TÉCNICO LISBOA

Reducing FaaS Latency with Storage-backed Caching

Pedro Caldeira Alexandre Próspero Luís

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Rodrigo Fraga Barcelos Paulus Bruno

Examination Committee

June 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 dissertation supervisor Prof. Rodrigo Bruno for always being available to help and sharing his knowledge and insight.

I would like to thank my family, who have always supported me emotionally and in my academic endeavors.

I would like to thank my girlfriend, for her unconditional love and support, and for always motivating me to work on this thesis.

I want to thank my friends, whose companionship has made working on this thesis easier.

Finally, I want to thank the reader for taking the time to read this.

Abstract

Cloud computing has become an immensely popular technology in the last few years. Function-as-a-Service (FaaS) is a recent cloud computing service that speeds up and facilitates application development. The developer only has to write the code for the application and the FaaS provider fully manages the infrastructure (serverless). However, FaaS applications suffer from cold starts. A cold start is when a function from an application is invoked but its runtime is not loaded. When this happens, the function has to be initialized and resources have to be loaded and this can a lot of time. Minimizing the incidence of cold starts is the key to reducing application latency in FaaS.

To tackle this problem, we developed a caching algorithm that extends the already existing memory cache FaaS services use to also incorporate other storage devices. To test this algorithm, we created a simulator and an emulator. This simulator allows us to test FaaS workloads with a high volume of invocations. It also enables us to experiment with multiple variations of the caching algorithm and different storage device technologies. The emulator allows us to test the algorithm in a more realistic environment.

Keywords

Serverless; Caching; Cloud Computing.

Resumo

A computação em nuvem tornou-se uma tecnologia extremamente popular nos últimos anos. Função-como-um-Serviço (FaaS) é um serviço recente de computação em nuvem que acelera e facilita o desenvolvimento de aplicações. O programador só tem de escrever o código para a aplicação e o fornecedor do serviço gere totalmente a infraestrutura (sem servidor). No entanto, as aplicações FaaS sofrem de inícios frios. Um início frio é quando uma função de uma aplicação é invocada mas o seu ambiente de execução não está carregado. Quando isso acontece, a função precisa de ser inicializada e os recursos precisam de ser carregados, o que pode demorar muito tempo. Minimizar a incidência de inícios frios é a chave para reduzir a latência de aplicações em FaaS.

Para resolver esse problema, desenvolvemos um algoritmo de cache que estende o cache de memória já existente que os serviços FaaS usam, para incorporar também outros dispositivos de armazenamento. Para testar este algoritmo, criámos um simulador e um emulador. Este simulador permite-nos testar cargas de trabalho com um elevado volume de invocações. Também nos permite experimentar múltiplas variações do algoritmo de cache e diferentes tecnologias de dispositivos de armazenamento. O emulador permite-nos testar o algoritmo num ambiente mais realista.

Palavras Chave

Sem servidor; Caching; Computação em nuvem.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Contents

Function-as-a-Service (FaaS) is a new cloud computing service that allows clients to provide code in the form of functions to be executed in the cloud. Clients set the triggers that determine when the functions are invoked and then the cloud provider is responsible for launching the functions and managing the servers. This type of service simplifies the development and deployment of software applications because the cloud provider is responsible for all of the infrastructure management and as such, the client can focus more time on the development part of the application. It can also reduce the costs for clients because no idle time is billed, only computation time. On top of this, FaaS services also provide a great level of elasticity, scaling up and down depending on the needs of the application. Due to these benefits, FaaS services are rapidly increasing in popularity and all the major cloud providers have their version: AWS Lambda [2], Azure Functions [3], Google Cloud Functions [4], etc.

The clients want their functions to be executed as fast as possible from the moment a trigger is activated, and cloud providers aim to meet the client's demands while keeping the infrastructure cost to a minimum. FaaS is still relatively new and, as such, there are still many open problems, one of which is the focus of this work.

1.1 Problem/Motivation

One of the critical drawbacks of FaaS services is the function start latency. FaaS services use containers or microVMs [5] to encapsulate and execute the functions. When the function finishes executing, the container is kept for a while (keep-alive period). If the function is invoked again, it will be assigned to that container and there won't be any startup latency. However, if the container has been shut down, a new container will have to be launched, causing a cold start. Launching a new container can take hundreds of milliseconds, and this can be disruptive for some applications, considering that 50% of functions take less than 1s to execute [6]. One way to reduce cold starts is to raise the keep-alive period, but this rapidly becomes too costly in terms of wasted resources (note that the user does not pay for idle memory time but the provider still needs to cope with infrastructure costs). An alternative to long keep-alive periods is pre-warming which requires predicting function invocations and loading the resources needed for the function execution before the function is invoked [6] [7] [8]. However, a lot of functions are hard to predict and this can cause unnecessary pre-warmups [6]. Finding a balance between reducing cold starts and keeping resource usage low is an important aspect of FaaS [6].

1.2 Goals

The main goal of this thesis is to find a way to reduce cold start latency in FaaS services without having to rely on function forecasting techniques and with minimal infrastructure costs for the provider. By doing this, we will provide a way to improve the client experience in FaaS services and reduce the costs for the Cloud Providers. We aim to contribute to the research of an emerging Cloud Computing research topic.

1.3 Drawbacks of existing works

Many works already propose multiple ways of reducing the function start latency. Many of them, try to predict when a function invocation will happen and warm up a container (load the function runtime and libraries necessary). These works can still struggle when functions are called at irregular time intervals, something that is not uncommon in serverless workloads. Other works propose keeping snapshots of the containers or VMs on disk or remote storage. However, previous works don't take into account the fact that storage and bandwidth are limited, and with the number of function invocations per day, it quickly becomes unrealistic.

1.4 Solution

This thesis will explore the impact of using disks of different types, with various bandwidths, and a network object storage service to save container checkpoints to minimize the function start latency present in FaaS services. We will extend the already existing in-memory cache to transfer functions in memory to a disk cache and a network cache. We will estimate the amount of bandwidth required and the currently available bandwidth. Using these values, we estimate the time needed to pull a checkpoint from the disk. We then compare the estimated times with the cold start latency to decide if we create a new container and initialize the function, or pull and restore a checkpoint (lukewarm/remote start). To test the impact of saving checkpoints on disk and in a network, we will use a real workload of Azure Functions. This workload was published recently [6].

To validate our approach, we build both a simulator and an emulator. In this simulator, we can choose the parameters we want, such as the memory available, bandwidth and storage space of the disks, the bandwidth of the network, and more. The emulator is limited by the hardware of the machine it runs in. Still, we can limit the disk storage size and memory. Even though the Azure trace only provides the number of invocations per minute for each function, we generate a sequential trace giving us invocations in certain timestamps. The simulator allows us to find the impact and optimal settings of saving container checkpoints to disk/network. The emulator allows us to run a workload in a more realistic environment and verify if the simulator's results are realistic.

2 Background

Contents

In this section, we will review the concept we deem fundamental for this thesis. We will provide the necessary background for the reader to better understand the rest of the work.

First, we go into detail on cloud computing. This thesis has FaaS as a main topic and FaaS is a type of cloud computing model. As a consequence, we start by discussing what cloud computing consists of and its ideals. We also have to provide a solid overview of virtualization. Virtualization is the foundation layer on top of which cloud computing builds. Furthermore, we introduce different approaches to virtualizing resources, from VMs to containers, and microVMs. We then provide background on the serverless computing model. FaaS is an implementation of the serverless model. To get a better understanding of the workings of FaaS services, it is necessary to know how serverless operates. Finally, we also discuss the evolution of storage devices. This is so the reader has an idea of how the different hard disk technologies compare with each other and how future ones may look.

2.1 Virtualization

Virtualization is a technology that allows you to multiplex the hardware resources of a single physical computer into multiple virtual computers. Virtualization surged in the 1960s as a solution to optimize the resources of mainframe computers. Today, it's the driving force behind cloud computing. Cloud providers have their data centers and they create virtual environments that use hardware resources. This allows clients to pay only for the resources they need, as opposed to entire servers.

Until recently, the only main form of virtualization used was Virtual Machines. A system can have multiple Virtual Machines running, each with an instance of an OS. Although Virtual Machines provide several benefits such as efficiency and security, they have some drawbacks, such as the slow start-up time. Software programs that enable virtualization are called hypervisors. Hypervisors create and manage VMs running guest OSs. There are two types of hypervisors: bare-metal and hosted. Bare-metal hypervisors run directly on the host's hardware without the need for an OS while hosted hypervisors run on top of an existing OS. An example of a hypervisor is Xen [9]. Xen offers two types of virtualization: paravirtualization and full virtualization. In paravirtualization, the OS knows it's being virtualized, which allows for efficient communication with the hardware resources. Full virtualization is fully hardware-assisted with emulated devices, where the CPU provides support for virtualization extensions. In this mode, Xen uses a modified version of QEMU [10] to emulate full PC hardware. QEMU is an emulator that emulates the machine's CPU and provides multiple hardware models for the machine. Containerization is another form of virtualization. The difference between containerization and traditional virtualization is that containerization runs multiple user spaces with isolated processes using only one OS, whereas, in traditional virtualization, multiple OS instances are running in one system. Container technology has existed for a long time, but in 2013, with the introduction of Docker [11], it gained a lot of popularity. Docker provided an easy way to create, manage, and deploy containers and quickly became the industry standard.

Containers play a critical role in cloud computing. They are lighter than VMs, so they start much faster and occupy less storage, which reduces latency in a lot of cloud services and allows cloud providers to reduce resource waste. They provide an efficient way for developers to package and deploy their apps. There are also container orchestration technologies that simplify this process. Although containers bring these advantages to the table, one big disadvantage is security. As containers share the same OS, as opposed to VMs where each one has its own OS, they have less isolation and thus offer less security. An attack on the host operating system can affect every container present in it. The inverse can also happen, an attack on a container can affect the host OS. There needs to be a balance between performance and isolation. An example of a container sandbox designed to improve security is gVisor [12]. They isolate the containers from the host OS by providing a virtualized environment to sandbox the container. gVisor intercepts the system calls and acts as a guest kernel. By doing this, they limit unauthorized access to the host OS and provide a higher level of isolation. Containers are the foundation of FaaS services. The functions are run in containers, isolated from other functions. Without containers, the functions would have to be executed in Virtual Machines and the start-up time would be too much of a problem.

Recently, alternatives to containers have surged such as microVMs [5] and unikernels. Firecracker [5] is a microVM implementation. It's a hypervisor that uses the Linux Kernel-based Virtual Machine to create and run microVMs. It removes unnecessary components from the VMs to make them more lightweight. Functions and container workloads can be run inside these. In cloud services, microVMs keep the resource efficiency of containers with the added benefits of isolation and security. Unikernels are machine images that pack the necessary components to run an app. Each deployment unit contains a small kernel. Because of this, they can run on bare-metal hypervisors. Compared to containers, unikernels are more secure and less resource-intensive.

2.2 Cloud Computing

Cloud computing has emerged in recent years as a revolutionary technology. It refers to the delivery of computing services over the Internet. Although it has gained popularity in the last few years, it has been a concept for longer. The term "cloud computing" first appeared in 1996 in a Compaq internal document [13]. It was linked to distributed computing. In 1999, the first instance of a SaaS (Software as a Service) service emerged as Salesforce started to offer its applications over the Internet. In the 2000s, Amazon launched AWS and EC2 (Elastic Cloud Compute) as one of its services. Microsoft also launched its cloud, called Azure. Since then, cloud computing has continued to grow and evolve.

The main goal of cloud computing is to leave infrastructure management as the responsibility of the cloud provider. This way, clients only pay for the resources they effectively use as opposed to paying for resources they might use. Its main benefits are:

- **Flexibility**: It is simple to scale the services in tune with the client's needs. With on-premises infrastructure, an entity may own too few servers to support an unexpected surge in activity. This doesn't happen with cloud services as the cloud providers have a vast amount of resources and can easily allocate more to the entity. The opposite can also happen, and the entity may have overestimated the demand for its services and own more servers than it needs. This also isn't a problem in the cloud as the client can simply choose to use fewer resources. Depending on the type of cloud service, the cloud provider can even eliminate the need for the client to estimate the number of resources they need and be the ones who handle the scaling of the infrastructure;

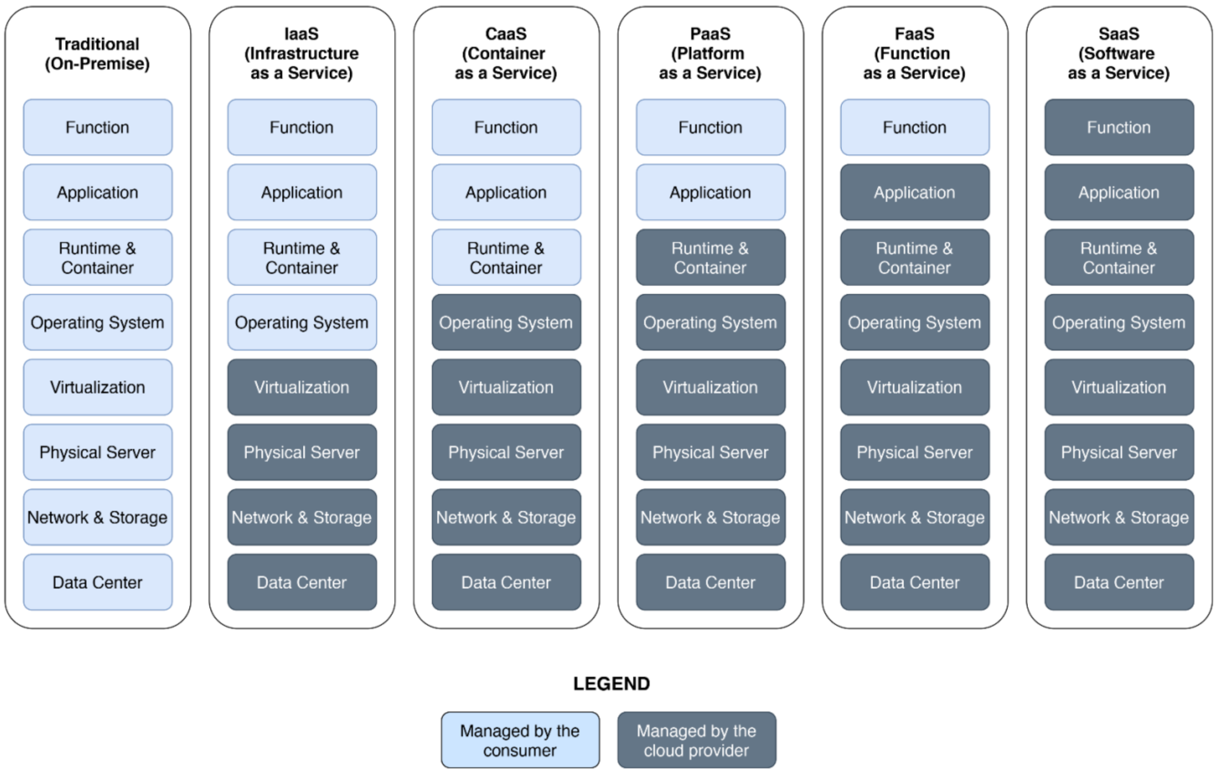
Traditional (On-Premise)
IaaS (Infrastructure as a Service)
CaaS (Container as a Service)
PaaS (Platform as a Service)
FaaS (Function as a Service)
SaaS (Software as a Service)
Function
Application
Runtime & Container
Operating System
Virtualization
Physical Server
Network & Storage
Data Center
LEGEND
Managed by the consumer
Managed by the cloud provider

Figure 2.1: Responsibilities in cloud services.

- **Efficiency**: Infrastructure management is the responsibility of the cloud provider. This allows clients to dedicate more manpower to other important tasks, adding to the efficiency of the development process;

- **Cost effectiveness**: With cloud services, the client no longer needs to build an infrastructure with usage spikes taken into account. The client also needs to spend less on DevOps as the cloud provider does most of the heavy lifting. Both of these factors contribute to a lower cost in the development of an application/service;

- **Security**: It is generally safer than using on-premises infrastructure. Cloud providers have a lot of experience in dealing with security threats, due to the magnitude of clients they have. It is also beneficial from a business standpoint, for the cloud providers to have good security in place. This is because a security breach damages the provider's reputation and can result in legal action. Ultimately, cloud computing can be considered safe as long as you trust the cloud provider.

Due to these benefits, many companies are moving or have moved their digital assets to the cloud. The main types of cloud computing service models are the following:

- **IaaS (Infrastructure as a Service)**: IaaS is a cloud service model that offers on-demand infrastructure resources. It's the type of service that gives the user the most control over the resources.

An example of an IaaS offering is AWS EC2, which allows the user to rent a virtual computer. The user chooses the hardware specifications they deem necessary to run their application. A popular use case of IaaS is running resource-intensive workloads. The user can easily launch a powerful virtual machine and set a policy for automatically launching extra instances if needed. The users are billed for the time the instance is running;

- **CaaS (Container as a Service)**: CaaS is a type of cloud service that allows the user to manage containers at a large scale. The user configures the containers and the cloud provider manages the underlying infrastructure. It works similarly to IaaS but the unit is a container as opposed to a virtual machine. Amazon ECS is an example of a CaaS service. The user provides the container images and files called task definitions, which contain the container configurations, such as which ports to open, what data volumes to use with the containers, and parameters for the operating system. Users are billed for the vCPU and memory resources their containerized application uses;
- **PaaS (Platform as a Service)**: PaaS is a cloud service model that offers tools for developing and managing apps. The service provides all the hardware and software resources the developer needs. The developer needs only to write the code. It offers less control than IaaS services as the provider is the one responsible for the Operating System, container orchestration, language runtime, and more. Because of this, it also makes app development faster and easier than with IaaS services.

  One example of a PaaS service is Google Cloud Run. You provide the code for the App and when there's a request, Cloud Run takes care of the containers and infrastructure. The client can choose for the app to scale to zero, which is cheaper but can cause slow requests if there isn't at least one container running. The client can choose the billing mode: Request-based or Instance-based. Request-based depends on the number of requests and Instance-based on the lifetime of container instances;
- **FaaS (Function as a Service)**: FaaS is a type of cloud computing service that allows users to execute code in response to events that have a trigger associated. In FaaS services users provide code in the form of functions and triggers that when activated invoke those functions. FaaS is frequently associated with serverless because the user only provides the code and has no control over the infrastructure, creating the illusion of no servers. An example of a FaaS service is AWS Lambda. Users only provide the functions and the amount of memory they want to execute that function. They also specify the events that should trigger the function. Amazon then automatically provisions the infrastructure and scales it up or down depending on the demand. Users are billed for the number of requests and execution time of the functions;
- **SaaS (Software as a Service)**: SaaS is a type of cloud computing service where users connect

to cloud-based apps over the Internet. The infrastructure, software, and data are managed by the service provider. An example of SaaS is Microsoft Office.

In Figure 2.1, you can find the differences between component management in the cloud models we talked about. Note how the responsibility of different layers of the software stack shifts towards the cloud provider allowing developers to focus on core domain logic.

2.3 Serverless Computing

Serverless computing is a cloud computing execution model where app developers don't have to worry about managing any of the server infrastructure or the tasks that it brings. In serverless, when an app is not executing, there are no resources allocated to it. Some apps are entirely serverless, and others use serverless components to complement the rest of the app. It's up to developers to choose the most suitable architecture depending on their use case. The key benefits of serverless computing are scalability and elasticity. The provider allocates the needed resources for the app and scales them up or down depending on the demand. Because of this, serverless apps have high availability and are cost-efficient, since clients are only billed for the time the app is executing.

Providers often offer FaaS platforms implemented on top of a serverless infrastructure: runtimes that execute application logic but don't store data. FaaS makes use of containers to execute the functions provided by the client. In Figure 2.2, there's a diagram of the basic architecture of a FaaS service. There are two sources of requests, events and API requests. An example of an event is data being written to Amazon S3. An example of an API request is an HTTP GET request. When there's an event, the event mapper finds the correct function to execute and sends a request for the scheduler to execute that function. In the case of an API request, the API gateway finds the correct function for that specific request and sends the function request to the scheduler. After this, the scheduler allocates the needed resources and executes the function inside a container/microVM. If it's necessary, the API gateway receives the result of the function's execution.

As we mentioned, FaaS services create containers when functions are invoked through events or API requests and destroy them when it is deemed no longer advantageous to keep them alive. Finding the right moment to destroy containers has been a highly researched topic in Serverless. If the container is destroyed too early, the next invocation will suffer from a cold start. However, if a container is left alive for too long and the function has no invocations, the time it was kept in memory is useless resource wastage.

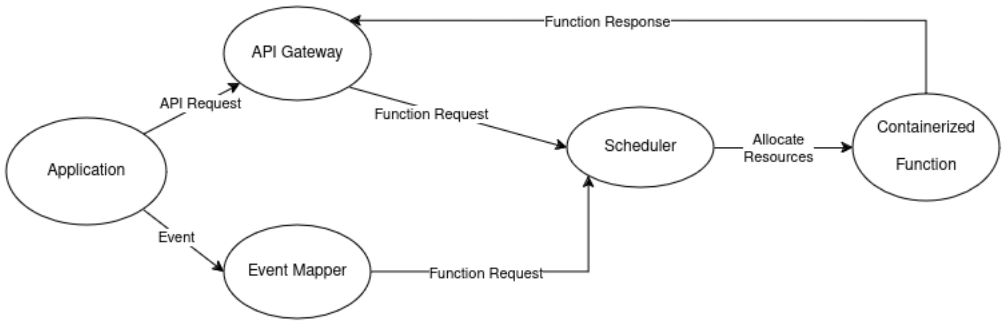

Figure 2.2: The architecture of FaaS services.

2.4 Serverless Worker Caching

Caching is the action of storing data in a temporary storage location for faster access. In the context of this thesis, we will look at caching in FaaS services. Caching is necessary for FaaS, due to the slow launch time of containers/microVMs and the long time it takes to initialize the runtime. FaaS providers maintain a cache of containers in memory. Usually, containers are kept alive for a few minutes after the function finishes executing [14]. AWS Lambda keeps containers alive for 10 minutes and Azure for 20 [6]. While this can reduce cold starts, it is not the most effective solution because it can waste a lot of memory that could otherwise be used to execute more invocations. A lot of research is being done to optimize this. When there's an invocation, there are three alternatives. The first is to use a cached container/microVM. This is the ideal scenario because the start latency is minimal, however, the container/microVM has to be cached in memory. Another alternative is to launch a new container/microVM and load the function runtime. As we mentioned before this can be very slow. Finally, we can restore the container/microVM from disk or remote storage. Before doing this, we have to determine if it's faster than a cold start. This can be a hard task as it depends on disk/network bandwidth, the restore latency, and more.

A problem with traditional FaaS caches is that RAM has a small storage capacity in comparison to disks. Due to this, FaaS vendors may reach the limit of containers stored faster and need to destroy containers still within the keep-alive window. Another issue is that containers, while lightweight compared to VMs, still occupy significant storage space. This could be resolved by caching with a lower level of abstraction.

2.5 Evolution of Persistent Memory Devices

Compared to RAM, disk access time is always going to be slower. However, due to RAM storage space being small, you might need to remove functions from the cache prematurely. In this thesis we've talked about storing container checkpoints on disk, however, this can only be advantageous if the disk is fast enough so that the access time is minimal. As you can see in Figure 2.3 [1], new SSDs with

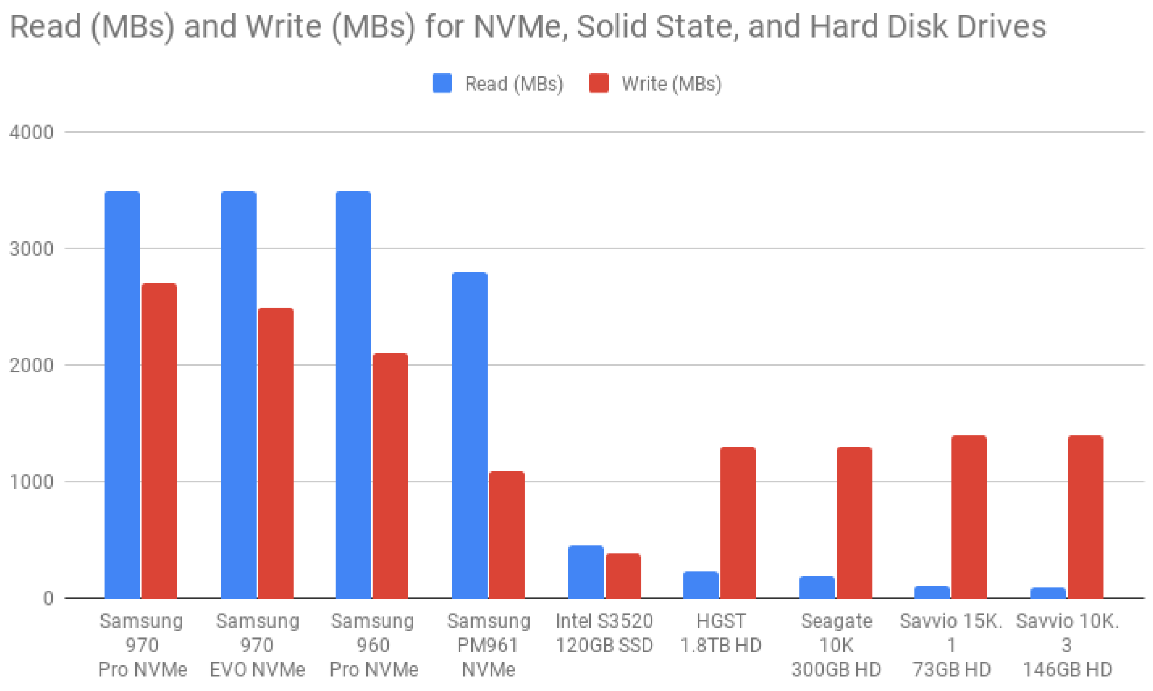

Figure 2.3: Comparison between different persistent memory devices read and write speeds [1].

the NVMe protocol are multiple times faster than regular SSDs. These technologies also each have different prices. At the time of writing this document, a 16 GB RAM stick is typically around $80. An HDD is usually around $35 per terabyte. SSDs are more expensive: a 1TB SATA SSD is usually priced at around $90 and a 1TB NVMe SSD can cost around $130. The cost per storage increases with the storage bandwidth.

More recently, a new technology has emerged: Compute Express Link(CXL). CXL maintains memory coherency between the CPU memory space and memory on attached devices. This results in better performance and lower system costs. This technology can be applied to persistent memory devices. One example of this is Samsung, which has unveiled a CXL SSD that bolsters a 20x performance increase over traditional SSDs [15].

In this thesis, we want to find the speed necessary for our method to be useful and find if it's applicable in the present. However, because of the constant evolution of storage devices, it makes sense to test disk and network speeds still not commercially available.

3 Related Work

Contents

In this section, we will explore work related to this thesis. These works ultimately have the same objective as this thesis, optimizing function start times in FaaS platforms. They can be divided into a few main areas of research: caching functions; function scheduling and snapshotting. The first one is about finding function invocation patterns and loading the function runtime to memory before the invocation occurs, preventing a cold start. The second one consists of finding an optimal policy for which servers to allocate the invocations. Preferably you would allocate an invocation to a server that already has the runtime loaded. The last one consists of saving the state of a container/microVM in storage and restoring it when a cold start occurs. In principle, this is faster than loading the function runtime from zero, which is what happens when there is a cold start. We will give a brief explanation of each work and find their limitations and what we can use for this thesis.

3.1 Caching Functions

As we mentioned before, reusing a container/microVM to execute a function gives minimal start latency. By predicting function invocations, it's possible to optimize the usage of the memory cache and consequentially reduce overall function start latency. This optimization can be done by extending the keep-alive window for a specific function or by pre-warming it. If the provider predicts when a function is going to be invoked, and the function is already in the memory cache, they can extend the keep-alive window to catch that invocation. If the function is not in the memory cache, the provider can pre-warm the function, this is, launch a new container/microVM and load the runtime before the invocation happens. In this case, the resource usage is the same for the provider, but from the perspective of the client, the invocation has a lower start latency. When using these methods, it is important to have a high level of accuracy in predicting the invocations. Otherwise, it leads to unnecessary keep-alive extensions and pre-warms, causing resource wastage.

3.1.1 Serverless in the Wild

This paper [6] is one of the most important for this thesis. It provides a real workload trace of Azure Functions. This trace contains important data such as function execution duration, memory, and function invocation count. This was the first public trace of its kind, which opened many doors for research. This paper collects data on all function invocations across Azure's entire infrastructure between July 15th and July 28th, 2019. They collected 4 datasets:

- Invocation counts, per function, in 1-minute bins;
- Trigger per function;
- Execution time per function: average, minimum, maximum, and count of samples, for each 30-second interval, recorded per worker;
- Memory usage per application: sampled every 5 seconds by the runtime and averaged, for each worker, each minute. Average, minimum, maximum, and count of samples, for allocated and resident memory.

They make three main observations: the vast majority of functions execute under a few seconds; the vast majority of applications are invoked infrequently and many applications show wide variability in their IATs (inter-arrival times). Based on these observations they propose a function invocation prediction model to reduce the number of cold starts. They use a range-limited histogram to determine a pre-warming window and a keep-alive window. When the pattern is uncertain they use a standard keep-alive window. When the histogram is not large enough they use a time-series analysis.

This paper has some limitations such as the granularity of the function invocation data. Invocation counts are binned in one-minute intervals. This presents a problem because FaaS services usually operate with millisecond granularity, and functions can be invoked at any millisecond. Another issue is that although pre-warming a function reduces function start latency for the service user, from the provider's perspective, it's still effectively a cold start because they have to create the container/microVM and load the function runtime. Also, if there's no invocation when the function is pre-warmed, there's a needless waste of resources. This brings us to our last point, which is that this paper doesn't take into account memory limitations. They don't have a solution if there's no more space for keeping containers in memory.

3.1.2 Icebreaker

Icebreaker [8] is another paper that presents a solution for reducing function start latency based on predicting function invocations. They identify two major limitations with current approaches: fixed keep-alive cost during the keep-alive period and lack of robustness to frequently changing patterns and concurrency degrees. They aim to reduce the keep-alive cost (by mixing the type of nodes: more powerful and expensive or less powerful and cheaper, used to warm up functions), reduce the service time (by reducing the incidence of cold starts), and take into account the level of concurrency the function when it's going to be invoked. To achieve their goals, they propose two mechanisms that work together:

- **Function Invocation Prediction Scheme (FIP)**: This scheme predicts if a function is going to be invoked in a certain time interval and its concurrency. They show that function invocation concurrency can be periodic and so, treat it like a pattern. They consider this pattern a time domain series, with the amplitude being the number of invocations in a time interval, and the period as the frequency with which the amplitude is repeated. With the Fourier transform, they compute the harmonics and the amplitudes in the time domain.
- **Placement Decision Maker (PDM)**: This component is responsible for keeping costs down. It assigns a utility score to each function with predicted invocations in a time interval by the FIP and based on this score chooses the type of node the function is going to be warmed up in. This utility score is based on a few factors: true negative prediction rate, false positive prediction rate, inter-server speedup(effect on the execution time of moving from one type of node to another), and memory footprint. If this utility score is above the high-end cutoff, the function is warmed up on a high-end server; if it's below the high-end cutoff but above the low-end cutoff, then it's warmed up on a low-end server. If it's below the low-end cutoff it's not warmed up at all.

Overall, Icebreaker shows great improvements. It reduces keep-alive cost by 43% and service time by 27% over the state-of-the-art approaches. In comparison to Serverless in the Wild [6], on high-end servers, memory wastage is reduced by 20%.

3.1.3 Orion

It would be an asset for FaaS services if they could estimate the E2E latency of DAG applications and allocate resources in a way that meets guarantees for a user-specified limit. Also, if you correctly determine when a stage of the DAG application ends, you can pre-warm functions reducing the E2E latency. These problems have not been solved due to three main issues: high variability and correlation in the execution time of individual functions, skew in execution times of the parallel invocations, and incidence of cold starts. This paper's [7] goals are to provide a workload characterization for serverless DAGs and to reduce E2E latency in these applications, meeting the client's requirements.

They first model the E2E latency distribution. They do this by modeling the functions' runtimes. They then combine these functions' latency to estimate the DAG E2E latency distribution. They also handle correlation among functions: they consider two types of statistical correlation in the DAG: in-series and in-parallel correlation. Using this E2E latency distribution model they design a per-function performance model. This per-function model determines the function latency for multiple VM sizes. Using this last model they can choose a configuration to execute the DAG that meets a user-specified latency objective and reduces the cost. They then bundle multiple parallel invocations in one stage within a larger VM. Finally, they determine when to start pre-warming the functions in each stage of the DAG. They do this by selecting the vector of pre-warming delays for each stage (time between the start of the DAG execution and the start of the VM initialization for that stage) that minimizes the DAG E2E latency while maintaining the target resource utilization (ratio between VM busy time and VM total time) set by the provider. Their evaluation shows that compared to competing approaches, ORION achieves up to 90% lower P95 latency without increasing cost, or up to 53% lower cost without increasing P95 latency.

3.2 Function Scheduling

Scheduling is an important part of Serverless. It is the process of allocating resources for a function's execution. When a function is invoked, the Serverless platform assigns it to a server. A good scheduling policy optimizes resource allocation and guarantees small idle times. A policy that does these things reduces function start latency and keeps the costs low.

3.2.1 Hermod

The authors of this paper [16] identify a few problems with current Serverless approaches. Late binding is a commonly used method consisting of waiting for the worker to be ready before assigning a task (invocation). They find that this method is sub-optimal for highly variable workloads: short invocations may have to wait for long invocations to finish executing. They also find that both random and locality-based load balancing are unsuitable for highly variable workloads as they can cause load imbalance across workers. Lastly, they determined that at low loads, least-loaded balancing can cause an increase in cold starts and use more servers than needed.

To address these problems, they developed a scheduler they call Hermod. Hermod is a hybrid scheduler. At low loads, it packs each worker with an invocation per core. When a worker is filled, a new worker will get packed. This policy guarantees no queuing at low loads. When all the servers are packed, it's considered high load and Hermod switches its policy. At high loads, they use least-loaded balancing. This reduces queue times in overload situations. Hermod is also locality-aware: it maintains a list of available warm containers and if there's one for the current function it directs the invocation to it. This reduces cold starts. Hermod shows promising results. They demonstrate that it can provide up to 85% lower slowdown and support a 60% higher load compared to existing approaches.

3.2.2 Palette

This paper [17] proposes a solution for the lack of locality present in current serverless workflows. In current FaaS services, data-intensive applications achieve low performance. This is because most external and cross-function interactions happen across network hops, as opposed to local data access which is much faster. Although this happens due to a lack of locality, there still isn't a way to maintain locality across executions. There have been multiple proposals to solve this problem, such as fast remote storage, specializing the system, and using functions as containers. The authors of this paper consider these methods are either inefficient or give up important benefits of the Serverless architecture.

The solution proposed is a color system. The app developers assign colors to their invocations. These colors work like hints, not as constraints. The FaaS provider will try to place invocations with the same color in the same instance. However, because colors are only hints, this might not happen and the apps still have to work correctly. There are three color scheduling policies, chosen by the app developer:

- **Hashing**: This policy is the simplest, as it is equivalent to randomly assigning colors to instances. However, this policy causes load imbalance which can negatively affect functions' runtimes;
- **Bucket Hashing**: In this policy, they hash colors into buckets and then assign the buckets to instances with load balance in mind;
- **Color Table (Least Assigned)**: Here they use a table to map colors into instances. When there's a new color, it's assigned to the instance with the least amount of assigned colors.

They evaluate Palette with a web app with a local cache and with data analytics apps. They conclude that for web apps, Palette has a near-perfect scaling of cache hit ratios, improving run times. For data analytics, they also improve run times, closing the gap between serverless and serverful apps.

3.3 Snapshotting

Snapshotting refers to the process of saving a copy of the state of a system at a specific point in time, usually to a persistent storage device. In the context of Serverless, it usually refers to saving images of containers/microVMs after they are loaded with the function runtime. Recently, snapshotting has emerged as an effective tactic to mitigate cold start times in FaaS services. Instead of having to launch the VM/container and initialize the runtime, snapshotting bypasses both of these. It loads an initialized function from persistent storage.

3.3.1 REAP

This paper [18] presents an analysis of snapshotting in FaaS services. To aid their research, they built a framework called vHive. This framework allows experimentation with serverless architectures. Using vHive, the authors of this paper characterize a state-of-the-art snapshot-based serverless infrastructure. Based on this characterization, they make three main observations:

- Loading a function from a snapshot has a smaller memory footprint than cold-booting the function. This happens because when loading a snapshot, only the resources needed for the function execution are loaded. When a function is cold-booted, it loads unnecessary resources.
- Snapshot-restored functions' execution times are heavily influenced by page faults. Pages are inadequately mapped into the guest and the host has to bring pages individually from the backing file on disk. Due to the lack of spatial locality of the guest accesses, these file accesses possess a high overhead.
- Multiple invocations of the same function usually access the same pages.

Based on these observations, they introduce a mechanism called REAP (REcord-And-Prefetch). This mechanism's objective is to reduce cold start times in snapshot-based serverless services. It has two phases. The first phase is the Record phase. During this phase, REAP inspects the page faults caused by the guest OS and identifies the location of the pages in the host OS. When an invocation is complete, two files are created. One of the files is the working space file and contains a copy of the guest memory pages that were accessed. The other file is the trace file and contains the offsets of the original pages inside the guest memory file. The second phase is called the prefetch phase. During this phase, function invocations have faster load times. This happens because when there's a new invocation, REAP fetches the working space file and installs it into the function instance's guest memory, reducing page faults. When compared to Firecracker [5], REAP reduces cold start times by 3.7x on average. Although it has promising results, this paper does not take into account memory limits.

3.3.2 FaasSnap

This paper [19] also uses a snapshot-based approach to solve the cold start problem. They focus on reducing page fault times and improving function loading time. They first analyze the Firecracker and REAP snapshot restoring techniques. Through this analysis, they take a few conclusions:

- The OS page cache is an important element in accelerating VM page faults;
- The function's working set can highly differ from one invocation to another due to changes in input data or function execution flow. Because of this, the previous page accesses can't be taken as absolute, and restoring a snapshot should adapt to this case;
- Due to the semantic gap between the guest and the host, the page allocation in the guest memory is translated to an unnecessary page fault.

The solution that this work proposes contains multiple optimizations to improve on REAP and Firecracker. A few examples of these techniques are:

- Instead of blocking the invocation while waiting for the prefetch, the FaasSnap daemon starts the VM and then launches a thread that takes care of the prefetch;
- Instead of recording the order of page accesses in the first invocation, and accessing the pages in the same order, they divide the working set pages into several working set groups by their access order. When reading, these groups and the pages inside them are accessed by access order. By doing this, there's a bigger chance for the loader to access a page earlier than the guest and still preserve disk access locality.

Measuring the latency of two functions, in one of them REAP takes 1408ms to start while FaasSnap takes 1070ms. In the other, REAP takes 480ms against FaasSnap's 136ms. When using remote storage, FaasSnap is on average, 2.06x faster than Firecracker and 1.20x faster than REAP.

3.4 Summary

All of the papers we reviewed propose a solution to the same problem: reducing function start latency. Icebreaker [8], Serverless in the Wild [6], and Orion [7] try to predict when the functions will be invoked and pre-warm the servers accordingly. As in this thesis, we don't try to predict invocations, these improvements are orthogonal to ours. We could use these methods in our simulator to compare results and find which works better with our solution. Hermod [16] and Palette [17] propose different scheduling techniques that reduce the incidence of cold starts. We could also use these methods in our simulator. REAP [18] and FaasSnap [19] are the papers most similar to this thesis. The main difference is that this thesis has a big focus on reducing RAM usage and storage limits. Another difference is that they are storing microVMs and we will store container checkpoints. Finally, they focus on improving the restore speed by reducing and improving page faults and we don't cover this topic.

4 Architecture

Contents

In this section, we will describe the proposed architecture in depth and explain our design choices.

4.1 Overview

Our system aims to reduce the impact of cold starts in serverless platforms. Cold starts are one of the main problems of FaaS services. They can introduce delays as high as 5 seconds in a simple HTTP server and 40 seconds in a Tensorflow image classifier [20]. These delays are so long because the Serverless provider has to load all the resources that make the execution possible.

Instead of using a keep-alive window as most FaaS services currently use, we set a maximum amount of memory that the RAM cache can occupy. We extend the cache to also use disk and network

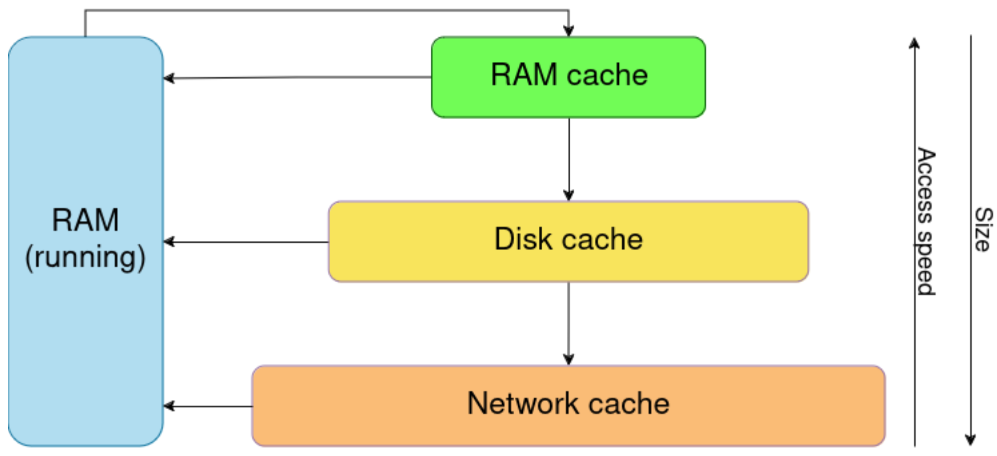

Figure 4.1: Relationship between the caching layers.

storage. We also have to consider that storage and bandwidth are finite and find the method that achieves the highest performance to deal with the case when the storage limit is reached. In this section, we start by discussing the different caching layers that we will make use of, followed by an architecture and design discussion of a caching simulator. We wrap up this section with a description of our emulator that allows us to benchmark our proposal in a realistic setting.

4.2 Caching Layers

Current FaaS services have an in-memory cache. When a function is invoked for the first time, a container with the function's runtime is created. When the function finishes executing, the container is usually kept running for a few minutes. If the function is invoked again, it will use that container and prevent a cold start. However, as we said before, this approach has a few limitations:

- A container can usually only run one function at a time. So if there's only one container with the runtime loaded and multiple invocations, only one invocation avoids a cold start;
- RAM is limited. With the high dimension of function invocations FaaS services experience, it might not be possible to keep all the containers in memory, and they would have to be destroyed prematurely;
- Running a new container and loading the function is more resource-heavy than restoring a container checkpoint.

Our caching system aims to solve these limitations. We use three caching layers, RAM, disk, and network. In Figure 4.1 we show a simple illustration of how these layers are organized. When a function finishes executing, we keep the container alive. This is what already happens in real FaaS services.

When we need to free up RAM, we create a checkpoint of the container and, if it's not already, save it on the disk. When the disk gets full and we want to save new functions to the disk cache, we start uploading checkpoints to the network, or deleting them if they're already there. In Figure 4.2, we have a flowchart of how the caching layers work when there's an invocation. First, the worker checks if the function is in the RAM cache, this is, if a running container already has the function initialized. If it is, the worker will reuse that container. If it's not, it will look in the disk and network caches and predict an invocation start time for each of these caches. If the function is in the disk cache, and its predicted start time is lower than the network's and cold start, a lukewarm start occurs. A lukewarm is when a checkpoint containing the function runtime is restored into a new container. If the function is in the network cache, and its predicted start time is lower than the disk's and cold start, a remote start occurs. The worker downloads the checkpoint into RAM and restores it into a new container. If the function is either not present in the network and disk caches, or the cold start is faster than the other predicted times, a new container is created, started, and initialized. We can assume that network storage is very big and thus we don't need to worry about filling it as much. It still is not infinite so there has to be a method to minimize wastage. We can allow the checkpoints to stay on the network for a certain amount of time without being downloaded. The amount of time depends on the needs and infrastructure capabilities of the service provider. Although disk storage is much bigger than RAM, it's not infinite and we still have to define a policy for when the limit is reached. There are three traditional options for this policy [21]:

- **Least recently used (LRU)**: The function present in the cache that hasn't been accessed for the longest time is evicted. This policy, while generally efficient, performs poorly if there are functions that are invoked occasionally but periodically and others that are invoked frequently but only for a short period.

- **Least frequently used (LFU)**: The function that has the least amount of accesses is evicted from the cache. This policy works well with functions that are invoked repeatedly over time. It is also usually less memory intensive than LRU, as it stores counters instead of time stamps. However, this policy is inappropriate for functions that are invoked frequently during a short period and then never again, as they can get stuck in the cache.

- **First in first out (FIFO)**: This policy is the simplest. It removes the item that has been in the cache for the longest time. It is less intensive than the other two since it only needs to keep the order in that items have been cached, but it's also the least specialized.

We will test these policies and see which performs better.

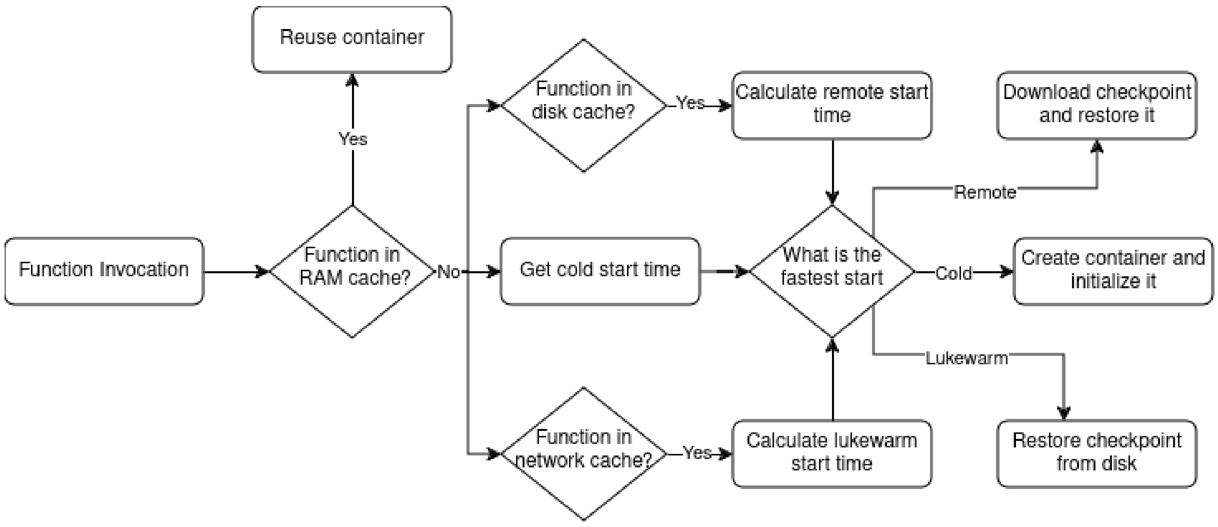
Reuse container
Yes
Function in disk cache?
Yes
Calculate remote start time
Download checkpoint and restore it
Remote
Function Invocation
Function in RAM cache?
No
Get cold start time
What is the fastest start
Cold
Create container and initialize it
Lukewarm
Function in network cache?
Yes
Calculate lukewarm start time
Restore checkpoint from disk

Figure 4.2: Caching layer functionality when there is a function invocation.

4.3 Simulator

To test the caching layer, we need a simulator. A simulator allows us to test many combinations of hardware and use a large-scale trace. This simulator will be implemented using Go. We chose this language because of its efficient, highly scalable, and easy-to-use concurrency model [22]. It is also a language frequently used in cloud services.

The simulator receives as input a workload generated from the Azure Function trace. Although there are more recent traces, this one contains the information we need, such as memory usage and duration. This workload contains an invocation on each line. The important information we extract from each invocation is the hash (function id), memory, duration, and timestamp. This is all the information needed to fully simulate the workload and test our caching system. The memory is in megabytes and both the duration and timestamp are in milliseconds. At run time, the user can set several different parameters to use during the simulation:

- **Input file**: The file that contains the workload to be simulated;
- **RAM**: The amount of RAM available (GB);
- **Disk Memory**: The amount of disk storage available (GB);
- **Cold latency**: The duration of a cold start (ms);
- **Read bandwidth**: The read speed of the disk (GB/s);
- **Write bandwidth**: The write speed of the disk (GB/s);

- **Threshold**: Maximum percentage of RAM occupied by RAM cache, before it starts to get offloaded to the disk cache;
- **Net cache**: Choose whether to use the net cache or not;
- **Net bandwidth**: The network upload and download speed (Gbit/s);
- **Cache Policy**: What policy to use to free the disk cache;
- **Checkpoint latency**: The amount of time it takes to create a container checkpoint (ms);
- **Restore latency**: The amount of time it takes to restore a container checkpoint (ms);
- **Max concurrent invocations**: The maximum amount of invocations executing at the same time.

When the simulator starts, if the net cache option is selected, then the simulator will read a file that represents the net cache. This file contains the function names and storage usage. It will use this information to populate the net cache and disk cache data structures for future usage.

Algorithm 4.1: Invocation Allocation Loop (simulator).

```
1: $invocation \leftarrow getNextInvocation()$
2: $update(invocation.timestamp)$
3: $startType \leftarrow "cold"$
4: $latency \leftarrow coldLatency$
5: if $tryWarm() == true$ then
6:    $startType \leftarrow "warm"$
7:    $latency \leftarrow 0$
8: else
9:    if $reserveMemory(invocation.memory) == false$ then
10:       $registerFailed()$
11:       $continue$
12:    end if
13:    $lukewarmLatency \leftarrow tryLukewarm()$
14:    $remoteLatency \leftarrow tryRemote()$
15:    if $lukewarmLatency > 0\ \&\ lukewarmLatency < latency$ then
16:       $startType \leftarrow "lukewarm"$
17:       $latency \leftarrow lukewarmLatency$
18:    end if
19:    if $remoteLatency > 0\ \&\ remoteLatency < latency$ then
20:       $startType \leftarrow "remote"$
21:       $latency \leftarrow remoteLatency$
22:    end if
23: end if
24: $registerStart(type)$
25: $addInvocationToExecuting(invocation, latency)$
```

The main loop of the simulator is presented in Algorithm 4.1. We will now explain it further. The simulator reads the workload in a loop iterating it line by line. If the maximum number of invocations has already been reached then the simulator will delay this invocation until one of the executing ones finishes. By doing this, we can test the amount of time a certain number of invocations take to execute. The first thing it does after reading an invocation is to update the simulation to the same time as the invocation's timestamp. What this means, is that it will check which invocations have finished and which read and write operations to the disk and network have ended. It will also check if the RAM threshold has been exceeded. If it has, it will start removing functions from the cache. While doing this, the simulator does the necessary calculations, updates the related values accordingly, and does the necessary operations between caches. The update operation is shown in Algorithm 4.2.

Algorithm 4.2: Update the time (simulator).

```
{Update network cache}
$i \leftarrow 0$
while $netCache.writeQueue[i].end < timestampandnetCache.writeQueue[i] \neq NULL$ do
    $addToNetCache(netCache.writeQueue[i])$
    $netCache.writeQueue[i] = NULL$
end while
while $netCache.readQueue[i].end < timestamp\ \&\ netCache.readQueue[i] \neq NULL$ do
    $netCache.readQueue[i] = NULL$
end while
{Update disk cache}
$i \leftarrow 0$
while $diskCache.writeQueue[i].end < timestamp\ \&\ diskCache.writeQueue[i] \neq NULL$ do
    if $diskCache.memory < diskCache.writeQueue[i].memory$ then
        $freeDiskCache(diskCache.memory - diskCache.writeQueue[i].memory)$
    end if
    $addToDiskCache(diskCache.writeQueue[i])$
    $diskCache.writeQueue[i] = NULL$
end while
while $diskCache.readQueue[i].end < timestamp\ \&\ diskCache.readQueue[i] \neq NULL$ do
    $diskCache.readQueue[i] = NULL$
end while
{Update RAM cache}
if $ramCache.occupied/ram.totalMemory > THRESHOLD$ then
    $freeRam()$
end if
{Update executing functions}
$i \leftarrow 0$
while $executingInvocations[i].end < timestamp\ \&\ executingInvocations[i] \neq NULL$ do
    $addToRamCache(executingInvocations[i])$
    $executingInvocations[i] = NULL$
end while
```

After the update, the simulator will check if the function is present in the RAM cache. If it is, there is no need to subtract the available RAM, it simply removes the function from the RAM cache, adds it to the executing functions list, and registers a warm start. The start latency in this case is 0ms. If the function is not in RAM, it subtracts the necessary memory. If there's not enough memory available, the simulator will try to free memory. It does this by first removing functions from the write queue to the disk and if not enough, by deleting items from the RAM cache. If after this there's still not enough memory, the invocation fails.

If the memory was successfully reserved, the simulator will look for the function in the disk cache and in the network cache. If the function is found in the disk cache, the time to copy the checkpoint from the disk to RAM is calculated. If the function is found in the network cache, the time to download the checkpoint from the network to RAM is calculated. The type of start depends on the times calculated. If the function is neither in the disk nor network caches or if the cold start time is the faster of the three, a cold start will occur. If the time to bring the function from the disk cache is the shortest of the calculated ones, then the function is added to the disk read queue and the simulator will register this invocation as a lukewarm start. If the time to download the function from the network cache is the fastest, then the function is added to the network download queue and the simulator will register this invocation as a remote start.

4.4 Emulator

As we said before, we need an emulator to validate the simulation results. The language we chose to develop this emulator is C because we need a great level of control over the OS memory and good efficiency and C provides us with that. We had to choose a tool to manage the containers. While Docker is the most obvious choice, its checkpoint functionality is riddled with bugs and it is currently not possible to restore a checkpoint into new containers. Because of this, we opted for Podman [23]. Podman allows us to export checkpoints into files and use those files to import the checkpoints into new containers. We communicate with Podman using its REST API [24]. We can use this API inside our C program by using libcurl [25] to build and send the HTTP requests. For our FaaS platform, we use Apache OpenWhisk [26]. We launch the containers with the OpenWhisk images. We can then initialize and run functions by using the OpenWhisk API. We communicate with the API by sending HTTP requests to the containers. Finally, for our network cache, we use MinIO [27]. MinIO is a high-performance object storage. It is also easy to run in a container on a machine which is enough for the scope of this thesis. We communicate with the MinIO server by using the C++ API. The input the emulator receives follows the same format as the simulator. At run time, like the simulator, the user can set several different parameters to use during the emulation:

- **Input file**: The file that contains the workload to be simulated;
- **RAM**: The amount of RAM available (GB);

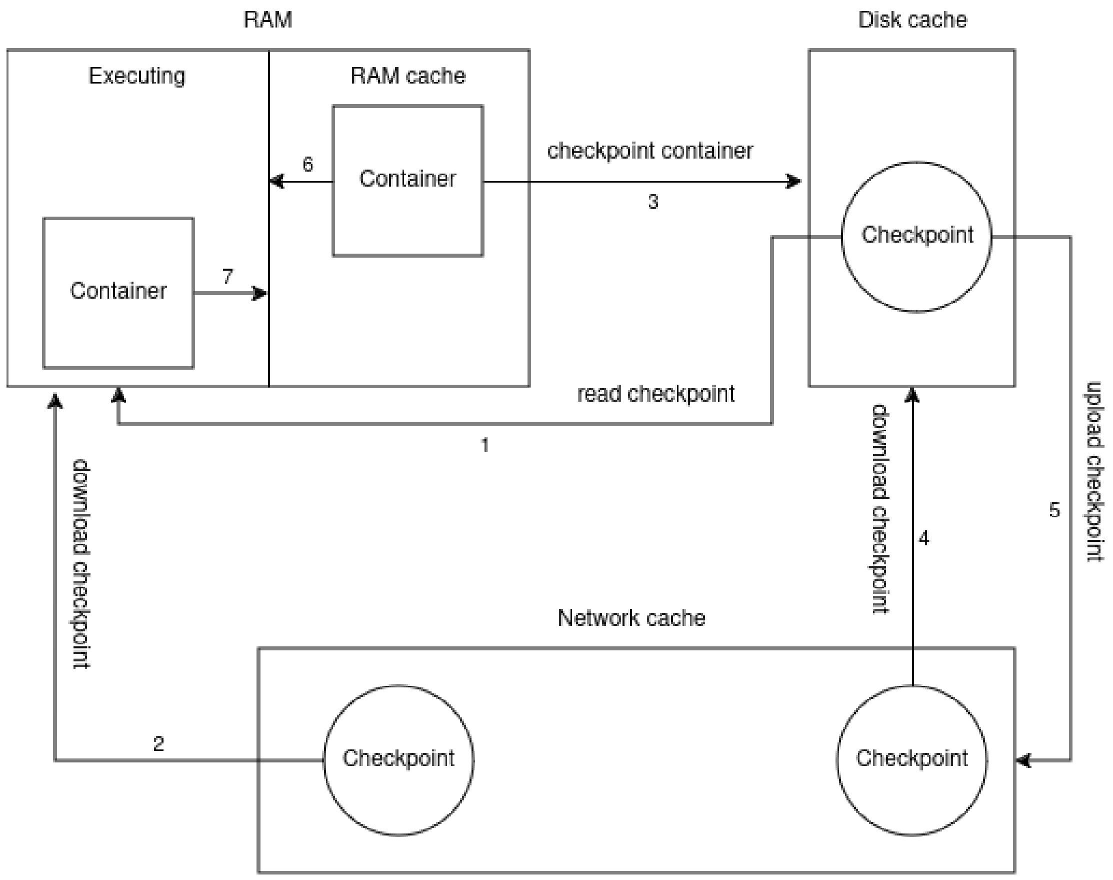

Figure 4.3: Operations between the caching layers in the emulator.

- **Disk memory**: The amount of disk storage available (GB). If it's set to 0, the disk cache won't be used;
- **Threads**: Number of threads allocating invocations;
- **Threshold**: Maximum percentage of RAM occupied by RAM cache, before it starts to get offloaded to the disk cache;
- **Net cache**: Choose whether to use the net cache or not.

In Figure 4.3, we present the operations between the caching layers in the emulator. During this section, when we talk about operations between cache layers and/or the execution layer, we will refer to the arrows of the figure that represent that operation. In the beginning, we create several files occupying a certain memory, write them to disk, and then read them into memory. We also create some checkpoints and restore them. We measure the time these operations take to obtain an estimate of the disk read and write speeds and of the restore checkpoint latency.

Our emulator is composed of several components that deal with different tasks:

Algorithm 4.3: Disk read thread (emulator).

```
1: while $readQueue.size == 0$ do
2:     $waitForSignal()$
3: end while
4: $invocation \leftarrow readQueue[0]$
5: $checkpointFile \leftarrow readFile(invocation.function)$
6: $notifyAllocateThread()$
```

Algorithm 4.4: RAM cache controller (emulator).

```
1: $itemsToRemove \leftarrow []$
2: while $ramCache.occupied/ram.totalMemory > THRESHOLD$ do
3:     $item \leftarrow getNextRamCacheItem()$
4:     $itemsToRemove.append(item)$
5:     $ramCache.remove(item)$
6: end while
7: $i \leftarrow 0$
8: while $itemsToRemove[i] \neq NULL$ do
9:     $item \leftarrow itemsToRemove[i]$
10:    $inDiskCache \leftarrow findInDisk(item.function)$
11:    if $inDiskCache == True$ then
12:        $killContainer(item.containerId)$
13:        $continue$
14:    end if
15:    $inNetworkCache \leftarrow findInNetwork(item.function)$
16:    if $inNetworkCache == True$ then
17:        $killContainer(item.containerId)$
18:        $downloadCheckpointToDisk(item.function)$
19:        $continue$
20:    end if
21:    $createCheckpoint(item)$
22:    $killContainer(item.containerId)$
23: end while
```

- A component that handles the transfers from the disk cache to RAM. This component is notified when the disk read queue is updated. One by one, it reads the necessary files in the disk to memory (arrow 1). When it finishes each read, it notifies the thread that made the request. This component is presented in Algorithm 4.3;
- A component that handles the transfers from the network cache to RAM. This thread is similar to the previous one but downloads files from the MinIO server into a buffer in memory (arrow 2);
- A component for writing to the disk cache. This component is represented by Algorithm 4.4. It periodically checks if the RAM cache threshold has been exceeded. It moves items from the RAM cache to the disk cache until the RAM cache is below the threshold. This component either creates checkpoints (arrow 3) or downloads them from the MinIO server as described above (arrow 4). It's also responsible for freeing disk memory if needed. This is achieved by either deleting the files corresponding to disk cache items or if using the network cache, adding these files to the network cache write queue (the file will be deleted by the next component in this case);
- A component that uploads checkpoint files to the network cache. This thread is notified when an item is added to the network write queue. It uploads the items one by one to the MinIO server (arrow 5). After uploading the files or if the file is already in the network cache it deletes them;
- The main component that contains the loop that reads the invocations one by one. It also keeps track of the time passed since the emulation started. If the invocation's timestamp is later than the current time, this component waits until the proper time is reached. After we convert the text to an invocation data structure, we send it into a thread pool to be processed. If all the allocation threads are occupied, this invocation will have to wait until one of them finishes;
- The component that allocates the invocations. It is made up of a number of threads (selected by the user). When one of the threads is free it grabs a request from the thread pool queue and it starts executing the allocation algorithm.

The invocation allocation is presented in Algorithm 4.5. We will now detail it. The first part is similar to the simulator. It first looks for the function in the RAM cache. If it's present, the invocation will have a warm start. If it's not, we have to reserve memory. If there's not enough memory free at this point, we need to find it. Usually, we wouldn't reach this point due to the thread that checks the RAM cache threshold. But if we're using a small amount of RAM, moving items from the RAM cache to the disk cache can be too slow. So the emulator deletes items in the back of the disk write queue and in the RAM cache if needed. If we manage to reserve the memory, we will then look for the function in the disk and network caches. If we find it, we calculate the time it would take to bring it to RAM from either of the caches. If it's better than a cold start, this invocation will be either a lukewarm or a remote start.

Algorithm 4.5: Allocate invocation (emulator).

```
1: $startType \leftarrow getStartType(invocation)$
2: **if** $startType == "warm"$ **then**
3:     $invocation.containerId \leftarrow getRAMContainerId(invocation.function)$
4: **else**
5:     **if** $ram < invocation.memory$ **then**
6:         **if** $freeRam(invocation.memory - ram) == False$ **then**
7:             $return$
8:         **end if**
9:     **end if**
10:    $invocation.containerId \leftarrow createContainer()$
11:    **if** $startType == "lukewarm"$ **then**
12:        $start \leftarrow getTime()$
13:        $addToReadQueue(invocation.buffer, invocation.function)$
14:        $signalReadThread()$
15:        $waitForRead()$
16:        $restoreCheckpoint(invocation.containerId, invocation.buffer)$
17:        $latency \leftarrow getTime() - start$
18:    **else if** $startType == "remote"$ **then**
19:        $start \leftarrow getTime()$
20:        $addToDownloadQueue(invocation.buffer, invocation.function)$
21:        $signalDownloadThread()$
22:        $waitForDownload()$
23:        $restoreCheckpoint(invocation.buffer, invocation.checkpoint)$
24:        $latency \leftarrow getTime() - start$
25:    **else if** $startType == "cold"$ **then**
26:        $start \leftarrow getTime()$
27:        $startContainer(invocation.containerId)$
28:        $initContainer(invocation.containerId, initFile)$
29:        $latency \leftarrow getTime() - start$
30:    **end if**
31: **end if**
32: $saveLatency(latency)$
33: $runFunction(invocation.containerId, invocation.arguments)$
```

If the function is not in any cache or if it's not beneficial to bring it to RAM, then a cold start will occur. When there's a cold start, a container is created and started. After this, a curl request containing the necessary file is sent to the container to initialize the function. The invocation latency is the time these three operations take. In case of a lukewarm start, a new container is created but instead of starting it, a checkpoint is restored into it. The invocation latency is the duration of the container creation, reading the checkpoint file from the disk and restoring it to the created container. A remote start is similar to the lukewarm, but instead of reading the file from the disk, we download it from MinIO. A warm start doesn't do any of those operations, it simply marks a certain container as in use by that invocation (arrow 6). The execution of the invocation is a curl request containing the arguments to the respective container. When the execution ends, the container is automatically added to the RAM cache (only program logic) (arrow 7). We decided to use the Least recently used policy for the emulator, due to results obtained using the simulator.

4.5 Optimizations

We implemented several optimizations to the emulator to be able to test it using the hardware available to us. Before the main loop, we download the network cache into the disk cache, until it's all downloaded or the disk cache is full. This is so we don't have a period at the beginning of an emulation where there are no lukewarm starts. Checkpointing containers using the Podman API with the export option takes a long time because we're not able to cancel the compression. Checkpointing when done in excess also negatively impacts the other Podman calls, causing them to slow. Because of this, to minimize the number of checkpoint creations, when possible, instead of creating a checkpoint we download it from MinIO into the disk.

Removing containers is another resource-heavy task. Similarly to creating checkpoints, it negatively impacts the rest of the emulator. Due to this, we have a thread that periodically prunes the containers. Pruning the containers is a command that removes every unused container. We have this thread to improve performance because removing containers individually is a costly operation. Instead of removing containers, we kill their process and then this thread will remove every killed container. Unfortunately, this command also kills created containers that have not finished starting up. Because of this, we need to coordinate this thread and the others. Before the thread prunes the containers, it notifies the allocation threads and the disk read thread. If any of these threads are in the middle of starting a container, the prune thread waits. If these threads have been notified, they will wait for the prune thread to be finished before creating a new container. When all threads finish starting their containers, the prune happens.

5 Evaluation

Contents

In this section, we explain how we plan to evaluate our solution. We first expose the questions we are looking to answer. We then define the experiments we will run to answer these questions.

5.1 Research Questions

Our main objective with this work is to reduce function start latency in serverless services. Because of this, the main question we have is whether or not our solution accomplishes that objective. In other words, we want to know if our multi-level caching layers provide advantages over a traditional system that only uses a memory cache. The answer to this question, however, may depend on multiple factors. Because of this, we divide this question into smaller questions:

1. In what situations is restoring the checkpoints from the disk or network beneficial? This means analyzing the caching layer performance with different disk cache sizes and bandwidths and multiple network speeds.

2. Considering the first question, what is the overall impact on a system of the caching layer? How does it affect the latency?

3. Maintaining similar performance, how much RAM can we save by using our algorithm?

By answering these questions, we get an overall idea of the efficiency of our solution and the ability to answer the main question.

5.2 Methodology

To test our solution, we take advantage of the Azure Functions' trace [6]. Since the trace is too long (2 weeks worth of data) and too large (it would require a large cluster to execute it), we downsized the trace. To do so we rely on two operations. The first is setting a start and end minute. Instead of testing a whole day, the period originally provided by the Azure Functions Trace, we can test any number of minutes we find appropriate. We also limit the maximum number of concurrent invocations possible. Each line on the trace contains an invocation with the memory usage, the function ID, the duration, and the timestamp. When we limit the number of concurrent invocations, that means the trace has a maximum of invocations with overlapping timestamps and duration. Because of the function start latency, in reality, more invocations can overlap, but as we have mentioned before, we can set a maximum number of concurrent invocations in the emulator and simulator. In this case, there's an invocation request but the maximum number of concurrent invocations is already being executed, instead of allocating the invocation immediately, the emulator or simulator delays that invocation.

For the simulator, there's no complex setup needed, and it can run on low-end hardware. It's single-threaded and the user only needs to have Go installed. We run it with an Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz and 16GB RAM. For the emulator, we need multiple threads. We use two different VMs in a server. The server has a Corsair 1TB NVMe, Intel(R) Xeon(R) CPU E5-2697 v4 @ 2.30GHz, and 10Gbps Ethernet Intel x540. One of the VMs is running the emulator and the Podman service that receives API calls from it. The other VM is the network cache. It's running a container with the MinIO server image that receives MinIO API calls.

5.3 Simulator Results

To test the simulator, we use a workload with 64 concurrent invocations over 2 hours. In this trace, half of the invocations have the same function. To better observe the results, we ignore these invocations. With these invocations, there would be too many warm starts and the impact of changes in the simulator's settings would be harder to measure. In the last section of the evaluation, we run an experiment including these invocations. Like the emulator, the invocations are delayed if the maximum concurrent functions are being executed. We will see if the simulator is consistent with the emulator and test the impact of changing multiple parameters. Unless referred to otherwise, the parameters used are:

- RAM size: 32 GB;
- Disk size: 50 GB;
- Cold latency: 250 ms;
- Warm latency: 0 ms;
- Restore checkpoint latency: 100 ms;
- Disk read speed: 4 GB/s;
- Net bandwidth: 10 Gbit/s;
- Cache policy: least recently used;
- Checkpoint latency: 350 ms.

5.3.1 Caching Policies

Here we will test the three disk caching policies. In these tests, we set a very low network bandwidth to maximize the possible lukewarm starts. We also use a lower amount of disk cache, 25 GB, so the policy is more influential.

In Figure 5.1, we observe that LRU is the best policy out of the three since it grants us the most amount of lukewarm starts and so has the least overall start latency. In Table 5.1 we have the exact numbers of lukewarm and cold starts.

Policy	lukewarm	cold
FIFO	24801	97921
LRU	51611	70831
LFU	36348	86338

Table 5.1: Number of starts for each cache policy in the simulator.

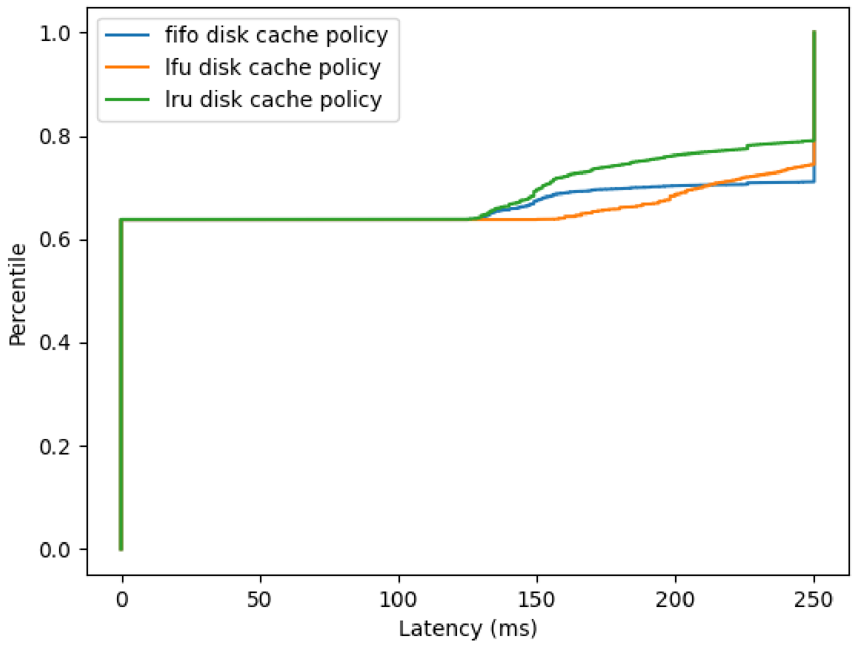

Figure 5.1: Simulator latency cumulative distribution function for each caching policy.

5.3.2 Changing cold start and restore latencies

Here we experiment with different cold start latencies. These latencies are a big part of our algorithm, as they are included in the equation to determine what type of start to use. We determined by experimenting with the emulator that usually, the longer the cold start, the longer the checkpoint restore operation. Because of this, we alter them linearly. In Table 5.2 we observe that with higher cold start times, our algorithm has a bigger impact. Just by changing the cold latency from 250 ms to 500 ms while raising the restore latency from 100 ms to 200 ms, we almost get 3x less cold starts. Both remote and lukewarm starts increase when raising these values. In Figure 5.2, we have the CDF for the different latencies. This figure shows that the curve cause by lukewarm starts increases and is more gradual with the latency increase. This is because there's a broader range of lukewarm latencies possible.

cold latency (ms)	restore latency (ms)	warm	lukewarm	remote	cold
250	100	216203	66662	26889	28810
500	200	214997	75210	37577	10780
750	300	214014	77521	39101	7928
1000	400	213679	80217	40535	4133

Table 5.2: Number of starts for different values of cold start and restore checkpoint times.

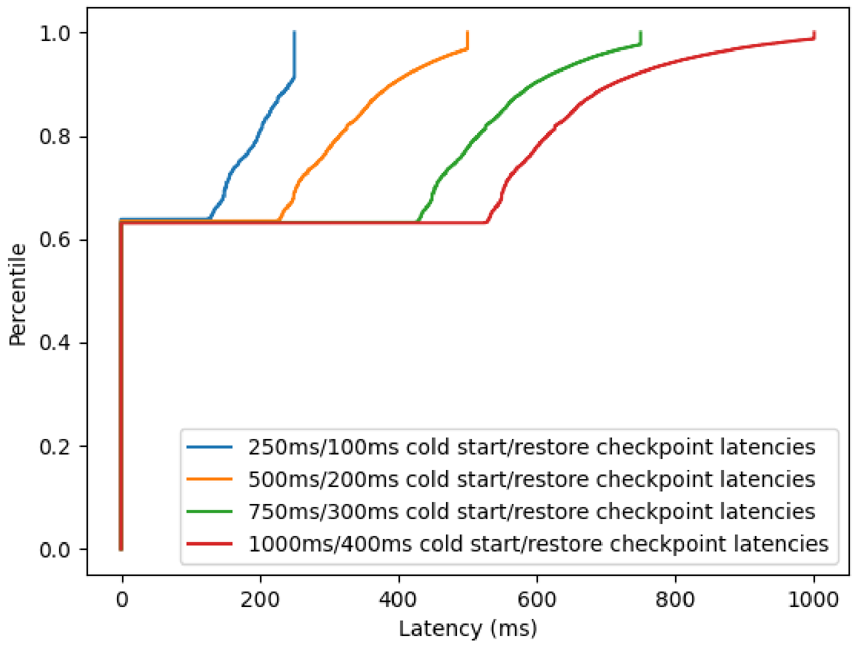

Figure 5.2: Simulator latency cumulative distribution function for different cold start and restore checkpoint latencies.

5.3.3 Changing disk read speed

With this experiment, we analyze the difference changing the disk read speed has on the system. In Figure 5.3 we have the invocation start latency CDF obtained. We can observe that the disk read speed also has a big impact on the system's efficiency. By increasing the read speed, each requested disk read gets executed faster, updating the time to read and opening the way for more lukewarm starts. In Table 5.3, we show the exact number of each start type.

As expected from analyzing Figure 5.3, the number of lukewarm starts increases when increasing the disk read speed. The remote starts slightly decrease because, with a higher disk bandwidth, some starts that would be faster as a remote are faster as a lukewarm. We can verify this by examining the average lukewarm and remote start latency. For 0.5GB/s, the average lukewarm is 250ms, the same duration as a cold start, and the average remote is 218ms. For 4.5GB/s, the average lukewarm is 168ms and the average remote is 215ms.

5.3.4 Changing network bandwidth

With this experiment, we aim to analyze the impact of the network bandwidth in our system. By looking at the CDF obtained in Figure 5.4, we can see that it's very similar to the one when changing disk

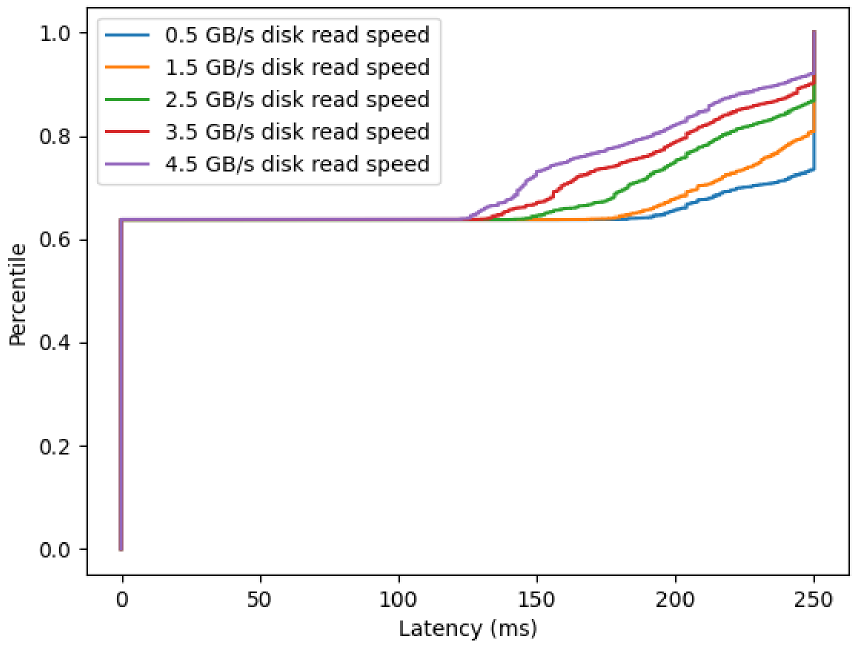

Figure 5.3: Simulator latency cumulative distribution function for different disk read speeds.

Disk read speed (GB/s)	warm	lukewarm	remote	cold
0.5	215793	146	33659	88966
1.5	215829	30243	29253	63239
2.5	215909	50407	28289	43959
3.5	216011	62627	27377	32549
4.5	216172	69920	26418	26054

Table 5.3: Number of starts for different values of disk read speed.

read speeds. The same that happens to lukewarm starts by changing the disk read speed, happens to remote starts by changing the network bandwidth. The more bandwidth the faster a download request is satisfied and the more remote starts can happen.

In Table 5.4, we can see that with the increase of remote starts, the cold starts and lukewarm starts decrease. This happens for the same reason as the opposite. More remote starts will be predicted to be faster than lukewarm and cold starts. We confirm this by again looking at the average lukewarm and remote start latency. For 5Gbit/s, the average lukewarm is 175ms and the average remote is 229ms. For 25Gbit/s, the average lukewarm is 164ms and the average remote is 172ms.

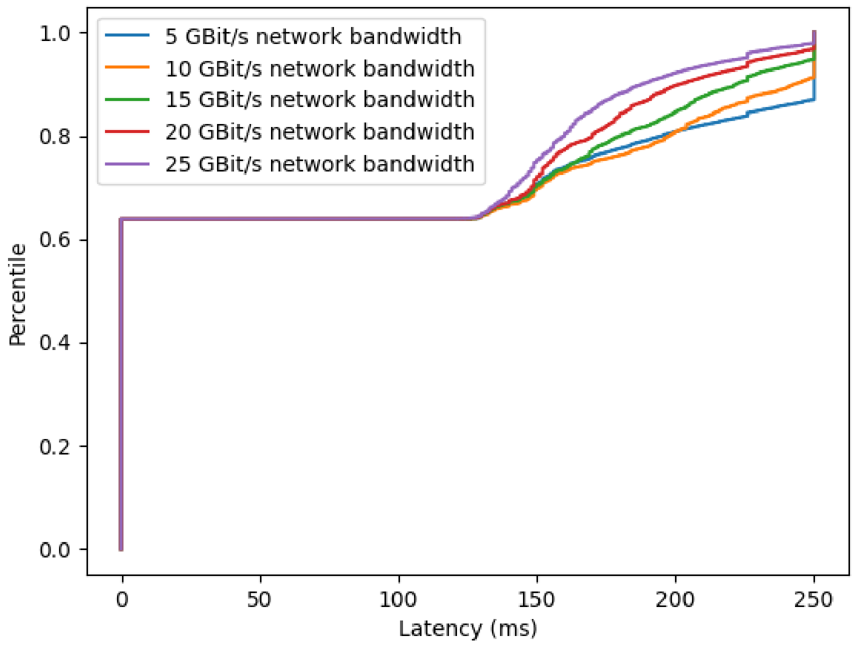

Figure 5.4: Simulator latency cumulative distribution function for different network bandwidths.

Network bandwidth (GBit/s)	warm	lukewarm	remote	cold
5	216096	77902	649	43917
10	216244	66662	26889	28810
15	215909	61694	43247	17379
20	216375	57271	54170	10748
25	216457	53037	61940	7130

Table 5.4: Number of starts for different values of network bandwidth.

5.4 Emulator Results

Although at the beginning of each emulation, we measure the bandwidths, the restore checkpoint latency, and the cold start latency, these values fluctuate between executions and the bandwidths specifically are lower than the optimal ones provided in the hardware specifications. For the sake of maintaining consistency throughout the experiment, we fixated average values to use on the emulator invocation start latency calculations:

- Disk read speed: 1.25 GB/s;
- Network speed: 3.59 Gbit/s;
- Cold latency: 1196 ms;
- Restore checkpoint latency: 999 ms.

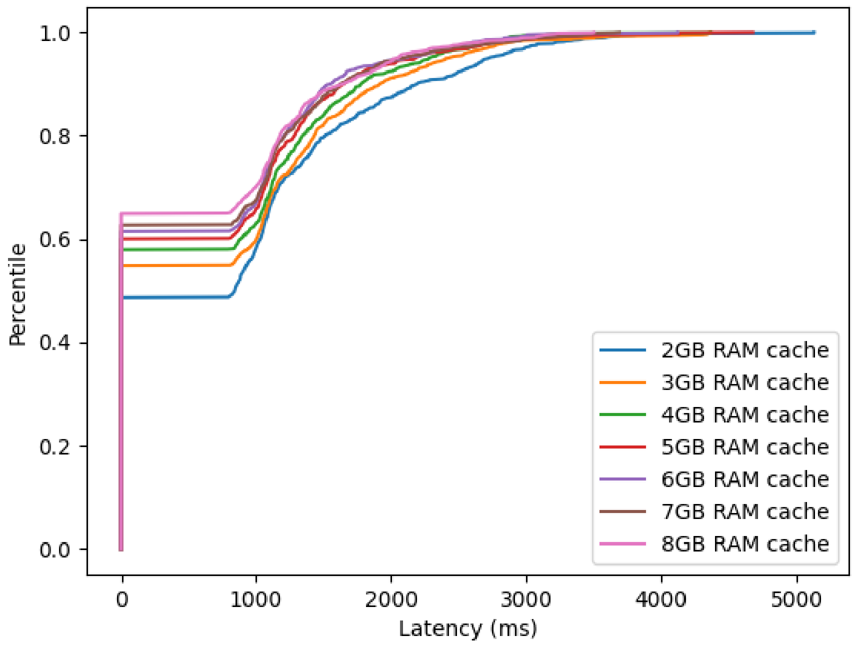

Figure 5.5: Emulator latency cumulative distribution function (changing RAM).

Since the restore checkpoint latency and the cold latency are close in the server, and our bandwidths aren't high, there's a high chance that reading or downloading a checkpoint and then restoring it won't be beneficial, so we won't obtain the best results. But it can still be used to check the validity of the simulator. To prove that better results can be obtained in with different hardware, we ran a few tests on a faster CPU with less cores (Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz). Using the same runtime we use in our emulations, we measure an average cold start of 950ms and restore latency of 646ms. With a heavier runtime, we measured an average cold start of 3850ms and an average restore of 2578ms.

To test the emulator, we use a workload with 4 concurrent invocations over 5 minutes. The emulation can last longer than 5 minutes, when max concurrency is reached, it delays the next invocation. We will test the impact of changing the size of the RAM and of the disk. Note that due to the slow network bandwidth and high restore latency time, no remote starts occur in these experiments.

5.4.1 Changing RAM size

This experiment shows us the impact that altering the RAM size has on the system. We used a 20GB disk cache. In Figure 5.5 we have a cumulative distribution function (CDF) of the invocation start latency for each RAM size tested. Every function starts with a vertical line at 0 ms latency. This line is caused by the warm starts, which have minimal latency. As expected, the bigger the RAM, the higher the number

RAM (GB)	warm	lukewarm	cold
2	553	426	157
3	623	367	146
4	658	340	138
5	682	324	130
6	698	319	119
7	712	308	116
8	737	290	109

Table 5.5: Number of starts when changing RAM size in the emulator.

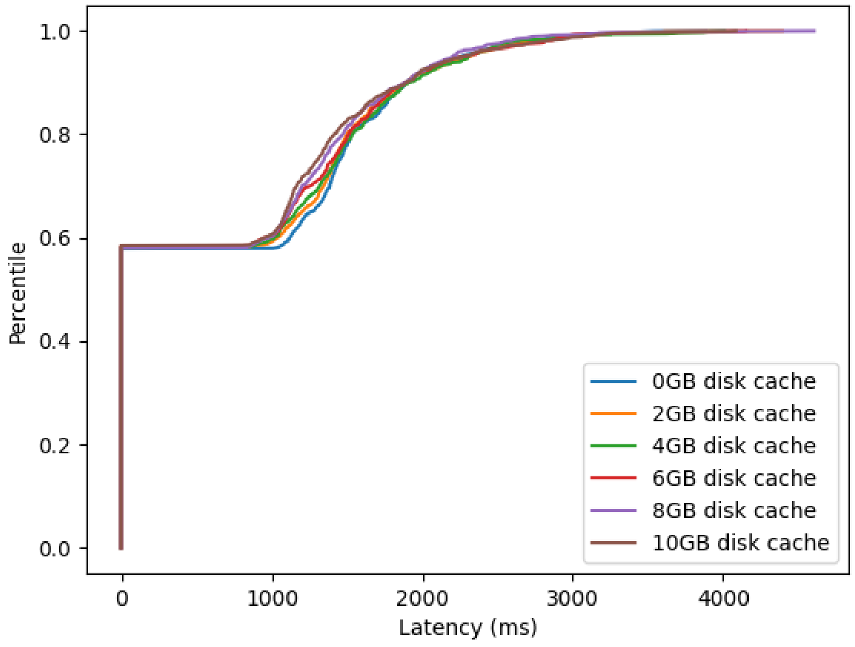

Figure 5.6: Emulator latency cumulative distribution function (changing disk).

of warm starts and the lower the overall start latency. We have the exact number of starts in Table 5.5. The higher amount of RAM also means a lower number of lukewarm starts (since warm starts always take precedence). From this, we can conclude that the impact of our caching layers is bigger the less RAM there is available.

5.4.2 Changing disk cache size

This experiment shows us the impact that altering the disk cache size has on the system. We used 4GB RAM size. In Figure 5.6 we have a CDF of the invocation start latency for each disk size tested. We can see that the higher the disk cache, the more invocations with a start latency from 1100ms to 1700ms. This is more visible when we zoom in on the CDF, in Figure 5.7. This happens because of the

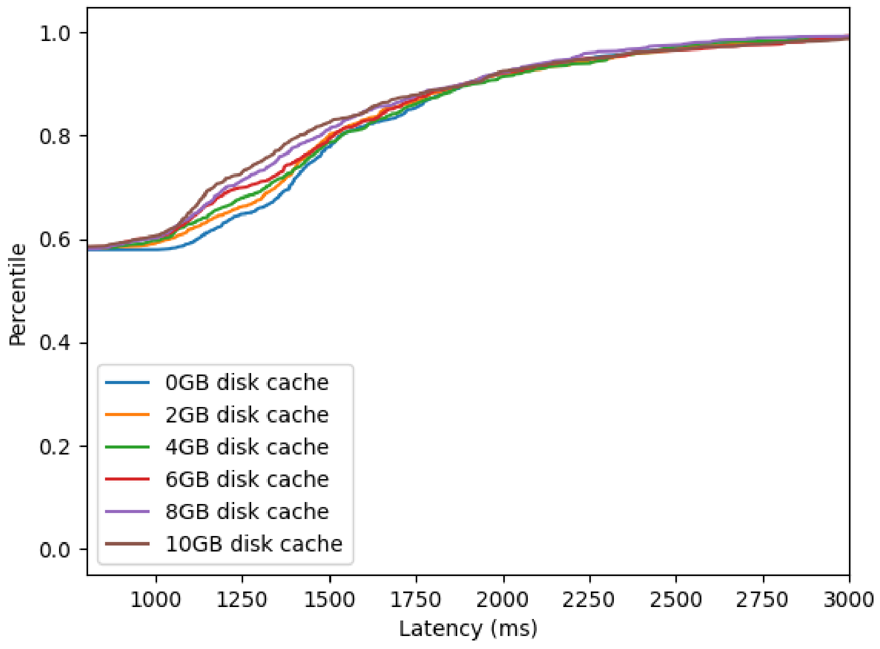

Figure 5.7: Emulator latency cumulative distribution function (changing disk) zoomed in.

higher incidence of lukewarm starts. An unexpected result is that the bigger the disk cache, the more invocations have a very high start latency. To understand this, we determine the average start latency of a lukewarm and a cold start. We looked at the case where we don't use disk cache and the case where we use the most. With no disk cache, the average cold start takes 1673ms and there are no lukewarm starts. We then examined the 10 GB cache execution and found that the average lukewarm start is 1241 ms and the average cold start is 1811 ms. This means that the problem is not the lukewarm latency prediction but instead the impact that reading files from disk and restoring checkpoints has on the rest of the system. We suspect that this is related to disk and CPU usage and suspect that with better hardware this problem wouldn't be so pronounced. Overall, as expected, the higher the disk size the bigger the number of lukewarm starts and consequentially a lower number of cold starts. You can observe this in Table 5.6.

Disk (GB)	warm	lukewarm	cold
0	657	0	479
2	660	41	435
4	660	81	395
6	663	124	349
8	659	158	319
10	664	192	280

Table 5.6: Number of starts when changing disk size in the emulator.

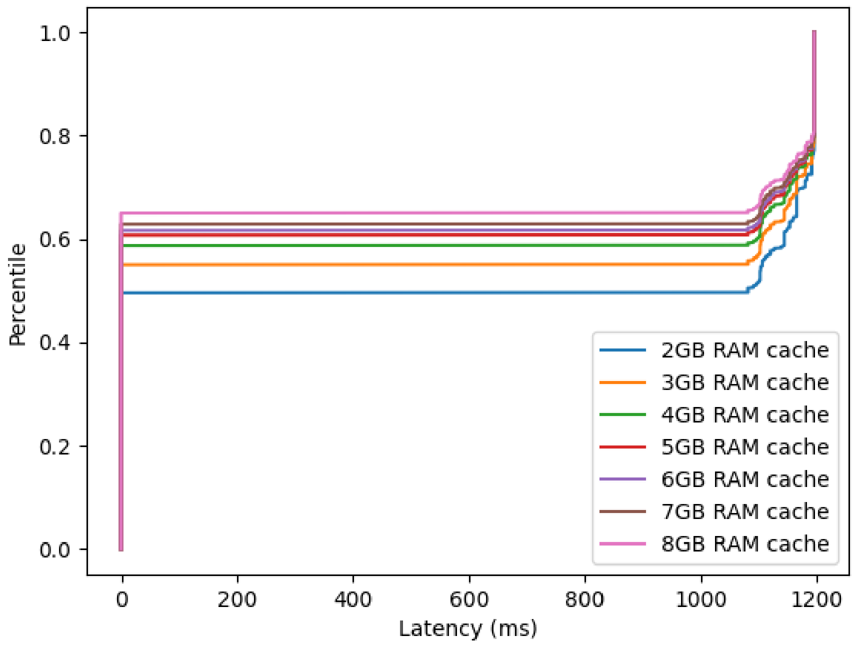

Figure 5.8: Simulator latency cumulative distribution function with emulator settings (changing RAM).

5.5 Comparing the simulator and the emulator

For this experiment, we use the emulator's experiments' input and fixed values in the simulator. We then compare the results obtained here with the results of the emulator experiments. It is expected for the simulator to have lower function start latency than the emulator. This is because the simulator doesn't face the performance issues that the emulator does, which we talked about previously in this chapter. More specifically, the cold start times are highly variable in the emulator but in the simulator, they're a set value. What we expect to find is that although the latency may be different, the results follow the same logic.

5.5.1 Changing RAM size in the simulator

In Figure 5.8, we have the CDF for when we change the RAM size in the simulator. We can compare it with the emulator's CDF in Figure 5.5. Like in the emulator, the percentage of invocations with a 0 ms start latency increases with the RAM size. We then have no invocations with a latency smaller than around 1100 ms. All the invocations between 1100 ms and 1200 ms latency are lukewarm starts and all the functions with 1200 ms latency are cold starts. We know this because the cold start time in the simulator is a set value. To compare the two CDFs, the most important part is the curve caused by the

RAM (GB)	warm	lukewarm	cold
2	563	311	262
3	625	266	245
4	667	238	231
5	690	216	230
6	700	209	227
7	714	197	225
8	738	177	221

Table 5.7: Number of starts when changing RAM size in the simulator using the emulator settings.

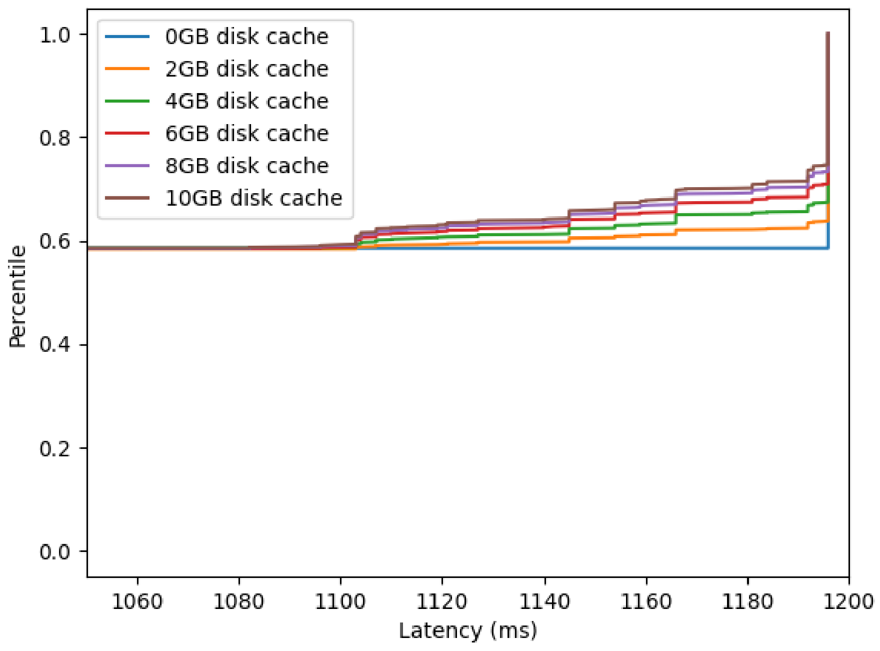

Figure 5.9: Simulator latency cumulative distribution function with emulator settings (changing disk).

lukewarm starts. This curve is present in both graphs in the same range but in the emulator's graph, it continues because the cold start times vary. In Table 5.7, we have the number of starts of each type. The number of warm starts is extremely similar to the corresponding experiment in the emulator, as we can observe by comparing with Table 5.5 . The number of lukewarm starts is lower in the simulator. Because of the higher invocation start latency in the emulator, the emulation takes longer than the time simulated, having more invocation delays. Because of these delays, the disk in the emulator has more time to complete each transfer, which causes a higher probability of lukewarm starts.

Disk size(GB)	warm	lukewarm	cold
0	664	0	472
2	662	62	412
4	664	101	371
6	663	142	331
8	663	169	304
10	664	183	289

Table 5.8: Number of starts when changing disk cache size in the simulator using the emulator settings.

5.5.2 Changing disk size in the simulator

In Figure 5.9, we have the CDF for when we change disk size in the simulator. We can compare it with the corresponding emulator's experiment's CDF in Figure 5.7. We selected a range so it's easier to see the lines. If we look at the latency range we chose here in the emulator's graph, we observe that the lines follow the same hierarchy. The emulator's curve continues due to the problems we specified before. By looking at this experiment's distribution of starts, in Table 5.8, and the emulator's distribution which is in Table 5.6, we see that the number of lukewarm and cold starts is similar to the emulator's. By doing these comparisons, we observe that the simulator is, despite some differences from the emulator, a sufficient way to test our solution. We can also reiterate the idea that with more RAM available our solution causes less of an improvement, and the contrary is true for disk size.

5.6 Comparing performance with traditional system

With this experiment, we aim to find the overall impact of our system, when compared to a traditional system, that only uses a memory cache. We use the simulator with the settings mentioned above. We first experiment with no disk and network caches, with 32GB RAM. We also want to find the amount of RAM needed in a traditional system to reach similar performance to our system. We test with the same input we have been using and also without removing the function that is invoked. We do this second set of tests to compare our system's impact when there's less variety in the invocation functions.

System	RAM (GB)	average latency (ms)
Traditional	32	91
Caching Layers	32	73
Traditional	45	72

Table 5.9: Average latency with different systems.

As you can see in Table 5.9, our system improves the average invocation start latency from 91ms to 73ms. This is a 19.7% improvement. If we wanted to get this performance increase without using our system, we would have to increase the RAM size from 32 GB to 45 GB. This means that while maintaining performance, our system saves 28.9% RAM. In Figure 5.10, we see that with our system,

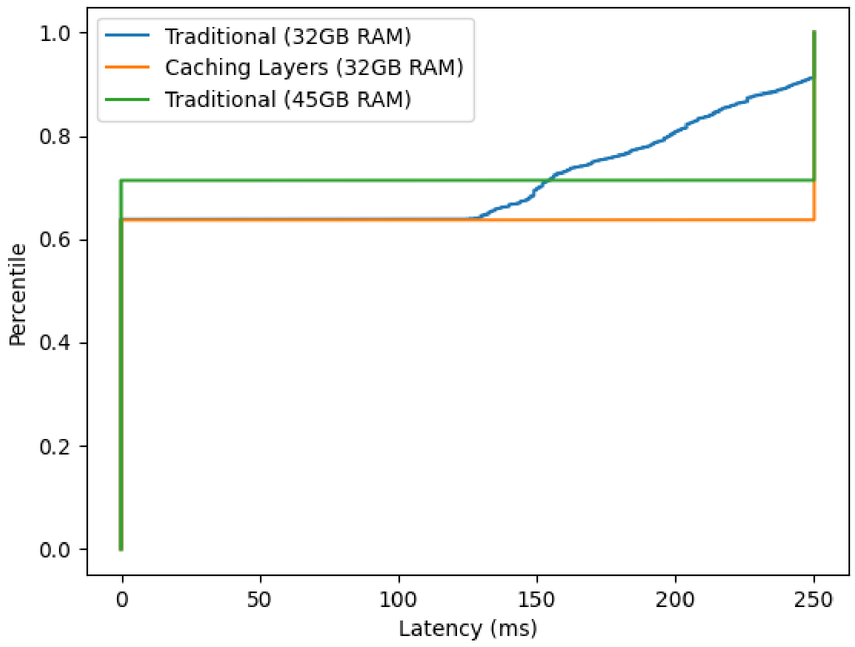

Figure 5.10: Simulator latency cumulative distribution function comparing caching layers system and traditional (only memory cache) system.

System	RAM (GB)	warm	lukewarm	remote	cold
Traditional	32	215682	0	0	122882
Caching Layers	32	216203	66662	26889	28810
Traditional	45	241649	0	0	96915

Table 5.10: Type of start distribution with different systems.

the function start latency has a gradual rise from around 125 ms (shortest lukewarm start) until 250 ms (cold start). The traditional system has a sudden jump from 0 ms (warm starts) to 250 ms(cold starts). Although the traditional system with 45 GB RAM has similar performance, we can see in Table 5.10, that it still has more cold starts than our system. It simply reaches similar average latency by having more warm starts.

System	RAM (GB)	average latency (ms)
Traditional	32	55
Caching Layers	32	44
Traditional	45	43

Table 5.11: Average latency with different systems (without removing invocations).

In Table 5.11, we show the results when we don't remove the invocations which are all of the same function. These invocations are about half of the workload. Our system has a similar performance

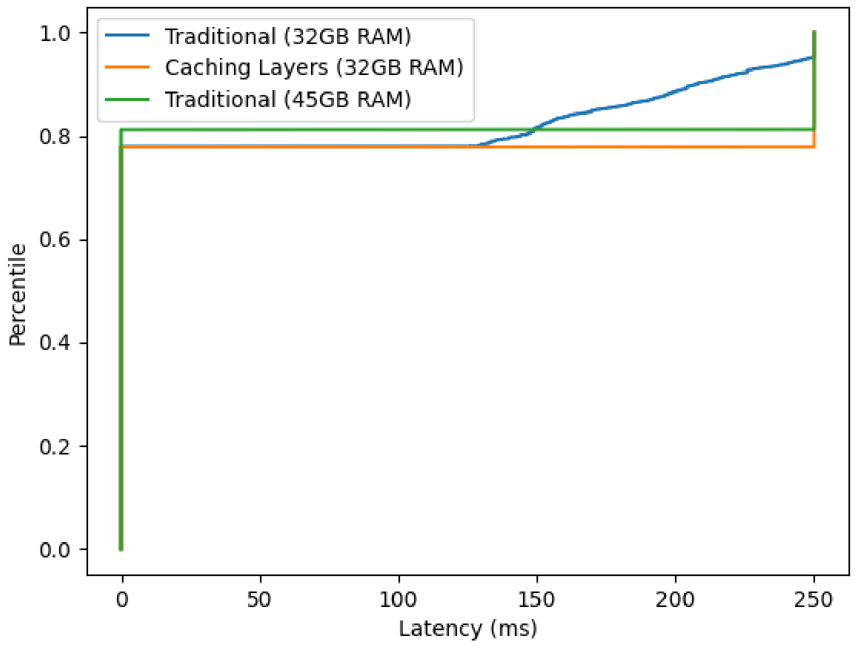

Figure 5.11: Simulator latency cumulative distribution function comparing caching layers system and traditional (only memory cache) system, with more repeated invocations.

System	RAM (GB)	warm	lukewarm	remote	cold
Traditional	32	492451	0	0	140521
Caching Layers	32	493251	79989	29887	29845
Traditional	45	513670	0	0	119302

Table 5.12: Type of start distribution with different systems (without removing invocations).

percentage increase, 20%. The memory we can save while maintaining the performance is the same for both cases. In Figure 5.11, we have the latency CDF for this experiment. This is similar to Figure 5.10. The main differences are that there are more invocations with 0 ms latency (more warm starts), and the gradual rise caused by lukewarm covers a smaller percentile. In Table 5.12, we have the distribution of start types. Again, our system drastically reduces cold starts.

For this experiment, we would obtain different results depending on the settings chosen. As we saw before, disk size, cold start latency, restore latency, disk read speed, and network bandwidth all have a significant impact on our system.

5.7 Summary

In this section, we presented the questions we consider crucial to evaluate our algorithm. We also provided the methodology we used to answer those questions. Our first question was in what situations is restoring the checkpoints from the disk or network beneficial. To answer this, we need to look at the experiments where we tested with different values of the same variable. The answer this question, we need to estimate the cold start time, the restore latency, the disk read speed, and the network bandwidth. For our prediction, we calculate the time it would take to read or download a checkpoint containing the function, taking into account other pending disk and network operations. We then see if this value is smaller than the cold start. As a rule of thumb, the more it takes to cold start a function, the more difference there is between the cold start and the time to restore a checkpoint with that function. Due to this, if a function has a long cold start, there's a higher chance that it's worth restoring it from the disk or network. With a faster disk read speed and faster network, fewer cold starts occur. Also, the more invocations received concurrently, the more bandwidth is needed to maintain the caching layers' performance.

To answer the second question, which was to find the overall impact on a system of the caching layer, we analyze the comparisons with a traditional system. We see that our system bolsters around a 20% average latency reduction. Note that as we referred before, this value can increase or decrease depending on the settings used. The third question was to find how much RAM we can save without losing performance. We want to find this because we use storage for our caching layers and storage is cheaper than RAM. This means that a provider could provide the same performance and have less costs. To answer the question, we analyze the same experiment. We determine that for a system without the proposed caching layers to reach the same performance, a 28.9% RAM increase would be needed. Again, this value can change depending on the settings used in our system. After answering these three questions, we can answer our main question. We determine that our system does reduce function start latency in serverless services.

6 Conclusion

Contents

6.1 Achievements

In the first stage of the work, we identified the tasks necessary before starting to implement a system. We identified a problem currently present in real-world services. More specifically, the high function start latency in serverless services. We then analyzed works related to solving this problem and identified what they were missing. We found that a lot of them were either scheduling or prediction-based and so, can be used together with our work. The one most similar to ours did use disk storage but did not study different disk bandwidths and how they can impact the feasibility of a disk cache.

We then designed an algorithm to solve the problem we identified. We described the functionality of the caching layers and of the simulator and emulator we used to test it. We executed experiences to answer our research questions and found that our system managed to reduce serverless function start latency, by approximately 20% on average, with the specifications we used in the simulation. Alternatively, one could keep the same performance and by using our proposed caching layers, save 28.9% RAM.

6.2 Future Work

There are some factors we didn't address in this thesis. One of them is that the function start time prediction is not perfect in real systems. We assume that the disk and network bandwidths remain constant but this is not always true. It is necessary to determine a method to capture changes in disk and network bandwidths during the system's execution. Another important aspect of finding the best start type is predicting the cold start and restore checkpoint latencies. We simply used average values. For our algorithm to be implemented in a real system, a method to accurately calculate these is needed.

Another factor is that we only tested our system with one worker node. In a real system, there's a scheduler that chooses which node to send the invocation to. We could take advantage of our network cache and propagate popular checkpoints to some nodes. We could keep a list of the most popular functions and when a node is launched, populating its disk cache immediately with checkpoints containing their runtime.

Finally, we could have base checkpoints that work for more than one function. The way this would work is that for functions that share resources, such as libraries or the same coding language and version, they could use a base checkpoint. In this case, we would propagate the base checkpoint to all nodes and then only fetch the specific part for each function when needed. This would make remote starts faster because we would download a smaller file. By doing this, we could support a high number of functions in our disk and network caches by using a low amount of storage.

Bibliography

[1] J. Wells and J. Coulter, “Research computing at a business university,” 07 2019, pp. 1–5.

[2] “Serverless computing - aws lambda - amazon web services,” https://aws.amazon.com/lambda/, Last accessed on 2023-05-28.

[3] “Azure functions - serverless functions in computing,” https://azure.microsoft.com/en-us/products/functions, Last accessed on 2023-05-28.

[4] “Cloud functions — google cloud,” https://cloud.google.com/functions, Last accessed on 2023-05-28.

[5] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, “Firecracker: Lightweight virtualization for serverless applications,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/agache

[6] M. Shahrad, R. Fonseca, I. 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)*. USENIX Association, Jul. 2020, pp. 205–218. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/shahrad

[7] A. Mahgoub, E. B. Yi, K. Shankar, S. Elnikety, S. Chaterji, and S. Bagchi, “{ORION} and the three rights: Sizing, bundling, and prewarming for serverless {DAGs},” in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 303–320.

[8] R. B. Roy, T. Patel, and D. Tiwari, “Icebreaker: Warming serverless functions better with heterogeneity,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 753–767.

[9] “Xen project,” https://xenproject.org/, Last accessed on 2023-05-28.

[10] "Qemu," https://www.qemu.org/, Last accessed on 2023-05-28.

[11] "Docker: Accelerated, containerized application development," https://www.docker.com/, Last accessed on 2023-05-28.

[12] "The container security platform — gvisor," https://gvisor.dev/, Last accessed on 2023-05-28.

[13] A. Regalado, "Who coined 'cloud computing'?" 2011, https://www.technologyreview.com/2011/10/31/257406/who-coined-cloud-computing/, Last accessed on 2023-05-28.

[14] A. Fuerst and P. Sharma, "Faascache: keeping serverless computing alive with greedy-dual caching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 386–400.

[15] "Samsung cxl ssd," https://www.tomshardware.com/news/samsung-memory-semantic-cxl-ssd-brings-20x-performa
Last accessed on 2024-05-17.

[16] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Hermod: principled and practical scheduling for serverless functions," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 289–305.

[17] M. Abdi, S. Ginzburg, C. Lin, J. M. Faleiro, I. Goiri, G. I. Chaudhry, R. Bianchini, D. S. Berger, and R. Fonseca, "Palette load balancing: Locality hints for serverless functions," in *Proceedings of the 18th European Conference on Computer Systems (EuroSys)*. ACM, May 2023. [Online]. Available: https://www.microsoft.com/en-us/research/publication/palette-load-balancing-locality-hints-for-serverless-functions/

[18] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 559–572.

[19] L. Ao, G. Porter, and G. M. Voelker, "Faasnap: Faas made fast using snapshot-based vms," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 730–746.

[20] P.-M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach," *arXiv preprint arXiv:1903.12221*, 2019.

[21] R. Fares, B. Romoser, Z. Zong, M. Nijim, and X. Qin, "Performance evaluation of traditional caching policies on a large system with petabytes of data," in *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*. IEEE, 2012, pp. 227–234.

[22] P. Dymora and A. Paszkiewicz, “Performance analysis of selected programming languages in the context of supporting decision-making processes for industry 4.0,” *Applied Sciences*, vol. 10, no. 23, p. 8521, 2020.

[23] “Podman,” https://podman.io/, Last accessed on 2024-05-13.

[24] “Podman rest api,” https://docs.podman.io/en/latest/_static/api.html, Last accessed on 2024-05-13.

[25] “libcurl,” https://curl.se/libcurl/, Last accessed on 2024-05-13.

[26] “Apache openwhisk,” https://openwhisk.apache.org/, Last accessed on 2024-05-14.

[27] “Minio,” https://min.io/, Last accessed on 2024-05-13.