ASSESSING SCALABILITY AND PERFORMANCE ISOLATION OF LIGHTWEIGHT VIRTUALIZATION SYSTEMS

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

Andrej Velichkovski (Student id: 10835521)
Supervisor: Dr. Pierre Olivier
Department of Computer Science

2023

Contents

Word Count: 10925

List of Figures

Abstract

The popularity of cloud computing has been vastly increasing in the last couple of years. Cloud environments rely on the concept of virtualization. Lightweight virtualization systems were created to resolve the problems with traditional heavy virtual machines. They are known to have better performance characteristics than traditional VMs. Two of the most popular lightweight virtualization systems are containers and unikernels.

In this project, we explore the performance of Docker, one of the most popular container engines, and Unikraft, a novel open-source unikernel development kit. We focus on assessing the scalability and performance isolation of these two systems. In addition, we create a benchmark framework to prototype and evaluate new experiments quickly.

In the first performance evaluation part, by incrementally launching many instances on the same machine, we identified different scalability bottlenecks in both Docker and Unikraft, which led to decreased performance. These bottlenecks include a linear increase in Docker container boot time and a network bottleneck in the Unikraft libraries.

In the second performance evaluation part, we investigate the effect of different misbehaving virtualization instances on other well-behaved user applications. We identified different attackers, which led to an 800% slowdown for a user application running in a Docker container. In addition, the effect of different file system attackers was explored, and several performance bottlenecks were identified.

Declaration

No portion of the work referred to in this report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

I want to thank my project supervisor, Dr. Pierre Olivier, for their continuous support and guidance and for giving me many learning opportunities throughout the work on this project. Furthermore, I am thankful for the support from the Unikraft open-source community in helping me resolve various issues while testing and analyzing Unikraft.

Finally, I am very grateful to my family and friends for their help and support throughout my studies at the University of Manchester.

Chapter 1

Introduction

1.1 Motivation

The popularity of cloud computing has been vastly increasing in the last couple of years. As a result, the most significant cloud providers generate over 200 billion dollars annually [30]. Due to the increasing popularity, more and more companies are switching towards using public cloud providers instead of managing their hardware infrastructure. In addition, using cloud environments for deploying software provides many benefits for users. For example, users can deploy their software products anytime without worrying about buying, installing, and maintaining many hardware devices. Furthermore, users can always scale their usage based on their needs, whether renting new hardware devices or stopping the rent for devices no longer needed.

Cloud environments rely on the concept of virtualization. For years, virtual machines were the central concept of virtualization, running the cloud environments [8]. However, due to their high overhead, the focus has been on more lightweight virtualization systems in the last few years. Containers and unikernels are the most popular lightweight virtualization systems. Lightweight virtualization systems outperform traditional systems in all scenarios. Due to the significant performance benefits of lightweight virtualization systems, they are becoming the preferred tools in cloud environments.

Performance is the most crucial characteristic of software systems. Since all engineers aim to maximize their software's performance, the overhead of any virtualization system should be minimized. Motivated by this, we explore the performance of lightweight virtualization systems in a more realistic cloud environment. We use different types of applications to simulate different realistic scenarios, which are most likely to be happening every day in cloud environments.

1.2 Project Aims

This project consists of two well-connected performance analyses, Assessing Scalability and Performance Isolation.

The primary aim of the first part is to observe how different virtualization systems behave in realistic large-scale cloud environments. Therefore, the three main aims of this part are:

- Establish a model for benchmarking different virtualization systems on a larger scale to simulate realistic environments.
- Use this model to identify scalability bottlenecks and issues.
- Generate analytics from the model to decide which virtualization systems are best in different scenarios.

The second part aims to understand the impact two or more virtual system instances might have on each other. Similarly to the experiment above, we aim to:

- Establish a model for benchmarking the impact of one system on another system running simultaneously.
- Generate analytics on how misbehaving systems impact well-behaved systems.
- Use these analytics to detect potential bottlenecks in the systems.

1.3 Report Structure

This report is organized into six chapters. In the first chapter, we introduce the motivation behind the study performed and provide our work's main aims and objectives.

The second chapter covers the background knowledge for our project and describes the internal designs and architectures of the systems we benchmark. The third chapter overviews our benchmark process, including the software system used and the guidelines for benchmarking the systems.

This report analyzes two performance elements critical for building suitable cloud environments. In the fourth chapter, we discuss the first element, assessing scalability. Moreover, in the fifth chapter, we look at performance isolation, the second element we analyze.

Finally, in the sixth chapter, we conclude our work, describe the limitations of our analysis, and look at future work that might be performed on the topic of our study.

1.4 Literature Review

Many engineering teams are interested in creating scalable systems. However, only a few experiments are performed to examine specific systems' scalability. Typically, authors focus on one particular system and explore the issues in the system in more detail. For example, *Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor* [24] explores three different issues impacting the Xen Hypervisor scalability. The authors implement changes that improve the scalability of the hypervisor's boot and destruction time. *An Analysis of Linux Scalability to Many Cores* [2] looks at the Linux kernel's scalability. The authors propose application and kernel code changes to improve the system's scalability. *Opportunistic Spinlocks: Achieving Virtual Machine Scalability in the Clouds* [12] analyses the scalability of execution time on virtual machines from different cloud providers. The paper suggests a new type of spinlock that resolves various scalability bottlenecks in VM scalability. However, the paper only looks at one type of benchmark: the time required to compile the Linux kernel. Most research papers on scalability only analyze one system or benchmark in more detail. Our project aims to test different systems and user applications and simulate more realistic user behaviors.

Performance Isolation is a more popular research topic. Different authors have performed performance isolation experiments with containers [42, 43] and virtual machines [6, 9]. However, due to the popularity of containers, most of the new research focuses on the performance isolation of containers. In our report, we try to extend the most common performance isolation benchmarks for containers to unikernels and discuss the differences.

Significant work has also been done on improving the performance isolation of containers through unique mechanisms. For example, *PINE: Optimizing Performance Isolation in Container Environments* [14] suggests a mechanism to allocate storage resources to containers based on the performance behaviors of the container. Similarly, *Characterizing and Optimizing Kernel Resource Isolation for Containers* [41] offers a Pareto-based identification of misbehaving containers and allocation of resources based on this identification. Both of these papers suggest approaches that improve the performance isolation of containers. However, they add certain performance overheads to the system. Furthermore, significant engineering effort is required for their implementation.

Chapter 2

Background

2.1 Virtualization

Virtualization is one of the main drives of cloud computing. Virtualization refers to abstracting hardware resources into virtual software versions that are easier to use. One of the most common forms of virtualization is virtual machines. Cloud providers can rent the same hardware machine to several consumers simultaneously using virtual machines.

Virtual machines are the perfect tool for cloud computing. By using virtual machines, the consumer does not know anything about the other users of the same hardware device. Furthermore, as each user can run different operating systems, they do not limit the user's actions. One of the virtual machines' most crucial benefits is their isolation. Each virtual machine running on the same hardware is isolated from the other virtual machines, and any security risk or attack is limited to only one instance.

The operating systems use hypervisors to make virtualization possible. Hypervisor is a software component of the operating system that controls the computing resources, such as networking, memory, disk, and processing resources. It exposes them to guest virtual machines through a simple interface. There are two different types of hypervisors:

- Type 1 hypervisor runs directly on the hardware and has access to the hardware resources, controlling them for the guest's virtual machines. Type 1 hypervisor is also known as bare metal or a native hypervisor [40].

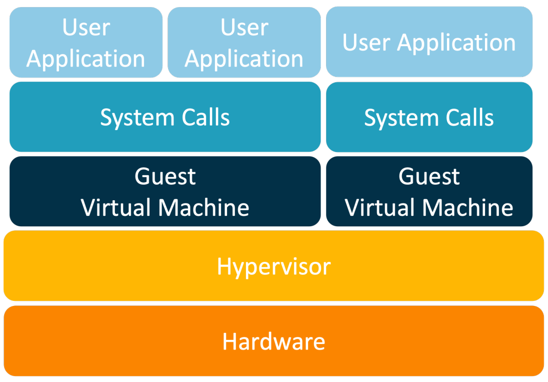
- Type 2 hypervisor runs as software on another operating system. Since it does not have direct access to the hardware resources, it performs worse than type 1 hypervisors, which are more efficient.

Typically, type 1 hypervisors are used for large data centers, while type 2 hypervisors are used more as end-user tools. Figures 2.1a and 2.1b present the schematic architectures of type 1 and type 2 hypervisors.

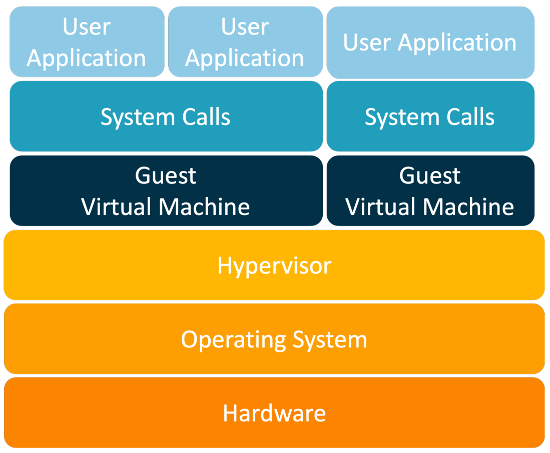
Virtual machines are great tools to abstract the hardware away from the user. However, virtual machines take up much storage space due to their large image size. Furthermore, virtual machines are slow due to their large size, taking a long time to modify, build, transport, or boot [44].

Lightweight virtualization systems were created to resolve the problems with heavy virtual machines. As a result, they provide much better performance results than traditional virtual machines.

Two of the most common lightweight virtualization systems are containers and unikernels. In this report, we will investigate and compare the performance of Docker [21], a tool for building and running containers, and Unikraft [13], an open-source development kit for unikernels.



(a) Type 1 Hypervisor



(b) Type 2 Hypervisor

Figure 2.1: Schematic architecture of different hypervisors

2.2 Containers

Containers are currently one of the most popular virtualization systems. Still, they are different from traditional virtualization systems. For example, containers do not require hypervisor software. Instead, the container engine runs each container as a separate process on the same host operating system.

The host operating system already contains many isolation mechanisms for separating two processes. Container applications leverage these mechanisms to build and run isolated processes efficiently. The two main components containers rely on for running are kernel namespaces and cgroups.

Kernel namespaces are a part of the Linux kernel, making it possible to split the available hardware resources so that each process can only see the resources allocated to them. In the Linux kernel, there are many different types of namespaces, such as user namespaces for controlling the users of the same operating system, process namespaces for managing the resources different processes can access, or network namespace for controlling the network access on the device.

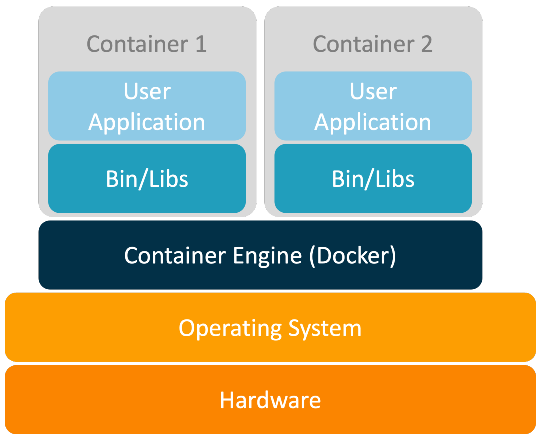


Figure 2.2: Overview of a containerized system

Control groups, or cgroups, are another part of the Linux kernel, allowing the host to modify and limit each process's resources. Cgroups allow easy control over specific physical resources' limitations, prioritization, and accountability. As a result, physical resources can be split between users and processes, allowing fair and efficient use of these resources. Nowadays, cgroups are known to be used by container engines. However, they are part of the Linux kernel and can be easily used by any other application.

In conclusion, kernel namespaces provide isolation between different resources, and cgroups allow for easy control and enforcement of this isolation [11]. Containers are built on top of these two features, and container engines heavily exploit them to provide isolation between many containers running simultaneously. Figure 2.2 presents the high-level architecture of a system that runs containers. The container engine is responsible for interaction with the host operating system. It creates and manages the containers running on the host.

2.2.1 Docker

There are many different container engines. However, one of the most popular nowadays is Docker [4]. Furthermore, Docker is open-source, simple, and free to use. These characteristics make Docker a perfect candidate for our study.

Docker is a set of tools to automate the process of building, running, and deploying containers. It is currently one of the most popular engines for running containers. It consists of a container engine that manages the whole process, a client CLI tool, and a container registry where many container images are stored. We present the architecture of Docker in figure 2.3. The container engine is also known as the Docker daemon; it runs on the host and can build Docker images, which can then be run as separate containers. Furthermore, it can pull data from the Docker registry, which contains images for many user applications, such as databases, web servers, data stores, or other programming environments.

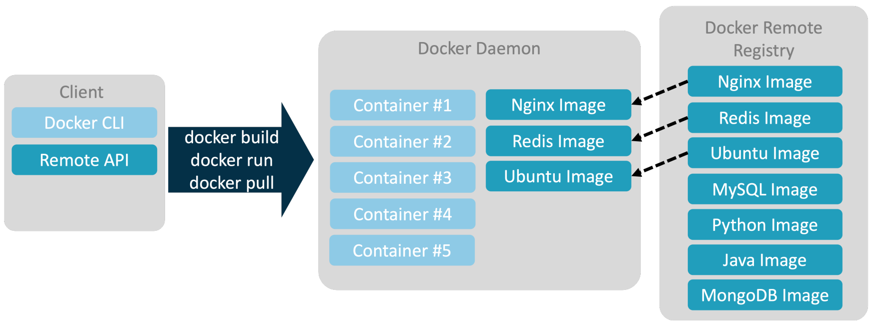


Figure 2.3: Docker system architecture

Docker also has a rich software ecosystem, such as tools for automating the build process or monitoring the containers' status. Because of their simplicity and the rich software ecosystem, Docker containers are the preferred deployment tool for many engineers.

However, since containers still run on the same operating system, the container engine does not provide complete isolation between different containers [41]. The incomplete isolation between different processes running as containers creates a significant security risk. In a hypothetical scenario, if one process bypasses the container engine's isolation layer, it can access all the resources the operating system manages. A study conducted in 2018 found more than 200 exploits effective on Linux containers [15]. Many teams have been working on improving container security, and there are new types of containers that aim to improve container security, like gVisor. However, the new layers of isolation this type of container introduces decrease the application performance significantly [42].

Another alternative to containers are unikernels. Unikernels are known to be very lightweight systems, requiring significantly fewer resources and giving much better performance. In the next section, we focus on exploring unikernels.

2.3 Unikernels

In traditional operating systems, the CPU works in two different execution modes: kernel mode and user mode. While running in kernel mode, the process has unlimited access to the CPU, memory, and other device resources. On the contrary, the process has limited access to hardware resources in user mode. Two separate kernel modes are an essential component of operating systems, as external applications might be harmful, and the operating system cannot trust such applications. In environments where multiple applications run simultaneously, having two kernel modes is non-negotiable.

However, each component will likely run only one application in cloud computing environments [16]. For example, such components might be web servers, data stores, databases, or other computing programs. If only one application runs in the whole operating system, having two separate kernel modes is redundant. The switches between kernel and user modes will only consume resources, while they will not provide any additional security or isolation benefit.

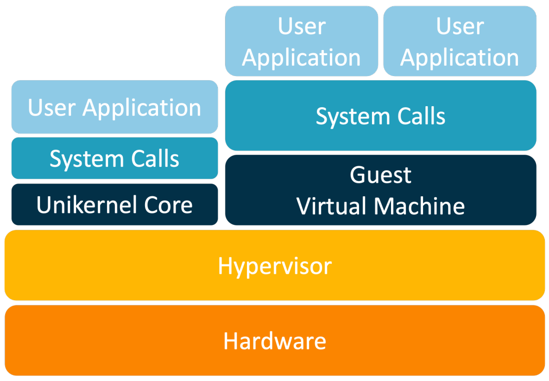
Unikernels are small virtual machine images that run only one application, and their kernel always runs in only one mode [17]. In unikernels, the software code is compiled with the kernel code resulting in a tiny virtual machine.

Figures 2.4a and 2.4b retrospectively present the high-level architecture of unikernels running on Type 1 and Type 2 hypervisors. Having only one kernel mode makes unikernels perform significantly better than containers and virtual machines [13]. In this report, we will focus on Unikraft. Unikraft is a novel open-source development kit for unikernels.

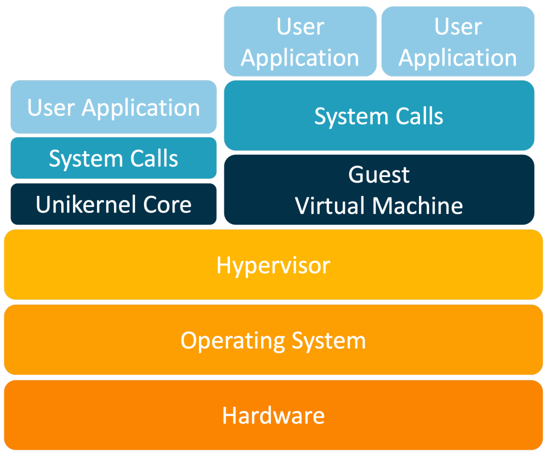
2.3.1 Unikraft

Unikraft is designed to be minimal in size. Therefore it supports full modulation of components, drivers, and libraries [13]. In addition, users have complete control over which kernel and library ecosystem components are included in the image they are building. Full modulation is impossible in traditional operating systems based on Linux, as all libraries in the Linux kernel are heavily dependent on each other. However, all Unikraft libraries are designed and implemented with full modulation and configuration in mind.

Unikraft supports system calls from the Portable Operating System Interface (POSIX) interface. This support makes it easier to port existing applications to work in Unikraft. Furthermore, it supports platform-independent image generation for different hypervisors and virtual



(a) Unikernel on a type 1 hypervisor



(b) Unikernel on a type 2 hypervisor

Figure 2.4: Unikernels running on different hypervisors

machine monitors. Such support is helpful in scenarios when we need to deploy the unikernel image to many devices. For example, instead of transporting a large source code or container image, we can build the image in one host and then transport only the compiled image to all other devices.

To build a unikernel that supports all these functionalities is difficult. As a result, Unikraft was built from scratch, as other projects did not have fully independent components and libraries.

Figure 2.5 shows the overview of building a new Unikraft image. On the left side, we can notice the whole pre-build setup. First, we have the Application code and Configuration files on top of the stack. Then the third-party libraries and the operating system/kernel code follow. Finally, we have the platform code and the hardware resources. Such application, OS, and kernel code setup is the typical setup required to run a user application on a traditional virtual machine.

Instead of using the whole code on the left, Unikraft takes the application code and fetches and prepares the required libraries. It then compiles the kernel, libraries, and application code separately. Then, finally, it links all the code together in a single unikernel image. The exciting and important thing here is that the compiled Unikernel image is tiny, as it only contains a few parts from the kernel functionalities, a few libraries, and the application code itself.

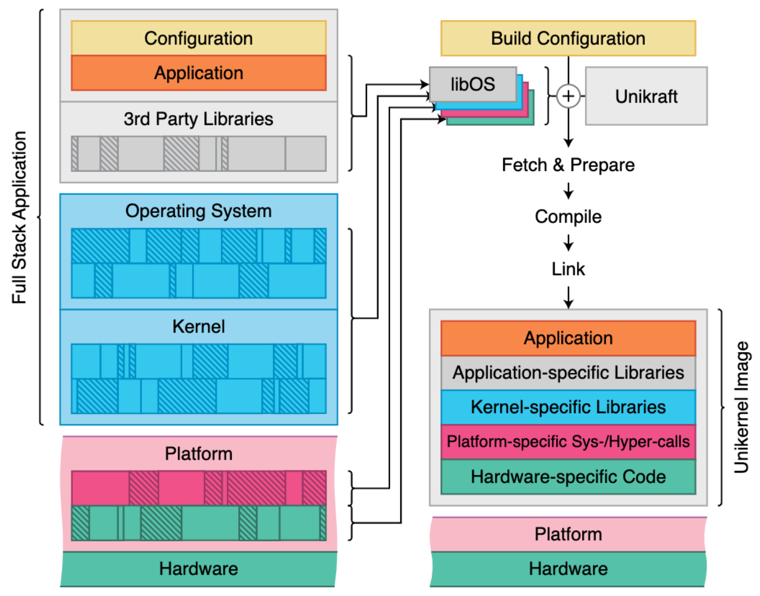


Figure 2.5: Overview of the Unikraft build process. Credits: Unikraft Documentation, `https://unikraft.org/docs/concepts/build-process/`

Chapter 3

Benchmark Overview

3.1 Infrastructure Configuration

We used a dedicated server machine to run all benchmark experiments. It has two Intel Xeon 520 processor chips, each running on a 2.2 GHz frequency. Each chip comprises 26 physical cores containing two threads, resulting in 52 per-chip threads or 104 total hyper threads. In addition, the processor has three levels of cache:

- L1 cache is shared by one core; It consists of two types: L1d 48KB for caching program data and L1i 32KB for caching program instructions.
- L2 cache is also shared by one core; It consists of a 1280KB total cache capacity shared by two threads on the same core.
- All cores share the L3 cache on the same chip; It consists of a 39 MB total cache capacity.

Besides the processing unit, the main memory plays a vital role in software systems' performance. Our device has 64GB DDR4 RAM running on 3200 MHz. As a disk storage, our device uses a 1TB Intel SSD.

Ubuntu 22.04 LTS was installed on our machine, and all experiments were compiled with GCC v11.3.0. To automate the execution of the benchmarks, Python 3.8.10 was used. In all of the benchmarks, we used Docker v23.0.1. Since Unikraft is developing more, we used v0.10.0 in the scalability experiments. However, for the performance isolation experiments, we update the version to v.0.12.0 as it contains relevant updates. All Unikraft images were run on QEMU emulator version 6.2.0 (Debian 1:6.2+dfsg-2ubuntu6.7).

3.2 Benchmark Software

We implement a CLI tool named experiment runner to simplify the benchmark process.

3.2.1 Design Decisions

The main goal of the experiment runner CLI was to make it possible to quickly prototype new experiments and easily extend them to larger-scale benchmarks. Bash is the most straightforward language for implementing automatic Linux scripts. However, Bash does not support advanced language functionalities. On the other hand, Python is one of the most popular scripting languages and supports all functionalities we need. Therefore, we decided to use Python for our scripts. The CLI tool was written using the Click library for Python. Click stands for Command Line Interface Creation Kit, and it is a popular Python library that simplifies writing a more extensive CLI tool.

Creating modular and reusable components was our second priority while implementing the scripts. Reusable components let us quickly change our experiments and adapt the scripts to new requirements.

Finally, we tracked the code base changes using Git source control. We hosted the whole project on a private repository on GitHub [34]. We decided to open-source the project once it was finished to ensure our results were reproducible and valuable for others.

3.2.2 Architecture

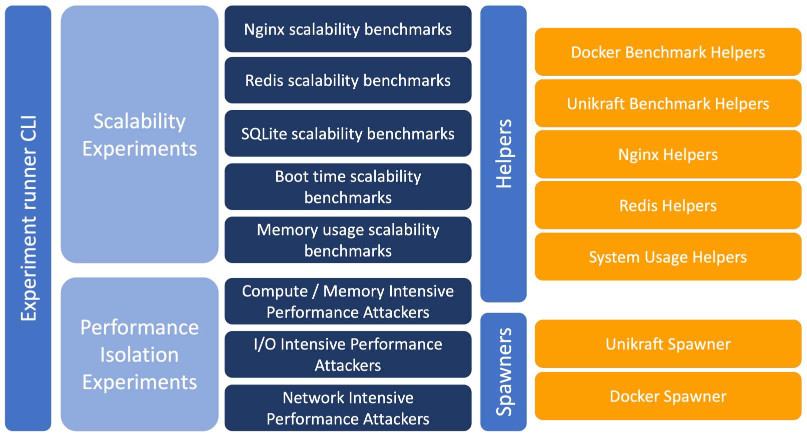

Experiment runner CLI
Scalability Experiments
Performance Isolation Experiments
Nginx scalability benchmarks
Redis scalability benchmarks
SQLite scalability benchmarks
Boot time scalability benchmarks
Memory usage scalability benchmarks
Compute / Memory Intensive Performance Attackers
I/O Intensive Performance Attackers
Network Intensive Performance Attackers
Helpers
Spawners
Docker Benchmark Helpers
Unikraft Benchmark Helpers
Nginx Helpers
Redis Helpers
System Usage Helpers
Unikraft Spawner
Docker Spawner

Figure 3.1: High-level architecture of the experiment runner CLI

Figure 3.1 presents the architecture of the experiment runner we described above. Every interaction is started by running the experiment runner CLI tool. Once the experiment runner is called, it activates one of the two types of experiments: scalability or performance isolation experiments. There are different types of benchmarks, depending on the application or parameter we are interested in analyzing. Irrelevant of which experiment we decided to run, each uses the helpers' library we implemented. The benchmark helpers library consists of two types of helpers: spawner helpers and general helpers.

The spawner helpers have various functions for running Docker containers and Unikraft VM images. There are multiple arguments that we control through these spawner helpers: name and version of the image we are starting, network parameters we pass to the image (IP address and port), whether we want to mount a file system to the image or not, or setting up any limits on the CPU or memory usages for the image.

The general helpers comprise another group of functions, primarily focusing on simplifying the benchmarking process. The helpers in this group include wrk helpers for benchmarking Nginx servers, redis-benchmark helpers, SQLite benchmark helpers, and CPU and memory usage helpers. All these helpers make it possible to automate the whole process and quickly run extensive benchmarks, such as spawning thousands of images and benchmarking their performances.

Another essential feature of the experiment runner is the benchmark cleaner. This part closes all the Docker containers or Unikraft images started during the benchmark process. Having such functionality was very helpful, as sometimes unpredicted behavior happens when working with applications on a larger scale. For example, some containers or VM images might crash in various situations. Without the automated cleaners, the whole server would be unusable, requiring a hard restart to unblock.

3.2.3 Usage

During the scalability benchmarks, we often launch many container or unikernel instances. Therefore, we need to launch more instances for each benchmark and then measure some performances iteratively. The CLI tool we created takes four input arguments to run:

1. Experiment name: each experiment we have designed is given a unique name. This name helps identify the experiment we want to run. Table A.1 in the appendix contains the code names for each experiment we have created.

2. Repetition times: specifies the number of repetitions for our benchmarks. We ran each experiment at least five times to get more detailed results.

3. Benchmark times: specifies the number of times we perform the benchmark inside each

experiment repetition. This argument is essential in the scalability experiments because it defines how long we run each repetition.

4. Instances per benchmark: specifies the number of instances we launch between two consecutive benchmarks in a single repetition. This argument only applies to scalability benchmarks.

For example, running: `python3 experiment_runner.py --runs 5 --name uk_ng_s --benchmark_time` `3 --instances_per_benchmark 10` will execute the Unikraft Nginx Scalability experiment 5 times. In each of the five runs, it will first start one Nginx instance and measure its performance using wrk benchmark. It will then launch ten additional stress instances (as specified in the `instances_per_benchmark` argument) three times (as defined in the `benchmark_times` argument). After every ten new instances, it will perform a new wrk benchmark. This command will collect 20 benchmark results, 4 in each of the five repetitions it executes. These four results represent the performance of Nginx on Unikraft:

- When it runs as a single process on the machine.
- When there are ten stress instances in the background.
- When there are 20 stress instances in the background.
- When there are 30 stress instances in the background.

We discuss the scalability benchmark process in Chapter 4 and the performance isolation benchmark process in Chapter 5

3.3 Benchmark Guidelines

To ensure the validity of our results, we repeated each scalability experiment at least five times. On each graph, we plot the average execution time, including the standard deviation observed in the results.

For the performance isolation experiments, we repeated them 20 times. This decision was made because the performance isolation experiments require significantly less completion time. In comparison, the scalability experiments require longer to execute due to the high load we are generating.

Chapter 4

Assessing Scalabiliy Methodology & Results

Scalability typically refers to a system, either hardware or software, to handle an increased load in work or operations. For example, scalable systems operate well when the load or traffic doubles in size or volume. Scalability is a general term, and there are many different types of scalability in computer systems, such as load, space, space-time, or structural scalability [1].

Scalability is of crucial importance in modern computer systems. Users expect 100% availability from the service they are using, and even short downtimes due to increased network load mean a loss of money for the business. Furthermore, in cloud computing environments, cloud consumers expect optimal performance irrelevant of the number of consumers using the same hardware machine. In this chapter, we will investigate and analyze how the performance varies as the number of user applications on the same machine increases. We will compare the performance of applications run on Docker and Unikraft at different scale levels.

4.1 Benchmark Design Overview

Our experiments simulate a realistic cloud computing environment with a single hardware machine. Increased load in a cloud environment typically refers to more users accessing the resources on the device. To provide a good user experience for all cloud consumers, increasing the load on a single machine should not impact any user's system performance.

We have a primary user application whose performance we observe in our scalability benchmarks. We run different user applications in each of our benchmark analyses. This user application is always relevant to the behavior we are trying to observe. We observe the performance of the user application when it is a single process running on the machine. We then load the system by starting a various number of background instances. Once more instances are started in the background, we observe the performance again. To collect fine-grained data, we observe the application's performance after every ten instances started.

We are interested in something other than the performance of a single virtual system instance. Such performance evaluation is another research topic, and both Unikraft and Docker have been carefully evaluated [13]. Instead, we are interested in how the behavior of these systems changes as we increase the operation scale by launching large amounts of instances on a single machine.

Launching new instances and measuring the user application's performance is controlled by the benchmark CLI, described in Chapter 3.

Typically benchmark experiments can be divided into two categories: microbenchmarks and macrobenchmarks. Microbenchmarks usually track and test small characteristics of a particular system. Microbenchmarks work in isolation, and they only focus on one specific overhead. On the other hand, macrobenchmarks are more extensive tests that aim to simulate real-world user load. As a result, micro and macrobenchmarks give us valuable insights into different aspects of the systems. This chapter first examines two scalability microbenchmarks: boot time and memory usage. Then, we look at two more significant macrobenchmarks that aim to thoroughly test the performance of the two virtualization systems we are analyzing.

4.2 Microbenchmarks

4.2.1 Boot Time Analysis

The first experience of cloud consumers with a rented cloud machine is booting their virtual system. In this experiment, we will compare the boot times of Docker and Unikraft on different scale levels. We will focus on the changes in boot time as more and more background systems are launched.

To design a fair experiment for measuring the boot time of any virtualization system, we first need to look at what happens when a particular virtualization system starts a new instance. To understand the boot process, we will examine what happens when a new Unikraft and Docker instance is run.

Firstly, we will look at the boot process of a Unikraft virtual machine. It consists of three different stages:

1. Boot time of the emulator (QEMU). Once the emulator is booted, it initiates the boot of the kernel.

2. Boot time of the Unikraft kernel. Once the kernel is successfully booted, the user application boot process starts.

3. Boot time of the user application.

To fairly measure the boot time of a Unikraft instance, we need to remove the overhead of the user application code. To do this, we modified the `helloworld` Unikraft template [33]. The `helloworld` Unikraft template is a tiny user application that can quickly launch a new Unikraft instance that only prints hello world to the terminal and exits. This application adds a negligible application overhead to the boot time as it compiles with the rest of the kernel code.

Secondly, we will look at the boot process of a Docker container. Similarly to Unikraft, it also consists of three separate components:

1. Container creation. The docker daemon creates a new container instance and initiates the boot process of this instance.

2. Boot time of the container. The container instance boots and initiates the boot command described in the `dockerfile` specification.

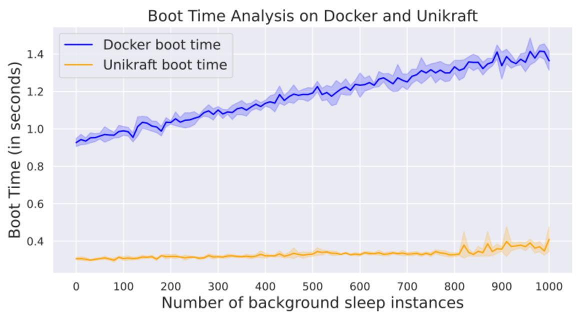
3. Boot time of the user application.

Similarly to what we described for Unikraft above, we want to exclude any boot time of the user application for Docker containers. To achieve this, we compiled a tiny C code that only prints hello world to the terminal and exits. Again, similarly to Unikraft, this only adds a negligible performance overhead and allows us to measure the boot time precisely.

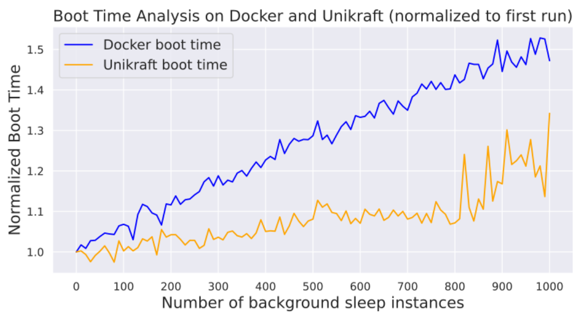
Measuring time precisely in an operating system is difficult because the system schedules things on and off the processor, giving only an approximate time measurement. To measure the boot time as precisely as possible, we use the `gettimeofday` system call. We compiled and used a small program that executes any application program and measures the time it takes for this execution [25].

Our analysis focuses mainly on the change in boot time as the number of background instances increases. Therefore, we do not measure the exact boot time for a single instance, as this involves more in-depth analysis and measurements of all software parts involved in the boot-up process. Other authors have performed a more in-depth analysis of the boot times of a single instance for both Docker [26] and Unikraft [13].

To perform a scalability analysis of the boot time, firstly, we measure the time required to boot a single instance of Unikraft or Docker when the system is free and not doing any work. Then, we start a new ten instances in the background, which only stress the container engine and the hypervisor, as they only contain a small C program that executes the `sleep()` command. Once the ten new instances are started, we again measure the boot time of a small `helloworld` instance. We repeat this 100 times, giving us the change in boot time as we launch up to 1000 background instances.



(a) Raw data analysis

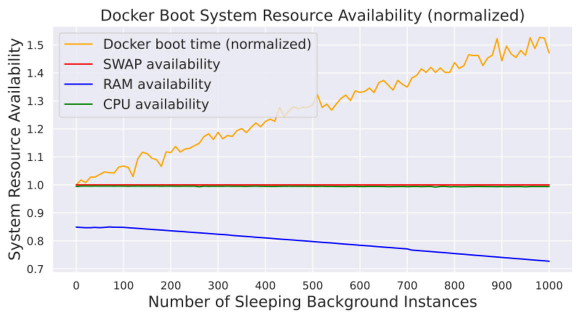


(b) Data normalized to first run

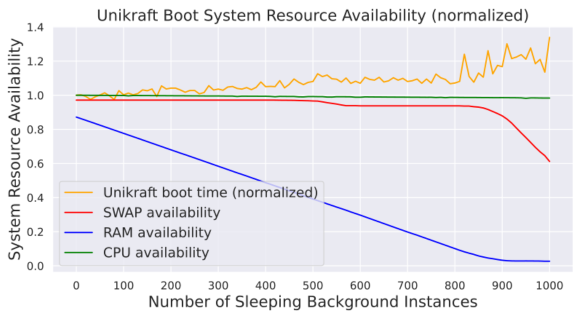
Figure 4.1: Combined boot time anlysis of Docker and Unikraft (lower is better)

Figure 4.1a presents the combined boot times of both Docker and Unikraft. To better understand how the boot times change compared to the boot time of the first instance, we also present Figure 4.1b, where the boot times are normalized based on the first run.

Firstly we will look at the change in the boot time of Docker containers. In general, booting a new Docker instance takes much longer than booting a new Unikraft instance because unikernels are more lightweight than container images, making their boot times much shorter. Moreover, we can observe that the boot time of a new Docker container increases linearly. In addition, figure 4.1b shows that booting the 1000th container takes roughly 50% longer than booting the first container on a single machine.



(a) Docker



(b) Unikraft

Figure 4.2: System resource availability in the boot time experiment

In Figure 4.2a, we present the change in system resource usage as we perform the boot time experiment for Docker. From this figure, this experiment does not generate any significant CPU, memory, or swap activity. Therefore, the slowdown is likely a bottleneck in the Docker container engine. Other authors have found similar bottlenecks in hypervisor engines, such as the Xen hypervisor engine [24].

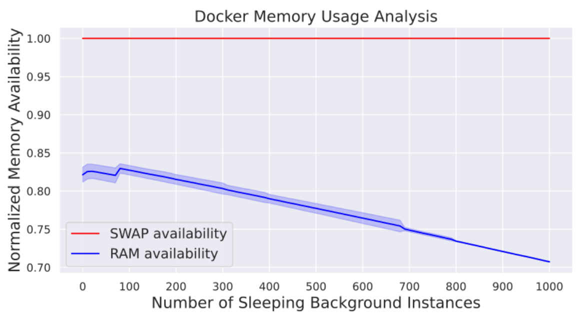
The change in boot time for Unikraft contains more exciting patterns than the boot time benchmark for Docker. First, we can observe that the boot time till around the 800th instance remains stable and does not increase by more than 10%. However, once we go over this threshold, the boot time significantly increases. To understand this in more detail, we must also look at the system resource usage patterns while performing this benchmark.

Figure 4.2b presents the CPU, main memory, swap memory usage, and boot time change for Unikraft. This figure shows that around the 800th instance, the RAM availability has decreased to less than 20%. Furthermore, the operating system has started using the swap memory, whose availability is also falling. Accessing the swap memory includes accessing the disk storage device, which is much slower than accessing the machine's main memory. Therefore, memory availability is the leading cause of the boot time slowdown of new Unikraft instances.

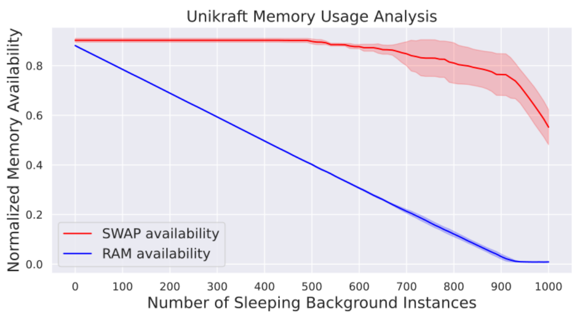
In conclusion, Unikraft outperforms Docker in the boot time scalability analysis. Furthermore, it boots in a significantly shorter time. Such difference makes Unikraft a better choice for applications with crucial boot time.

Since primary memory availability plays a vital role in the performance of the virtualization systems we analyze, in the next subsection, we take a more detailed look at how these systems use memory.

4.2.2 Memory Usage Analysis



(a) Docker



(b) Unikraft

Figure 4.3: Scalability memory usage analysis of Unikraft and Docker

Apart from just benchmarking the boot time, we also look at the memory usage of both lightweight virtualization systems on a larger scale. Similarly to the boot benchmark, we are not interested in the detailed memory usage of a single Unikraft or Docker instance. Instead, we focus on how the memory usage changes as we increase the scale of instances launched on the same device.

To measure the change in memory usage of Unikraft and Docker on a larger scale, we used a similar experiment as in the boot time benchmark. In turn, we start ten new instances and measure memory usage. Then, we repeat this 100 times, observing the change in memory usage as we launch 1000 instances. We want to avoid the memory overhead of the application running inside the virtual systems. To do this, inside the Unikraft image and Docker container, we run a C program that infinitely executes the sleep function. When this program is compiled, its size is minimal.

Furthermore, since it does not interact with any data, its memory consumption is negligible. Such load of spawning many virtual systems puts the most stress on the hypervisor, QEMU for Unikraft, and Docker's container engine. Figures 4.3a and 4.3b show the differences in memory usage of Docker and Unikraft.

It is essential to mention that the pattern of using large amounts of memory is not because of the Unikraft kernel. Instead, most of the memory overhead is used by the hypervisor, QEMU. QEMU uses as much as possible of the available memory. Once it does not have enough memory, it adapts and uses less memory. Such an approach is reasonable if only a few instances run on the same device. It uses more memory to provide better application performance. However, the memory gets filled up quickly, and the operating system has to activate the swap functionalities. Swap memory access is slower than main memory and decreases the application's performance. On the other hand, Docker has a slightly more conservative memory consumption and only uses a minimal amount of memory required to run the application.

The boot time and memory usage are essential characteristics of computer systems. However, these metrics give us little insight into how these systems perform in realistic scenarios. Furthermore, isolated metrics like boot time do not provide us any indication of the performance of the application after it boots. To understand how the applications perform after the boot process finishes, we need to look at macro benchmarks that measure the overall performance of Unikraft and Docker. Motivated by their popularity, we look at two categories of applications: network-intensive and storage-intensive.

4.3 Network Intensive Applications

Many teams of engineers build network applications, either as web services, data, or other types of services. As a result, network applications are one of the most popular applications hosted in cloud environments. People usually associate network applications with client-server web applications. However, another group of typical network applications is data stores. Data stores are network applications that store and manage information in different formats. SQL or NoSQL databases, file storage systems, or in-memory databases are typical data stores.

Since web servers and data stores are among the most common applications used in cloud environments, we chose to benchmark two different applications and see how their performance changes as we increase the benchmark scale.

4.3.1 Nginx

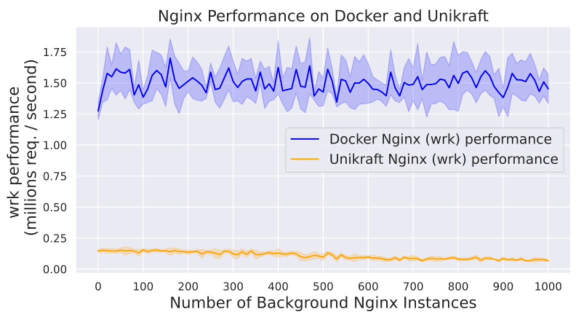
Firstly, as a web server, we chose to benchmark Nginx. Nginx is an open-source software offering an extensive list of functionalities to users. It can be a simple HTTP server, load balancer [23], mail proxy [22], or HTTP cache server. As a result of all the functionalities it can perform, according to some analysis, it is currently the most popular web server [31]. Another study by Datadog[1] suggests that Nginx is the most popular application run in Docker containers [5]. Nginx is also ported and available in Unikraft, making it a perfect candidate for our study.

There are many popular benchmark tools for HTTP web servers. However, we chose to use the wrk benchmark tool. It is an open-source modern HTTP benchmark tool and is easy to configure and use [7]. However, even though wrk is a simple tool, it can generate a significant load to the web server we are testing. Furthermore, benchmarks for Unikraft have already been performed using wrk [13], and using the same benchmarking tool keeps these benchmarks consistent. In addition, the following wrk configuration was used: 30 parallel connections running on 14 threads for 20 seconds.

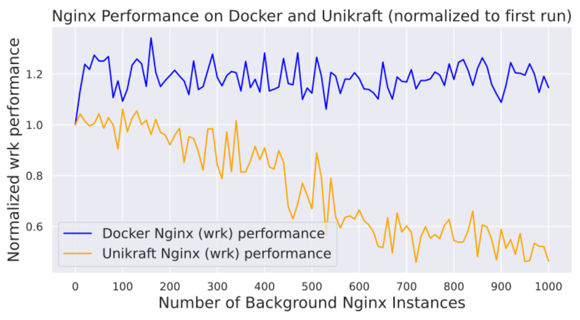
The main problem we want to understand is whether many Nginx instances launched on the same host influence the performance of a single instance running on the machine. To get more insights into this issue, we first benchmark Nginx's performance on both systems when it runs as a single instance on the machine. We then launch ten additional Nginx instances on either Unikraft or Docker, and we observe the performance of this one instance again. Finally, we repeat the process of launching ten more instances and benchmarking the one instance until we launch up to 1000 background instances. The instances we launch in the background are Nginx

[1]Datadog is a company providing tools for monitoring servers, databases and services

servers that do not perform any activity, only wait in the background. Figure 4.4a presents the combined performance of Unikraft and Docker when benchmarking a single Nginx web server.



(a) Raw data analysis



(b) Data normalized to first run

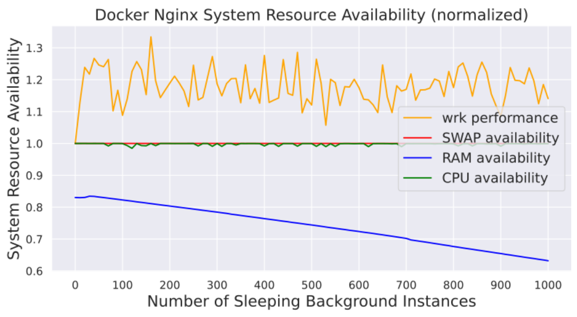
Figure 4.4: Combined wrk benchmark performance of Nginx on Docker and Unikraft (higher is better)

We notice that Unikraft processes slightly fewer requests per second than Docker. This performance difference happens because our experiment used the default Unikraft configuration to generate the Nginx image. However, Unikraft is fully configurable, and many parameters must be configured correctly to achieve optimal performance. This is explored in more detail in the original Unikraft paper, where the optimal performance for each application on a single instance is benchmarked [13]. In our work, we focus on how the performance changes when many applications are run simultaneously.

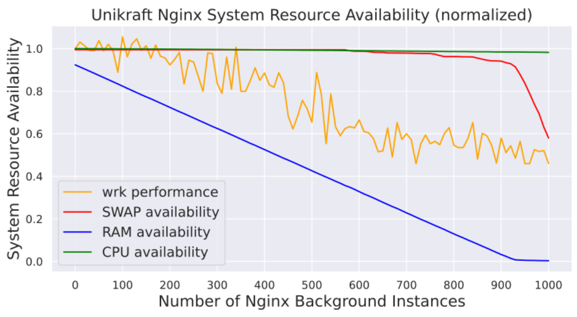
To get a clear image of how the performance changes as we increase the number of background images, we present the performance of both Docker and Unikraft normalized to the single instance performance in Figure 4.4b. From this figure, we can notice that the performance of Docker remains stable, and any background instances do not impact the performance of the server we are benchmarking. On the other hand, we can observe that the performance of Unikraft decreases as we increase the number of background unikernels. Overall, after 1000 background unikernels, it reduces by around 50%. A scalability bottleneck in the Unikraft kernel or network library likely causes this. Another reason for the reduced performance might be the overhead of the Qemu hypervisor.

Another pattern we can observe is that the Docker experiment of Nginx performs slightly better as more instances are launched. This performance improvement happens because the first instance starts with a cold cache, and nothing from the main memory is cached. However, once we start performing more experiments, the operating system caches much information required to execute the experiment, improving its performance.

Figures 4.5a and 4.5b present the system resource availabilities in Docker and Nginx when



(a) Docker



(b) Unikraft

Figure 4.5: System resource availability in the Nginx scalability experiment

testing Nginx. In addition, we can observe the use of swap memory in the Unikraft experiment. The use of swap memory likely decreases the instance's performance slightly, as the operating system has to spend some time transferring the data from the main memory in the temporary swap storage. On the other hand, the RAM usage in the Docker experiment remains stable when running this experiment.

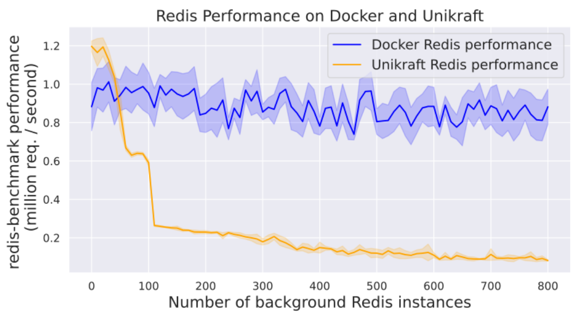
4.3.2 Redis

Secondly, as a data store, we decided to evaluate the performance of Redis. Redis stands for Remote Dictionary Server, an open-source in-memory key-value data store. It is used extensively in many large software systems due to its fast read (GET) and write (SET) operations. In addition, most systems use Redis as a cache for more expensive database operations. The same study conducted by Datadog suggests that Redis is the second most popular container application, just after Nginx [5]. Similarly to Nginx, Redis is also ported to Unikraft.

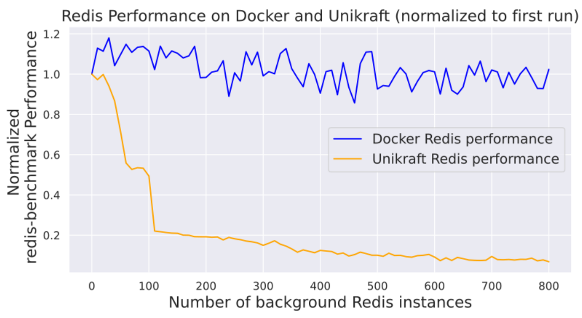
We used the open-source Redis benchmark (`redis-benchmark`) tool to evaluate the performance of Redis launched on Docker and Unikraft. It is a simple tool that simulates many different Redis commands [27]. The following benchmark configuration was used in our benchmark analysis: 10 million requests running on 30 clients in 16 pipelines.

Figure 4.6a presents the combined performance of a Redis server hosted on Unikraft and Docker. Figure 4.6b presents the normalized performance. The performance of Unikraft shows some interesting patterns. Unikraft outperforms Docker early in the experiment, but its performance drastically decreases after starting a few more instances.

Figures 4.7a and 4.7b present the system resource usage of both systems. In the chart for the Unikraft performance, we can see that apart from the decrease in the redis-benchmark performance, the availability of CPU also gets decreases quickly. Furthermore, after 100 instances,

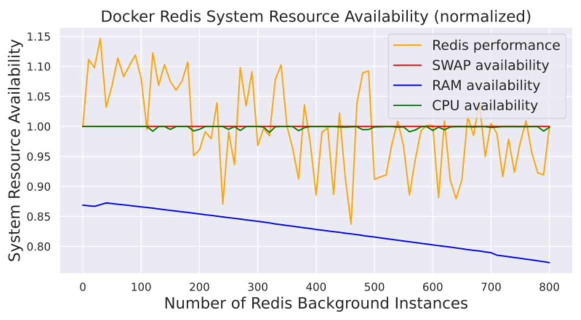


(a) Raw data analysis

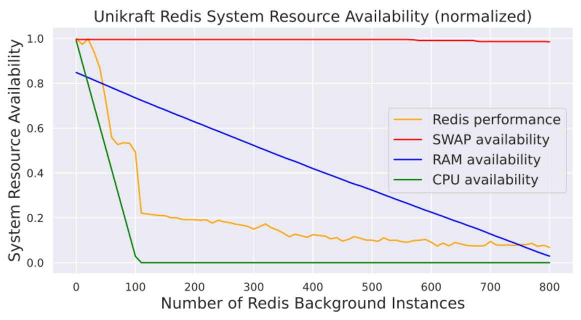


(b) Data normalized to first run

Figure 4.6: Combined Redis benchmark performance on Docker and Unikraft (higher is better)



(a) Docker



(b) Unikraft

Figure 4.7: System resource availability in the Redis scalability experiment

we can observe that the CPU availability is zero. This is because our machine has 104 cores, and none can execute any other work. To understand the cause of this, we run a single Unikraft image with a Redis server inside. We observed that the core to which we pin the Unikraft image started gets 100% usage. When we start 100 instances, all the available cores get filled up, and the CPU availability decreases to zero. We reported this issue to the Unikraft community, and it was fixed in the newer Unikraft version, the v0.11 release [36].

On the other hand, Docker's performance with Redis remains stable throughout the experiment. Docker is an older system, and its popularity has been growing ever since. Therefore, much engineering effort has been put into optimizing its performance and stability [14, 29].

Network applications are critical; optimizing and understanding their performance is crucial for high-performance computing. However, this type of application often requires some form of permanent storage. For example, most websites need to store user details in some form of a database. Therefore, to understand the performance impact of using lightweight virtualization systems, we explore storage-intensive applications in the next section.

4.4 Storage Intensive Applications

Cloud storage is a service cloud providers offer to store data permanently without managing hardware devices. As a result, cloud storage gives users better scalability, flexibility, and security. However, one of the essential features of cloud storage is redundancy. Providers replicate the same data on multiple machines and sometimes even multiple disks on the same device to ensure no data is lost. Such storage services are essential for many applications which rely on them. Cloud storages typically offer storing data in a database or a virtual file system, making them a good choice for many cloud consumers. Motivated by the importance of cloud storage and its popularity over the last few years, we explore the performance of storage-intensive applications on a larger scale below.

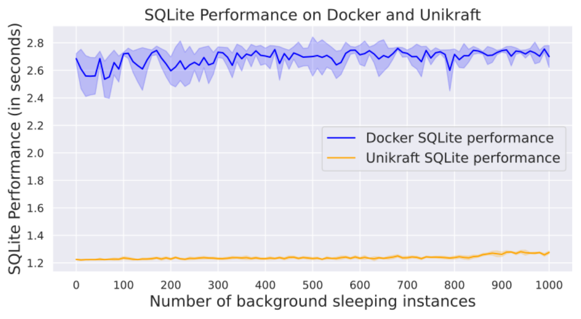
Storage-intensive application access the hardware storage device through a standard system call interface. These applications make use of the file systems on the operating systems. In chapter 5, we explore file systems on a lower level in more detail.

Similarly to the experiments performed above, we wanted to evaluate the performance of Docker and Unikraft in a realistic scenario. One of the most common uses of cloud storage is database storage. Therefore, we decided to evaluate the storage scalability performance using a database benchmark. SQLite was the perfect choice for our experiments. It has a minimal design, can be compiled quickly, and supports all standard SQL functionalities [10].

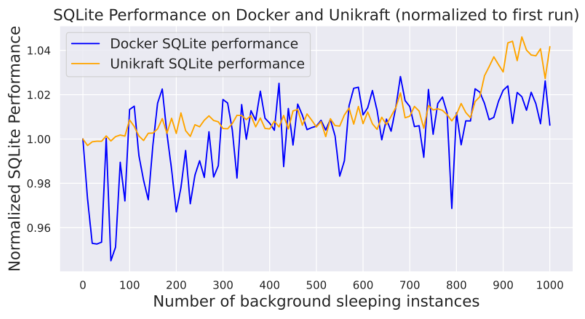
Unlike network-intensive applications we discussed above, which have many standard benchmarking tools, there is no common SQL benchmark tool. Therefore, to provide consistency with other benchmark analyses, we used a benchmark designed by the Unikraft community, already used to evaluate the performance of SQLite ported on Unikraft as a single instance running. Unikraft is an open-source project, and they have open-sourced all the code for their benchmarks. In particular, for this benchmark, it is a simple C program that executes 60000 SQL queries that insert new rows in a table [32]. In addition, we also ported this benchmark to run it inside a Docker container.

Figure 4.8a presents the performance of both Docker and Unikraft on an SQLite benchmark. We observe that Unikraft outperforms Docker significantly in this experiment. The performance difference is because Unikraft runs as a separate virtual machine, and all system calls executed by the benchmark directly go to the underlying hardware. Furthermore, Unikraft is an unikernel VM that does not waste resources on switching between kernel and user modes. On the other hand, Docker containers spend many computational resources as each system call has to go through the underlying operating system before accessing the hardware.

In Figure 4.8b, we can observe the normalized change in the benchmark time. The benchmark times of both Unikraft and Docker vary by around 3% from the first run. Overall, the



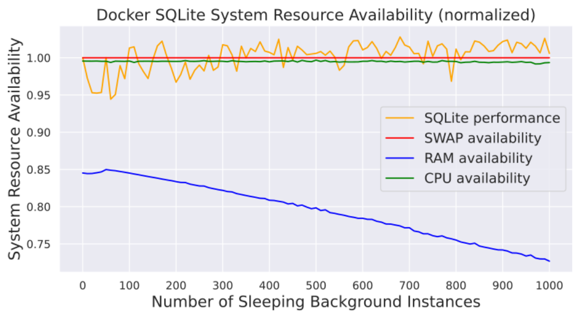
(a) Raw data analysis



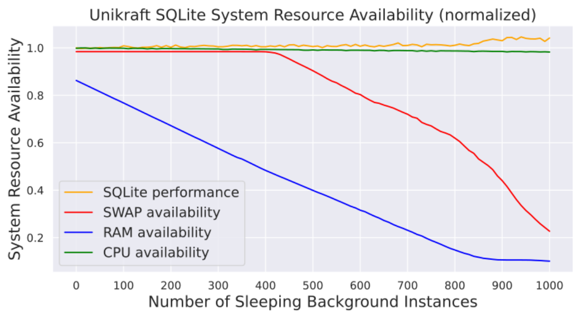
(b) Data normalized to first run

Figure 4.8: Combined SQLite benchmark performance on Docker and Unikraft (lower is better)

benchmark times remain stable even though we increase the load to the virtualization systems by launching 1000 instances.



(a) Docker



(b) Unikraft

Figure 4.9: System resource availability in the SQLite scalability experiment

To understand the change in performance in more detail, we need to look at the system resource availability. Therefore, we explore the system resource availabilities in Figures 4.9a and 4.9b. For example, in Figure 4.9b, we notice that the RAM becomes full at around 800 Unikraft instances, and the performance slightly reduces after this. However, this degradation is minor, only at about 4%. On the other hand, in Figure 4.9a, we do not notice any performance degradation for Docker. Docker's performance remains stable as this experiment only uses around 30% of the available RAM. Therefore, the benchmark still has access to all system resources.

Chapter 5

Performance Isolation Methodology & Results

Performance isolation is the capability of the operating system or hypervisor to reasonably isolate the resources between all the entities trying to use them. Performance isolation in cloud computing refers to the responsibility of the cloud provider and the machine kernel to isolate the different systems' performance effects on each other. For example, if one misbehaving user starts a container that actively abuses the hardware resources, simultaneously running containers should not see any slowdown because of the misbehaving user.

Performance isolation is well-studied for both containers [28] and virtual machines [18]. In this report, we extend existing studies and compare the performance isolation across containers and unikernels.

5.1 Benchmark Overview

Performance isolation studies typically involve two-step measurement:

1. An uninterrupted application benchmark is run, and performance is recorded.
2. An additional misbehaving instance is run in the background on the same hardware resources.
3. The application benchmark is rerun, now running together with the misbehaving instance.

The performance of the interrupted application is also recorded and analyzed compared to the uninterrupted performance. The percentage ratio between the interrupted and uninterrupted performance is known as a slowdown measure.

In our experiments, we also make use of the described model. In addition, we suggest an alternative model for evaluating performance isolation experiments to generalize our work. Namely, we propose a general model where we have two instances participating in each experiment:

1. Application instance: running a specific program. This program might be a web server, data server, or a simple program executing some work.
2. Misbehaving instance: program designed to intentionally misuse the operating systems resource and slow down the application instance.

Later in the text, we sometimes refer to the application instance as a worker and to the misbehaving instance as an attacker. Using this model, we abstract away the logic of running performance isolation experiments described above. Instead, we can focus on designing new performance attackers and studying their effects on the workers.

To avoid missing any information from the complete data, we do not generalize each benchmark to a single number representing the slowdown but present the observed data, including the average and the standard deviation in the data. Some authors use the Equation 5.1 to describe the slowdown [41]. Unfortunately, this equation fails to capture the difference between two possible types of benchmarks:

- Benchmarks where a smaller value indicates better performance, for example, benchmarks that return a time value required to complete.
- Benchmarks where a smaller value indicates reduced performance. Such as benchmarks that return a number of requests per second.

Furthermore, the equation ignores any deviation in the collected performances. Meanwhile, capturing the standard deviation in the observed data is essential, as it allows us to reason about the observed behaviors confidently.

$$\text{Slow Down} = \left(1 - \frac{\text{performance}_{\text{misbehaving}}}{\text{performance}_{\text{baseline}}}\right) \cdot 100\% \tag{5.1}$$

Once we establish the method for designing and evaluating performance isolation experiments, we can start analyzing different types of attackers and their effects on various applications.

5.2 CPU and Memory Intensive Attackers

One of the primary resources the operating system controls are the CPU and memory resources. Much engineering effort has been put into isolating the CPU and memory between virtual instances fairly and efficiently. Moreover, the effect of CPU and intensive programs has been studied in detail [39, 6]. However, most existing studies only measure the performance of one type of virtualization system and do not compare the performance across different types of systems.

This section analyzes the first type of performance attackers, the CPU and memory attackers. We start by looking at two well-known high-level application workers, Nginx and Redis. We then extend our analysis also to include the SQLite worker benchmark.

Designing a CPU-intensive attacker is a relatively straightforward process. Our experiments used a program that checks whether a number is a prime number for each value between 1 and 1 billion. This program involves many arithmetic operations, such as division, modulo, and branching. Running this program on only one CPU core always resulted in 100% CPU usage, which confirmed it was a good attacker for our studies.

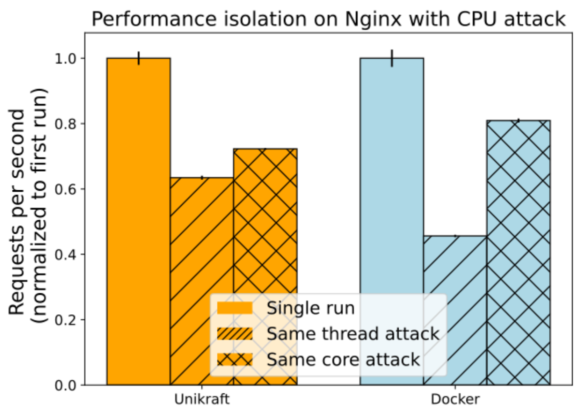
On the other hand, designing a memory-intensive attacker is slightly more complicated. Therefore, to ensure reliable results, we adapted STREAM, a standardized memory benchmark, as an attacker. STREAM is a simple benchmark program that computes the memory bandwidth in MB/s [20, 19]. It was written in 1995, but it was modified several times through the years to adapt to newer trends in computing. It consists of four different vector operations: copy, scale, sum, and triad, which are all computationally simple operations but put high stress on the memory system.

Furthermore, we were not interested in the outputs STREAM produces for our system, as this is irrelevant to our study. Instead, we adapted the operations STREAM uses to stress the memory as a memory-intensive attack. Finally, we ported STREAM to Unikraft and Docker and used it in our experiments.

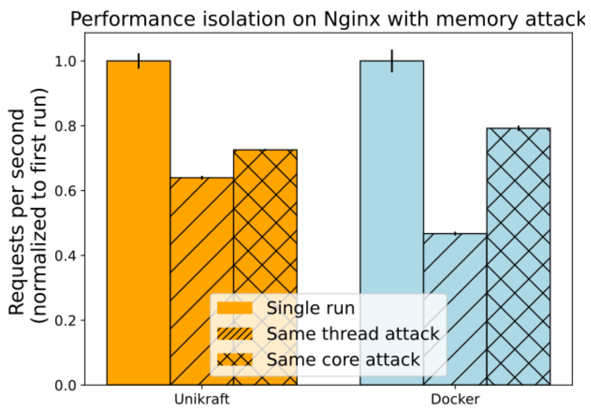
Once we create the CPU and Memory intensive attackers, we can observe the difference in the performance when the instance runs on its own and when it runs attacked.

5.2.1 Nginx

We first run the Nginx server virtualized in Unikraft or Docker system in our setup. To ensure we stress this instance, we pin it to a specific core in the processor. We then benchmark its performance using wrk benchmark. The performance we get serves as a baseline performance of the Nginx server. Then, the attacker is launched alongside the server, pinned on the same thread. The performance is again observed, and the slowdown is noted down. Finally, in the third run, we run the attacker on the other hyper thread belonging to the same core using the same process. We use the same wrk and Nginx configurations from chapter 4.



(a) CPU attack results



(b) Memory attack results

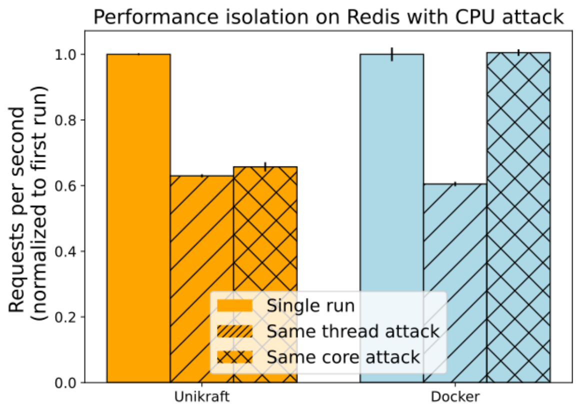
Figure 5.1: Performance Isolation Results of Nginx on Docker and Unikraft (higher is better)

Figures 5.1a and 5.1b show the normalized wrk benchmark performances for CPU- and memory-intensive applications. When using these types of attackers, Unikraft provides better isolation, and the slowdown is smaller than Docker when the attacker is pinned to the same thread as the application. On the other hand, when the attacker is pinned to the other hyper thread in the same core, Docker slightly outperforms Unikraft. Finally, there is no significant difference in the results of the CPU and memory attacks in this experiment.

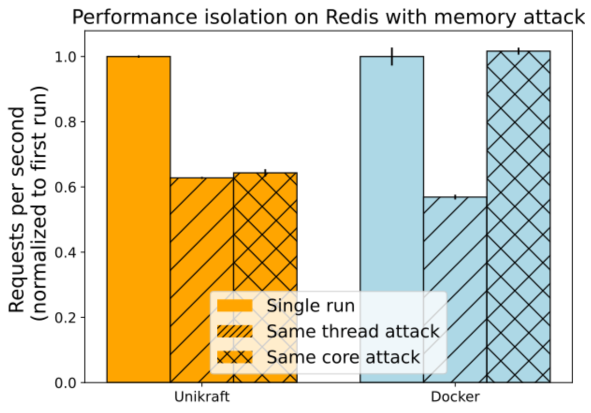
5.2.2 Redis

The same procedure for benchmarking the performance isolation of Redis is repeated too. In figures 5.2a and 5.2b, we present the performance of Redis under the CPU and memory-intensive attackers.

Firstly, when the attacker is run on the same thread as the application, the slowdown of Unikraft is slightly smaller than the slowdown of Docker. However, this difference is only 2.5% for the CPU attacker and 5.8% for the memory attacker and is insignificant. On the other hand, when the attacker is run on the same core but on a different thread, Docker outperforms Unikraft. Furthermore, the attacker running on the other thread does not give any slowdown to the main application container. Such behavior is an excellent performance isolation characteristic of Docker. Redis is a popular application, and unique engineering effort has likely been put toward optimizing its performance and handling such performance attackers.



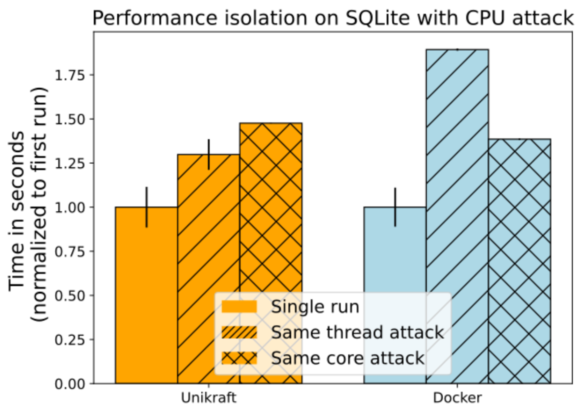
(a) CPU attack results



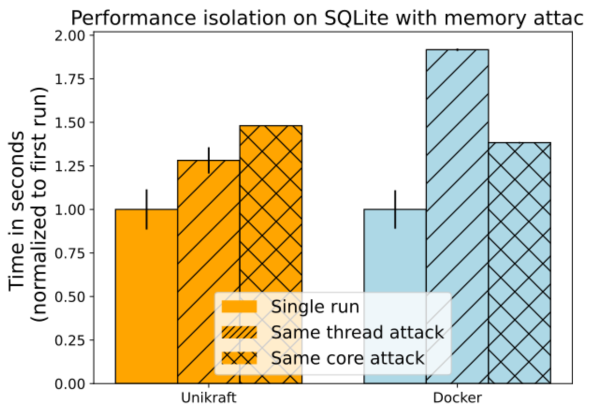
(b) Memory attack results

Figure 5.2: Performance Isolation Results of Redis on Docker and Unikraft (higher is better)

5.2.3 SQLite



(a) CPU attack results



(b) Memory attack results

Figure 5.3: Performance Isolation Results of SQLite on Docker and Unikraft (lower is better)

SQLite is a storage-intensive application, and it heavily depends on fast memory. One of the essential storage parts in computer devices is the cache itself. Since the cache is organized in a topology, different cache levels are shared with different chips, and we add another level at which we examine the performance isolation for SQLite. The new level of analysis we run places the attacker on the same physical core but on a neighboring thread. The two threads running on the same core share the same cache lines, among many other hardware resources, such as memory buses.

Figures 5.3a and 5.3b present the performance of SQLite under CPU and memory-intensive attacks. Firstly, Unikraft outperforms Docker significantly on the test when the application and attacker are placed on the same thread. On the other hand, when the attacker is run on the same core but a different thread, Docker performs only slightly better. To understand these results, we must look more closely at file systems and other types of attackers. Therefore, we explore file systems in more detail in the next section.

5.3 File System Attackers

Another essential feature of the operating system is the file system. A file is typically a collection of some data. On a lower level, we can view files as a list of bytes representing some information. However, Unix-based operating systems were built with the "Everything is a file." philosophy in mind. Since files are the base building blocks of all other functionality in the operating systems, maintaining an efficient file system is crucial.

The file system typically refers to the operating system's data structures and functions to control file creation, modification, and deletion. In addition, these data structures store metadata parameters, such as file name, size, type, permissions, and owner.

Containers rely on the file system managed by the host operating system. Therefore, every container on one host uses the same file system and data structures. Sharing the same file system is a well-known issue of containers, and both the performance and security impacts have often been analyzed [15, 41]. On the other hand, each virtual machine and unikernels uses an entirely different file system. As a result, each guest operating system stores different data structures for all file operations. Such separation allows for better isolation, both in terms of performance and security [13]. However, even though the file systems of unikernels are separated, they still compete for the same hardware resources, and a misbehaving instance might still impact the performance of a well-behaving instance.

Motivated by the importance of file systems in operating systems, we explore different ways of attacking a system running file storage-intensive applications. Our file system experiments use an SQLite benchmark as a well-behaving instance. Furthermore, we use the same benchmark to assess the scalability of storage-intensive applications in Chapter 4. This benchmark measures the time to execute 60000 insert SQL database queries.

Operations with files often involve accessing the disk, and disk storage is a well-known bottleneck in computer performance. We use a temporary in-memory file system to avoid the overhead of accessing the disk in our analysis. Unikraft supports RamFS (short for Random Access Memory File System), a simple file system that stores both the file system data structures and file contents in the main memory. Docker supports TmpFS (short for Temporary File System). TmpFS is very similar to RamFS, a simple file system that stores the data in the main memory instead of storing it on a permanent disk. TmpFS is a more recent version of an in-memory file system, and it supports more advanced features like setting a storage limit. Still, the main functionalities in both TmpFS and RamFS are the same.

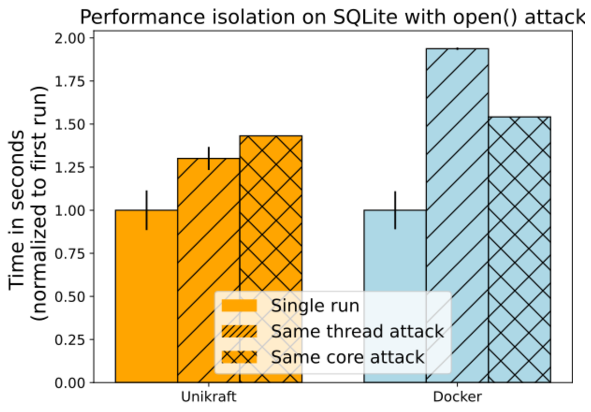
In this subsection, we will explore five different file system attackers. Each of the five attackers runs an infinite loop in which a different system call is executed:

- `open()` and `close()` system calls
- `read()` system call
- `write()` system call
- `stat()` system call
- `fork()` system call

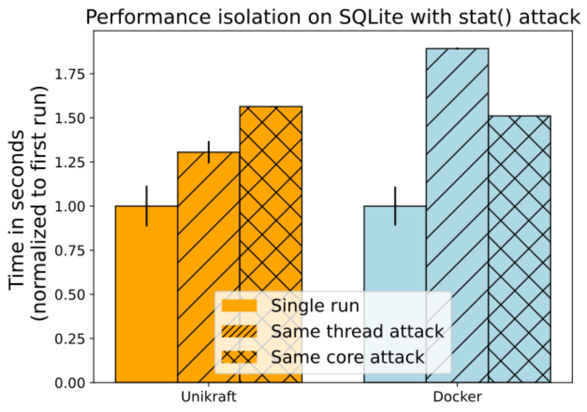
Figures 5.4a, 5.4b, and 5.4c present the benchmark slowdowns with the `open()`, `stat()`, and `fork()` attackers. We observe that the slowdowns for both Unikraft and Docker are similar to the CPU and memory attacks from the subsection 5.2.3 above. However, these system calls are well-engineered, and containers and virtual machines are optimized to handle such attackers with minimal influence.

Figures 5.4d and 5.4e present the benchmark slowdowns with the `read()` and `write()` attackers. For Docker, we notice similar behavior to the other attackers, without the new attackers significantly affecting the performance. However, for Unikraft, we can observe a more significant slowdown for the `write()` system call attack. The slowdown difference happens because the other system calls make great use of the cache available in the processor. On the other hand, the `write()` system call cannot make as extensive use of the cache as the other system calls, as it needs to write to the main memory frequently. Therefore, we speculate that the significant slowdown for Unikraft `write()` attack happens because both the application and the attacker are accessing the memory in parallel. Furthermore, the write system call is much more memory-intensive than the others. Table 5.1 presents the slowdowns of all the system call attackers discussed above for both Docker and Unikraft.

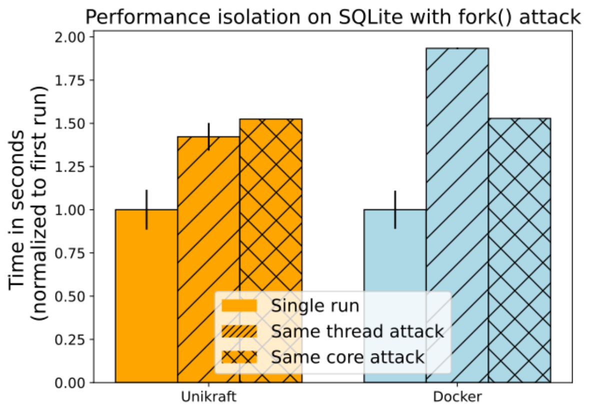
Although these experiments gave us good insights into the ability of performance isolation of Docker and Unikraft, we did not observe any significant bottlenecks by running the application and the attacker on the same thread or the same core. Therefore, we design a larger performance isolation experiment to put more stress on the systems. Namely, on the same processor, instead of launching the attacker on just one core, we launch an attacker on every hyper thread the processor has, apart from one core that we use for the application benchmark. By



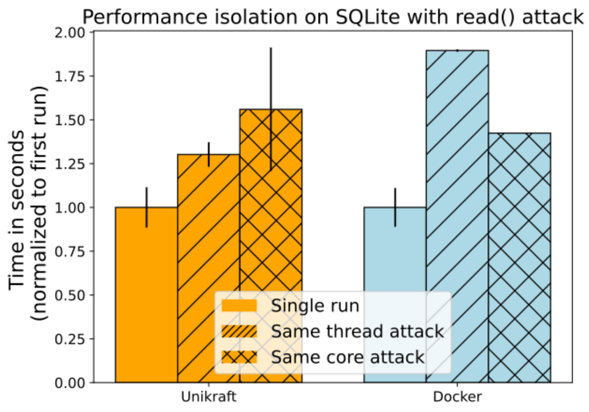
(a) open() attack results



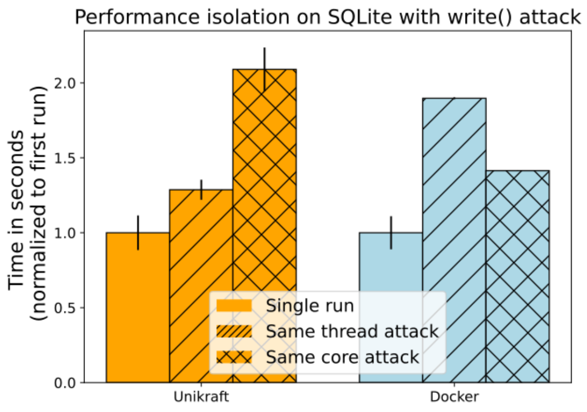
(b) stat() attack results



(c) fork() attack results



(d) read() attack results



(e) write() attack results

Figure 5.4: Performance Isolation Results of SQLite on Docker and Unikraft with different system call attacks (lower is better)

System call stressed	Thread slowdown	Core slowdown
`open` (Unikraft)	30.05%	43.15%
`open` (Docker)	93.72%	54.16%
`stat` (Unikraft)	30.53%	56.36%
`stat` (Docker)	89.24%	51.08%
`write` (Unikraft)	28.7%	109.02%
`write` (Docker)	89.73%	41.44%
`read` (Unikraft)	30.2%	55.98%
`read` (Docker)	89.56%	42.41%
`fork` (Unikraft)	42.2%	52.45%
`fork` (Docker)	93.37%	52.82%

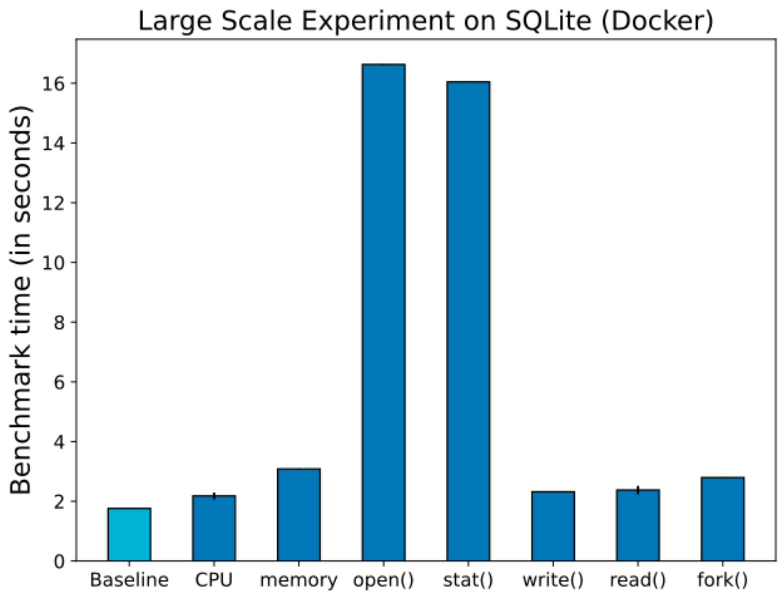
Table 5.1: Slowdowns for each SQLite system call attack

having a whole core for the benchmark application, we ensure the benchmark has full access to the cache and computing resources available at this core. However, all the attackers running on the processor share the primary L3 cache and the other shared memory resources.

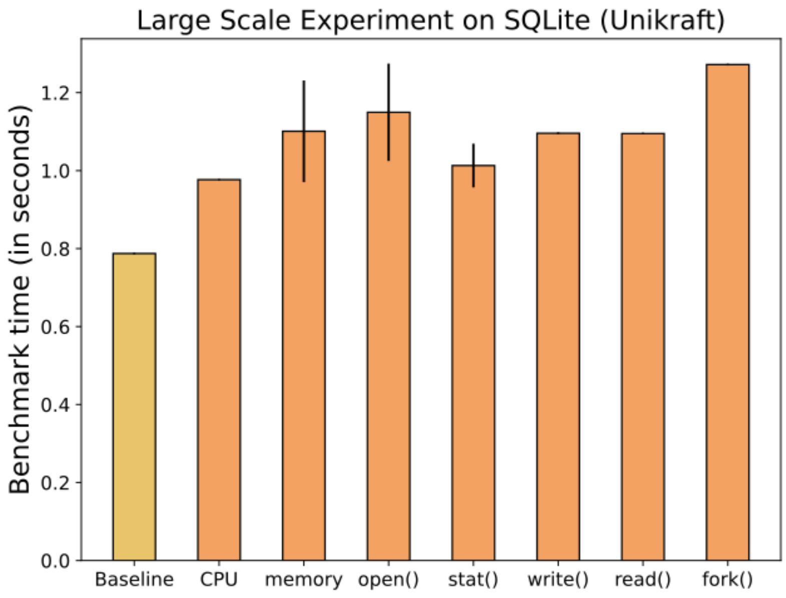
Figures 5.5a and 5.5b present the large-scale performance isolation experiment results for both Docker and Unikraft. Firstly, for Unikraft, we observe a significant slowdown between 10% and 50%. This slowdown is likely because all the attackers access the memory in parallel, and the memory becomes the device's bottleneck. Secondly, we observe a much more significant application slowdown for Docker. We can notice this for the `open()` and `stat()` system calls, giving around an 800% slowdown. The `open()` and `stat()` system calls are meant to be executed infrequently. Typical use cases for these system calls involve opening one file and performing many operations instead of constantly opening and closing the same file.

Misbehaving containers can often give such significant slowdowns. This has been well-researched and analyzed. Containers share the same host operating system. Therefore all containers share the same resources the OS kernel manages. These resources are limited, and executing one system call in many containers is an easy way to consume all the available resources [13].

Furthermore, all containers share exclusive resources, such as lock mechanisms required for all operating system functionalities. Misbehaving containers can easily stress both consumable and exclusive resources. Significant work has been done to improve container isolation [13]. However, such improvement requires significant engineering effort and analysis for every possible misbehaving container type. On the other hand, unikernels are entirely isolated by design, and most of their slowdowns happen because of physical limitations, such as the CPU or memory being unavailable.



(a) Docker Performance Isolation Results



(b) Unikraft Performance Isolation Results

Figure 5.5: Large Scale Performance Isolation Experiment of Docker and Unikraft (lower is better)

Chapter 6

Conclusions, Limitations and Future Works

6.1 Conclusions

Docker containers are inherently different from Unikraft virtual machines. Due to the significant design differences, the performance characteristics of Docker and Unikraft differ significantly.

In the scalability benchmarks, both systems perform better in different scenarios. For example, Unikraft outperforms Docker in the boot time and storage-intensive benchmarks. On the other hand, Docker performs better than Unikraft in the network-intensive benchmarks. The experiments we have designed also test the hypervisor QEMU. We observe that this hypervisor greedily uses memory and starts using swap memory after around 800 Unikraft instances. This heavily reduces Unikraft's performance in all benchmarks we have performed.

In the performance isolation benchmarks, we observe several different performance patterns. Unikraft outperforms Docker when the attacker runs on the same thread as the user application. However, Docker outperforms Unikraft when the attacker runs on the same core but a different hyper thread as the user application. Such behavior likely happens because Docker containers share more resources than Unikraft virtual machines, completely isolated by design. The initial performance isolation experiment did not stress the system enough, so we created larger-scale experiments. These experiments show the potential harms of using containers. Attackers using the `open()` and `stat()` system call slow down the user application by more than 800% on Docker. This slowdown suggests possible lock congestion, as these system calls heavily use locks to ensure only one process reads a file. Since all locks are shared between containers, lock congestions often happen.

In conclusion, we identified different performance bottlenecks in both Docker and Unikraft. We also identified scenarios in which the systems shine with their performance. Finally, as Unikraft is still in active development, we helped the community by identifying a list of issues. We list the issues discovered in section A.2 in the Appendix.

6.2 Challenges and Limitations

This project did not have a strict plan from the beginning. Since this project involved designing and trying new experiments, predicting how much time each would take was impossible. Furthermore, some of the scalability experiments took very long to execute. For example, completing the Redis on the Unikraft scalability benchmark took over 8 hours. Such slow execution is because of the issue in Unikraft's Redis library in v0.10.

Initially, the only focus of the project was assessing scalability. However, around the start of the second semester, we decided also to include performance isolation experiments, as these types of experiments are closely related and help us explore some characteristics of the systems in more detail.

Various difficulties were met while completing the work on this project. In the first part of the project, assessing scalability, the main issues focused on managing thousands of instances started on the same machine. In addition, there were often network issues connecting the QEMU hypervisor to the operating system network bridge. Furthermore, even though both systems are designed to perform well at large scale, they are designed to be set up for only a few instances. For example, Docker has a limit of 1000 instances to be connected to one network bridge. Therefore, we decided to limit all our scalability experiments to 1000 instances since 1000 instances is already a large enough number and much larger than the available computing units on the machine.

Minor issues like the ones described above were often easy and quick to resolve. However, there were often more significant issues that needed careful decisions. For example, since Unikraft is a recently developed tool, its support for user applications could be improved. In our search for a candidate for evaluating a storage-intensive application, we were interested in more advanced database tools, like MySQL or PostgreSQL. However, these databases must still be ported to Unikraft, and only SQLite is supported.

Furthermore, as Unikraft is still actively developing, its main features often change, and new versions constantly emerge. Therefore, we always tried to use the newest version. However, this was only sometimes possible, as the newest versions had issues stopping us from progressing.

Finally, many experiment ideas did not give any relevant results and were ignored. One such example is a parallel benchmark for testing many Redis instances simultaneously. We had to launch more than 100 simultaneous connections to get relevant results for the systems, while Redis ran on Unikraft breaks after starting more than 57 connections [3].

6.3 Future Works

The performance evaluation of virtualization systems has much potential for future research. Firstly, extending the project with more applications, such as more relevant database tools, would be suitable once they are ported and available in Unikraft. Secondly, modifying the benchmark CLI to test any unikernel, hypervisor, or container type independently would be highly beneficial for identifying issues and bottlenecks in new and different systems. Finally, going deeper into the results and identifying the root causes for some performance bottlenecks would be a significant step toward creating better virtualization systems. For example, such work might include exploring why the Docker boot time increases as we boot many instances or why the Unikraft Nginx performance decreases as we boot many instances.

Bibliography

[1] A. B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd international workshop on Software and performance*, pages 195–203, 2000.

[2] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. T. Morris, N. Zeldovich, et al. An analysis of linux scalability to many cores. In *OSDI*, volume 10, pages 86–93, 2010.

[3] J. Cameron. System breaks at 58 connections. 57 connections produces good results. `https://github.com/unikraft/app-redis/issues/9`, 2022.

[4] V. G. da Silva, M. Kirikova, and G. Alksnis. Containers for virtualization: An overview. *Applied Computer Systems*, 23(1):21–27, 2018.

[5] Datadog. 9 insights on real-world container use. `https://www.datadoghq.com/container-report/#6`, 2022.

[6] T. Deshane, D. Dimatos, G. Hamilton, M. Hapuarachchi, W. Hu, M. McCabe, and J. N. Matthews. Performance isolation of a misbehaving virtual machine with xen, vmware and solaris containers. *submitted to USENIX*, 2006.

[7] W. Glozer. wrk: Modern http benchmarking tool. `https://github.com/wg/wrk`, 2021.

[8] C. Gong, J. Liu, Q. Zhang, H. Chen, and Z. Gong. The characteristics of cloud computing. In *2010 39th International Conference on Parallel Processing Workshops*, pages 275–279, 2010.

[9] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing performance isolation across virtual machines in xen. In *Middleware 2006: ACM/IFIP/USENIX 7th International Middleware Conference, Melbourne, Australia, November 27-December 1, 2006. Proceedings 7*, pages 342–362. Springer, 2006.

[10] R. D. Hipp. SQLite, 2023.

[11] S. v. Kalken. What are namespaces and cgroups, and how do they work? `https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/`, 2021.

[12] S. Kashyap, C. Min, and T. Kim. Opportunistic spinlocks: Achieving virtual machine scalability in the clouds. *ACM SIGOPS Operating Systems Review*, 50(1):9–16, 2016.

[13] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ş. Teodorescu, C. Răducanu, et al. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 376–394, 2021.

[14] Y. Li, J. Zhang, C. Jiang, J. Wan, and Z. Ren. Pine: Optimizing performance isolation in container environments. *IEEE Access*, 7:30410–30422, 2019.

[15] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou. A measurement study on linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 418–429, New York, NY, USA, 2018. Association for Computing Machinery.

[16] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 461–472, New York, NY, USA, 2013. Association for Computing Machinery.

[17] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, page 461–472, New York, NY, USA, 2013. Association for Computing Machinery.

[18] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens. Quantifying the performance isolation properties of virtualization systems. In *Proceedings of the 2007 workshop on Experimental computer science*, pages 6–es, 2007.

[19] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[20] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, Dec. 1995.

[21] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[22] Nginx. Configuring nginx as a mail proxy server. `https://docs.nginx.com/nginx/admin-guide/mail-proxy/mail-proxy/`, 2023.

[23] Nginx. Http load balancing. `https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/`, 2023.

[24] V. Nitu, P. Olivier, A. Tchana, D. Chiba, A. Barbalace, D. Hagimont, and B. Ravindran. Swift birth and quick death: Enabling fast parallel guest boot and destruction in the xen hypervisor. *ACM SIGPLAN Notices*, 52(7):1–14, 2017.

[25] P. Olivier. Chrono. `https://github.com/olivierpierre/chrono`, 2016.

[26] B. B. Rad, H. J. Bhatti, and M. Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[27] Redis. Redis benchmark. `https://redis.io/docs/management/optimization/benchmarks/`, 2023.

[28] B. Ruan, H. Huang, S. Wu, and H. Jin. A performance study of containers in cloud environment. In *Advances in Services Computing: 10th Asia-Pacific Services Computing Conference, APSCC 2016, Zhangjiajie, China, November 16-18, 2016, Proceedings 10*, pages 343–356. Springer, 2016.

[29] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 121–135, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Statista. Worldwide market share of leading cloud infrasturcture service providers. https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/ last accessed 2 Apr. 2023, 2022.

[31] Z. Tavaria. Now the world's #1 web server, nginx looks forward to an even brighter future. https://www.nginx.com/blog/now-worlds-1-web-server-nginx-looks-forward-to-even-brighter-future/, 2023.

[32] Unikraft. Unikraft eurosys'21 artifacts. https://github.com/unikraft/eurosys21-artifacts, 2021.

[33] Unikraft. Unikraft "hello world" application. https://github.com/unikraft/app-helloworld, 2023.

[34] A. Velichkovski. Assessing scalability and performance isolation github repository. https://github.com/andrejvelichkovski/assessing-scalability, 2022.

[35] A. Velichkovski. Nginx stops processing requests on new version of uk. https://github.com/unikraft/app-nginx/issues/9, 2022.

[36] A. Velichkovski. Redis on unikraft uses 100% of the cpu available. https://github.com/unikraft/app-redis/issues/12, 2022.

[37] A. Velichkovski. Unikraft redis doesn't start with kraft run, works correctly when started with qemu. https://github.com/unikraft/lib-redis/issues/8, 2022.

[38] A. Velichkovski. Unikraft redis stops processing requests (crashes) on increased load. https://github.com/unikraft/lib-redis/issues/9, 2022.

[39] B. Verghese, A. Gupta, and M. Rosenblum. Performance isolation: Sharing and isolation in shared-memory multiprocessors. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, page 181–192, New York, NY, USA, 1998. Association for Computing Machinery.

[40] VMWare. What is a bare metal hypervisor? https://www.vmware.com/topics/glossary/content/bare-metal-hypervisor.html, 2023.

[41] K. Wang, S. Wu, K. Suo, Y. Liu, H. Huang, Z. Huang, and H. Jin. Characterizing and optimizing kernel resource isolation for containers. *Future Generation Computer Systems*, 141:218–229, 2023.

[42] X. Wang, J. Du, and H. Liu. Performance and isolation analysis of runc, gvisor and kata containers runtimes. *Cluster Computing*, 25(2):1497–1513, 2022.

[43] M. G. Xavier, I. C. De Oliveira, F. D. Rossi, R. D. Dos Passos, K. J. Matteussi, and C. A. De Rose. A performance isolation analysis of disk-intensive workloads on container-based clouds. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 253–260. IEEE, 2015.

[44] X. Xu, F. Zhou, J. Wan, and Y. Jiang. Quantifying performance properties of virtual machine. In *2008 International Symposium on Information Science and Engineering*, volume 1, pages 24–28, 2008.

Appendix A

Benchmark appendix

A.1 Experiment Code Names

Experiment code	Experiment description
`uk_boot`	Unikraft Boot Microbenchmark
`d_boot`	Docker Boot Microbenchmark
`uk_mem`	Unikraft Memory Usage Microbenchmark
`d_mem`	Docker Memory Usage Microbenchmark
`uk_ng_s`	Unikraft Nginx Scalability Benchmark
`d_ng_s`	Docker Nginx Scalability Benchmark
`uk_re_s`	Unikraft Redis Scalability Benchmark
`d_re_s`	Docker Redis Scalability Benchmark
`uk_sql_s`	Unikraft SQLite Scalability Benchmark
`d_sql_s`	Docker SQLite Scalability Benchmark
`uk_nginx_perf_iso`	Unikraft Nginx Performance Isolation Benchmark
`d_nginx_perf_iso`	Docker Nginx Performance Isolation Benchmark
`uk_redis_perf_iso`	Unikraft Redis Performance Isolation Benchmark
`d_redis_perf_iso`	Docker Redis Performance Isolation Benchmark
`uk_sqlite_perf_iso`	Unikraft SQLite Performance Isolation Benchmark
`d_sqlite_perf_iso`	Docker SQLite Performance Isolation Benchmark

Table A.1: Code name for each benchmark experiment implemented

A.2 Issues identified in Unikraft

A.2.1 Redis on Unikraft uses 100% of the CPU available

This issue was identified as part of the Redis Unikraft scalability experiment. It was already known to the authors of the Redis library for Unikraft. It is resolved in Unikraft v0.11 version [36].

A.2.2 Unikraft Redis stops processing requests (crashes) on increased load

This issue was identified in Unikraft v0.11 version, when trying to switch to newer version for the Performance Isolation experiments. It is resolved in Unikraft v0.12 version [38].

A.2.3 Unikraft redis doesn't start with kraft run, works correctly when started with Qemu

This issue was identified while prototyping different performance isolation experiments [37].

A.2.4 Nginx stops processing requests on new version of UK

This issue was identified in the same time as the issue described in subsection A.2.2. The root cause is the same in both issues, and it has been resolved in Unikraft v0.12 version [35].