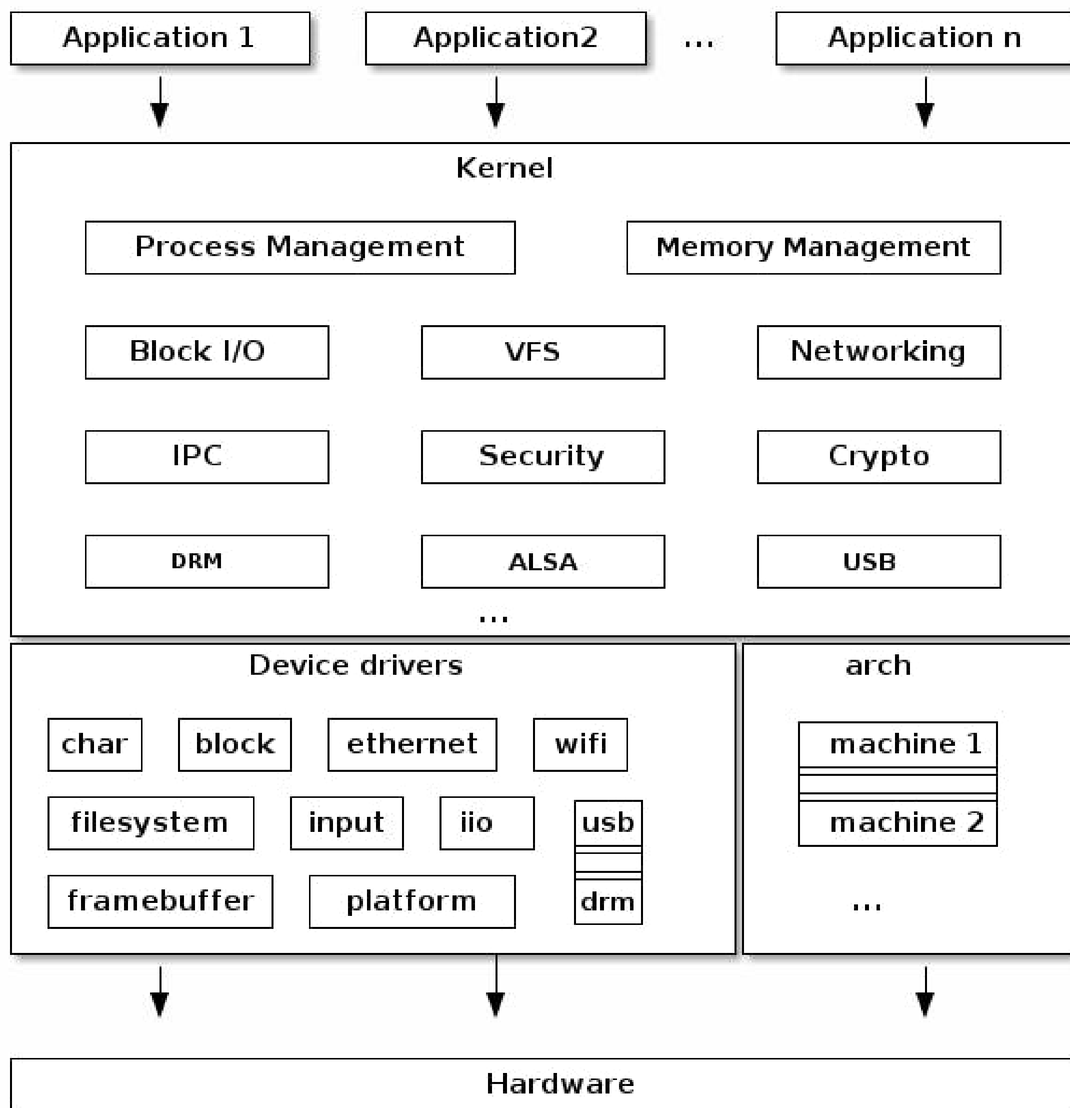

Figure 2.3: Kernel Responsibilities [9].

Syscall interception refers to the ability to monitor, modify, or prevent system calls (syscalls) made by a process. Syscalls are requests from a user-space program to the operating system kernel for low-level services like File System operations, memory allocation and, process management. By intercepting these syscalls, security tools, debuggers, and system monitors can observe or control the behavior of a

process, offering a layer of supervision or restriction.

Some of the most common techniques for syscall interception include tools like `ptrace`, which provides tracing and debugging capabilities, and `Seccomp`, which allows processes to define mechanisms to restrict the permitted syscalls.

System call interception is already a valuable tool for technologies used in Cloud Computing such as containers, which leverage Seccomp for security profiles and container isolation [15]. In this project, we'll leverage Seccomp to provide File System virtualization since we can use it to modify and intercept system calls that directly interact with or modify the File System.

2.2.3.A ptrace

The `ptrace` system call enables the tracer process to monitor and control the execution of another process. This system call is used by gdb to inspect the arguments made to other system calls as well as examine and change the memory and registers of the process or thread being traced. `ptrace` works by attaching a target process to the calling process and intercepting signals and syscalls, these signals can be used to place breakpoints inside the traced process' execution for debugging, or merely for system call tracing.

2.2.3.B Seccomp

Seccomp stands for Secure Computing and allows a process to define a mechanism to restrict the system calls available or intercept all the system calls performed by child processes or threads. This feature can be leveraged to reduce user privileges or to restrict actions within a given container or application. Seccomp is widely used by container runtimes, browsers and, other applications that require isolation or want to reduce the kernel attack surface.

In Strict Mode, Seccomp only allows `exit()`, `read()`, `sigreturn()` and `write()` to open file descriptors, killing the process if any other system calls occur. The strict mode offers a lot of security guarantees, however, its nature prevents it from being useful for more complex programs or systems where we want to provide a different set of privileges to different users or applications, or even multi-threaded applications where we want to isolate the different threads, as is the case in this Thesis.

With seccomp-bpf mode, Seccomp filtering is configured through customizable policies expressed as a BerkeleyPacketFilter(BPF) program, allowing a process to specify the allow-listed or forbidden calls, as well as specifying a default behavior. The Seccomp filter works through notifications, then the process can create a new process or thread(s) to handle the Seccomp notifications. Seccomp-bf provides a number of actions to be taken depending on the result of the process, like killing the process, allowing the system call, or notifying an attached tracer via `SECCOMP_RET_ALLOW`.

To interact with the filter, the process needs to use the `SECCOMP_FILTER_FLAG_NEW_LISTENER` as an argument to the seccomp() syscall: `seccomp(SECCOMP_SET_MODE_FILTER, SECCOMP_FILTER_FLAG_NEW_LISTENER, &prog);` , on success this will return the file descriptor we can use to poll the seccomp notifications. In order to process the notifications we will use the structures shown below.

```
    struct seccomp_notif_sizes {
    __u16 seccomp_notif;
    __u16 seccomp_notif_resp;
    __u16 seccomp_data;
};

struct seccomp_notif {
    __u64 id;
    __u32 pid;
    __u32 flags;
    struct seccomp_data data;
};

struct seccomp_notif_resp {
    __u64 id;
    __s64 val;
    __s32 error;
    __u32 flags;
};
```

These structures will allow us to process the system calls based on the information provided by them, in order to sandbox syscalls made by a given process or thread id, for example.

2.3 Summary

In this section, we went over the key technologies and concepts related to Cloud Computing, Virtualization, and how we can use these technologies to provide File System Virtualization, thus improving the performance of Serverless platforms by improving Runtime Sharing while also extending support for stateful functions by reducing the need for access to external data sources.

3

Related Work

Contents

3.1 Photons 16

3.2 FAASM [3] 16

3.3 GroundHog: Efficient Request Isolation in FaaS [4] 17

3.4 Pushing serverless to the Edge with WebAssembly runtimes [5] 18

3.5 RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing [6] 18

3.6 Container Hardening Through Automated Seccomp Profiling [7] 19

3.7 Discussion 19

In this section, we will dive into relevant research work done on the topic of Runtime Sharing and High-Performance Serverless Computing. As we will demonstrate, there are several points of contact and techniques we can leverage in our project.

3.1 Photons

[16] Photons reduce function startup time and memory footprint by allowing runtime and data sharing across invocations. This reduces memory consumption significantly without performance degradation as well as the number of cold starts. Photons leverage the fact that multiple invocations use the same code and need the same environment, pointing out that data and runtime isolation are particularly inefficient in terms of memory utilization. This happens as a result of the fact that memory is not shared among invocations, and runtime initialization since the runtime needs to be booted for each invocation. This inefficiency has cascading negative impacts: colt starts, lower throughput, higher latency, and higher cost.

In order to address this inefficiency, Photons provide an abstraction that allows the developer to specify and manage data to be shared across the different photons, while the system automatically ensures data separation. A photon is a lightweight function executor that contains the private state of a single invocation while sharing the runtime and common application state with the other photons. All photons within the same execution environment share the same object heap and the application runtime code cache, thus leveraging all the optimized code to achieve faster execution. By enabling runtime and application state sharing among concurrent invocations, Photons are able to reduce function memory consumption by at least 25% per invocation without performance degradation as well as reducing the number of cold starts by 52%.

In this project, we aim to build on this concept by allowing different function invocations to share the same File System. By doing this we will provide a much lighter solution compared to the traditional platforms since the File System does not need to be replicated for each invocation. To provide data separation among multiple executions within the same runtime, writes are performed on a local, invocation-specific copy of the given field, this is what we aim to do, but for the File System and with the use of File System Namespaces. Photons leverage unique identifiers to create a private temporary state, extending this concept to the File System.

3.2 FAASM [3]

Similarly, FAASM also aims to tackle the problem of stateless isolation, which prevents functions from sharing memory. The overhead generated by stateless isolation is particularly taxing for big data processing, one of the main applications of Serverless computing, due to its event-driven nature and easy connection with data sources. As mentioned previously, function isolation results in data duplication and performance as well as memory inefficiencies.

The authors propose an isolation mechanism that enables fine-grained control over memory with a two-tier system, combining a local tier of shared memory and a global tier for cross-host synchroniza-

tion. Sharing resources contradicts the isolation goal, hence, providing shared access in a multi-tenant Serverless environment is a challenge. Thus, FAASM proposes a memory isolation abstraction for a high-performance Serverless platform, isolating the memory of executing functions while allowing memory regions to be shared between functions in the same address space. As stated, the stateless nature of the containers forces any relevant computation state to be stored externally or passed between invocations and, thus the need for solutions that provide a more lightweight isolation mechanism.

Faaslets deliver lightweight isolation by enabling lightweight isolation for CPU and network using Linux cgroups while providing a host interface for file system access. However, we want to virtualize the file system and allow multiple applications to share the same file system. In addition to this, FAASM restores Faaslets from pre-existing snapshots to reduce initialization times. FAASM is able to perform at a high level without compromising isolation by leveraging Faaslets, which provide memory safety while sharing an in-memory state.

3.3 GroundHog: Efficient Request Isolation in FaaS [4]

GroundHog aims to address security concerns that may arise due to container reuse through isolation of sequential invocations for a function by efficiently reverting to a clean slate.

It leverages two properties of typical FaaS platforms: that each container executes at most one function at a time and legitimate functions do not retain state across invocations This enables GroundHog to use snapshots to restore function state between invocations independently from the runtime without introducing significant latency and throughput overhead. Sequential request isolation is critical if a function can be invoked by users with different permission sets since any information leak could have relevant security implications.

This system addresses relevant challenges but has a different aim since this thesis aims to ensure File System isolation for multiple functions running concurrently on the same container. Furthermore, the assumption that each container executes at most one function at a time does not hold for this project since our system is designed to improve the performance of Serverless platforms with multiple functions running on the same process.

Similarly, in this project, we will clean up the thread-local File System context after every function invocation and return the thread execution to the original file system context so every new function will observe the original state of the File System since the File System context will be built as needed for every invocation.

3.4 Pushing serverless to the Edge with WebAssembly runtimes [5]

While the Serverless model is ideal for handling variable workloads, cold-start latency, as well as CPU and memory limitations on hosts make it unable to support latency-critical services. WebAssembly, a portable, binary instruction format for memory-safe execution, makes it possible to build suitable container runtimes that provide great improvements in throughput and memory consumption. Its portability means that a function can be compiled whenever a runtime exists. Wasm, a contraction of WebAssembly, was intended to act as a compilation target for low-level languages such as C or C++ with the goal of enabling the execution of programs in those languages on the web. However, it has been recently considered for use in Serverless as it can be started very fast, and, despite being slower than native code, can be compiled to it.

This paper suggests replacing Docker-based containers with WebAssembly runtimes, enabling significant cold-start latency reduction while increasing throughput. Similarly, we aim to provide a model with finer-grained resource elasticity when compared to the traditional runtimes, by enabling file system sharing and running multiple applications sharing the same file system in a single VM we could improve performance while only requiring extra memory for the local copies of the files each thread is reading or updating.

3.5 RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing [6]

RunD is a lightweight secure runtime that aims to address low deployment density and slow startup performance at high concurrency in secure containers isolated through microVMs. Traditional containers are implemented based on Namespaces and cgroups [17] and thus cannot ensure the same isolation provided by traditional Virtual Machines. The single-container-per-VM secure container model isolates each function invocation at a cost of heavy memory overhead and footprint since each microVM needs to run its exclusive guest operating system.

With this project, we aim to achieve the same isolation level provided by RunD but without creating a new process, using isolates so that a segment of memory is dedicated to a given invocation, enabling a solution that is both faster and lighter in memory footprint when compared to RunD.

3.6 Container Hardening Through Automated Seccomp Profiling [7]

This paper proposes a mechanism to generate custom Seccomp filters for Production-ready systems by capturing all the System calls made by a container during the testing stage of the CI/CD pipelines and generating and deploying custom Seccomp profiles. This solution uses eBPF tracing capabilities to create a hook on the syscall and register it in an output file. Then, it whitelists all the syscalls detected in the Docker Seccomp Profile. By doing this, the paper facilitates incorporating Seccomp filtering within production systems, reducing the attack surface and increasing container Security.

In this thesis, we'll leverage Seccomp to restrict access to the File System and enable isolation for multiple threads executing function invocations within the same process while sharing the same File System.

3.7 Discussion

As we have discussed the systems mentioned above, there are systems that provide lightweight isolation for CPU and Network, but don't enable File System sharing and don't virtualize the File System. In summary, a number of techniques have been analyzed:

- Lightweight isolation for CPU and Network;
- Request isolation in systems where each container only executes one request at a time
- Runtime sharing with data separation among different invocations;
- Isolation among concurrent requests using a single-container-per-VM model and using a different process for each execution;
- Syscall Interception using Seccomp for Container Isolation;

However, none of these projects achieve our goal of virtualizing the file system and allowing it to be shared among concurrent function invocations running on a single process while ensuring isolation between concurrent and consecutive requests.

4

Solution Architecture

Contents

4.1 System Architecture 22
4.2 File System Isolation 23
4.3 Performance Discussion 24
4.4 Seccomp Interface 24
4.5 System call interception and emulation 26
4.6 Local Copy Mechanism 27
4.7 Post execution Cleanup 29

In this section we'll go over the System Architecture, describing how it ensures File System Isolation as well as the key performance aspects of the system and the trade-offs when compared to other alternatives. Next, we describe the Seccomp interface, how we use it to intercept system calls, and how it helps the system provide File System Virtualization. In Section 4.5 we explain how the system does System call interception and how it modifies the File System syscalls to achieve our goals. The last two sections describe the Local Copy mechanism used to store the thread-local File System context and the Post Execution Cleanup mechanism we use to ensure isolation between consecutive function invocations.

4.1 System Architecture

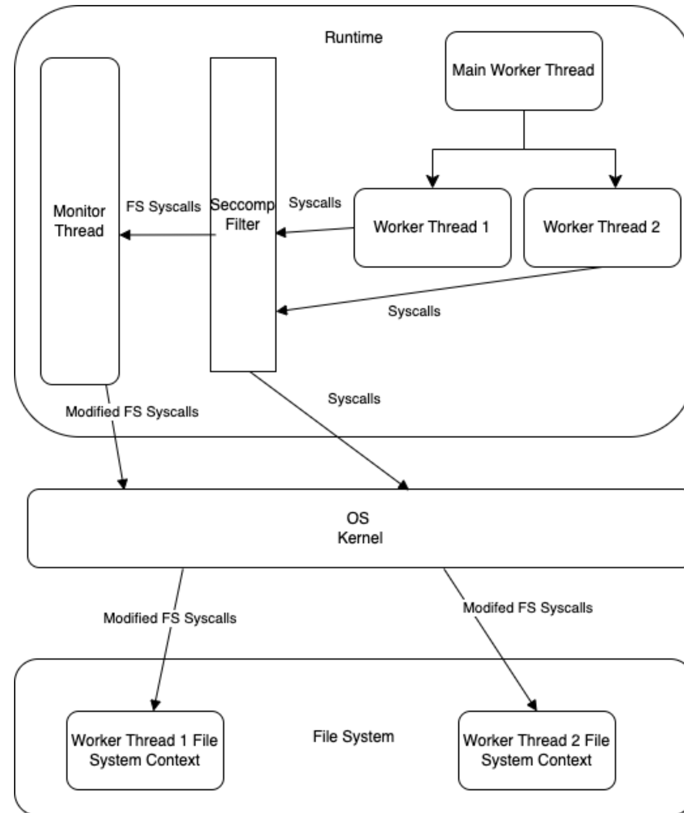

Figure 4.1: Architecture Model for our system.

As shown in Figure 4.1, the entry point to our system is the main worker thread which will be responsible for creating the worker thread pool to handle the benchmark workloads. This thread installs the Seccomp filter and updates a shared structure with the Seccomp file descriptor which will be used to poll the Seccomp notifications.

The system is then divided into two components, the worker threads which execute the benchmarks, and the monitor thread which polls the Seccomp notifications, processing File System related calls to ensure isolation between the concurrent threads. In order to isolate the threads, the monitor thread will intersect all `open()` and `open_at()` and use a copy-on-write mechanism, writing to a thread-local-copy. All subsequent reads to previously modified files will then be redirected to the local copies while reads to previously unaltered files will go through unmodified.

The monitor thread is responsible for enforcing the copy-on-write mechanism, ensuring the File System Syscalls will only modify the File System context of the calling Worker Thread to ensure isolation. If the calls don't depend on or don't affect the File System context, meaning that they are read-only calls to files previously unmodified, they will be allowed to proceed so that they can be executed by the kernel

without being modified. For every other case, they're adapted to ensure isolation, meaning that they will execute on thread-local copies of the files.

To preserve isolation between consecutive function executions in the same thread, at the end of each thread execution, the files created during that execution are deleted and the local copies are deleted.

This architecture supports multiple function executions running in parallel in the same process and sharing the same File System, thus providing a lightweight File System Virtualization mechanism.

4.2 File System Isolation

Without the isolation mechanism, concurrent File System executions could modify the files that other executions would depend on, corrupting the original File System state. If function executions depend on the state, without isolating concurrent and consecutive executions the system wouldn't be deterministic, in that the same series of function executions could result in different final states. Lack of isolation can result in inconsistent or incorrect output, data loss, or system crashes. For instance, one thread may open and modify a file while another thread is simultaneously reading it, leading to unpredictable behavior or a corrupted state that differs from the expected result.

In contrast, a system that provides File System isolation ensures that concurrent and consecutive executions are separated, preventing unintended side effects. Isolation allows for deterministic behavior because each thread operates on its independent copy of the File System, or in this case, local copies of previously modified files. With a copy-on-write mechanism, worker threads can modify their local copies of files without affecting the global File System or other threads' local copies. This guarantees that the system will behave predictably, ensuring that repeated function executions with the same inputs will yield the same outputs.

The benefits of File System isolation are particularly significant in scenarios that require strong consistency and repeatability, such as benchmarking, testing, and sandboxed execution environments. By isolating File System operations, systems can prevent data corruption, ensure that performance metrics are reliable, and maintain security by preventing unintended access to shared resources. Isolation mechanisms play a critical role in Cloud Computing environments ensuring that each execution executes in a clean, consistent, and isolated environment, protecting both system integrity and data accuracy.

Additionally, isolation reduces complexity for developers in handling concurrency, as the need for explicit locks or complex synchronization mechanisms around File System resources is minimized. This helps avoid common issues like deadlocks, race conditions, and contention, improving system stability and performance. By relying on a copy-on-write mechanism, and redirecting the File System syscalls we can avoid concurrency-related performance penalties. The copy-on-write mechanism ensures that local copies are created only for files previously modified by the executing worker thread.

4.3 Performance Discussion

As mentioned previously, without a File System isolation mechanism, a Serverless runtime cannot allow concurrent function executions to share the same File System. Currently, the only readily available mechanism is to create a new subprocess using `clone()`, copy all the data required for the execution to a new File System root created using `chroot` and, execute the function in the root.

This approach has several drawbacks, first, all the File System context required for the function execution needs to be copied before every function execution, including files that only need to be opened for reading. In addition to the performance costs there's a significant disk space usage increase in this approach. Additionally, every function execution requires a new process to be created, while a thread-based approach allows for thread reuse. Thread pools allow for efficient execution of short-lived functions, as threads can be recycled after completing a task. This approach dramatically reduces the startup cost and improves system throughput, making it ideal for the typical Serverless environment where functions execute quickly. On the other hand, creating a new process is more expensive than creating a new thread as it requires setting up a separate memory space. If we take into account that function executions tend to be short and don't usually require a lot of File System-related operations, the overhead becomes even more significant.

The drawback of the approach proposed in this document is that all the File System syscalls go through the monitor thread, however, if we take into account that function executions are usually short and don't primarily rely on File System operations the overhead is not as significant, particularly when compared to when compared with copying the whole File System context. Additionally, this system doesn't modify open calls for reading in files that weren't modified previously, which reduces the number of interventions by the monitor thread, and the overhead introduced. Furthermore, the implementation proposed in this document supports the creation of more than one monitoring thread, which would decrease the latency of File System-related requests and decrease thread waiting time. This could be done by installing more than one filter and portioning the worker threads so that the load is divided by the different monitor threads, each polling from a different Seccomp file descriptor without any additional need for synchronization.

4.4 Seccomp Interface

In this section we'll go over how our system interacts with Seccomp to intercept File System related syscalls to be emulated by the monitor thread.

4.4.1 Requirements

- **Function Isolation** File system-related calls from worker threads are intercepted and redirected to the appropriate copy of the file.
- **Efficient Syscall Interception** Only File System-related syscalls are intercepted, with all the other calls being allowed to proceed. By doing this, we minimize latency impact while maintaining security
- **User-Space Handling** File System-related calls are forwarded to user space to be patched by the monitor thread before being allowed to proceed to the kernel

4.4.2 Solution Design

The system proposed in this document leverages Seccomp to intercept File System related Syscalls, the main worker thread installs the Secommp filter and retrieves its file descriptor for notification polling. This Seccomp filter configuration intercepts File System related calls, sending a notification to the Seccomp file descriptor. The filter allows all the other calls, enabling them to be executed by the kernel without any additional modifications.

The filter is defined and installed as follows:

```
int install_seccomp_filter () {
    struct sock_filter filter[] = {
        BPF_STMT(BPF_LD + BPF_W + BPF_ABS, (offsetof(struct seccomp_data, arch))),
        BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, AUDIT_ARCH_X86_64, 1, 0),
        BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_KILL),
        BPF_STMT(BPF_LD + BPF_W + BPF_ABS, (offsetof(struct seccomp_data, nr))),

        BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, __NR_close,   2, 0),
        BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, __NR_open,    1, 0),
        BPF_JUMP(BPF_JMP + BPF_JEQ + BPF_K, __NR_openat,  0, 1),
        BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_USER_NOTIF),

        // default rule
        BPF_STMT(BPF_RET + BPF_K, SECCOMP_RET_ALLOW),
    };

    struct sock_fprog prog = {
        .len = (unsigned short)(sizeof(filter) / sizeof(filter[0])),
        .filter = filter,
```

```
    };

    if (prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)) {
        perror("error: failed to prctl(NO_NEW_PRIVS)");
        return -1;
    }

    int fd = syscall(SYS_seccomp, SECCOMP_SET_MODE_FILTER, SECCOMP_FILTER_FLAG_NEW_LISTENER, &prog);
    if (fd < 0) {
        perror("error: failed to seccomp(SECCOMP_SET_MODE_FILTER)");
        return -1;
    }

    return fd;
}
```

The monitor thread will then poll the Seccomp FD for notifications, and process them, redirecting open calls that intended to change a file or read from a previously modified file. At a high level, the Seccomp integration works as described in the pseudo-code below:

Algorithm 4.1: Seccomp Handler Pseudo-Code

$seccompFd \leftarrow$ `InstallSeccompFilter())`
while True **do**
`receiveSeccompNotification()`
if isReadOnlyCall() OR fileNotModified() **then**
`executeSyscall()`
else
`createLocalCopy()`
`executeSyscallOnLocalCopy()`

4.5 System call interception and emulation

As mentioned, the system will intercept and modify `open()` and `open_at()` syscalls. Whenever one of these is called, the monitor thread checks if this file is tracked, by acessing the Local Copy Mechanism 4.6 and, if the file will be opened only for reading. If both of these are true, the syscall is executed unmodified. This will reduce the overhead by ignoring calls that have no impact or dependencies in relation to the File System state.

Otherwise, the monitor thread gets the local copy path, modifies the file path in the File System arguments, and, tracks the open file. To track the open file, it checks all the open files for this thread and stores this file path if it isn't present.

4.5.1 Requirements

• **Selective Interception** As guaranteed by the Seccomp Filter, only `open()` and `open_at` syscalls are patched

• **Efficient copy-on-write** To avoid unnecessary duplication, we extract the open call flags and only track the file the call opens the file for writing or if it is reading from a previously modified

4.5.2 Implementation

When the Seccomp handler intercepts `open()` and `open_at()` syscalls, it calls the custom handler for this function. The handling then happens as described below:

1. The system checks if the file is being opened for reading only, if it is and the file wasn't modified previously the syscall proceeds to the kernel unmodified

2. If it is a read call but the file was modified previously, the read call is redirected to the local copy by modifying the pathname in the syscall args.

3. If it is a write call, the system creates a local copy, adds the file to the open file entries and, copies the original content to the local copy so that the thread can then modify the original context

4. The system returns the file descriptor of the accessed file, be it the original file or the thread-local copy

This syscall interception and emulation mechanism enables efficient File System Virtualization, showing only one File System state to the different threads, but allowing them to execute independently while minimizing overhead.

4.6 Local Copy Mechanism

To manage thread-local copies, the monitor thread converts the original pathname into the local copy pathname. To do this, we extract the worker thread ID from the seccomp notification. Then, each tracked file is placed in `/tmp/THREAD_ID/ORIGINAL_PATH`, thus, we are able to retain the original File System structure inside each local copy stored in the `tmp` directory.

4.6.1 Requirements

• **Function Isolation** Each function execution should only access its own File System state so that it can execute correctly without corrupting the original state

• **Efficient Copy-on-Write** To avoid unnecessary duplication, we leverage copy-on-write to ensure a local file copy is only created if the file was modified by the thread

• **File System Virtualization** Each function sees the same File System structure, but underlying access is redirected to local copies by the monitor thread

4.6.2 Solution Design

```
typedef struct {
    int fd;
    char pathname[MAX_PATHNAME];
    char local_copy[MAX_PATHNAME];
} open_file_t;


typedef struct {
    pthread_t thread_id;
    open_file_t open_files[MAX_OPEN_FILES];
    int num_open_files;
} thread_open_files_t;


thread_open_files_t thread_open_files_table[MAX_THREADS];
pthread_mutex_t thread_table_mutex = PTHREAD_MUTEX_INITIALIZER;
```

Files are stored in a `thread_open_files_t` array, where each entry of the array stores all the open files for a given thread within the `thread_open_files_t` structure. This struct stores the thread id, an array with the open files and, the number of open files. Each `open_files_t` contains the file descriptor, the original path and, the path to the thread-local copy.

Every time a new thread is created, a new entry the system creates a new entry in `thread_open_files_table`, mapped to its `thread_id`. The thread will then update this entry throughout the function invocation.

Once the function execution is complete, its entry in the table is cleared, and all the created files in `/tmp/THREAD_ID` are removed.

All modifications to `thread_open_files_table` are synchronized with a mutex to ensure that threads can safely create their entries. No further synchronization is needed since access to the table entries is based on `thread_id` and each function will operate on its own file system context.

4.7 Post execution Cleanup

After each execution function execution, the worker thread iterates over all the modified files, if the file existed previously, it only deletes the reference in the open files table. If the file didn't exist previously, it deletes the file from the thread-local `tmp` directory altogether.

5

Evaluation

Contents

5.1 Evaluation Plan 31
5.2 Evaluation Hypotheses and Expected Outcomes 32
5.3 Evaluation environment 33
5.4 Experiments 33
5.5 Seccomp installation overhead 33
5.6 Benchmarks 34
5.7 Memory Consumption 44

In this section, we'll go over the Evaluation plan, discuss the metrics we'll use to evaluate the system and, describe the benchmarks and how they are relevant to different aspects of the system. After that, we'll discuss the experiment results and draw conclusions from the performance of the system.

5.1 Evaluation Plan

As key performance metrics, we will evaluate latency, application throughput and, memory footprint. We accomplish this by using benchmarks and workloads related to the File System that approximate the

workloads of production systems. We will do so by using benchmarks and workloads based on the ones used by existing systems such as Photons [16].

As proposed in Shahrad et. al [18] we will use workloads based on real traces in order to properly assess the performance of our system. In order to evaluate our system we intend to collect the following metrics:

- Application throughput;
- Execution time per function invocation;
- Seccomp Filter installation time;
- Memory usage;

We will compare the performance with and without File System Virtualization and assess scalability by checking for overhead introduced as we increase the number of threads and executions. More specifically, we aim to measure how fast we can handle File System operations and if the virtualization mechanism has a significant impact on performance.

We intend to benchmark our system against a traditional system that doesn't ensure isolation, and against a virtualization alternative using chroot. The goal of these experiments is to determine if the system is able to introduce File System Virtualization without impacting performance and memory footprint.

5.2 Evaluation Hypotheses and Expected Outcomes

With our evaluation experiments, we aim to validate whether the goals of this project have been achieved by answering the following questions:

- **Is there any significant overhead introduced by System Call Filtering and interception?** We expect our system to introduce minimal overhead, particularly when compared to other isolation mechanisms like chroot. 5.5
- **How will the overhead scale as the number of threads and executions increases?** We expect that the system will scale efficiently and that, as workloads get larger the initial setup costs will be amortized. 5.6.6
- **What is the impact of our system on workloads that don't access the File System?** We expect negligible overhead for workloads that don't access the File System, as the virtualization layer should only affect file system-related operations.

- **How does our system perform on file system-intensive loads that depend on the thread-specific context?** We expect the system to handle file system-intensive operations efficiently, without introducing significant overhead, particularly when measured against chroot. 5.6.2 5.6.2 5.6.3

- **How does File System Virtualization affect the memory utilization of the Runtime** We anticipate that memory usage will increase slightly due to the virtualization layer when compared to non-isolated systems, but will be lower than other isolation alternatives like chroot. 5.7

5.3 Evaluation environment

The execution times were obtained using a Linux machine of the DPSS cluster at INESC-ID Lisbon with a **Intel(R) Xeon(R) CPU E5-2660 v4 @ 2.00GHz** processors with 56 cores. Due to availability constraints, we didn't have exclusive access to the instances at all times so the benchmarks use only 50 threads.

5.4 Experiments

In order to evaluate our system, we'll use three different types of function executions:

- **No Isolation** We'll run the benchmarks on a system without any file isolation to establish a baseline for the latency and overhead introduced.
- **Chroot** In this mode, all function executions will run on a child process executing on a new File System root, fetching all the required File System context at the beginning of the execution.
- **File System Virtualization** In this mode, we'll use the Seccomp Filter and the copy-on-write mechanism to ensure File System Isolation between concurrent and consecutive function executions

With these execution modes, we will be able to compare the performance of our system with a system that provides the same isolation guarantees and compare performance as well as comparing it with a system without any File System isolation to measure the overhead introduced.

5.5 Seccomp installation overhead

In order to assess our solution, we want to understand if adding Seccomp introduces any significant overhead to our system. Seccomp installation only needs to happen once and is done by the main worker thread. After measuring this in our benchmarks, we have observed that the average time to install the Seccomp filter is around 0.2 milliseconds.

5.6 Benchmarks

To assess the performance of the system, we'll use multiple benchmarks:

- **Hello World** This simple benchmark will allow us to measure the overhead introduced by the system for functions that do not interact with the File System. Additionally, this benchmark allows us to isolate the overhead that results purely from the virtualization mechanism.
- **Read Only** A read-only benchmark allows us to measure the performance impact of the virtualization mechanism on the system in workloads where functions only read from the File System.
- **Read and Write to Files** This benchmark allows us to fully test our system and assess the performance introduced by multiple concurrent reads and writes to the same files. The benchmark is a good test of the system's copy-on-write mechanisms, concurrency, and synchronization in file system operations, providing a realistic scenario that mimics the behavior of production systems.
- **Fibonacci** This benchmark allows us to test the performance of our system in a CPU-intensive task.
- **File Hashing** This benchmark allows us to assess the performance of our system in an example where functions read and modify a single file
- **Misc** To simulate a real-world benchmark we'll execute a function that randomly selects one of the previously described functions and executes it. This benchmark provides a holistic view of the system's performance by introducing a mix of operations, such as CPU-intensive tasks, read-heavy workloads, and file modifications. The randomness in function execution mirrors production workloads, providing a more reliable and comprehensive picture of the system's overall performance under diverse conditions.

5.6.1 Hello World

First, we want to understand how the execution time evolves as the number of executions increases to assess the scalability of the function, in figure 5.1 we can see that there's no visible overhead introduced by the system when compared to a simple solution without any File System Isolation. This benchmark shows that the system doesn't affect the parallel execution of simple programs that don't interact with the File System.

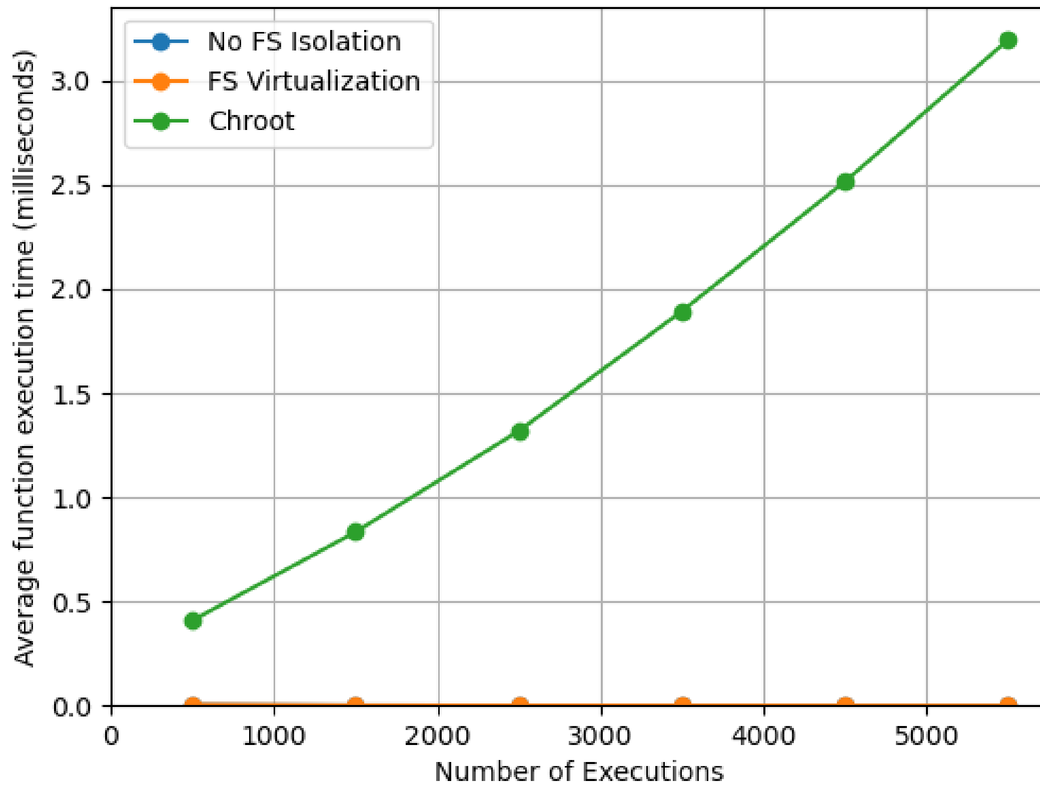

Figure 5.1: Average time to execute Hello World across different execution modes.

As we can see, by looking at figure 5.1 we can't detect any meaningful overhead introduced by our solution. To make sure there's no performance impact we'll evaluate only the base case and our system in figure 5.2. Our File System Virtualization solution vastly performs Chroot, as it needs to create and move to a new root in every invocation, while in the other modes of execution, the threads are reused.

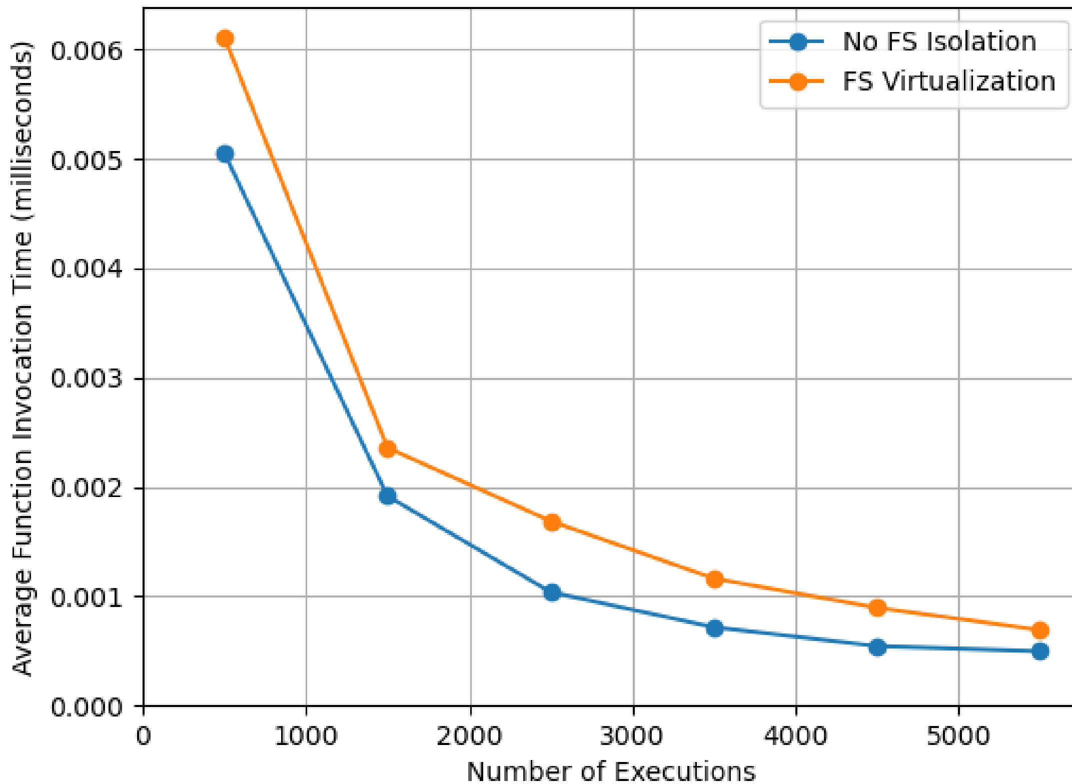

Figure 5.2: Average time to execute consecutive Hello World with and without FS Virtualization.

As we can see, there's no distinguishable overhead between the two executions, and as the number of executions increases the two average execution times converge. The execution time decreases because, unlike with Chroot, the threads are reused and so the initial thread creation overhead is amortized as the number of executions increases. This effect is more significant in Hello world than in other benchmarks due to its shorter execution time.

5.6.2 Read Only

First, we want to understand how the execution time evolves as the number of executions increases to assess the scalability of the FS Virtualization mechanism, in figure 5.3 we can see that the overhead introduced by the function scales well as the executions increase. Similarly to the previous benchmark, the time when using Chroot is significantly larger than in the other modes so the figure only displays the other two so we can compare the performance.

The exclusion of Chroot from the graph allows us to focus on the virtualization mechanism's performance compared to a baseline. Despite executing multiple read calls concurrently with a large number

of threads, the overhead remains low, and we observe that by tuning the number of monitor threads, the overhead can be further reduced. This benchmark demonstrates that the system handles read-heavy workloads efficiently, even under high concurrency.

The program executes 5 read calls so the overhead is between 0.1 and 0.15 milliseconds, even taking into consideration that we're running 50 worker threads and only one monitor thread, by increasing the number of monitor threads we can bring this overhead down significantly without the need introducing any synchronization as the Seccomp notifications would be polled from different file descriptors. Note that this benchmark only performs file system operations so the threads wait for responses from the monitor thread at the same time, mixing file system accesses with other operations as would be expected in a production workload would yield even better results.

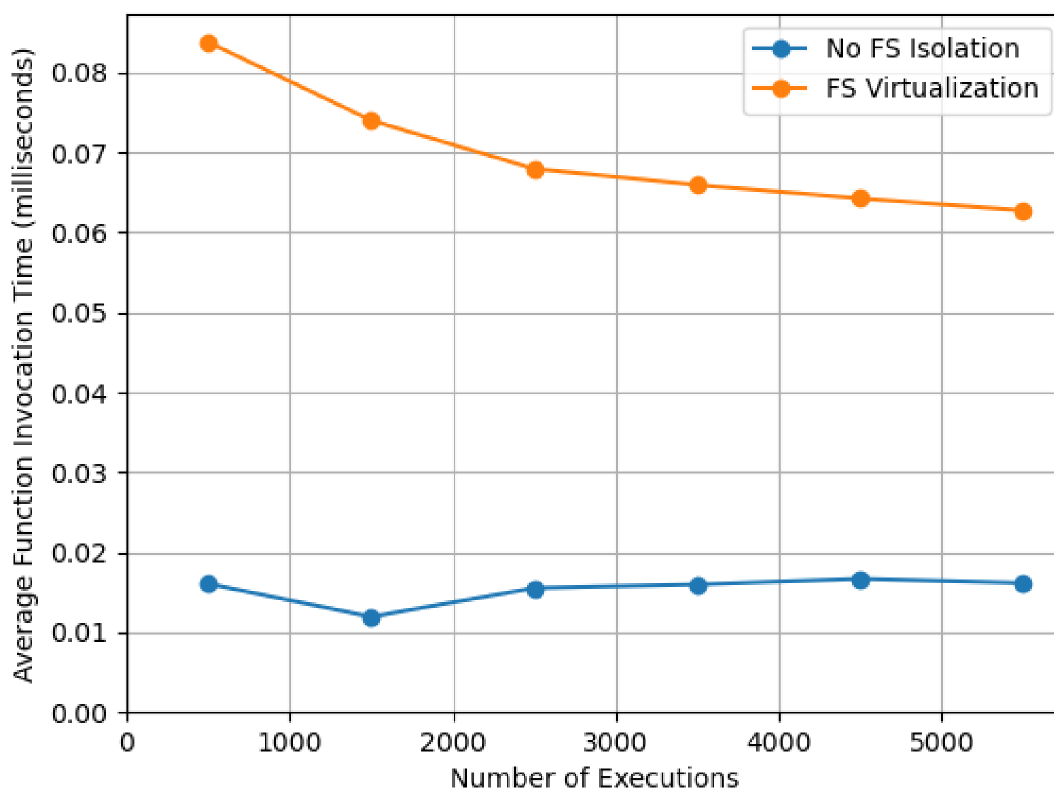

Figure 5.3: Average time to execute Read Only Benchmark with and without the filter.

Figure 5.3, shows a decrease in average execution time for the FS Virtualization mode as the initial setup costs of the virtualization layer get amortized.

5.6.3 Multiple Reads and Writes

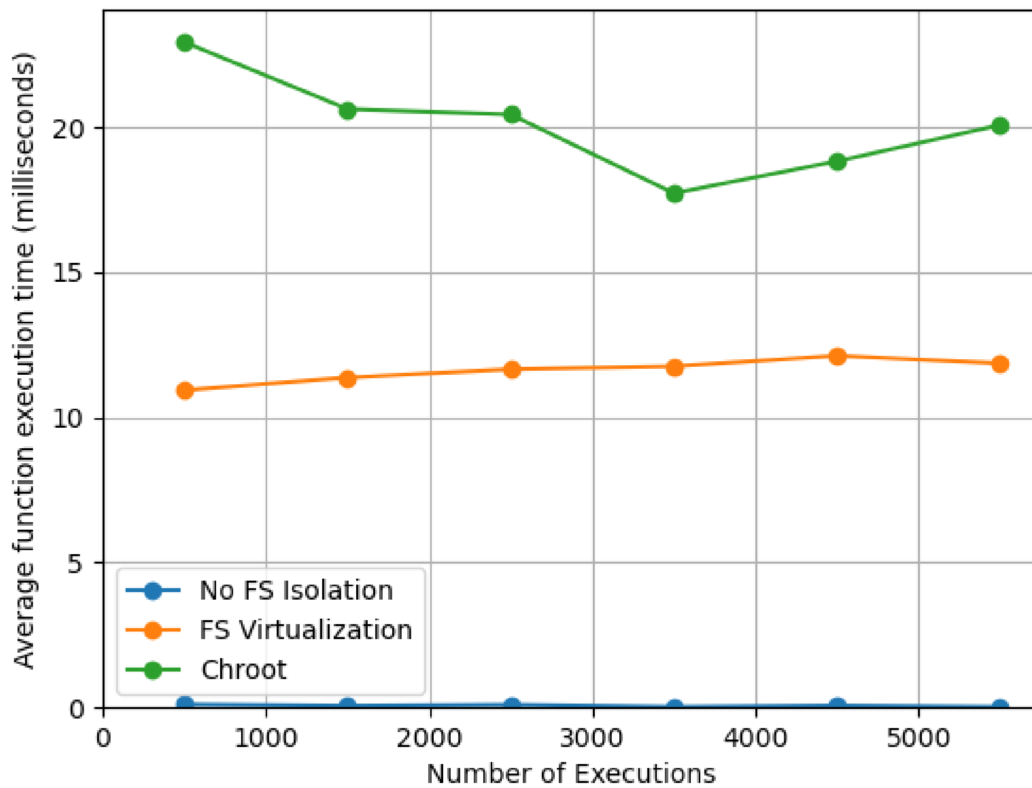

Figure 5.4: Average time to execute multiple read and write operations across execution modes.

We want to understand how the system behaves in the worst-case scenario, a workload composed exclusively of accesses to the File System, where the executing threads modify and read previously modified files, always triggering the copy-on-write mechanism. By analyzing the results in figure 5.4, we can see that the overhead introduced remains stable as the number of executions increases, showing that it scales well while performing better than Chroot, an alternative that provides the same isolation guarantees.

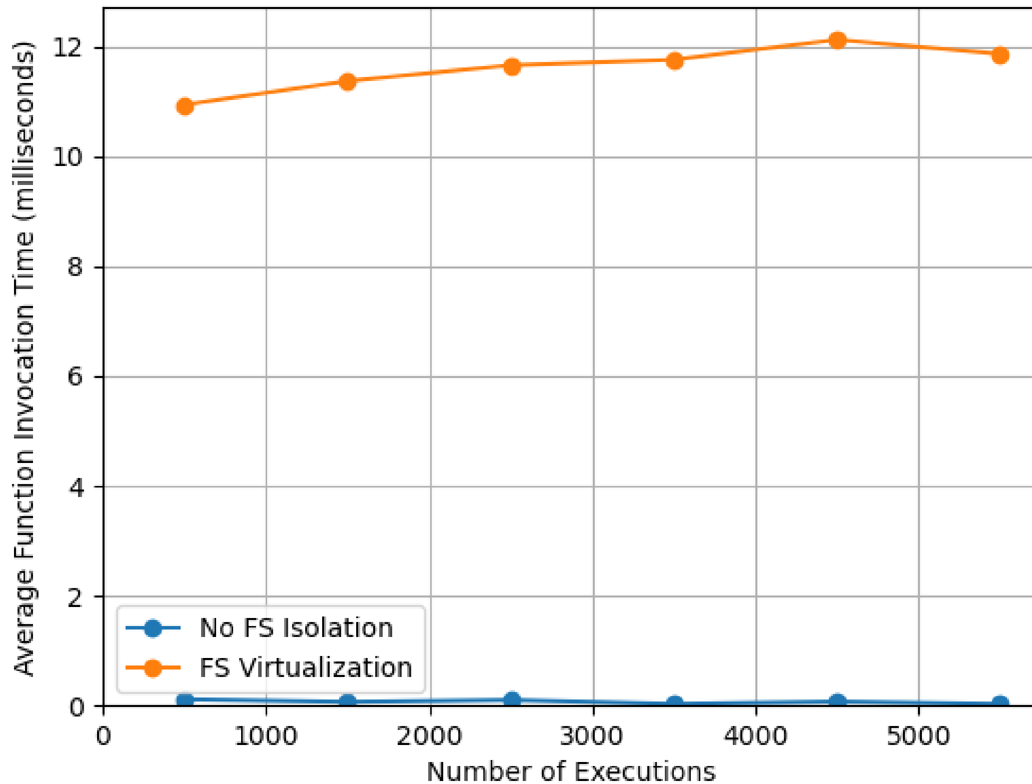

Figure 5.5: Average time to execute Reads and Writes and without FS Isolation.

Given that each function execution performs 13 File System operations, the overhead converges to around 0.92 milliseconds per File System operation. Taking into account that this is the worst-case scenario, where all the functions execute multiple context-dependent file system operations in parallel the overhead introduced is expected, as the waiting time increases as a result of a larger processing time by the monitor thread. The overhead could be significantly reduced by increasing the number of monitor threads, which would result in lower idle time by the threads. Note that the system only intercepts open calls and not read or write calls, which occur at a higher frequency and involve more data. These don't need to be intercepted as they operate based on the file descriptor and the file descriptor will always point to the copy selected by the monitor thread.

Even though the overhead introduced is significant, we expect it to be amortized when we combine multiple types of function invocations, as the monitor thread workload will be spread out across the benchmark time.

The stable overhead across an increasing number of executions highlights the efficiency of the system's copy-on-write mechanism, as it is able to handle concurrent workloads without degrading perfor-

mance.

5.6.4 Fibonacci

This benchmark allows us to understand how the system affects CPU-intensive functions that don't interact with the file system. Each function invocation calculates the first 50 numbers of the Fibonacci sequence and prints the result.

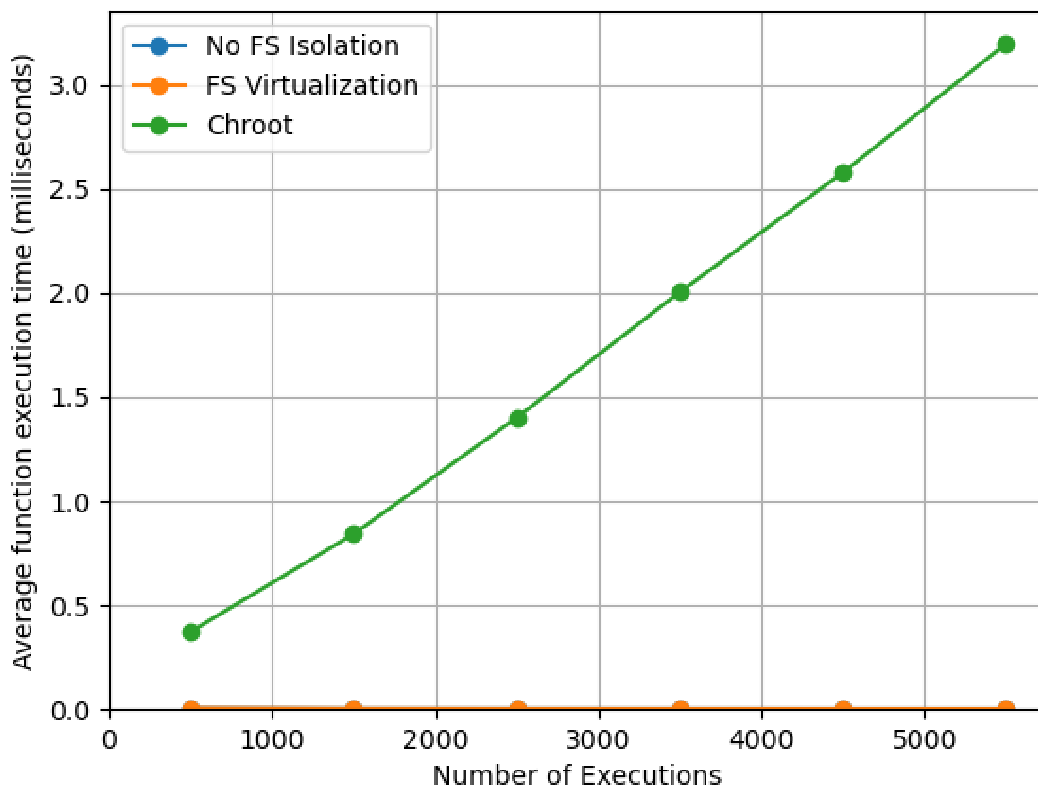

Figure 5.6: Average time to execute the Fibonacci benchmark across different modes of execution.

Similarly to what we observed with other functions 5.6, the system vastly outperforms chroot as the performance penalty incurred from creating and changing to a new root for every invocation significantly affects the execution time.

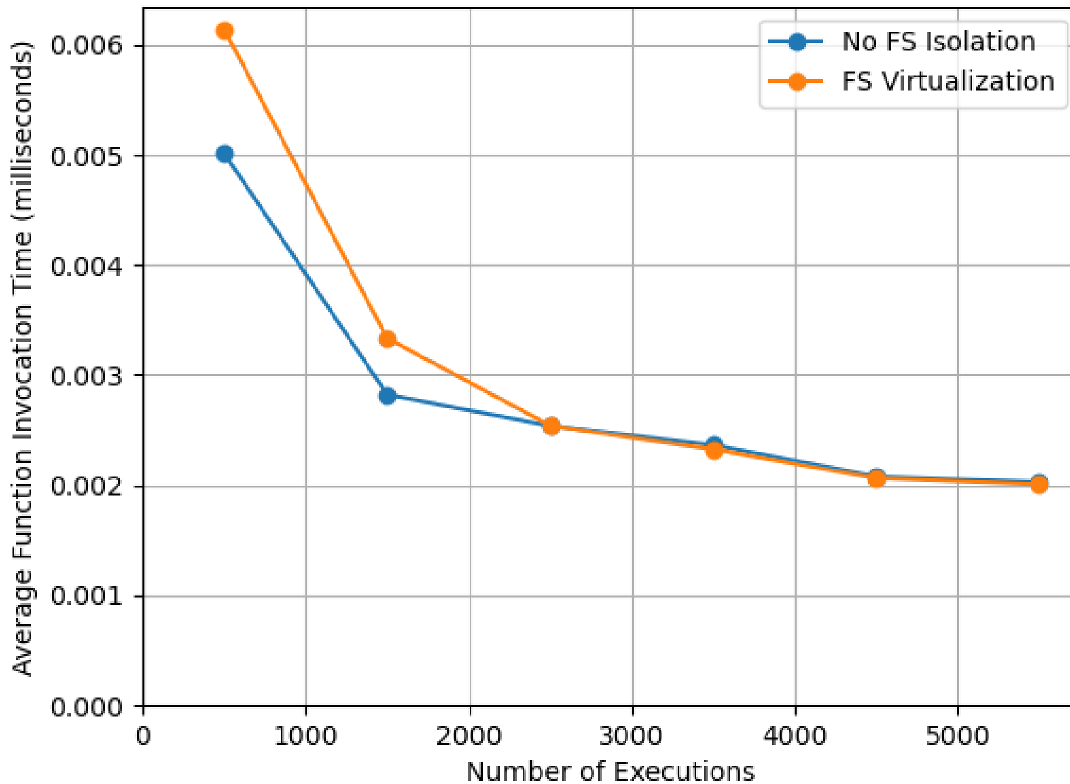

Figure 5.7: Average time to execute the Fibonacci benchmark with and without FS Virtualization.

As can be seen on 5.7, the system doesn't introduce any significant overhead when compared to an alternative without isolation and the execution times quickly converge.

The drop in average execution time is expected and reflects the fact that the threads are reused, so the thread creation time is progressively less significant to the overall execution time.

5.6.5 File Hashing

We'll use this benchmark to test function invocations that access the file system and perform some form of computation. The file hashing function reads the content of a file and replaces it with its SHA-256 hash.

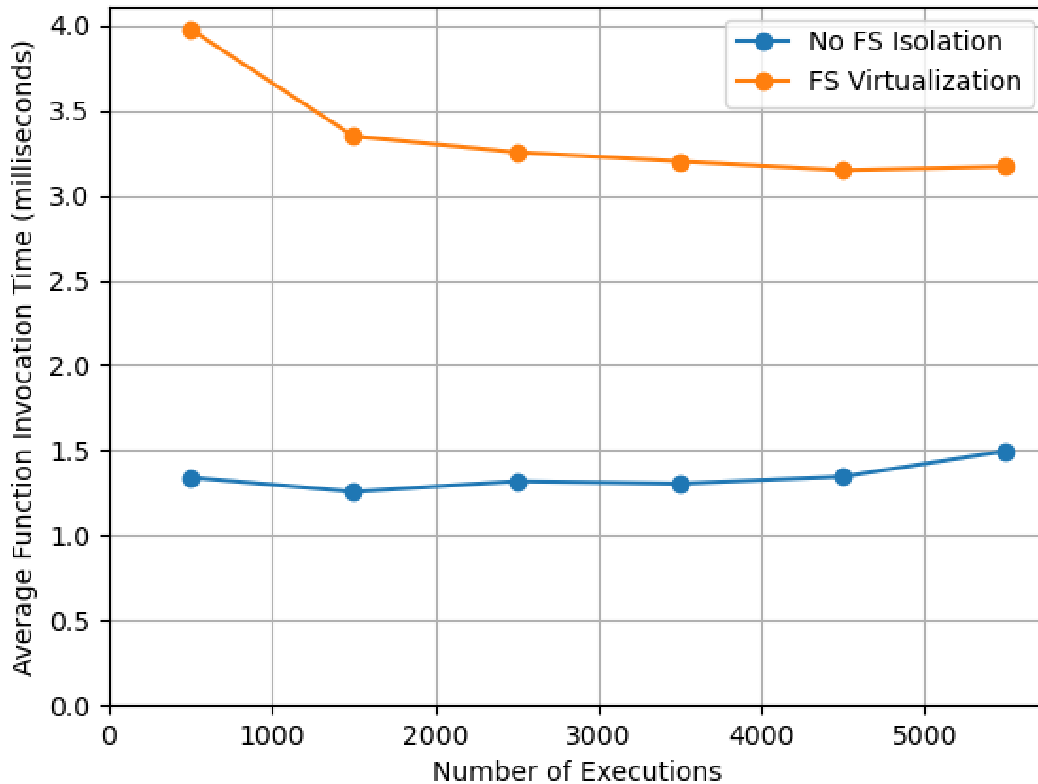

Figure 5.8: Average time to execute the File Hashing benchmark with and without FS Virtualization.

By looking at figure 5.8 we can see that the File Virtualization mechanism scales well and that the latency impact doesn't increase as the number of executions increases. As expected, the system introduces some overhead as the function modifies a file, triggering the copy-on-write mechanism.

The overhead is around 2ms, which can be significant for this function execution but can be explained by the fact that the worker threads have to wait for the monitoring thread to process the `open()` call before they can modify the file. As this benchmark only executes this function, the waiting time increases, causing the overhead we observe in the graph.

Similarly to the previous File System-related functions, we expect the system to be able to mitigate the overhead once we combine multiple types of functions in the next benchmark.

5.6.6 Misc

This benchmark randomly executes one of the functions defined above and aims at replicating a real-world production workload. It offers a more reliable picture of the system's performance as the monitor thread has a more balanced load and is neither free nor overloaded with notifications.

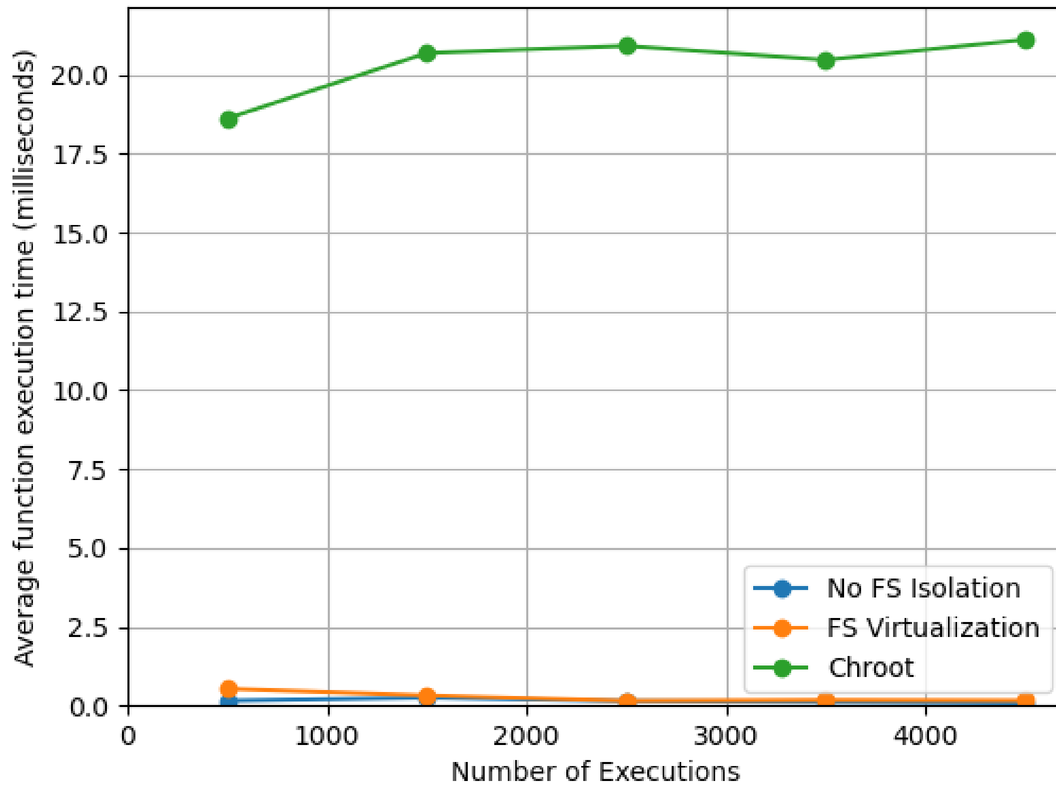

Figure 5.9: Average time to execute the Misc benchmark across execution modes.

Similarly to what we have seen on other benchmarks, figure 5.10 shows that Chroot performs significantly worse than the baseline and the system with File System Virtualization.

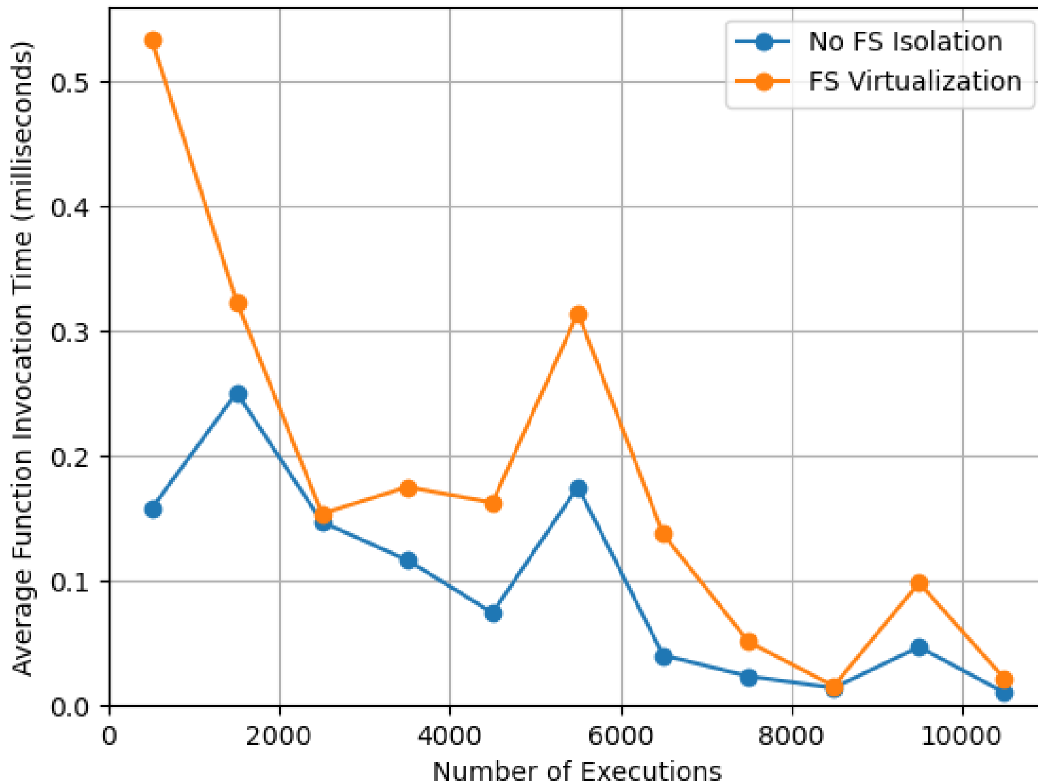

Figure 5.10: Average time to execute the Misc benchmark with and without FS Virtualization.

Figure 5.10 shows that as the number of executions increases, the setup and initialization overhead of the Virtualization system become less significant and overall don't impact the performance of the system, showing that we can introduce FS Virtualization, ensuring isolation without impacting performance. By varying the functions executed and combining functions with different degrees of File System utilization the overhead introduced becomes negligible and the average function invocation time converges. This indicates that FS Virtualization is a viable option for Serverless Platforms, as the initial overhead can be amortized over time, and performance can stabilize to match non-virtualized configurations.

5.7 Memory Consumption

In addition to execution time, it is important to measure the system's memory consumption to understand if introducing File System Virtualization impacts the memory footprint of the program.

In order to evaluate the memory consumption of our system, we'll run the Misc benchmark across all different execution modes and compare the resource utilization using `time -v`. To measure this, we'll

extract the Maximum Resident Set size, meaning the maximum amount of physical memory assigned to the process. For this experiment, we ran the Misc benchmark with 50 threads.

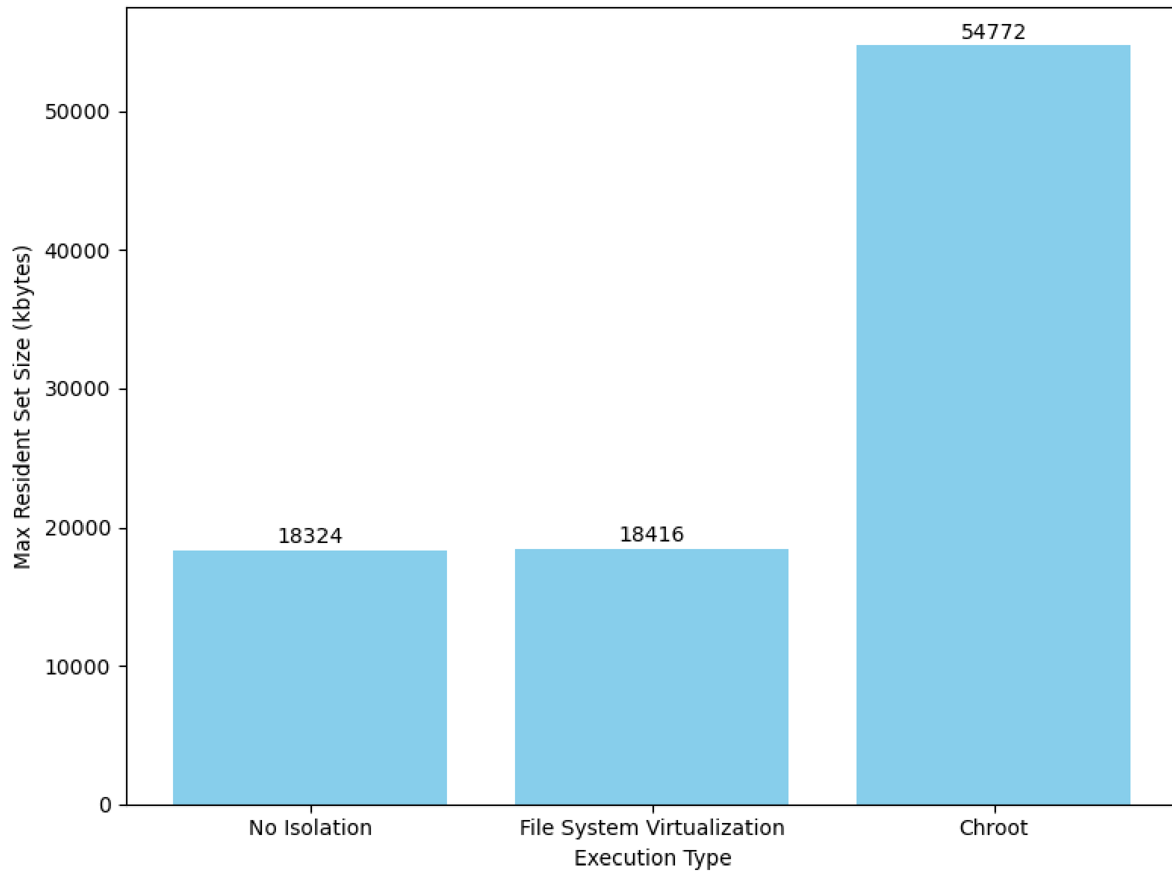

Figure 5.11: Memory consumption across different execution modes.

Figure 5.11 shows that the memory consumption overhead is minimal when compared to a solution without isolation, and significantly lower than the one observed for chroot. This shows that our system is able to provide File System Virtualization without significantly increasing the memory footprint of the program.

6

Conclusion

Contents

6.1 Conclusions 47
6.2 Future Work 48

6.1 Conclusions

There is a growing demand for Runtime Sharing in Serverless platforms as it reduces start-up latency and memory footprint. In order to fully virtualize the runtime, File System Virtualization remains to be addressed in order to ensure that function executions can execute concurrently without corrupting the File System state.

The goal of this project was to create a lightweight mechanism that enables File System sharing among concurrent invocations running in the same process without significantly increasing memory consumption or introducing significant latency overhead. Chroot provides an alternative that ensures the same isolation benefits but adds significant latency and memory overhead. Additionally, chroot works at the process level so it doesn't ensure isolation between concurrent threads running inside the same process making it unsuitable for platforms that aim to share the runtime efficiently across threads.

To address File System Virtualization, we introduced a system that leverages Seccomp to intercept and patch File System-related syscalls and implemented a copy-on-write mechanism, to ensure that each thread only modifies its local copy of any given file. The solution uses a monitoring thread to poll for Seccomp notifications and modify the syscalls so that they're redirected to the appropriate copy of the file.

We tested this implementation using different types of functions for benchmarking. By combining read-intensive benchmarks, with benchmarks that modified and used the file content and CPU-intensive benchmarks we were able to get an accurate picture of the system's behavior under production-like workloads. We found that the system was able to replicate the performance of systems that don't provide isolation for workloads that don't interact with the File System. For File System-intensive workloads we saw minimal overhead for read-only operations with some significant introduced for functions that modified and read previously modified files. If we take into account that it was only observed in workloads that relied almost exclusively on File System operations, with minimal computation, we expected the overhead to be mitigated once we diversified the function types, and this is what we saw. As we diversified the workloads, the monitor thread ceased to be overloaded and the working threads were able to complete the functions faster as the waiting time was drastically reduced.

Overall, when we tested the system using a benchmark that combines the different functions we saw that the performance of our system closely approximates solutions without File System isolation and vastly outperforms chroot, as it needs to create and move to a new root in every invocation as well as loading all the needed File System context. This shows that our approach provides the benefits of file system isolation without sacrificing performance. In contrast, Chroot-based isolation results in significantly higher execution times and memory usage, making it impractical for high-throughput, low-latency serverless applications.

In conclusion, this thesis presents a lightweight, efficient mechanism for File System Virtualization in Serverless Platforms. The system delivers isolation with minimal performance impact, providing a scalable solution that is able to handle large workloads efficiently.

6.2 Future Work

While this thesis presents promising results, there are several opportunities for future work and improvement.

- The system could be integrated into existing Runtime Sharing Platforms such as GraalVM;
- Although the current system handles basic file operations, future iterations could include support for more complex operations such as symbolic links;

- In the current implementation, one monitoring thread is responsible for processing all the intercepted syscalls. However, it can be modified to include multiple Seccomp filters with corresponding monitor threads polling from different file descriptors for notifications, thus improving the system performance and reducing idle time in the worker threads without the need for any additional synchronization;

Bibliography

[1] "What is a virtual machine." [Online]. Available: https://www.vmware.com/topics/glossary/content/virtual-machine.html

[2] "Use containers to build, share and run your applications." [Online]. Available: https://www.docker.com/resources/what-container/

[3] S. Shillaker and P. Pietzuch, "Faasm: Lightweight isolation for efficient stateful serverless computing," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 419–433.

[4] M. Alzayat, J. Mace, P. Druschel, and D. Garg, "Groundhog: Efficient request isolation in faas," *arXiv preprint arXiv:2205.11458*, 2022.

[5] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, "Pushing serverless to the edge with webassembly runtimes," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022, pp. 140–149.

[6] Z. Li, J. Cheng, Q. Chen, E. Guan, Z. Bian, Y. Tao, B. Zha, Q. Wang, W. Han, and M. Guo, "{RunD}: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 53–68.

[7] N. Lopes, R. Martins, M. E. Correia, S. Serrano, and F. Nunes, "Container hardening through automated seccomp profiling," ser. WOC'20. New York, NY, USA: Association for Computing Machinery, 2021, p. 31–36. [Online]. Available: https://doi.org/10.1145/3429885.3429966

[8] "What are cloud iaas, paas, saas, faas, and why we use them." [Online]. Available: https://pub.towardsai.net/what-are-cloud-iaas-paas-saas-faas-and-why-we-use-them-8af979dad141

[9] "Linux fundamentals: user space, kernel space, and the syscalls api surface." [Online]. Available: https://www.form3.tech/blog/engineering/linux-fundamentals-user-kernel-space

[10] "What is a micro vm?" [Online]. Available: https://www.techtarget.com/searchsecurity/definition/micro-VM-micro-virtual-machine

[11] "Operating lambda: Performance optimization – part 1." [Online]. Available: https://aws.amazon.com/blogs/compute/operating-lambda-performance-optimization-part-1/

[12] "Isolates and compressed references: More flexible and efficient memory management via graalvm." [Online]. Available: IsolatesandCompressedReferences:MoreFlexibleandEfficientMemoryManagementviaGraalVM

[13] "Types of cloud computing." [Online]. Available: https://aws.amazon.com/types-of-cloud-computing/

[14] "What is aws lambda?" [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html

[15] "Seccomp security profiles for docker." [Online]. Available: https://docs.docker.com/engine/security/seccomp/

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

[17] "cgroups(7) - linux manual page." [Online]. Available: https://man7.org/linux/man-pages/man2/cgroups.7.html

[18] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 205–218.