

Juhani Säteri

COMPARISON OF CONTAINER RUNTIMES IN EMBEDDED CONTEXT

Master's thesis
Faculty of Information Technology and Communication Sciences
Supervisor: David Hästbacka
October 2022

ABSTRACT

Juhani Säteri: Comparison of container runtimes in embedded context
Master's thesis
Tampere University
Master's Programme in Information Technology
October 2022

This master's thesis takes a look at low-level containerization components called container runtimes from the perspective of embedded devices. All of the runtimes chosen followed the Open Container Initiative specification. Container performance, memory usage, install size, dependencies and portability was evaluated during the testing phase. The performance was measured with 7z (compression / decompression) and Redis database benchmark. Portability was tested with a complex Python application that was moved across CPU architectures to test if any of the container runtimes offered portability benefits.

The testing identified differences in performance characteristics between hardware architectures, but it remained unclear whether hardware or resource constraints caused the difference. Performance was also shown to be affected by the added security mechanisms of more complex container runtimes. On portability none of the container runtimes were identified to have any benefits when it comes to portability. All of the runtimes had reasonable dependencies that can be fulfilled easily by any builds of Linux, but some of the runtimes required hardware features that might be harder to fulfill on embedded devices.

Memory usage showed different memory usage patterns for different runtimes and the overhead was estimated to be less than 50 MB at best. At worst the overhead was ≈ 460 MB on some runtimes. Turned out that some of the runtimes utilized less memory than others even though all of them had the same amount available.

Keywords: containerization, container runtime, embedded device, runc, crun, gVisor, Kata-containers, virtualization, Docker, embedded development

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

TIIVISTELMÄ

Juhani Säteri: Konttiajoympäristöjen vertailu sulautetussa laitteessa.

Diplomityö

Tampereen yliopisto

Tietotekniikan DI-ohjelma

Lokakuu 2022

Tämän diplomityön tarkoituksena on tutkia konttiajoympäristöjä sulautettujen laitteiden näkökulmasta. Kaikkien ajoympäristöjä pitää täyttää "Open Container Initiative"spesifikaatio. Konttien suorituskykyä, muistin käyttöä, asennuksen kokoa, riippuvuuksia ja siirrettävyyttä arvioitiin testausvaiheessa. Suorituskykyä mitattiin 7z suorituskykytestin avulla (pakkaaminen / purkaminen) ja Redis suorituskykytestin avulla. Siirrettävyyttä testattiin siirtämällä konttia prosessori arkkitehtuurilta toiselle, jotta löydetään tarjoavatko kontti ajoympäristöt siirrettävyys hyötyjä.

Testaus löysi eroja rautaarkkitehtuurien välillä, mutta jäi epäselväksi aiheutuivatko erot eri rautaarkkitehtuurista vai alustojen suorituskykyerosta. Turvallisuutta lisäävien ominaisuuksien todettiin huonontavan suorituskykyä. Siirrettävyyden osalta mikään konttiajoympäristö ei tarjonnut tukea suoraan eri rautaarkkitehtuureilta tullessiin kontteihin. Kaikkilla konttiajoympäristöillä oli järkevät riippuvuudet joiden täyttäminen sulautetulla laitteella ei pitäis tuottaa ongelmia. Osa konttiajoympäristöistä tarvitsi kuitenkin tukea rautatasolla.

Muistin käytön testaus paljasti erillaisia muistin käyttö tapoja ajoympäristöjen välillä ja ajoympäristöjen resurssikustannukset muistin osalta arvioitiin olevan parhaassa tapauksessa alle 50 megatavua. Huonoimmassa tapauksessa kuitenkin resurssikustannukset olivat noin 460 megatavua. Kävi ilmi että osa ajoympäristöistä käytti vähemmän muistia kuin toiset vaikka kaikilla oli saman verran muistia saatavissa.

Avainsanat: kontti, konttiajoympäristö, sulautetut laitteet, runc, crun, gVisor, Kata-containers, virtualisointi, Docker, sulautettujen järjestelmien kehitys

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

PREFACE

Firstly I want to thank Hannu Matti Järvinen for helping me reach out to Tampere University faculty members for subjects of this thesis. Secondly I would like to thank David Hästbacka for helping me find a subject that I found interesting. David also provided me with invaluable help during the writing process and helped me to narrow down the subject to a reasonable size. Finally none of this would have been possible without my family and friends encouraging me to finish my studies.

Tampere, 30th October 2022

Juhani Säteri

CONTENTS

1.	Introduction	1
2.	Background	3
2.1	Virtualization	3
2.1.1	Virtual machine	3
2.1.2	Virtualization in containers	4
2.2	Short history of containers	4
2.3	OCI (Open Container Initiative)	5
2.3.1	Image specification	5
2.3.2	Container runtime specification	8
2.3.3	Native runtime	10
2.3.4	Sandboxed runtime	10
2.3.5	Virtualized runtime	11
2.4	Related work	12
3.	Problem analysis	15
3.1	Challenges and benefits of using containers in embedded devices?	15
3.2	Ideal state	16
3.3	Testing methodology	18
3.3.1	Portability	19
3.3.2	Storage space and dependencies	20
3.3.3	RAM usage and performance	20
3.3.4	CPU performance	20
3.3.5	Database performance	20
4.	Testing results	22
4.1	Portability	22
4.2	Database performance (Redis)	26
4.2.1	ARM results	27
4.2.2	x86 results	28
4.3	7z benchmark results	30
4.3.1	x86 results	30
4.3.2	ARM results	31
4.4	Memory usage on Redis benchmark	32
4.4.1	ARM result	32
4.4.2	x86 results	36
4.5	Memory usage on the 7z benchmark	42
4.5.1	ARM results	42

4.5.2	x86 results	46
4.6	Dependency analysis	52
5.	Result analysis	53
5.1	Portability	53
5.2	Database performance	54
5.3	7z benchmark	55
5.4	Memory usage and storage space	57
6.	Conclusion	59
	References	60
	Appendix A: Appendix	63
A.1	Redis benchmark ARM results raw version	63
A.2	Redis benchmark x86 results raw version	66

1. INTRODUCTION

During the recent years we have seen a huge boom in IoT, edge computing and cloud based data management. It is more important to be able to deploy IoT devices that can do some of the computation on the device. Many of the current applications in the cloud run in containers for their ease of deployment and due to the rise of microservice architectures, but embedded devices have not yet seen wide adoption of containers. What this master's thesis aims to investigate, is the different container runtimes available for Open Container Initiative (OCI) compliant containers. These container runtimes can be divided into a few different types; native runtimes, sandboxed runtimes and virtualized runtimes. None are clearly marketed towards embedded devices, but one might expect native runtimes to offer higher performance compared to virtualized runtimes.

Container runtimes are the lowest level before actual container of Docker or Podman which most people commonly use when creating containers. The container runtime is ultimately the last piece of the puzzle before the operating system and hardware. On top of a container runtime is usually a container manager, that will use the container runtime to create new containers and manages networks.

The metrics used for evaluating the container runtimes suitability for embedded devices are resource consumption (storage space and RAM usage), runtime dependencies, performance and portability. These were identified to be the most common bottlenecks in embedded devices. Storage space and ram are quite self explanatory, but portability refers to the ability to deploy one containerized application to any type of embedded device or to the cloud. To achieve portability the container runtime would have to be able to convert x86 binary to ARM and vice versa. Portability can also be achieved with CPU architecture agnostic containers.

There is no perfect test to test all the wanted metrics at once. The testing has been divided to multiple smaller tests that measure individual metrics at a time. Testing directly on the hardware will be used as a baseline to determine what can be expected from containers. All runtimes are expected to be running on top of a Linux operating system. This masters thesis aims to answer the questions: Do current container runtime implementations provide portable environments that would be viable to use in embedded devices, where the storage space, RAM and performance is limited? How well do these container runtimes

perform in terms of memory usage (RAM and storage), database performance, and compression/decompression? The diagram below illustrates where a container runtime is in relation to high level tools such as Podman, or Kubernetes.

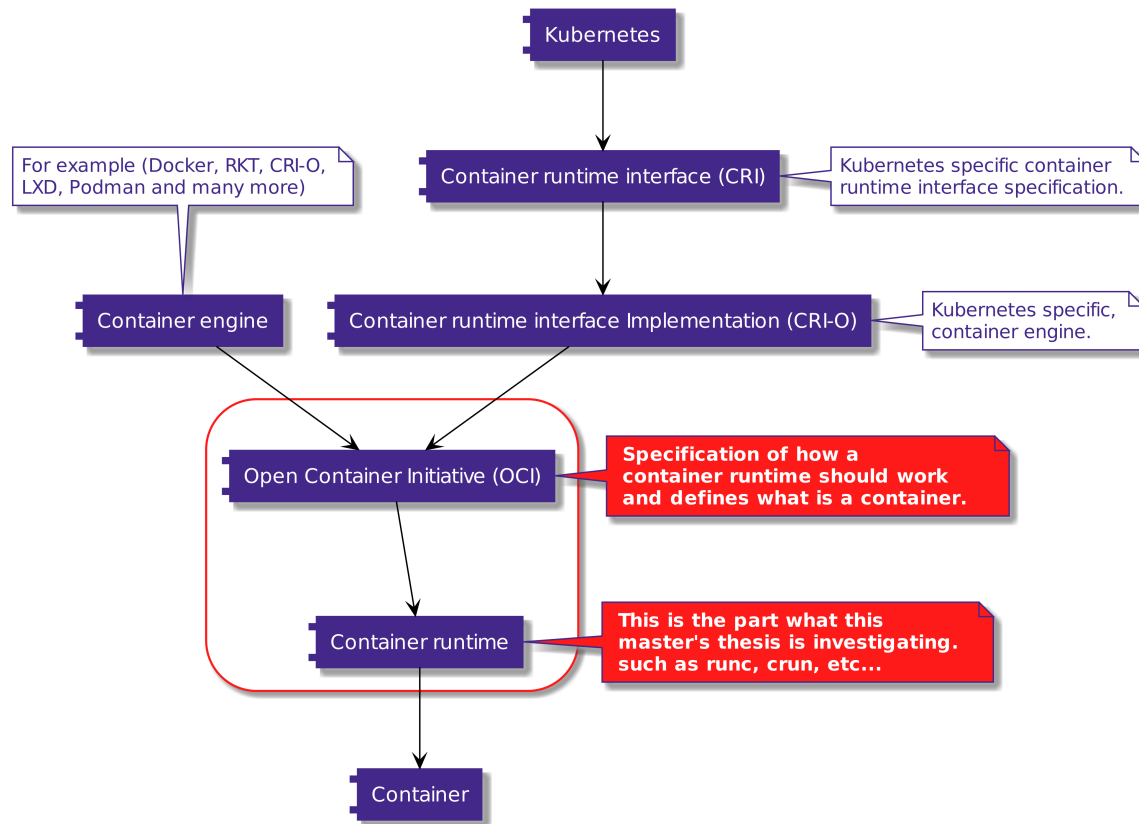


Figure 1.1. Component diagram representing connection from high level tools to containers. [42, 19, 12]

The testing of different container runtimes implementations was conducted on two different hardware platforms. One that was a Lenovo e480 laptop using Intel Core i5-8250U x86 processor and the other one was Raspberry Pi 4 model B with its Broadcom BCM2711 CPU. For benchmarking Redis-benchmark and 7z were used. In addition to raw benchmarks the portability was tested by moving a complex container across hardware architectures. Finally the dependencies and install size of the runtimes was calculated.

2. BACKGROUND

2.1 Virtualization

In order to understand containers and what they try to achieve we need to be able to understand what is virtualization? Wikipedia gives a good description of what is virtualization: "In computing, virtualization or virtualisation (sometimes abbreviated v12n, a numeronym) is the act of creating a virtual (rather than actual) version of something at the same abstraction level, including virtual computer hardware platforms, storage devices, and computer network resources" [57]. So instead of having a physical processor you would create a virtual one. The trick in the context of computing is to make the virtual one indistinguishable from the physical one. So in case of a processor the operating system should not be able to tell the difference between a virtual processor and a physical one.

2.1.1 Virtual machine

One of the more common applications for virtualization is a virtual machine. A virtual machine is a computer that is entirely or partially virtual. Meaning all the hardware is made with software instead of using physical hardware. To note in some cases it might be beneficial for the virtual machine to interact directly with hardware. The virtualization is achieved with a software component called a hypervisor. A hypervisor will manage and provide the virtual machine with all virtual hardware resources it needs.

There are two types of hypervisors available. The first one is called a type 1 hypervisor, which is located directly on top of the hardware compared to type 2 hypervisor which is running on top of an already existing operating system. There are many advantages to virtual machines that are especially important for big data centers where you are expected to run multiple applications or multiple operating systems on the same hardware. Since the same hardware can be used by multiple operating systems, you will not need one server for one task anymore. This will increase the density of data centers and most of the hardware won't have to sit idle. The hardware utilization will be better since applications can be scaled up on demand or brought down when they are not needed anymore. For example the company email server will not see much traffic during the weekend and can

be scaled down to let more computationally intensive tasks use the hardware. Availability will also increase since the virtual machine can be moved from one server to another. [40]

2.1.2 Virtualization in containers

You could say on containers that the virtualization is moved one level higher compared to virtual machines. A virtual machines relies on virtualized hardware to operate. To use this analogy in a container, it expects the operating system to be virtualized. How the operating system level virtualization is achieved is by providing the running containers with all the resources that they would usually request from the operating system. So for example when a container requests access to the memory it will be provided these resources as if it was running on a normal operating system instead of a container. [47]

2.2 Short history of containers.

The definition of a container according to google cloud is "Containers are packages of software that contain all of the necessary elements to run in any environment" [17]. How this is achieved though is left out of the definition. To understand how containers came to be we need to look briefly at their history. One of the first programs that could be considered to be a container, would be chroot, that was released on to the FreeBSD operating system source code control system on 18 of March 1982 [14]. Chroot runs a command in a different root environment for added security [15]. By current day standards chroot, is not a container, since the security it provides is not good enough for making a secure container. Even though chroot by itself is not a secure container it is a step that modern day containers still use and the principle of it has been expanded to make modern day containers.

The concept of chroot was expanded on the FreeBSD operating system by a program called jail. Jail added the containerization of files, processes and super user accounts. This meant an increase in the security of containers, by limiting what the process could interact with [5, Chapter 15.1 Jails]. A chrooted process can interact with the network stack or other processes, but a jailed process on the other hand has it's own network, file system and user accounts. In a chrooted process, one could easily tell which was the host system and which was the containerized process, but on a jailed process the lines are starting to become blurred. It is not so straight forward to tell whether the process is running in a container or natively.

After jail, came a patch proposal in 29 of May 2007 called process containers, made by developers from google domain addresses to include container functionality on the Linux kernel. The initial patch proposed an interface that would achieve the same functionality

as the existing cgroup pseudo file system [43]. Cgroup made it possible for the kernel to limit the amount of memory and processors available to the processes. The problem with this approach though was that memory could only be shared in NUMA (non-uniform memory access) systems [7]. In the new proposal this would instead be made possible without requiring NUMA hardware support. The patch saw a few iterations and was merged to the main line kernel in version 2.6.24 [6]. Currently this functionality is more commonly known as control groups (cgroups for short).

The first application to use the new functionality added to the Linux kernel was LXC (Linux Container). The goal of LXC was to make it possible to use the new kernel functionality with a user space program. The first release of LXC was made in 1 of August 2008 [25], but the 1.0 took until 20th of February 2014 to be finalized [48]. LXC uses control groups to isolate the containers, so that it can only use limited amount of resources. It also uses similar methods to FreeBSD jail so that processes have their own accounts, network interfaces, files and so on. What LXC achieves is a full container by today's standards.

For modern day developers a container is the same as Docker. But as a matter of fact when Docker was first launched it used LXC as the backend for containers. Later on Docker has developed their own container runtime, which it later open sourced and it is currently known by the name of runc [46, 10]. Runc is the current reference implementation of the Open container runtime specification.

2.3 OCI (Open Container Initiative)

As we saw on the previous section containers were not just suddenly made they were gradually built from the concept of chroot. There was no clear specification on what a container is and it is still up to debate what is considered a container. For the scope of this work though we are focusing on containers as defined by the Open Container Initiative OCI. As for what is OCI? "The Open Container Initiative is an open governance structure for the express purpose of creating open industry standards around container formats and runtimes." [12]" OCI manages the specification for container runtimes and images, called container runtime specification and image specification respectively.

2.3.1 Image specification

The image specification defines what a container image is, how it can be shared and how it is structured. All container images according OCI consists of a selection of container components. There are currently 11 component types supported by the specification and this number could increase in the future, when new versions of the specification are released. All of the components defined are stored as blobs. Blobs are just opaque data containers that can hold information connected to any of the component types defined.

Table 2.1. Layers described by OCI [54].

application/vnd.oci.descriptor.v1+json	Content Descriptor
application/vnd.oci.layout.header.v1+json	OCI Layout
application/vnd.oci.image.index.v1+json	Image Index
application/vnd.oci.image.manifest.v1+json	Image manifest
application/vnd.oci.image.config.v1+json	Image config
application/vnd.oci.image.layer.v1.tar	"Layer", as a tar archive
application/vnd.oci.image.layer.v1.tar+gzip	"Layer", as a tar archive compressed with gzip
application/vnd.oci.image.layer.v1.tar+zstd	"Layer", as a tar archive compressed with zstd
application/vnd.oci.image.layer.nondistributable.v1.tar	"Layer", as a tar archive with distribution restrictions
application/vnd.oci.image.layer.nondistributable.v1.tar+gzip	"Layer", as a tar archive with distribution restrictions compressed with gzip
application/vnd.oci.image.layer.nondistributable.v1.tar+zstd	"Layer", as a tar archive with distribution restrictions compressed with zstd

Below is a list of all the component types supported by the specification at the time of writing.

Starting from the top a content descriptor is the method used for connect components to each other. Below is a example of content descriptor to a image manifest.

```
{
  "mediaType": "application/vnd.oci.image.manifest.v1+json",
  "size": 7682,
  "digest": "sha256:5b0bcabd1...e94a4333501270"
}
```

This particular descriptor is pointing to a image manifest, with the size of 7682 bytes and the hash value of sha256:5b0b..., where SHA-256 is the hashing algorithm used. Other hashing algorithm can also be used and the specification currently supports SHA-512 in

addition to SHA-256. The hash value here is used to address the blob that holds the actual image-manifest component. The connections make up a tree structure called the Merkle tree [53]. In a Merkle tree every node has a hash value, as can be seen from the content descriptor. When you add the fact that nodes and leafs carry opaque payloads (blobs) you get a Merkle Directed Acyclic Graph [35]. This is the structure that is used to store a container image, or a collection of container images.

The highest level component in the tree structure is called image index, which is just a list of image manifests. The only container specific information that is available at this point is the targeted platform of the container. This information includes architecture, features and the OS of the targeted platform and it is related to a single image manifest. So for example when downloading a container for a Linux host, you would check that the architecture of your PC matches the one defined for this specific image manifest and that you have the required features available on your container host. Defining the platform on the image index is not mandatory. So if you were making a generic container that would work on any operating system no matter the features or version, you would leave this empty.

Moving down on the tree from image index. The next component is called image manifest. When making a container only the image manifest and the components connected to it are required. The image index is just a way to store multiple different types of containers, that might be targeted at different platforms or that serve different applications. For the image manifest to make a single container, it needs to record the files system changes and the configuration of the container. File system changes are represented with layers. Each layer represents file changes on the root file system of the container. There are numerous file types available in addition to regular files and folders such as sockets, block devices and FIFOs. Sockets are used to defined what kind of ports are available to the container, block devices refer to hard drive or solid state drives, and FIFOs (first in first out) are queues.

Configuration of the image is held in a component called image config. In the configuration; details such as operating system, CPU architecture, author, date of creation, exposed ports, volumes, working directory of the container are defined. As can be seen all the possible things that can be configured on a container are defined here. The OCI specification also describes how the image manifest can be converted to something that the container runtime specification can use. Below is an illustration of the above mentioned components as a component diagram. The illustration showcases the connections between different OCI components. [54] To note though there can be one to many image manifest connected to a single image index and the same applies to the connection from image manifest to image layer.

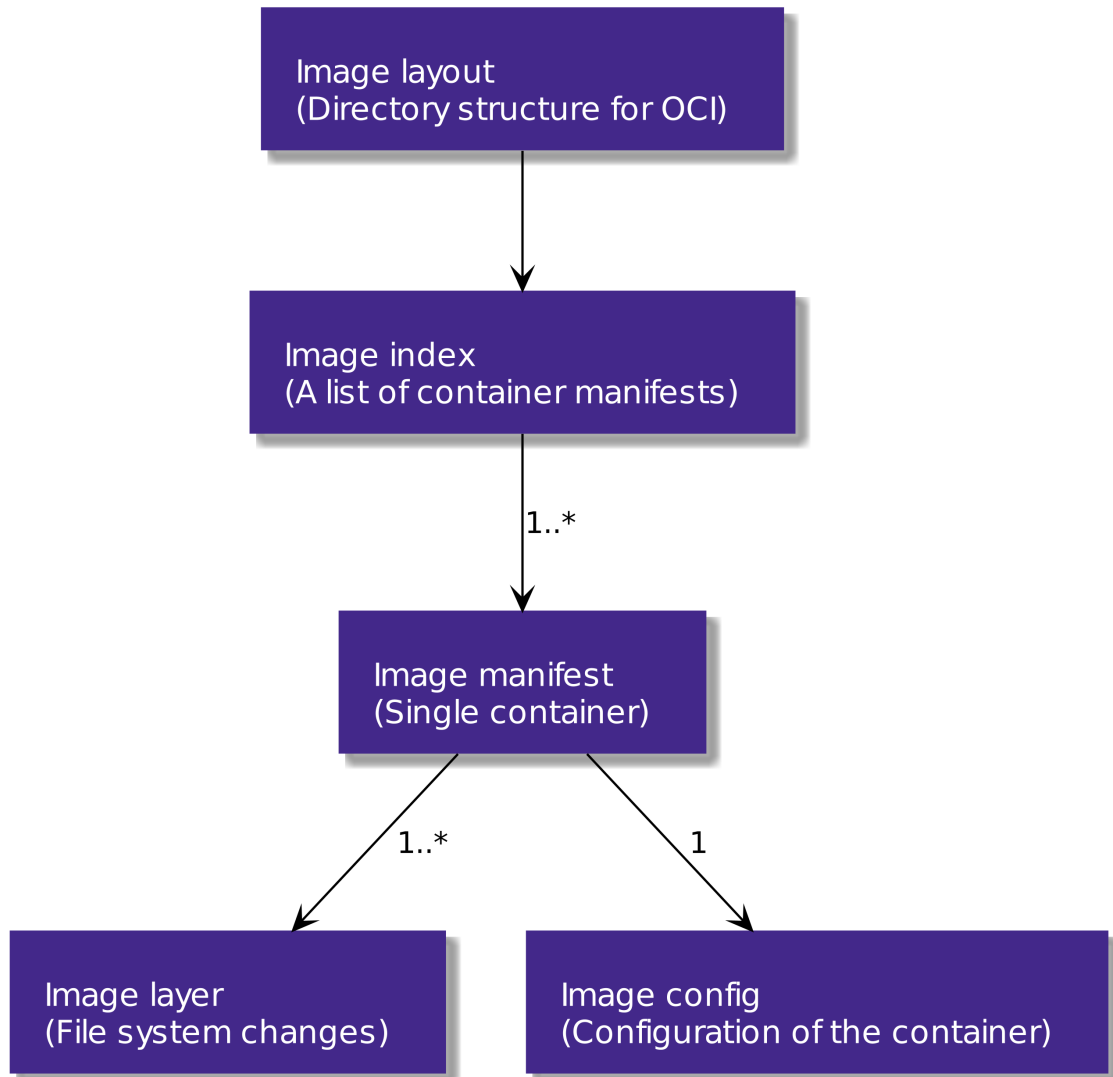


Figure 2.1. OCI image specification components

2.3.2 Container runtime specification

The container image specification defined a container and how it can be shared. The runtime will then use the information that is provided by the container image specification and create the actual container and do the steps required to make it run. When it comes to security the details are defined to be out of scope for the OCI runtime specification [55]. This fact needs to be investigated more, whether it has any effect on runtime security. It most likely means that the methods used in making containers, will need to provide the required amount of isolation. Security is up to the container runtime implementation! The methods to achieve a secure container runtime are implemented by the Linux kernel. Other operating systems also provide methods for making containers, but on this work we will be focusing on the Linux kernel, due to the open nature of Linux implementation and the wide use of Linux kernels on the cloud.

In order to make a working Linux container, some paths will need to be setup to make the container work. These paths cannot be provided by the host of the container since it would undermine the security of the container. A Linux container most likely expects these file paths to be set up:

<code>/proc</code>	Provides a view inside the Linux kernel where certain kernel parameters can be changed, but most importantly it shows the running processes. [2]
<code>/sys</code>	Shows information about the hardware devices and their kernel information related to them.
<code>/dev/pts</code>	Pseudo terminals.
<code>/dev/shm</code>	A temporary file storage.

Normally processes can see each other, users, networks and much more. To limit this visibility the kernel provides a method called namespaces to limit what the process can see. For example when setting up a container and giving it a process identifier namespace (PID), the container can only see other processes that are defined to be on the same namespace. The host system can see the processes of the container, but with different PIDs. Because of this limited visible inside the container the runtime specification outlines the use of POSIX sockets to send the process information outside of the container. The following namespaces are available to limit the containers ability to see the host system [31]:

<code>cgroup</code>	cgroups limit the RAM, CPU, devices, block device access and network priority available to the container. [3]
<code>IPC</code>	Limits how the container can use IPC (inter process communication) between processes. Isolates it from the host system. [20]
<code>Network</code>	Isolates network devices, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, port numbers (sockets) and so on. [37]
<code>Mount</code>	Limits what mounts the container can see. [36]
<code>PID</code>	Limits what processes the container can see [39]
<code>Time</code>	Create virtual instances of system clocks available. [49]
<code>User</code>	Isolates the user of the container. Inside the container users can have root privileges but outside of the container they are unprivileged users. [50]
<code>UTS</code>	Provides an isolated hostname and NIS (network information service) name. [51]

Cgroups take care of limiting device access, but it is up to the runtime how it will provide the allowed device access. In addition to limiting devices, system calls can also be limited.

To limit system calls the amount of information exposed by system calls a method called `seccomp` is available on the Linux kernel [32]. `seccomp` provides a way to filter system calls received by the process and this is used in the OCI runtime specification to limit the amount of information exposed by system calls. There are more complex details available on the specification, but this should provide a good overview on how access is limited and what kernel methods are used.

2.3.3 Native runtime

A native runtime as its name suggest runs natively on the system without the need for any hardware support. The container will use the same kernel as any other process. So when a container is made it is isolated with previously mentioned methods. A running container will use the kernel directly, but with limited visibility. From a security perspective a bug in the kernel may allow process to go outside of its defined bounds. This also means that once the process is running it will most likely have performance that is close to bare metal since it will use the kernel directly, only having limited visibility. The default runtime that ships with Docker called `runc` is a native runtime.

2.3.4 Sandboxed runtime

A sandbox as defined by wikipedia is "In computer security, a sandbox is a security mechanism for separating running programs, usually in an effort to mitigate system failures and/or software vulnerabilities from spreading" [56]. A sandboxed runtime will increase the isolation as was defined by the OCI specification by not allowing the container to directly interact with the Linux kernel, but through a layer. A sandboxed process cannot interact directly with the kernel, everything goes through the sandbox interface, which will increase the security. This interface layer in between the host kernel and the container will intercept the calls made to the kernel and decide whether they should go through or not. [18]

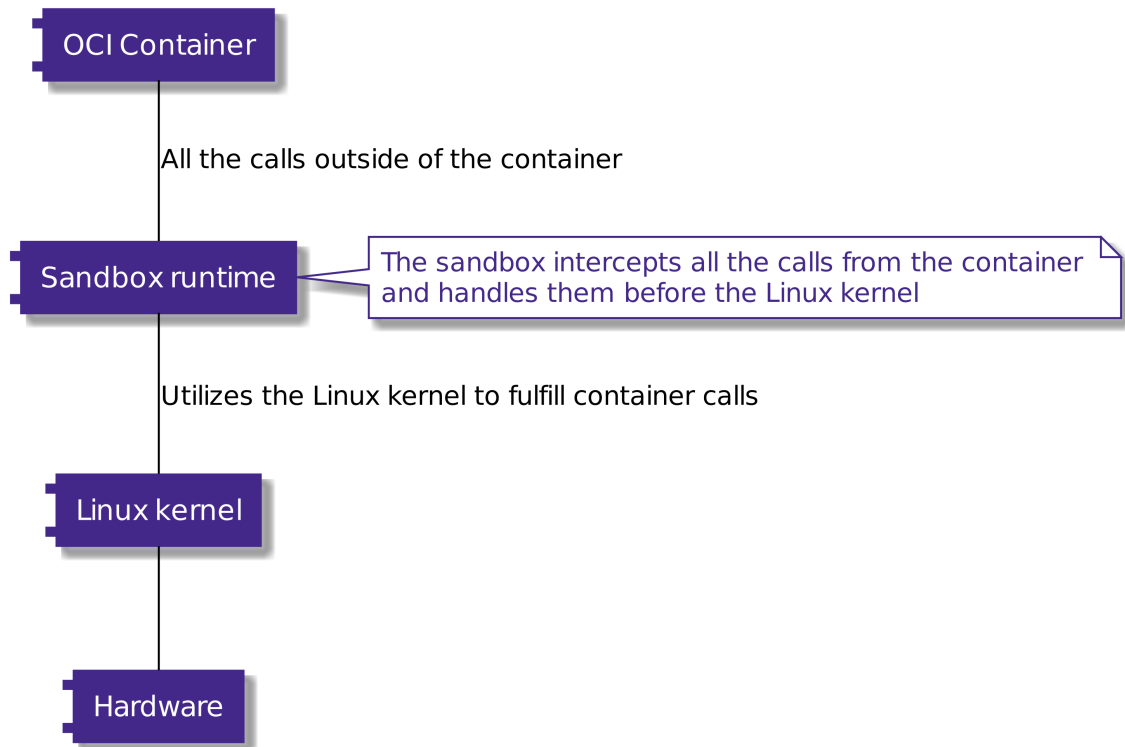


Figure 2.2. How a sandboxed runtime interacts with the Linux kernel.

2.3.5 Virtualized runtime

A virtualized runtime makes a lightweight virtual machine where the container will run. The security is increased by not allowing the process to interact with the kernel in anyway, but instead the container runtime interacts with the hypervisor. The downside of this kind of approach is that hardware support is required for the best performance. [22]

For this work the virtualized runtime in use is Kata containers which supports utilizing either KVM or QEMU as the virtual machine host. KVM (kernel virtual machine) provides a type 1 hypervisor for creating Kata containers. QEMU (a generic and open source machine emulator and virtualizer) on the other hand is a type 2 hypervisor. [21] The QEMU variant is used in testing.

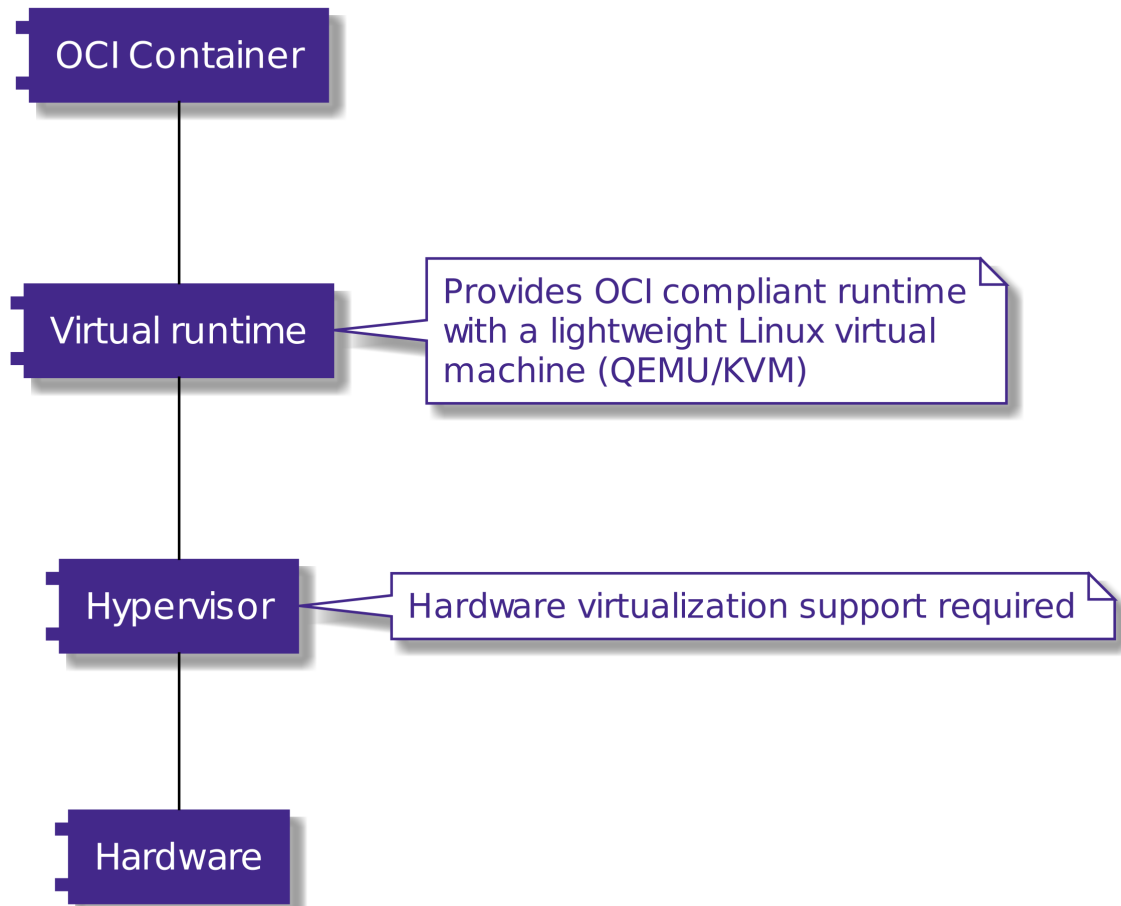


Figure 2.3. How a virtualized runtime interacts with the hardware.

2.4 Related work

Containers are not the first attempt at solving portability issues in embedded devices. Research has been conducted to use other techniques such as micro kernels. As can be seen from the research conducted by researchers in the University of Bucharest. They implemented a new native runtime for IoT devices that was using a VMXL4 microkernel to provide a native runtime environment for IoT devices [30].

Some existing research is available for container runtimes, but it mostly focuses on the cloud use-case of containers. Below a brief explanation of the different runtimes discussed.

runc	Docker developed runtime implementation that is now the reference implementation that follows the OCI container runtime specification
gVisor	"An application kernel, written in Go, that implements a substantial portion of the Linux system call interface. It provides an additional layer of isolation between running applications and the host operating system." Developed by google [58]
runcsc	"gVisor includes an Open Container Initiative (OCI) runtime called runcsc that makes it easy to work with existing container tooling. The runcsc runtime integrates with Docker and Kubernetes, making it simple to run sandboxed containers" [58].
gVisor ptrace	Implements runcsc with system calls
gVisor KVM	Implement runcsc with a kernel virtual machine (KVM). Requires hardware to support it.
Kata container	A container runtime implemented with a hypervisor providing hardware isolated virtual machines for containers.
Nabla containers	"Nabla containers use unikernel techniques to reduce the total number of system calls and thereby reduce the attack surface" [34].

The researchers at Xidian University performed an extensive suite of tests to measure the performance, security and isolation of RunC, Kata Containers and gVisor. gVisor had two variants on the research, gVisor ptrace, which uses ptrace to manage the intercept and redirect the system calls of the container to increase the security it provides. The other gVisor variant gVisor KVM on the other hand uses the Linux kernel virtual machine or KVM for short. What they found when doing this research is that when it came to performance RunC was the best, but it also provided the worst isolation and security. Both gVisor variants were shown to have the best isolation, but they suffered huge performance degradation with system calls, I/O operation and memory. [24] Similar research conducted in the University of Wisconsin found that the increased isolation of gVisor caused huge performance degradation. System calls were 2.2 times slower, memory allocations 2.5 time slower, large downloads 2.8 times slower and file operations 216 times slower. [60] Containers for high performance applications were reasearched by Martin, Kandasamy and Chandrasekaran. They compared RKT, Docker and Linux LXC containers on their research and found out that the RKT offered near native performance on high performance computing applications. [33] Unfortunately the RKT container project has ended [41]. Another comparison between runc and runcsc done in the university of Garching Germany, found that runcsc provided better security but caused per-

formance degradation. On the benchmarks CPU, memory and Disk I/O was measured. [9] Runsc is the gVisor OCI compliant runtime implementation.

Performance evaluated by Marvridis and Karatza with Kata, gVisor, runc, Nabra containers and Firecracker. Found out that out that the additional security of these runtimes did not hinder the CPU and network performance. For random reads / rights there was a small 2.3% performance penalty. Disk performance showed that Kata is significantly less performant than runc. Redis benchmark showed that gVisor and Kata offered 50% less performance when compared to runc. [34]

3. PROBLEM ANALYSIS

A look into what an ideal container should look like. And then outlining the testing procedure and tests that will be used in evaluating how close a container runtime is to the ideal state.

3.1 Challenges and benefits of using containers in embedded devices?

Normally when people talk about containers they expect that they will be running them in the cloud or maybe a development container on their machine. This is because the immediate use case for containers has been the cloud. Cloud will most likely remain the biggest selling use case for containers to come, but why couldn't they be utilized on embedded devices as well. There are of course issues when using containers on an embedded device. But at least some companies have been able to solve most of these issues and are selling products that support containers on an embedded platform. For example Wind River Linux [23] which builds custom embedded Linux images and these images do support container technologies such as Docker and the Docker provided runtime runc.

Not all embedded devices will benefit from having an operating system on board. Smaller devices where the battery is expected to last for years, or really simple devices that only measure the temperature have no need for a full-blown operating system. For these device coding them with just an event loop will remain desirable. But as we have seen with history the performance of computers will keep increasing and there will come a point when performance is not the issue but development time. At this point it might be viable to add an operating system to these smaller embedded devices. On some devices this might already be the case. This is why projects like Yocto project are becoming more interesting. The Yocto projects provides tools and a space to create custom Linux based systems no matter the underlying hardware or architecture [59]. If we were to add portable containers in to this mix we could have a software product that can be deployed in any device that has a suitable Linux image running.

Even with software we cannot get rid of the hardware limitations that most embedded devices face. The amount of CPU, RAM and memory in an embedded product will be

limited. A suitable container runtime for embedded devices should use as little as possible of these precious resources. Most of the resource usage should come from either the operating system or the application. The container runtime should use as little resources as possible.

When a new hardware release for the embedded devices becomes available parts of the software will most likely require to be heavily modified or re-written. But if the main application would be coded as a easily transferable container, this would not be such a huge issue anymore. Only the operating system that provides the container runtime would need to be ported to the new hardware. This is also the approach that Wind River Linux is taking [23]. For minor hardware changes, where a smaller part of the device changes a hardware driver would only require to be made, which in most cases could be provided by the manufacture of the said hardware module.

Another huge issue is flash memories of embedded device that will wear out over time. To avoid wearing out the flash of an embedded device a file system without any writing is beneficial, which can be done with containers. Because a container is only a collection of file system changes and the required application, and making it read only is already a standard feature of the file system.

3.2 Ideal state

When it comes to portability an ideal container would be usable in any type of device, no matter the underlying hardware architecture. None of the target platform specific parameters would matter for the container. This would mean that no matter the operating system, or the underlying processor architecture the container runtime would make any container work. To currently achieve this kind of portability the container runtime would have to be able to emulate other processor architectures so that binaries coded for x86, ARM or RISC-V would work. An alternative to this would be to use a high level programming language that would allow the code to be portable, but the execution environment would be architecture specific.

From the perspective of CPU performance, a containerized application should offer the same performance as a native implementation. However this is unlikely to be possible in practise due to the added overhead of containers. A more appropriate goal for container runtimes is to make the overhead to be so insignificant that it will not matter in the real world. If we were to give a numeric value the container runtimes should offer 95 - 99 % of the performance when compared to native application. According to research conducted by IBM comparing virtual machines and containers to native the CPU performance hit can be around 2% making this goal plausible . CPU performance hit of around 2% was reported by a 2015 study conducted by IBM comparing virtual machines and containers to native implementations [11]. Making this ideal goal even more of a reality.

When it comes to storage usage, an ideal container runtime should be as small as possible, because embedded devices have a small flash memory where both the operating system and the application need to fit. For example a decent consumer router offers around 32 - 64 MB of storage [45]. For a container environment to make sense in this kind of constrained environment the container runtime cannot take more than few percents of the space available. The amount of space a container runtime takes is not the only issue. The actual application container will also need to fit into the device. Since containers are independent of each other all of the dependencies of the application are included in the container. This might cause the container to require more space than the actual runtime implementation. In this kind of situation the container should be made as small as possible, which is left up to the developers of said application.

Similar to disk space usage the amount of RAM is also limited. When a container application is running the overhead of said runtime should not increase the amount of memory used. Achieving zero overhead is not possible. What can be achieved though is making sure that the runtime takes at most 1 - 5% more memory than the native version. For tiny applications this does not make sense. A container at minimum requires a certain amount of memory which is not affected by the size of the container. For tiny containers there needs to be a different ideal state for memory usage. For these kind of situations the runtime memory requirement should not be more than 1% of the entire memory pool. Of course this becomes harder and harder to achieve the less overall memory is available and 1% overhead would mean at most 100 containers on a device. For embedded devices we can assume that they will not require more than 10 containerized applications. This is because they are specialized devices designed to do few things only. So additional memory usage of 1-5% is acceptable. The findings done in previous research at the university of Ho Chi Minh indicate that this might be impossible [4]. The observed overhead was 20 % on high performance applications on a Docker container when compared to native.

One thing that can be widely ignored in cloud context is the amount of dependencies for a container runtime. On an embedded device some of the dependencies might be missing or take too much space to make sense. An ideal container would only depend on the Linux kernel. Or if the container runtime does depend on something the dependencies of that container runtime should be included in most embedded Linux images or build processes. If the dependencies are not included they should require as little space as possible to install them. When it comes to security an ideal container would be completely isolate from the other containers. The OCI specification for container runtimes does not specify a minimum level of isolation it is all up to the implementation of runtimes. So whether any of the current runtime implementations offer this complete isolation is highly unlikely.

3.3 Testing methodology

Testing was conducted on two hardware platforms x86 and ARM. The x86 hardware platform served as a baseline to validate that the tests produced reasonable results by comparing the results to results achieved by others. The x86 platform was also required because gVisor did not have support for ARM processor architecture. The ARM hardware platform was used as a representative of an embedded device. Another interesting hardware platform from the perspective of embedded development, would have been RISC-V. Unfortunately the author does not have access to such hardware. Also the support for RISC-V based Linux systems is not yet as mature as x86 and ARM. The use of two hardware platforms was also a requirement to test the portability of containers across architectures and to compare architectural performance characteristics between the two.

For testing on the x86 hardware a laptop was used that will represent an x86 based device. The laptop offered an Intel core 8th generation i5-8250U processor. Since the laptop had its own battery and power management features, the battery was disabled during testing. This decision was made so that any kind of battery management or power saving features of the laptop will not effect the testing results. As for the ARM side of things a Raspberry Pi 4 model B was used. Raspberry Pi four is a small credit card sized embedded computer that is really popular among hobbyist and commonly used for prototyping embedded products. The pi has multiple models available, but the most common one is one of the model variants B or B+. There are also lower end models available, but for comparing performance between the native implementation of a test to a containerized one, the underlying hardware does not matter.

As for the host operating system Ubuntu 20.04 64 bit server was installed on both systems. There is a Ubuntu 32 bit version available for the Raspberry Pi, but 64 bit was chosen so that the test environments are as close to each other as possible. Before testing all the latest updates were installed on both of the systems and no further updates applied once the testing had started.

Four different container runtimes were used for testing. Runc and crun representing native container runtimes, gVisor representing a Sandboxed runtime and Kata containers representing a virtual runtime. Runc is the reference implementation for OCI runtime specification, and management by the same entity. Crun on the other hand is a C implementation which boasts to provide better performance than runc due to the fact that it is using C instead of GO as the programming language [16]. GVisor on the other hand is a runtime implementation which creates an application kernel between the container and the Linux kernel. The application kernel is said to increase the security of the container [18]. The other non native runtime Kata containers is a runtime implementation which provides the runtime interface through virtual machines [22]. Unfortunately though gVisor does not support the ARM architecture so it can only be tested with x86 platform. The

other runtimes though are usable and the newest possible versions were used on the target device, to get the latest features and performance characteristics. As a reminder see table 3.1 with all the numerical software and hardware details.

Table 3.1. Hardware and software details for the used testing platforms.

	Lenovo thinkpad e480	Raspberry Pi 4 model B
processor	Intel Core i5-8250U 8th generation, 1.6 GHz base, boost of 3.4 GHz	Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
RAM	8 GB DDR4, 2400 MT/s	2GB RAM
Storage	512 GB pci-e m.2 ssd	16 GB Micro SD card
Host os	Ubuntu server 64-bit 20.04 LTS with the latest updates as of 23.04.2022	
Kernel	5.4.0-107-generic	5.4.0-1058-raspi
Docker	20.10.14	20.10.13
gVisor	1.0.2-dev	Not available
Kata-containers (QEMU)	1.13.0-alpha0	1.11.2
crun		1.4.3
runc		1.0.3

3.3.1 Portability

The portability test was using a portable high level programming language to achieve portability between ARM and x86. This was tested by adding a complex Python application with significant amount of dependencies to a container and trying to run the same container image on both systems. The complex Python application used in this test ended up being home-assistant[52], which is coded entirely in Python according to github metrics. Home assistant is a home automation software suite which makes it possible to control smart devices such as light bulbs, and switches. First home assistant was installed on both platforms and verified that it was working correctly. Then the container was moved to the other platform. On the other platform the container image was loaded and similar verification conducted to make sure everything worked correctly. Whether the container is portable or not, it is interesting to see what kind of layers the container has. The layers should tell us why it was not portable or why it ended up being portable. Because of these reason the container layers were looked at with Dive [8] after the portability test.

3.3.2 Storage space and dependencies

To make sure that the container runtimes are suitable to be used in embedded devices the dependencies and storage space requirements need to be analyzed. The dependencies were extracted with either debian package manager or by directly looking at the runtime symbolic links. Most of the software was installed with apt in which case the software package included the dependency information. Some of the runtimes were installed more manually and not packaged properly so the dependencies were evaluated manually, by looking at the runtime symbolic links.

3.3.3 RAM usage and performance

During the execution of all the tests memory usage of the system was monitored and logged with free utility [13]. This ram monitoring was included in all of the performance tests. The free utility provided 5 metrics for memory usage. The used memory, free memory (unused memory), shared memory (the amount of memory shared between processes), buffers / cache (kernel buffers, memory used by page cache and slabs) and finally available (memory available to be used by programs). During 7z and Redis benchmark the free was run with a bash script every 0.1 seconds to get adequate amount of data for comparison.

In addition to measuring RAM usage the performance can be evaluated to some extent by the 7z benchmark where especially the compression is dependant on RAM latency. 7z was also used in measuring the CPU benchmark as detailed in 3.3.4.

3.3.4 CPU performance

To measure the CPU performance 7z benchmark was used. 7z benchmark is a benchmark provided with the 7z compression and decompression utility. This benchmark was chosen due to its easy installation and the fact that compression depends on RAM latency and decompression is depended on the CPU. The benchmark provided a method to run the benchmark multiple times, and it was decided to run the benchmark 20 times per runtime to get a good average of performance over multiple runs.

3.3.5 Database performance

Database performance was evaluated using the Redis-benchmark [27]. "Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker" [26]. This test was run by making a container that hosts the actual database and a second container that launched the benchmarking tool. The Redis benchmark had a few available parameters to choose from that changed for example how

many clients or the size of requests. During testing the default parameters were used.

4. TESTING RESULTS

In this chapter all the test results are written down and figures and tables shown for every test run. Firstly the portability is evaluated in section 4.1, then database performance is evaluated in section 4.2, followed by CPU performance with 7z benchmark in section 4.3 and finally memory usage is evaluated during both Redis and 7z-benchmark in sections 4.4 and 4.5. Both of the chosen hardware platforms are present in every test case and all runtimes are used except for gVisor in ARM platform tests. GVisor does not currently support ARM in anyway.

4.1 Portability

The test was to create the container on one host and then move it to another. The container image that is used for this test is home-assistant [52]. First the image was pulled directly from Docker hub to the device and run with different types of container runtimes. For verifying that everything was working an account was created that is required when running the home-assistant Docker image for the first time. Below are the commands used when unloading the Docker image and loading it on the other end. And finally the command that were used when running the home-assistant container.

```
# Save the Docker image to disk
docker save -o ha.tar.gz \
  homeassistant/home-assistant

# On the host PC load the Docker image
docker load -i ~/ha.tar.gz

# Run the Docker image on the other machine
docker run \
  --name <some-name> \
  --privileged \
  --restart=unless-stopped \
  -e TZ=MY_TIME_ZONE \
  -v ~/.config/<runtime-name>:/config \
```

```
—network=host \
ghcr.io/home-assistant/home-assistant:stable
```

The only anomaly that was encountered during testing was that for some reason the ARM version of Kata runtime did not work with the `'--privileged'` flag. As mentioned before the gVisor was not included in this test for ARM since there is no support for it. During the loading Docker gave out a warning that the hardware platform differs from the platform indicated by the Docker image, but loaded the image nonetheless see table 4.1 for detailed warnings.

Table 4.1. Docker warning messages when loading home-assistant Docker image in ARM from x86 and vice versa.

```
WARNING: The requested image's platform (linux/amd64) does not
match the detected host platform (linux/arm64/v8) and no specific plat-
form was requested
```

```
WARNING: The requested image's platform (linux/arm64) does not
match the detected host platform (linux/amd64) and no specific plat-
form was requested
```

When trying to run the image in any of the runtimes all of them gave some form of exec format error. The error messages were slightly differently formatted, but contained the same error. Below a table for all the runtimes 4.2.

Table 4.2. Exec format errors when trying to run the home-assistant Docker image on ARM from x86 and vice versa.

runtime	x86 to ARM	ARM to x86
runc	standard_init_linux.go:228: exec user process caused: exec format error	standard_init_linux.go:228: exec user process caused: exec format error
crun	{"msg":"exec container process '/init': Exec format error","level":"error","time":"2022-04-23T10:00:21.000342507Z"}	{"msg":"exec container process '/init': Exec format error","level":"error","time":"2022-04-23T09:49:58.000693287Z"}
Kata-containers	error="standard_init_linux.go:211: exec user process caused exec format error" name=Kata-agent pid=1 source=agent	standard_init_linux.go:211: exec user process caused "exec format error"
gVisor	Not supported	starting root container: starting sandbox: creating process: failed to load /init: exec format error: unknown.

A further analysis on the container layers was conducted in order to dive deeper into why the containers are not portable. At a glance when checking the container images with

Docker inspect all the hash values were different indicating that all of the layers differed from one another. But a further inspection with Dive [8] revealed that most of the layers are identical, except binary layers where something is installed or build from source. See table 4.3 for details.

Table 4.3. Home assistant layers

Layer	What it is	Details	Difference	ARM size	x86 size
1	Setup working dir	-	None	0 B	0 B
2	s6 overlay	"s6 is a collection of utilities revolving around process supervision and management, logging, and system initialization" [1].	Different architecture	63 MB	76 MB
3	Copying some files	Add patches	None	3.6 kB	3.6 kB
4	Installing Python and its dependencies		Different architecture	124 MB	114 MB
5	Making links to Python		None	0 B	0 B
6	Installing pip		None	17 MB	17 MB
7	Installing libs and setting locale.		None	16 MB	17 MB
8	Installing requirements		Different architecture	87 MB	100 MB
9	Copying a requirements.txt for Python requirements		None	65 B	65 B
10	Installing requirements with pip		Different architecture	1.9 MB	2.0 MB

11	Installing ssocr	"Seven Segment Optical Character Recognition or ssocr for short is a program to recognize digits of a seven segment display" [44].	Different architecture	1.5 MB	1.6 MB
12	Installing arp-scan	The ARP scanner	Different architecture	1.5 MB	1.5 MB
13	Installing libcec		Different architecture	2.5 MB	2.5 MB
14	Installing Pico TTS	"Text to speech voice synthesizer from SVox, included in Android AOSP" [38]	Different architecture	8.6 MB	8.6 MB
15	Installing tolduss	Smart lite controller	Different architecture	1.6 MB	1.5 MB
16	Installing iperf	TCP, UDP and SCTP network bandwidth measurement tool	Different architecture	2.1 MB	2.1 MB
17	Copy mime types		None	44 kB	44 kB
18	Copy home-assistant requirements		None	453 b	453 b
19	Copy home-assistant package constraints		None	3.1 kB	3.1 kB
20	Install home assistant requirements		None	18 MB	19 MB
21	Copy all requirements		None	48 kB	48 kB
22	Install home-assistant requirements		None	818 MB	824 MB
23	Copy home-assistant files		None	42 MB	42 MB
24	Compile home-assistant Python		None	23 MB	23 MB

25	Install sql-lite		Different architecture	1.3 MB	1.2 MB
26	Copy service files		Same	2.2 kB	2.2 kB
27	Set up work dir		Same	0 B	0 B

4.2 Database performance (Redis)

Redis is an in memory benchmark and according to the documentation these factors can affect the performance of Redis; network bandwidth and latency, CPU, Speed of RAM. In addition to these easy to explain things other factors also effect the results. Such as whether TCP/IP loop back is used or unix domain sockets are used. Unix domain sockets are reported to provide 50% more performance compared to TCP/IP. Whether the Redis instance is running on a VM or native will also have impact on the results. For the testing though all of the hardware details will remain the same since comparison happens between container runtimes. Most of the overhead in our case is the added overhead of the current container runtime.

Table 4.4. *Commands tested by Redis benchmark. [28]*

Command	Explanation
PING_INLINE	Older format of pinging when using telnet or similar old tool.
PING_MBULK	Newer ping that is sending binary.
SET	Set a key value in the Redis database.
GET	Return a value behind they key from Redis.
INCR	Increment a integer value by one.
LPUSH	Insert values to the beginning of the list.
RPUSH	Insert values to the end of the list.
LPOP	Remove and get the first element.
RPOP	Remove and get the last element.
SADD	Add value to the set specified by key.
HSET	Set the string value of a hash field.
SPOP	Remove and return one to many from a set.
ZADD	Add one to many members to a sorted set.
ZPOPMIN	Remove and return members with the lowest score from a sorted list.
LRANGE_100	Get 100 elements from a list.
LRANGE_300	Get 300 elements from a list.
LRANGE_500	Get 500 elements from a list.
LRANGE_600	Get 600 elements from a list.

4.2.1 ARM results

When looking at the request per second results Kata shows the worst performance in every category (9.2% - 31.4% worse than native) except RPOP where crun is the worst (marked in red). Only GET, LPUSH and RPOP and LRANGE_100 are close to runc and crun (marked in yellow). Runc and crun on the other hand trade blows quite evenly. Runc is slightly better at PING_INLINE, SET, RPOP, RPOP, HSET, LRANGE_300 and LRANGE_600. And crun is slightly better at the remaining metrics (marked in green). Funnily enough runc and crun seem to provide better performance than native in LRANGE_500 and LRANGE_600, but other than those few anomalies they are 10 - 15% worse than native (marked in purple). For more detailed results check A.1.

Table 4.5. ARM results on Redis benchmark RPS (request per second)

test	Native (rps)	runc (rps)	kata (rps)	crun (rps)
PING_INLINE	15951.51	14810.43	12072.92	14727.54
PING_MBULK	5775.36	14553.92	13104.44	14583.64
SET	16113.44	14695.08	12615.11	14367.82
GET	16007.68	14577.26	14035.09	14817.01
INCR	15893.2	14376.08	12884.94	14764.51
LPUSH	15898.25	14440.43	13768.42	14444.61
RPOP	15862.94	14478.07	14403	14376.08
LPOP	15883.1	14771.05	12465.72	14817.01
RPOP	15933.72	14903.13	13422.82	14677.82
SADD	16189.09	14688.6	13943.11	14471.78
HSET	15810.28	14330.75	13294.34	14194.46
SPOP	15976.99	14386.42	12948.34	14446.69
ZADD	15880.58	14594.28	13603.59	14639.15
ZPOPMIN	16168.15	14547.57	13798.81	14677.82
LPUSH (needed to benchmark LRANGE)	16095.28	14537	13664.94	14652.01
LRANGE_100 (first 100 elements)	11234.69	10564.12	10130.69	10674.64
LRANGE_300 (first 300 elements)	6191.95	5226.03	4858.61	5195.61
LRANGE_500 (first 500 elements)	4560.38	4576.66	3676.88	4590.74
LRANGE_600 (first 600 elements)	3989.63	4125.07	3417.87	4021.39
MSET (10 keys)	16194.33	14499.06	11117.29	14598.54

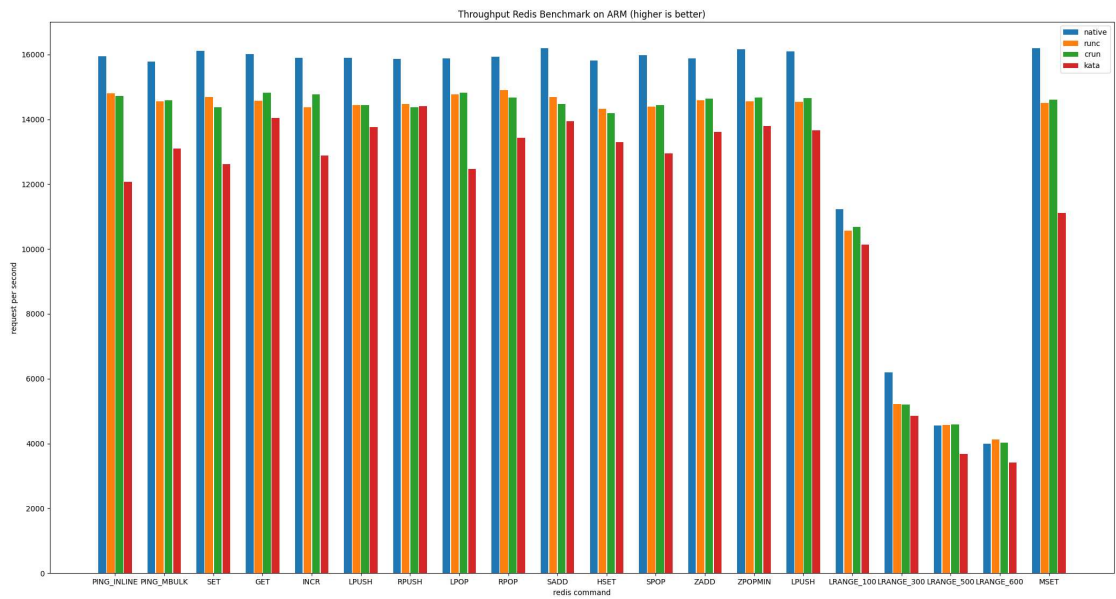


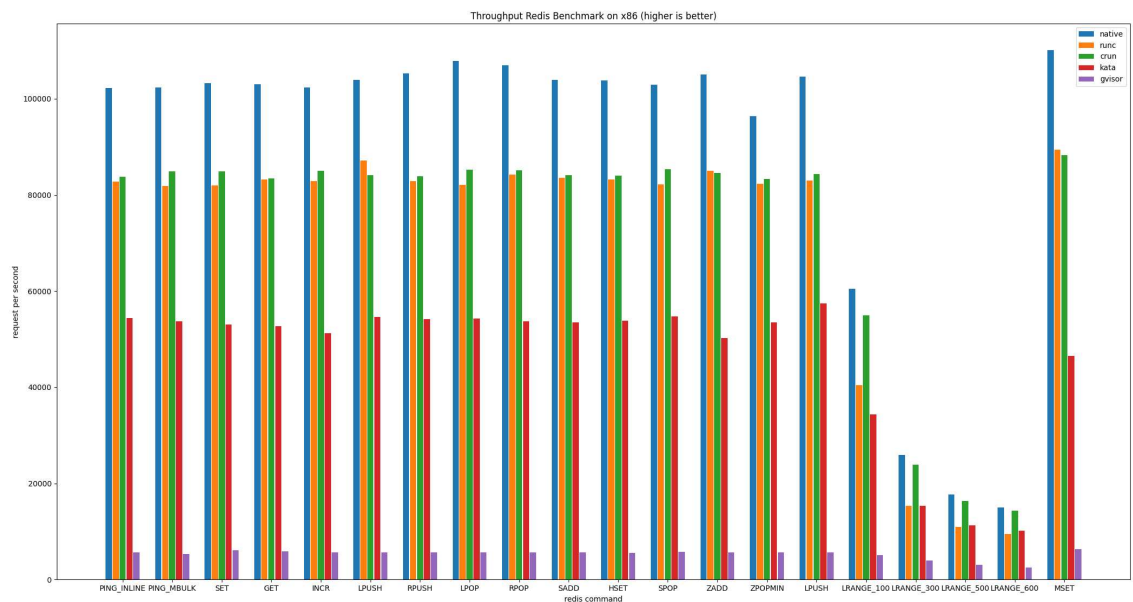
Figure 4.1. Request per second Barchart on ARM.

4.2.2 x86 results

For x86 the results for request per second show native being clearly the faster than runc, crun, gVisor or Kata (marked in green). Kata is clearly the worst performing offering at best 17.5% (marked in red) of what the native can offer followed by gVisor with 67.6% (marked in purple), crun with 85.3% and runc with 95.7%. Crun does manage to outperform runc in LPUSH and ZADD and MSET (10 keys). Green indicating the best performance excluding native and yellow the second best then followed by purple and finally red indicating the worst performer. For more detailed results check A.2

Table 4.6. x86 results on Redis benchmark RPS (request per second)

test	Native	runc (rps)	gVisor (rps)	Kata (rps)	crun (rps)
PING_INLINE	102249.49	83752.09	5662.83	54406.96	82781.46
PING_MBULK	102354.15	84889.65	5288.49	53705.69	81833.06
SET	103305.79	84961.77	6060.97	53022.27	82034.45
GET	102986.61	83472.46	5890.67	52742.62	83263.95
INCR	102354.15	85034.02	5641.11	51203.28	82850.04
LPUSH	103950.1	84175.09	5658.03	54644.81	87183.96
RPUSH	105263.16	83892.62	5644.62	54141.85	82850.04
LPOP	107874.87	85251.49	5643.02	54259.36	82101.8
RPOP	106951.88	85106.38	5651.63	53763.44	84245.99
SADD	103950.1	84104.29	5650.36	53475.94	83612.04
HSET	103842.16	84033.61	5492.69	53821.31	83194.67
SPOP	102880.66	85324.23	5753.08	54764.51	82169.27
ZADD	105042.02	84530.86	5639.84	50276.52	85034.02
ZPOPMIN	96432.02	83333.33	5659.63	53475.94	82304.52
LPUSH (needed to benchmark LRANGE)	104602.52	84388.19	5693.46	57471.27	82987.55
LRANGE_100 (first 100 elements)	60459.49	54945.05	5039.81	34352.46	40453.08
LRANGE_300 (first 300 elements)	25893.32	23877.74	3923.88	15372.79	15384.62
LRANGE_500 (first 500 elements)	17658.48	16380.02	3088.61	11309.66	10979.36
LRANGE_600 (first 600 elements)	14981.27	14341.03	2492.21	10130.69	9452.69
MSET (10 keys)	110132.16	88339.23	6297.63	46533.27	89445.44

**Figure 4.2.** Request per second Barchart on x86.

4.3 7z benchmark results

The 7z benchmark measures the CPU performance achieved by the container runtime. Compression depends on RAM latency, Data Cache size/speed and Translation lookaside buffer. Decompression on the other hand is dependent on the CPU integer operations. 7z benchmark provides a good view on how the added overhead of containerization effects CPU performance. The rating is show in MIPS (million instructions per second). The R/U column shows the results normalized for 100% CPU usage. Which is calculated by dividing the rating with the usage. [29]

4.3.1 x86 results

When looking at the ratings for different runtimes runc and crun are nearly the same as the native test, less than a percent performance difference. Kata on the other hand provides the worst performance only offering at best 23% of the performance that is available natively in decompression. GVisor on the other hand offers 88.8 - 97.9% of native performance.

Table 4.7. x86 results for 7z benchmark.

Compression average			
runtime	Usage	R/U	Rating (MIPS)
native	651	2491	16205
runc	658	2445	16100
crun	654	2479	16218
Kata	100	3324	3323
gVisor	755	1905	14394

Compression total			
runtime	Usage	R/U	Rating (MIPS)
native	715	2200	15550
runc	722	2171	15495
crun	720	2185	15535
Kata	100	3423	3422
gVisor	773	1874	14490

Decompression average			
runtime	Usage	R/U	Rating (MIPS)
native	780	1910	14896
runc	785	1896	14891
crun	785	1896	14891
Kata	100	3522	3521
gVisor	791	1843	14585

4.3.2 ARM results

Similar story to x86 Kata is the worst performing with respect to Rating, only offering at best 40.6% of the native performance. Runc with on ARM seems to offer better performance than native 2.4 - 5.5% to be exact. Crun on the other hand is at worst 2.6% slower in decompression than native and at best 0.7% better at compression than native.

Table 4.8. ARM results for 7z benchmark

Compression average			
runtime	Usage	R/U	Rating (MIPS)
native	328	1039	3405
runc	332	1051	3487
crun	331	1037	3430
Kata	100	1382	1382

Compression total			
runtime	Usage	R/U	Rating (MIPS)
native	339	1271	4342
runc	345	1302	4527
crun	344	1238	4286
Kata	100	1631	1630

Decompression average			
runtime	Usage	R/U	Rating (MIPS)
native	351	1504	5279
runc	358	1553	5567
crun	357	1439	5141
Kata	100	1880	1879

4.4 Memory usage on Redis benchmark

As mentioned before the memory usage was measured during the some of the benchmarks. This section provides the results when the Redis benchmark was running. The figures show all the different memory types that the free utility provides. Used indicating how much memory is in use, free how much is available, shared is the amount of shared memory, buff / cache is the amount of memory used by kernel buffers page caches and slabs, and available is an estimation of how much memory is available for new programs [13]. The memory was measured every 0.1 seconds with the help of sleep. All the graphs show how many measurements were taken so for example 200 samples is roughly equal to 20 seconds.

4.4.1 ARM result

Below are the results for the memory usage during Redis benchmark on ARM on different runtimes. From all the results we can see some fluctuation in the beginning when the

containers are created which is to be expected. The native Redis benchmark has no overhead, runc uses 101 MB of memory when starting the container, crun uses 82 MB and Kata 405 MB. After the container creation we see saw-blade pattern for the memory usage. On native the used memory fluctuates between 220 - 249 MB, runc 263 - 287 MB, crun 265 - 288 MB and Kata 602 - 635 MB. The most memory hungry in this case is the Kata runtime which shows biggest swings in memory usage. See figures 4.3 for native, 4.4 for runc, 4.5 for crun and 4.6 for Kata.

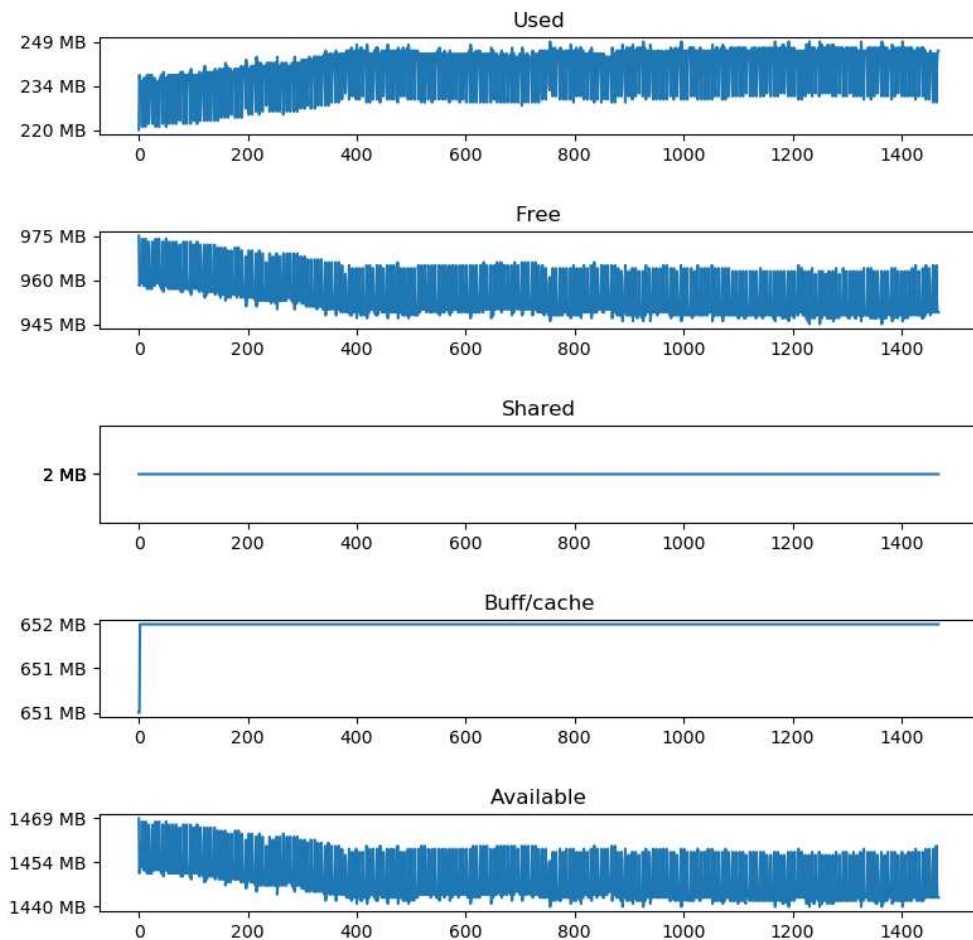


Figure 4.3. Memory usage on ARM native when running the Redis benchmark.

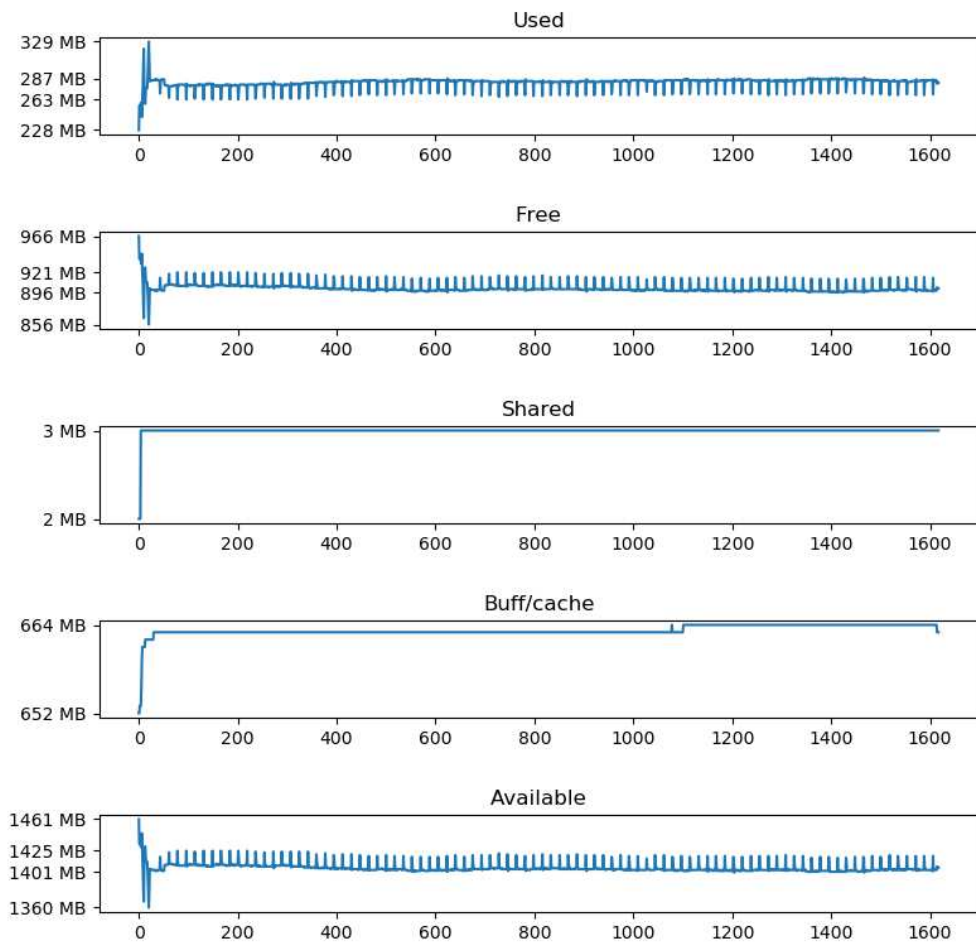


Figure 4.4. Memory usage on ARM runc when running the Redis benchmark.

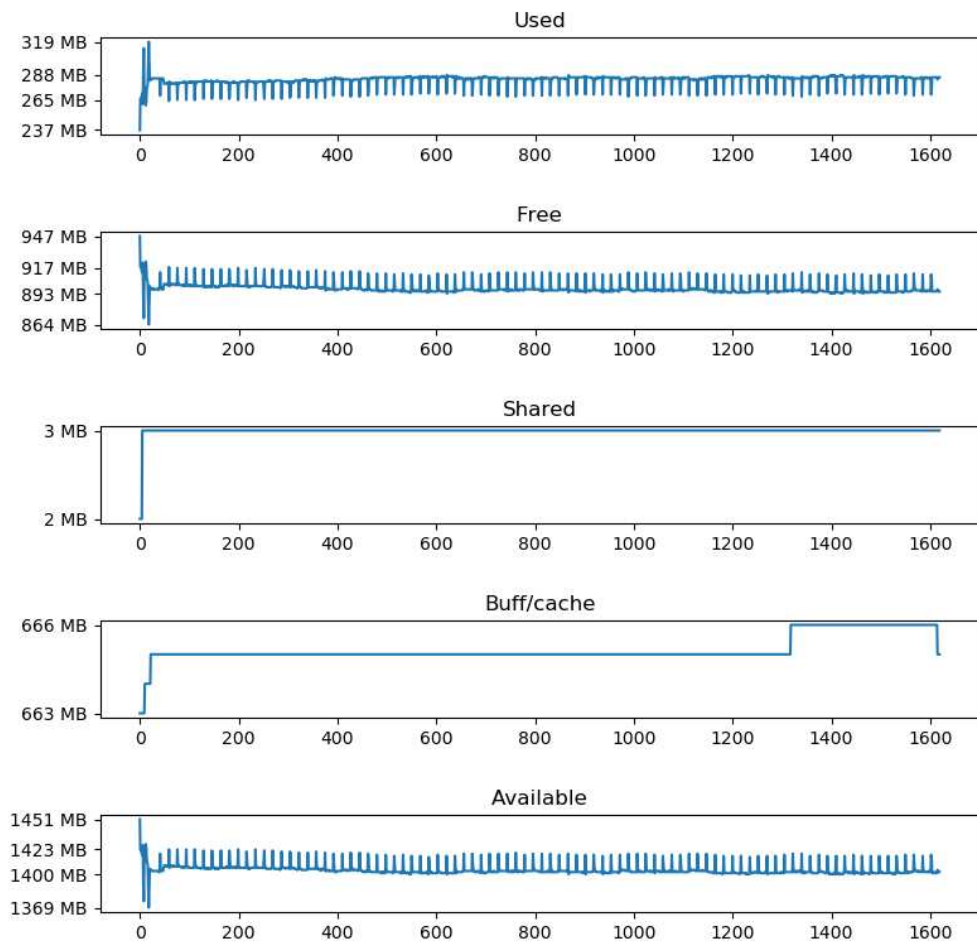


Figure 4.5. Memory usage on ARM crun when running the Redis benchmark.

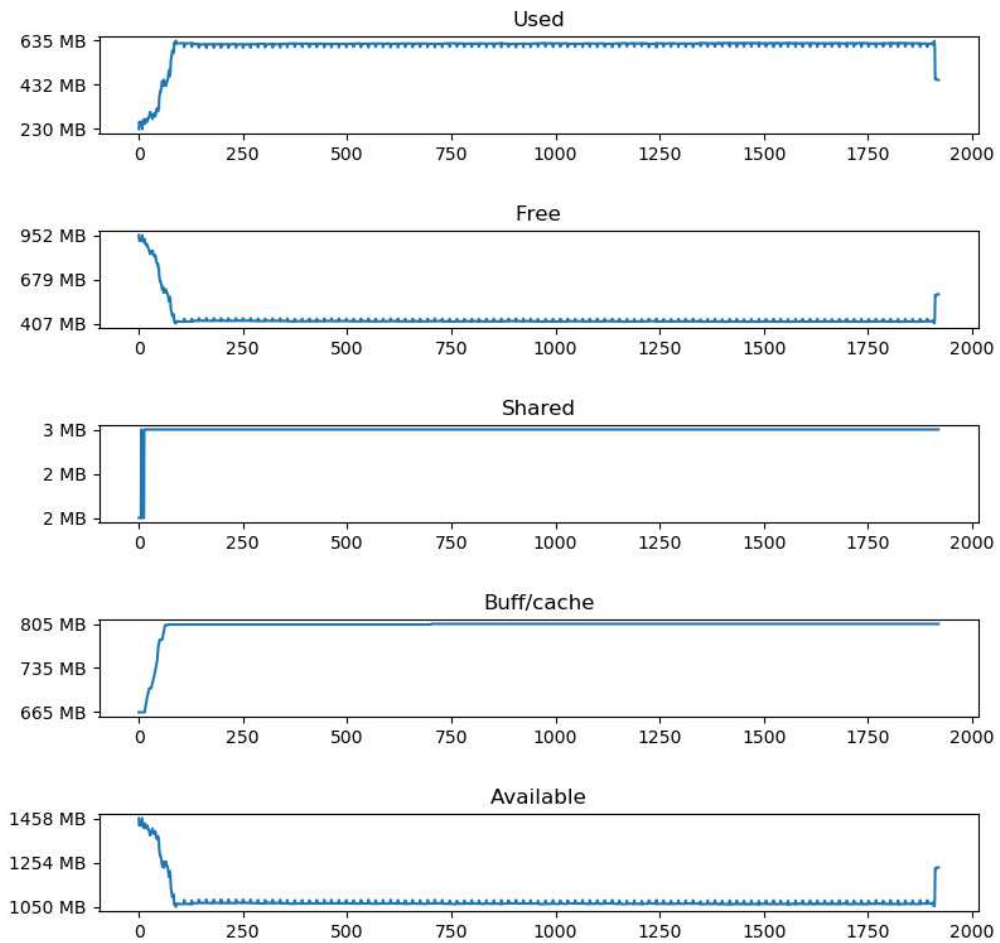


Figure 4.6. Memory usage on ARM Kata when running the Redis benchmark.

4.4.2 x86 results

The same memory usage measurements were also conducted on the x86 platform. On the native there is no added overhead and instead we directly move to the saw-blade pattern. Runc required 88 MB memory when creating the container, crun 77 MB, gVisor 97 MB and Kata 499 MB. The used memory on native fluctuated between 391 - 420, runc 390 - 478 MB, crun 431 - 477 MB, gVisor 395 - 492 MB and finally Kata 852 - 882 MB. See more details on figures 4.7 for native, 4.8 for runc, 4.9 for crun, 4.10 for gVisor, 4.11 for Kata.

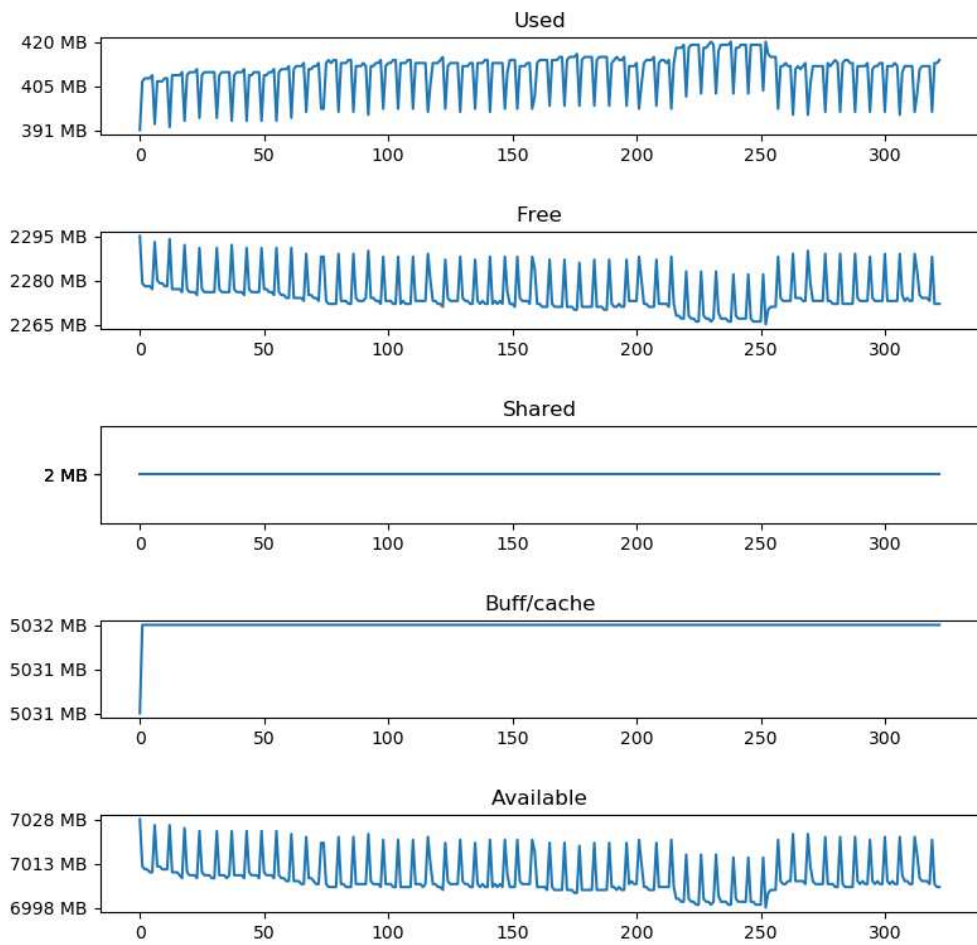


Figure 4.7. Native x86 memory usage when running the Redis benchmark.

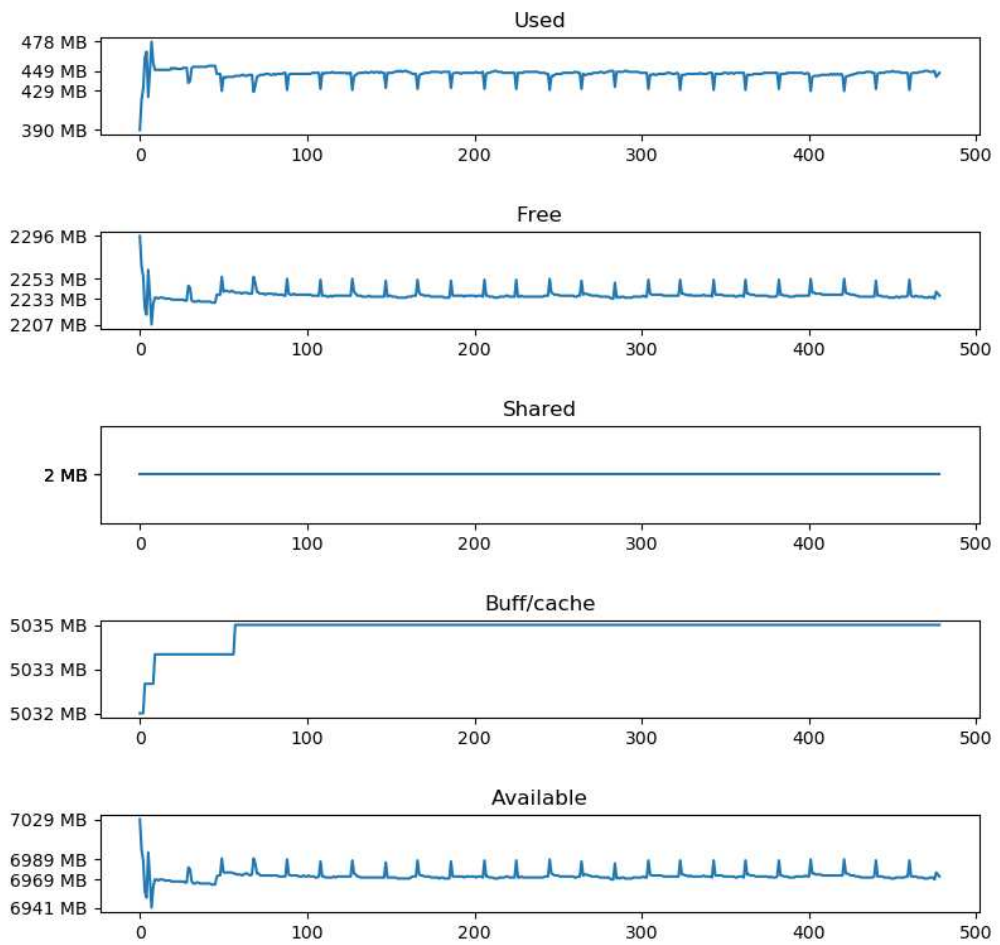


Figure 4.8. Runc runtime memory usage on x86 when running the Redis benchmark.

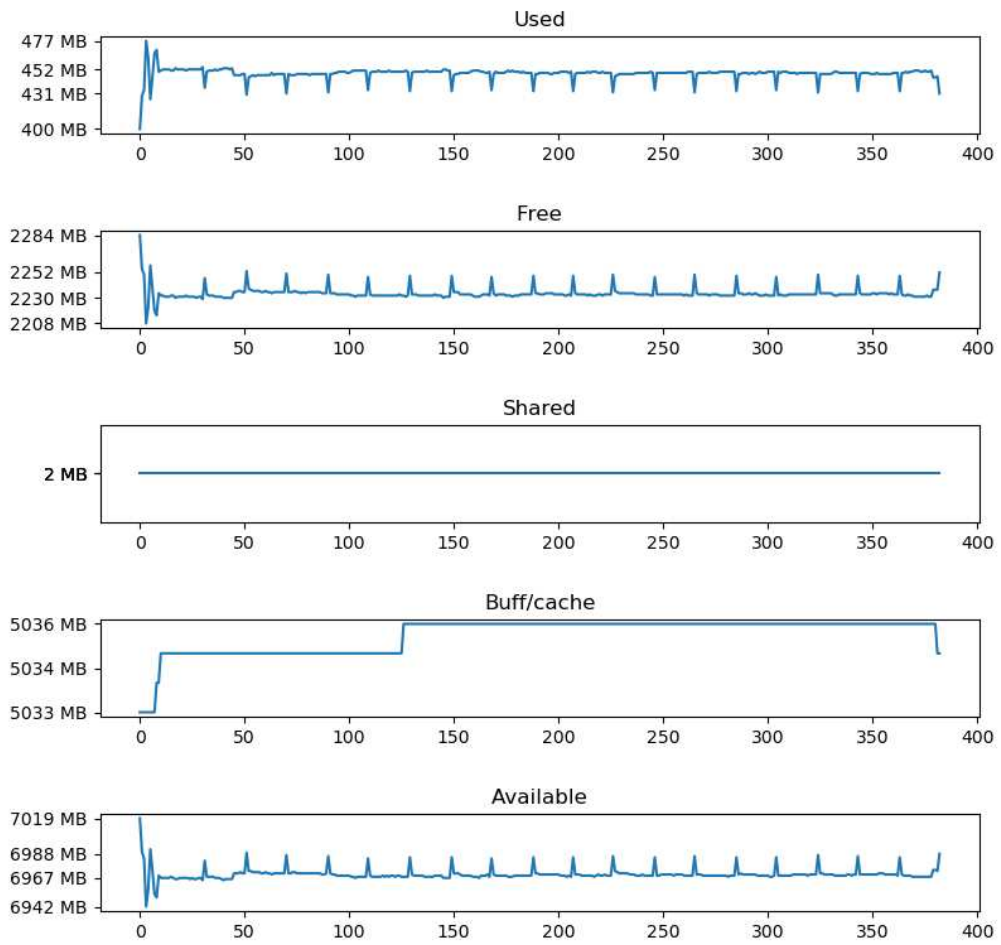


Figure 4.9. Crun runtime memory usage on x86 when running the Redis benchmark.

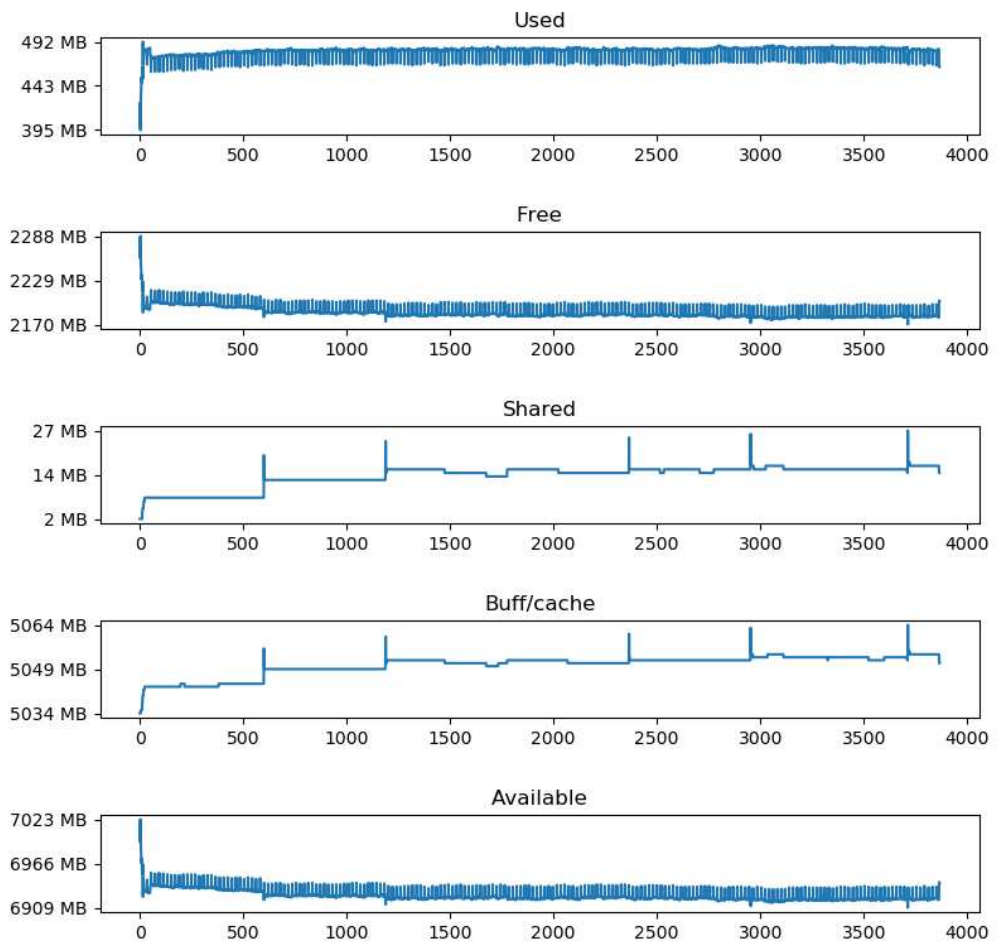


Figure 4.10. GVisor runtime memory usage on x86 when running the Redis benchmark.

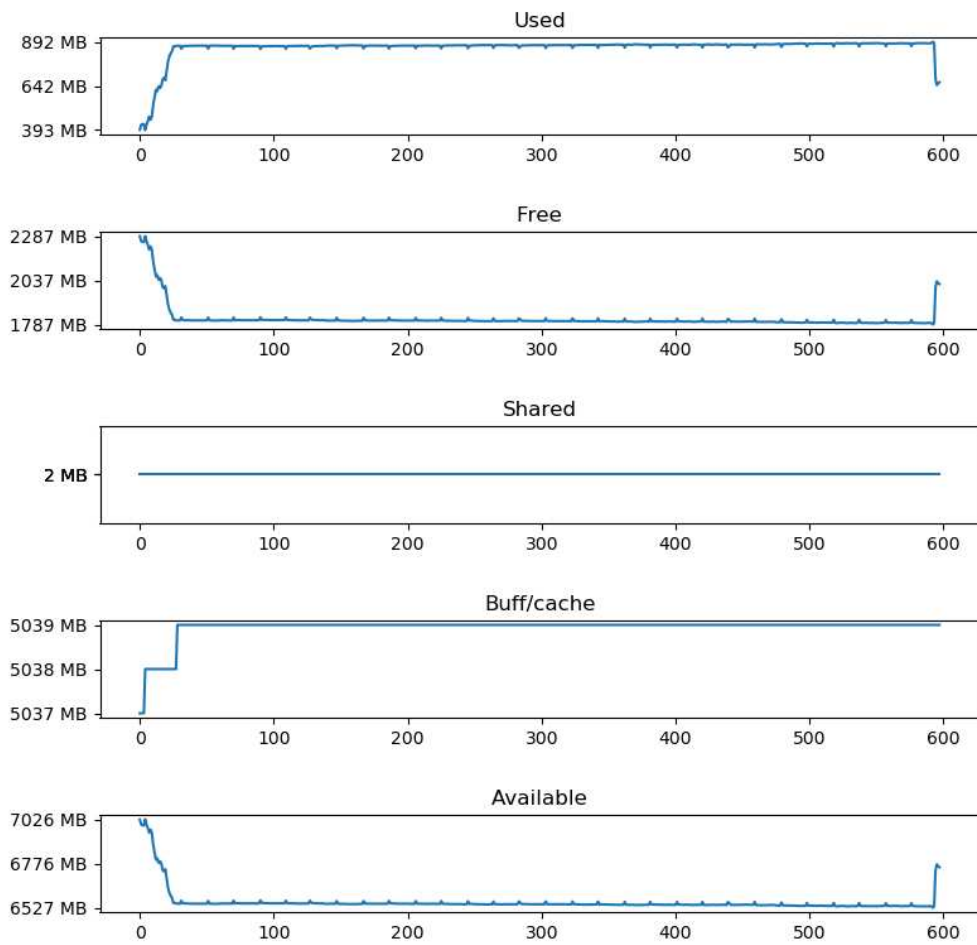


Figure 4.11. Memory usage on x86 Kata when running the Redis benchmark.

4.5 Memory usage on the 7z benchmark

As with the Redis benchmark the memory usage was also measured during the 7z benchmarks. Again the same 0.1 sleep timing was used when measuring the memory usage, meaning 200 on the x-axis translates to roughly 20 seconds. The figures show all the different memory types that the free utility provides. Used indicating how much memory is in use, free how much is available, shared is the amount of shared memory, buff / cache is the amount of memory used by kernel buffers page caches and slabs, and available is an estimation of how much memory is available for new programs [13]. The 7z benchmark uses more memory and is a bigger workload so any memory overhead created by runtimes is less visible on the figures.

4.5.1 ARM results

Most of the figures look pretty similar to one another again a small fluctuation in the beginning when the container is created and then a saw-blade pattern. The exception here is the Kata runtime which seems to reserve all the memory it needs in the beginning and does not release until the container is shutdown. Overall the entire workload takes 984 MB of memory at maximum when running natively, 988 MB with runc, 1000MB with crun, 611 MB on Kata. On the Buff/cache usage there is a spike in the beginning for every runtime and the major difference is when this memory is released. Native and runc release this memory after around 4-5 seconds, crun on the other hand release after around 95 seconds and Kata does not seem to release this at all, but it most likely happens after the container is shutdown. Native uses 327 MB of Buff/cache, runc 174 MB, crun 36 MB and Kata 136 MB. See figures 4.12, 4.13, 4.14, 4.15 for details on native, runc, crun and Kata respectively.

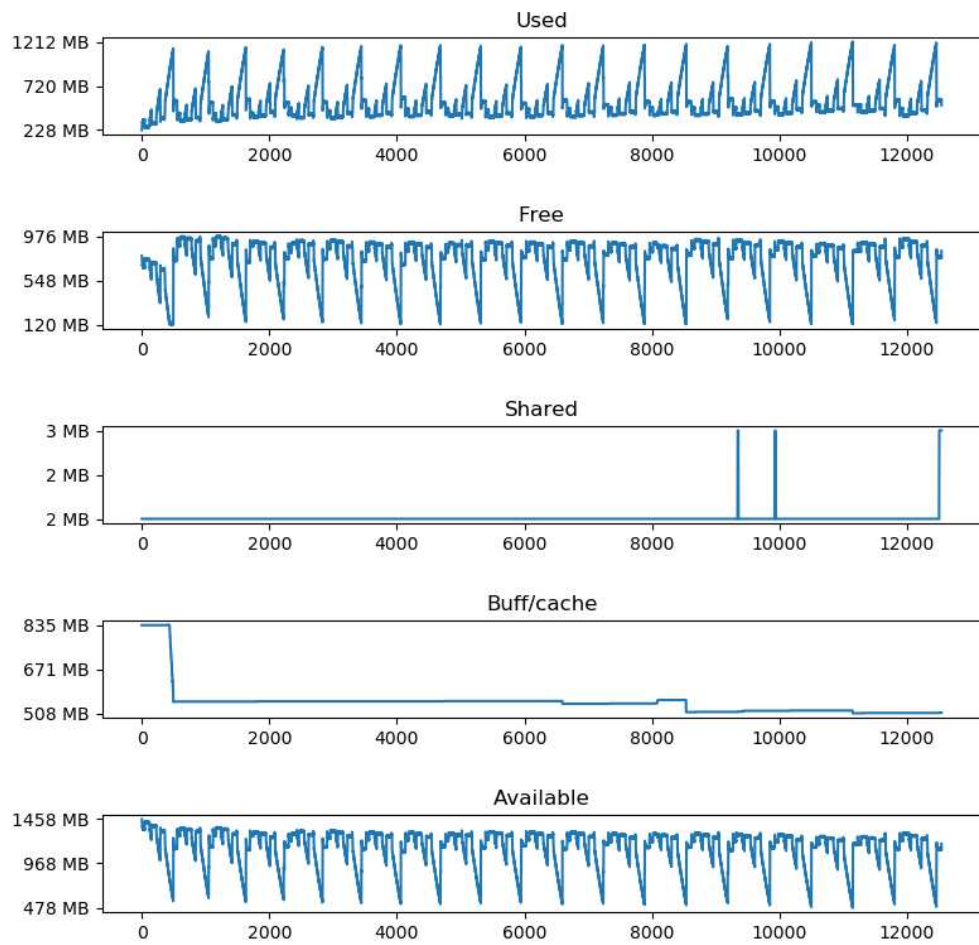


Figure 4.12. Memory usage on ARM native when running the 7z benchmark.

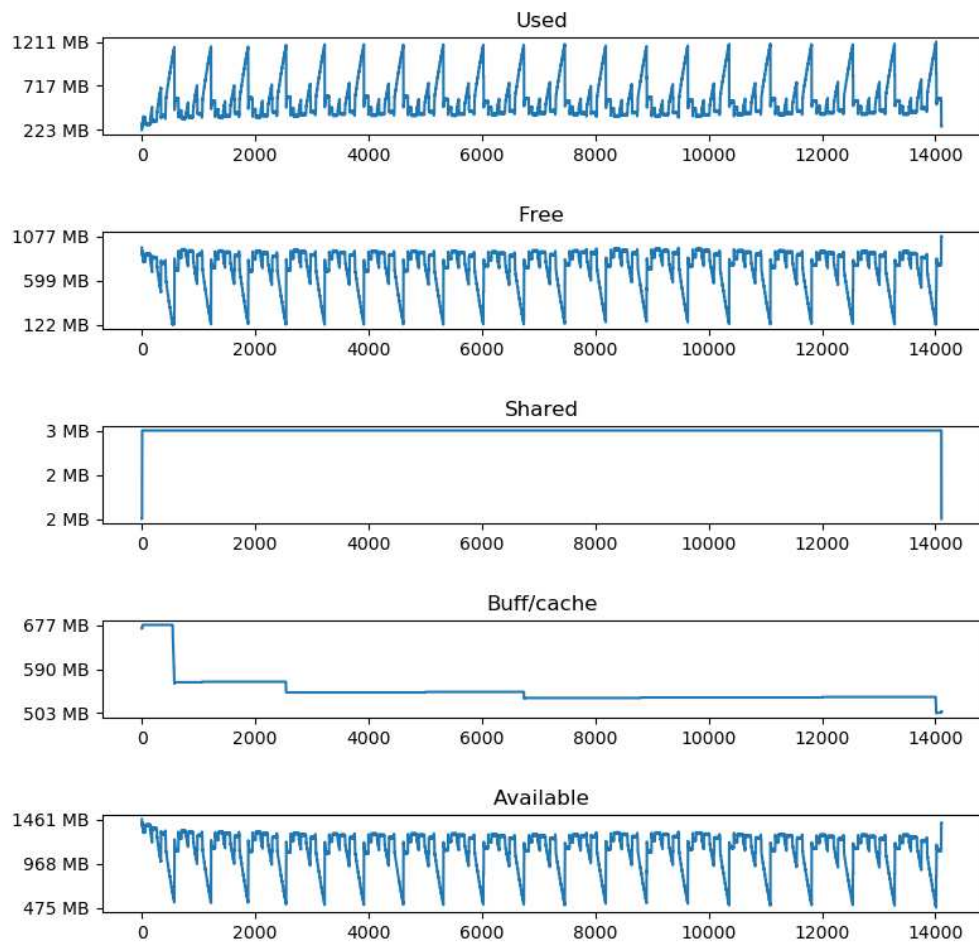


Figure 4.13. Memory usage on ARM runc when running the 7z benchmark.

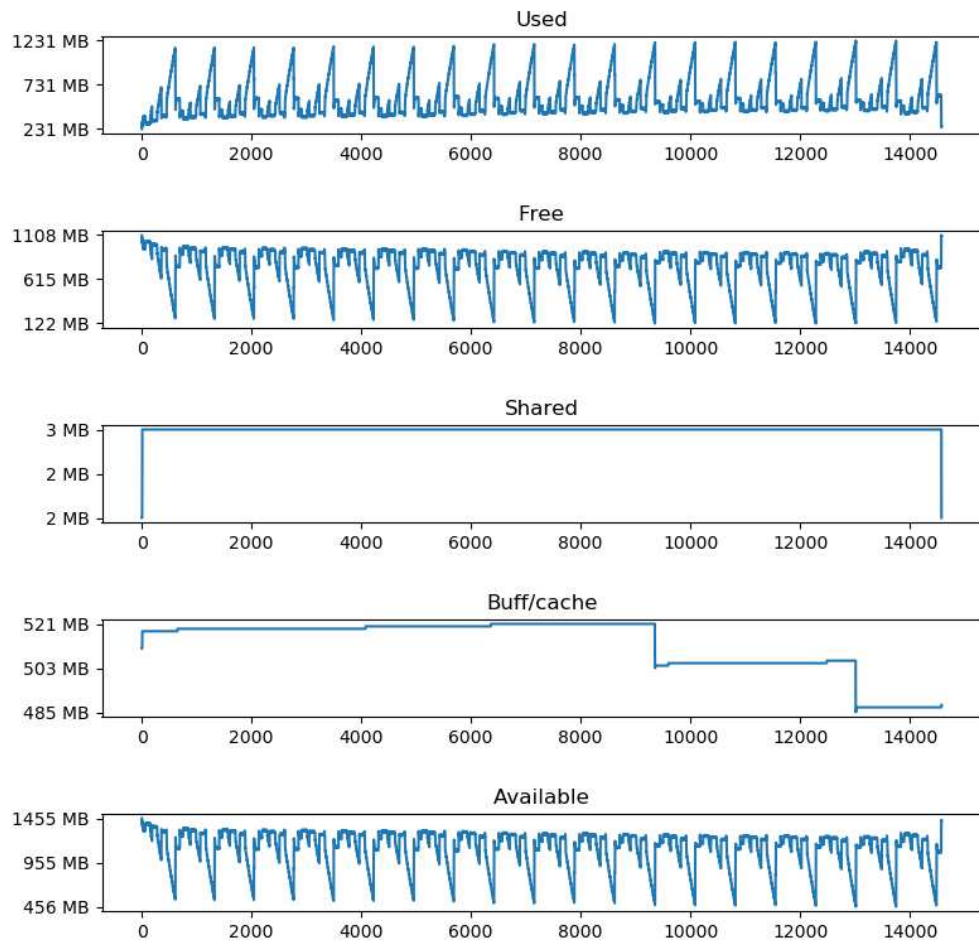


Figure 4.14. Memory usage on ARM crun when running the 7z benchmark.

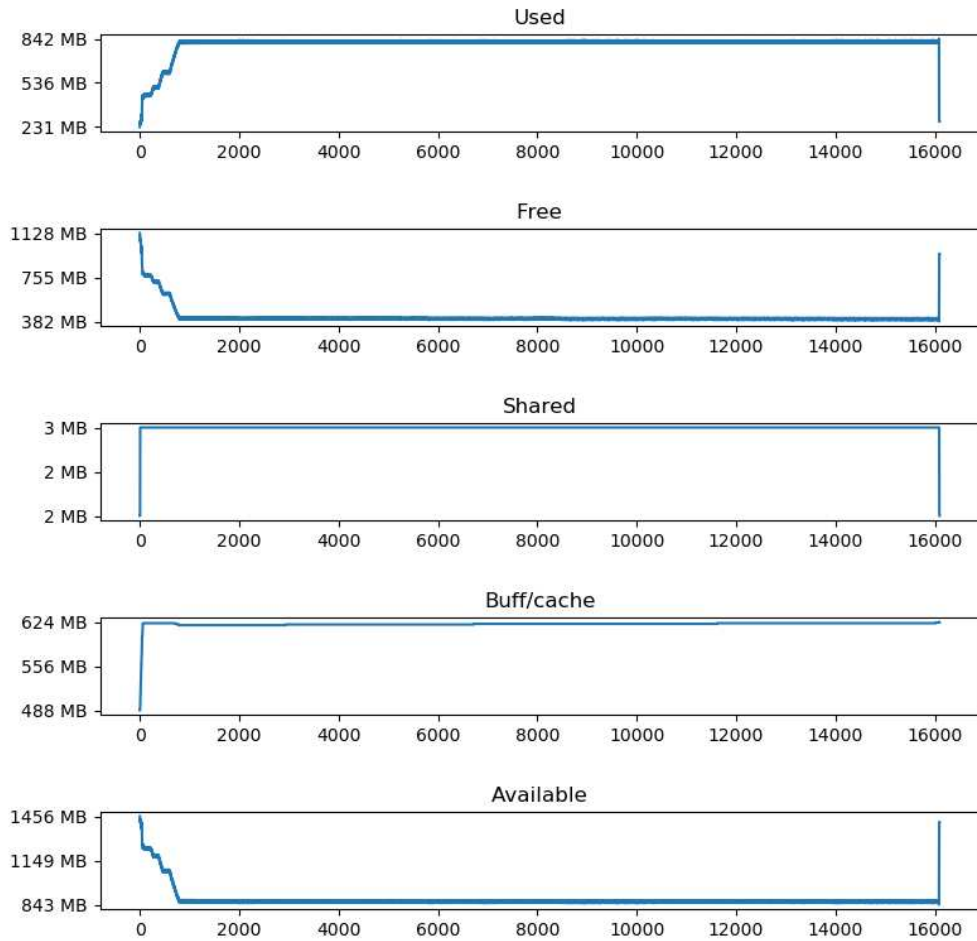


Figure 4.15. Memory usage on ARM Kata when running the 7z benchmark.

4.5.2 x86 results

Similarly to the ARM counterpart the measurements make a saw-blade pattern on the memory usage. And again the Kata runtime reserves all the memory beforehand and releases it once the container is shutdown 4.20. The memory behaviour with gVisor is also different when compared to native, crun or runc. To complete the benchmark native requires 1841 MB of memory, runc 1883 MB, crun 1895 MB, gVisor 72 MB, Kata 669 MB. gVisor heavily uses the shared memory Buffer / cache. The shared memory follows the same pattern as the free memory on other runtimes. For more details see figures 4.16 for native, 4.17 for runc, 4.18 for crun, 4.19 for gVisor, 4.20 for Kata.

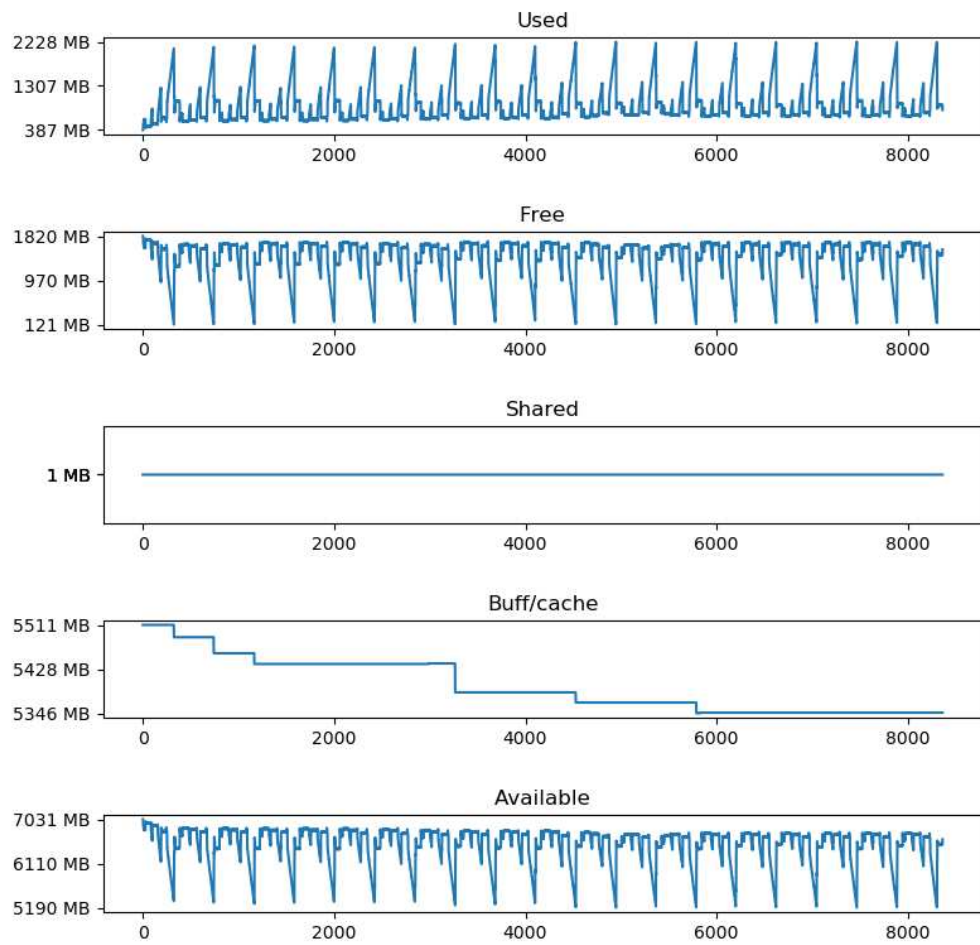


Figure 4.16. Memory usage on x86 native when running the 7z benchmark.

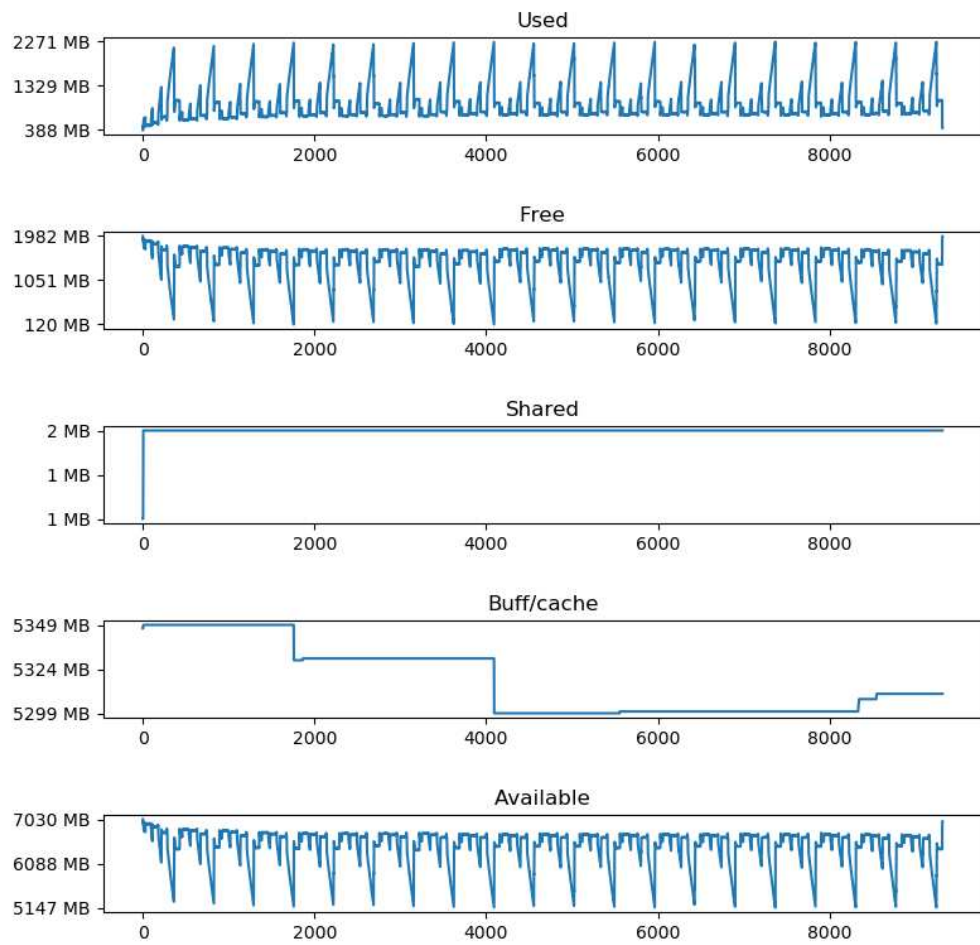


Figure 4.17. Memory usage on x86 runc when running the 7z benchmark.

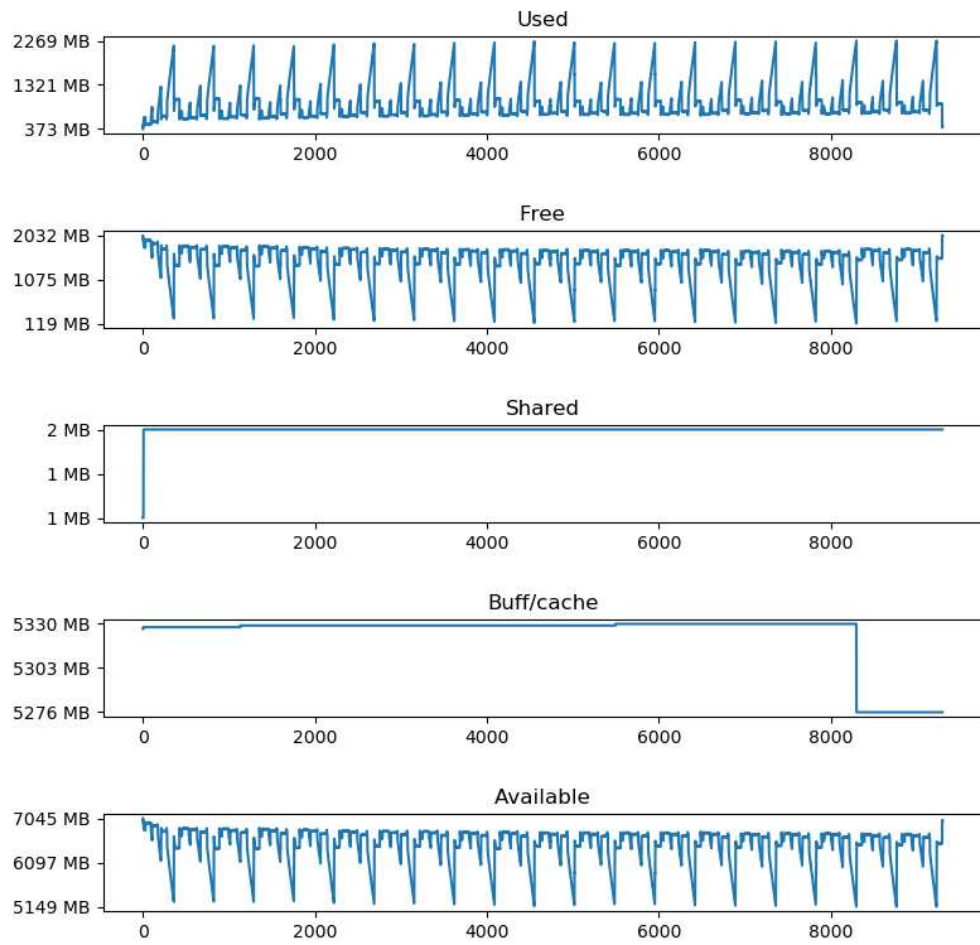


Figure 4.18. Memory usage on x86 crun when running the 7z benchmark.

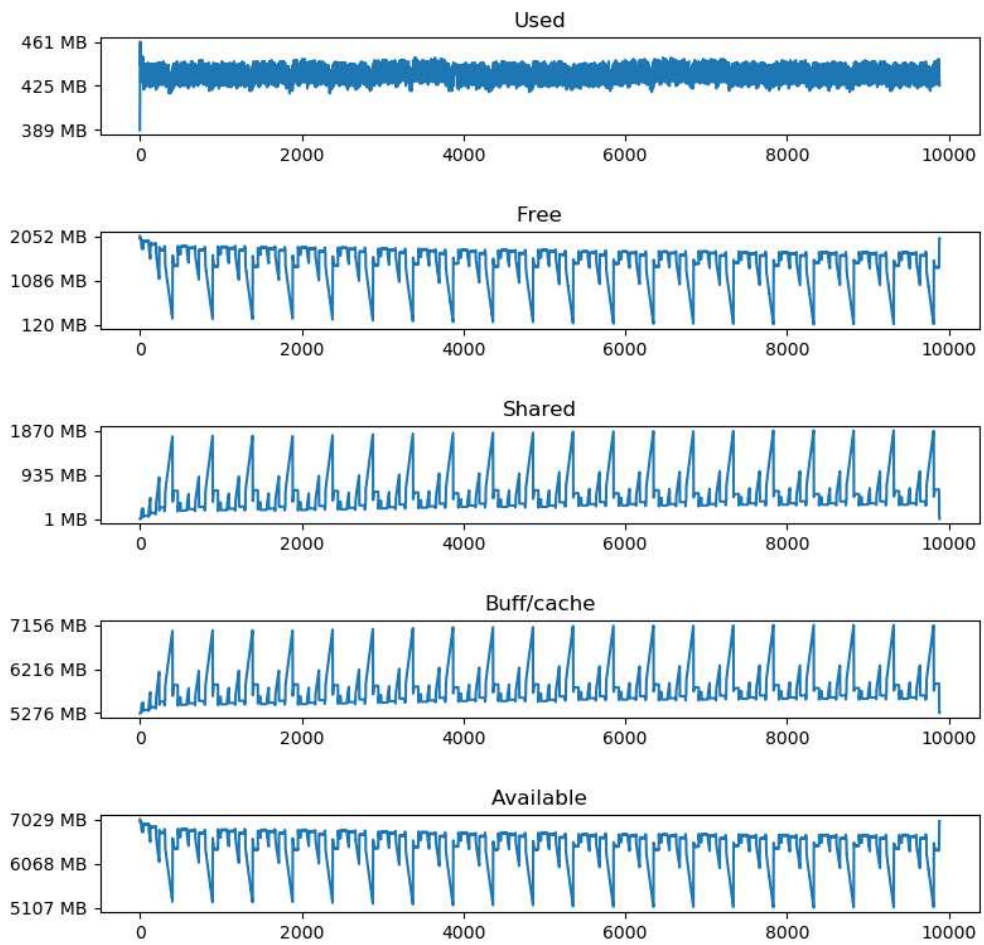


Figure 4.19. Memory usage on x86 gVisor when running the 7z benchmark.

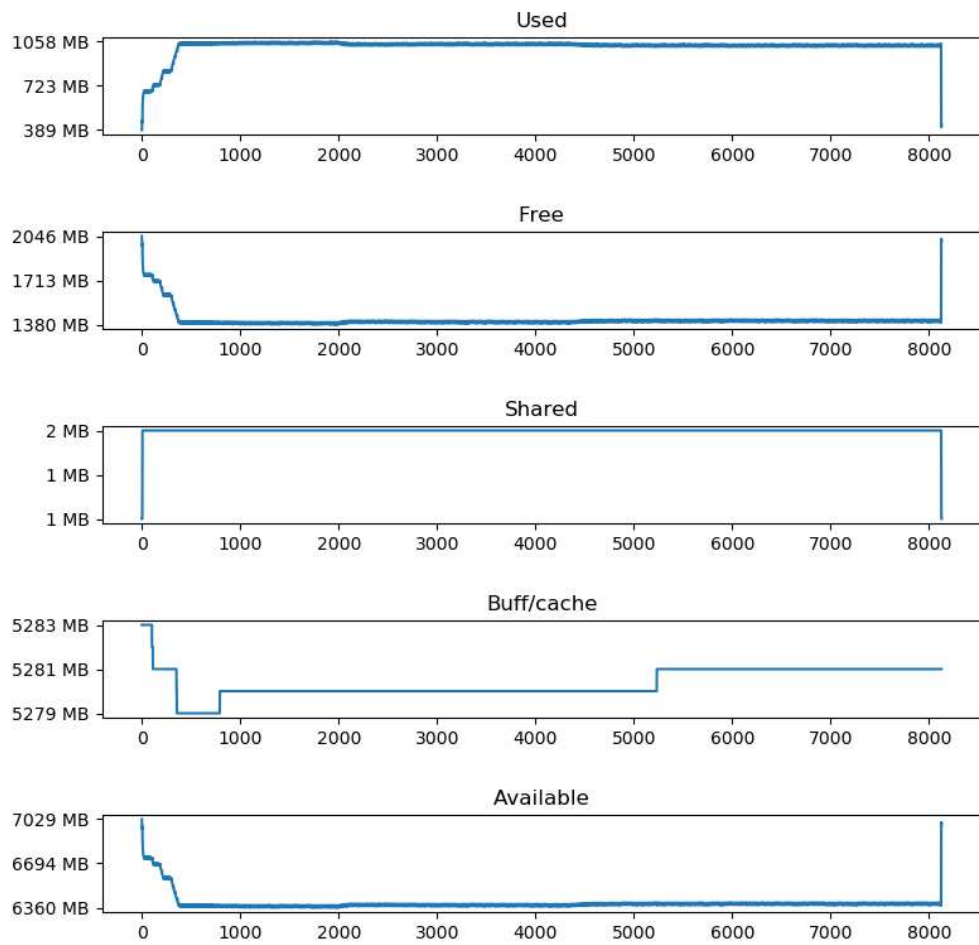


Figure 4.20. Memory usage on x86 Kata when running the 7z benchmark.

4.6 Dependency analysis

From the five used runtimes runc, Kata, and gVisor were installed with debian package manager so their dependencies were directly available. For runc on the other hand, which was built from source the dependencies were identified manually with the help of symbolic links. Kata on the ARM was installed with snap an universal Linux package format and there was no clear way to extract the dependencies. So instead the dependency list that was acquired from x86 was used and the size calculate with that.

When it came to size of dependencies the crun runtime was clearly the smallest only requiring 17.3 MB of space including dependencies. The next one was gVisor that only required 64.3 MB, followed by runc and finally Kata as the largest one. When it came to the amount of dependencies runc and crun only required 6, but Kata on the other hand required 52 on x86 and 85 on arm and gVisor had none.

Table 4.9. Execution dependencies for the used container runtimes.

runtime	ARM		x86	
	size / MB	packages	size / MB	packages
runc	≈ 113.3 MB	6	≈ 135.1 MB	6
crun	≈ 17.3 MB	6	≈ 19.7 MB	6
Kata	≈ 172.7 MB	85	≈ 718.1 MB	52
gVisor	Not available		≈ 64.3 MB	0

5. RESULT ANALYSIS

Given the ideal state as defined in section 3.2 the results in chapter 4 are evaluated. Theoretical solutions to these problems are also discussed briefly.

5.1 Portability

When it comes to portability none of the runtimes offered any kind of portability. All the error messages are saying that the binary cannot be executed on this processor architecture. There is no emulation support of any kind added directly to the runtimes. Whether this level of portability could be achieved with some kind of tricks though still remains open. For example the benefit of portability and easier deployment could still be achieved, by using binary emulation on the developer machine. Meaning that the application would be coded for ARM on the developer machine by using some sort of emulated container development environment that is emulating ARM on x86. Or the developer could use an ARM based laptop as his development environment from the start. This would not be a perfect solution since the application would have to be ported for different processor architectures, but would make easier to deploy applications to suitable embedded devices. ARM devices to be specific in this example.

When digging deeper to the layers of the container image it turned out that the portability of Python is deeply engrained into the libraries, meaning that even though the language is portable the libraries are not always so. Pure Python libraries are portable since they are transformed to machine code during runtime, but many of the Python libraries have bindings to C / C++ libraries. This is a design choice of Python that was made to make better performing libraries. If one wished to achieve portability with Python on containers the container should only include pure Python libraries. So even though the OCI runtime specification defines processor architecture agnostic containers, in practice creating such a container is not easy. You cannot include any binaries in the container, but instead everything needs to be on a higher level in a format that can be run on any processor architecture. It was envisioned that Python could achieve this as it can be transformed easily across architectures. Whether any other high level programming language such as JavaScript programming language is up to the task remains to be seen.

5.2 Database performance

As one might have expected none of the containers are faster than native on all the metrics. There are some cases on ARM where the crun or runc runtimes are performing better than native, but that can be explained by variations across tests. On x86 there is clear distinction where native is faster across all the metrics. The cause for weird ARM performance can be that it is a lower end platform or maybe the processor architecture has some effect on how the runtimes operate. More research needs to be done to identify the root cause and the extent of the effect.

The ideal state for a runtime was defined to offer 95 - 99% of the native performance. Based on this the best performing is runc were at worst you get 84% of the native performance on ARM and 80% on x86 when looking at requests per second 5.1. This might be acceptable for some some but there is clearly room for improvement. Other runtime solutions such as gVisor and Kata that offer better isolation offer much worse performance that makes them less usable for an embedded environment. One thing to note is that the Redis database benchmark needs to do a lot of communication between two containers. Section 5.3 shows performance when the benchmark is not relying on communicating with the outside world.

Table 5.1. *RPS (request per second) / RPS native * 100% minimum maximum*

CPU architecture	Runc	crun	gVisor	Kata
ARM	84 - 104%	83 - 100%	-	68 - 90%
x86	79 - 96%	59 - 85%	5 - 18%	42 - 68%

The methods described in achieving container isolation in section 2.3.2 make it seem like the process is like any other process, but just limited visibility. One might expect the performance to be very close to native. At least based on the Redis benchmark, there is significant overhead when the container is communicating with the outside world. This is an ongoing struggle and will likely improve in the future. When looking at the latencies the story is similar to requests per seconds where adding security measures makes the performance worse see 5.1 and 5.2. This is especially the case for gVisor which is \approx 10 times worse than native. The addition of a second kernel between the processor and process is clearly adding a lot of overhead. This is always a trade off when considering security and performance and it applies to container runtimes as well.

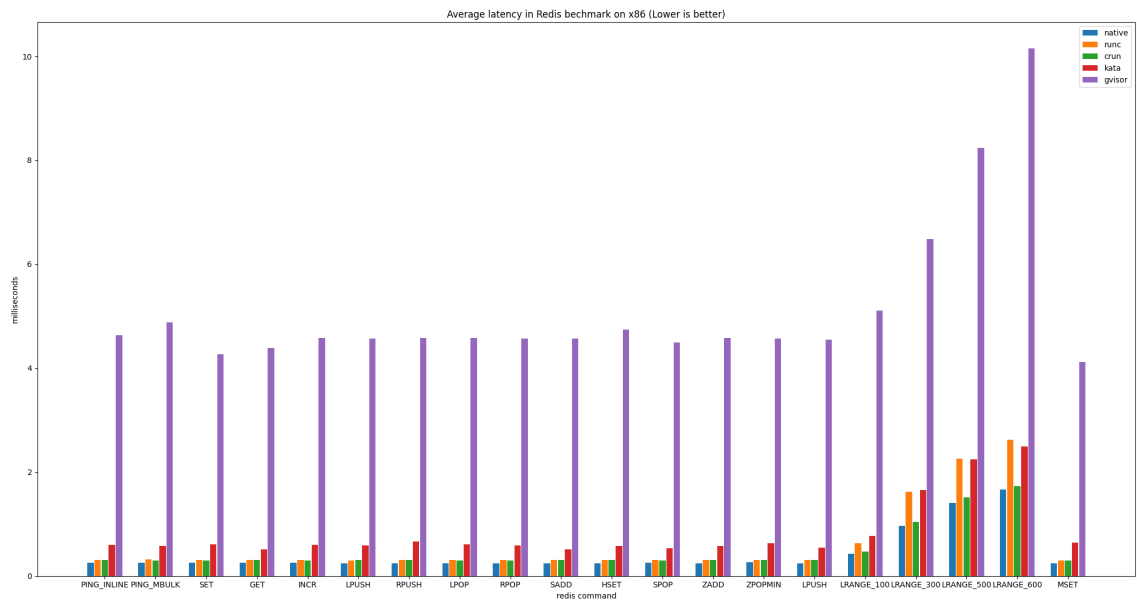


Figure 5.1. Latency Barchart on x86.

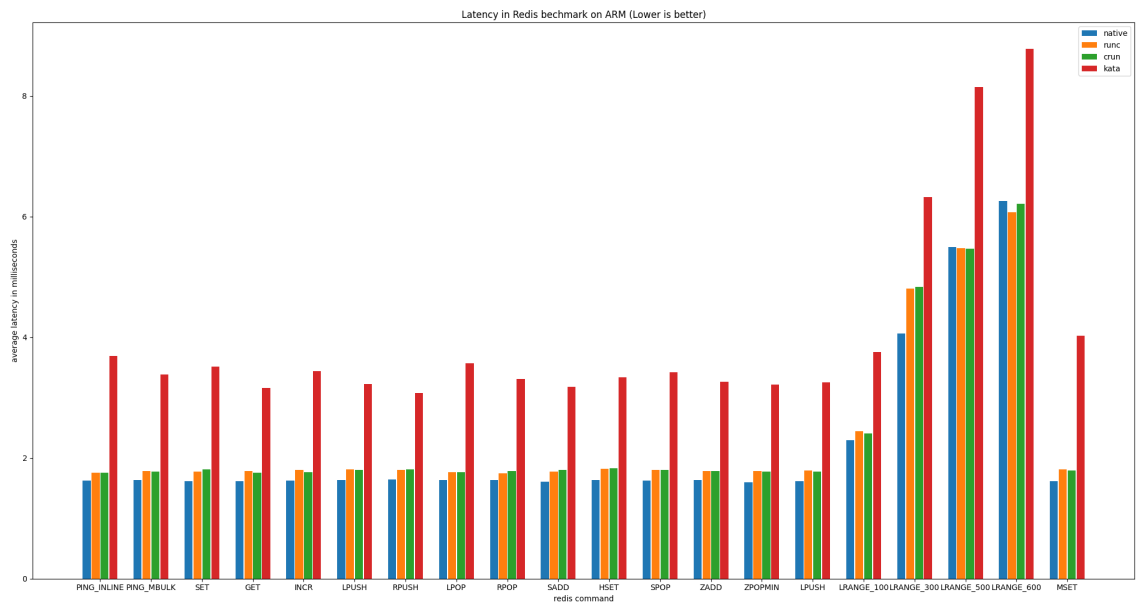


Figure 5.2. Latency Barchart on ARM.

5.3 7z benchmark

Where the database performance (Redis benchmark) was more dependant on the containers ability to communicate with the outside world 7z will remain inside the container

the whole time. With Redis benchmark runtimes were clearly inferior to native implementation the story is quite different on 7z. On ARM runc is showing better rating on both compression and decompression than native. Crun on the the hand is slightly worse than native. The ideal state where performance is 95 - 99% of native is fulfilled by runc and crun and gVisor is not far of either with at worst 88% of native performance on both architectures 5.2.

Table 5.2. *Rating / Rating native * 100%*

CPU architecture	Runc	crun	gVisor	Kata
ARM	102 - 105%	97 - 100 %	-	35 - 40%
x86	99 %	99 - 100%	88 - 97%	20 - 23%

Looking at the R / U column (table 4.8 and 4.7) though tells a different story where Kata outperforms every single other runtime including the native (see also 5.3). This is a bug in the 7z / Kata runtime benchmark. The Kata runtime shows usage of 100 meaning that one core is utilized 100%. But when we look at the other runtimes their usage is 600 - 800. What is probably happening is that when the calculation is done for R/U column the usage is reported incorrectly by the Kata runtime which causes the wrong R/U value. Kata is using more than one core, but reports the Usage incorrectly.

Table 5.3. *(R/U) / (R/U native) * 100%. Wrongly reported Kata marked in red.*

CPU architecture	Metric	Runc	crun	gVisor	Kata
ARM	compression average	101%	99%	-	133%
x86	compression average	98%	99%	76%	133%
ARM	compression total	102%	97%	-	128%
x86	compression total	98%	99%	85%	155%
ARM	decompression average	103%	95%	-	125%
x86	decompression average	99%	99%	96%	184%

When the incorrectly calculated Kata is taken of from the results we can see that crun and runc are well within the ideal state, but gVisor this time at worst 76% of native performance. Based on these results it can be said that on this benchmark the container runtimes offer good enough performance to be used in embedded devices. The better performance can be explained by the 7z benchmark not requiring any communication outside of the container. This makes it completely viable to use containers for embedded application based on this performance test.

5.4 Memory usage and storage space

Measuring the memory usage during Redis benchmark showed us that the containers have a memory overhead when starting the container. The memory overhead was the smallest with runc and crun on both CPU architectures. GVisor and Kata required the most overhead as one might have expected based on their architecture. If we compare these results to the ideal state setup where for a tiny application the memory overhead should be <1% of the overall memory pool. When this is put to a numeric value the memory overhead on ARM should be < 18 MB and < 77 MB on x86. On ARM the overhead is between between 4% - 22% (82 - 405 MB) which more than the ideal state. On x86 the overhead is between between 1% - 7% (77 - 499 MB) and which is also more than the ideal state of 1%. Meaning that even in the higher end system where there is 8 Gigabytes of memory the ideal state of <1% of RAM usage cannot be achieved.

To achieve the ideal state there might be some optimization to be done to make runtimes utilize less memory, but with the current runtimes less than 1% of memory usage entire memory pool is not possible in the beginning. The initial spike is created by the container creation and starting, which is not as important as overhead after starting. The overhead when the container is running calculated with this simplified calculation

$$m_{runtime} - m_{native} = m_{overhead}$$

where $m_{runtime}$ is the amount of memory used by runtime, m_{native} the amount of memory used by native and $m_{overhead}$ is the amount of memory overhead. For example on ARM native memory fluctuated during Redis benchmark between 220 - 249 MB and runc fluctuated between 263 - 287 MB the runtime overhead is 38 - 43 MB. This assumes that the the peak memory usage is otherwise similar except the runtime overhead. Below overhead estimation calculated to all the other runtimes on all the tests 5.4.

Table 5.4. Memory overhead estimation.

Architecture	runtime	benchmark	overhead estimation
ARM	runc	Redis	38 - 43 MB
	crun	Redis	39 - 45 MB
	Kata	Redis	382 - 386 MB
x86	runc	Redis	(-1) - 58 MB
	crun	Redis	40 - 57 MB
	Kata	Redis	461 - 462 MB
	gVisor	Redis	4 - 72 MB
ARM	runc	7z	(-5) - (-1) MB
	crun	7z	3 - 19 MB
	Kata	7z	(-370) - 19 MB
x86	runc	7z	1 - 43 MB
	crun	7z	(-14) - 41 MB
	Kata	7z	(-1170) - 2 MB
	gVisor	7z	(-1767) - 2 MB

During the Redis benchmark Kata seems to have the biggest overhead of all the runtimes. The story is completely different on 7z benchmark where many of the runtimes seem to have negative overhead. The 7z benchmark does not act the same way as in native when it is put to Kata or gVisor container. Based on the performance numbers Kata and gVisor performed worse than the other two. This could explain some of the performance drop, where the runtime cannot properly utilize all the available memory as runc and crun can. If we look at the ideal state of using < 1% of the memory pool the x86 platform meets the criteria for every runtime except Kata. On ARM though, where there is around 2 Gigabytes of memory available the ideal state cannot be achieved. The memory usage is still less than < 3% of all the memory which is probably enough for many uses cases. Storage space on the other hand was lowest for runc with 17.3 MB on ARM and 19.7 MB on x86 making it well within the ideal state on ARM and x86. The other runtimes do not make it under the threshold of < 1% of the entire memory pool. The dependencies are also quite reasonable, but none of them can beat gVisor which has no other dependencies making it ideal for adding to any Linux image.

6. CONCLUSION

None of the runtimes were identified to be perfect for embedded devices and further research is needed in the matter. Performance in constrained environment or ARM also showed differences when compared to a higher level system with more resources and x86 architecture. Is the only cause for these performance differences the architecture or could the runtimes operate differently on resource constrained environments?

The testing revealed that adding security features increases the performance overhead. This makes runtimes such as runc and crun more appealing for embedded devices. But gVisor seems to be reasonable candidate as well for beefier devices where there is room to leave performance on the table. gVisor offered added security features and provided good enough performance on some use cases. It also had the least amount of dependencies with a second smallest install size. The downside for this runtime currently is that it only supports x86 CPU architecture, which makes it harder to be utilizable on embedded devices. Dependencies and install size of all the runtimes are all reasonably sized and could be installed on smaller devices. Kata runtime would be the most unlikely candidate to be used in an embedded device for its poor performance in many of the applications and hardware support requirements.

Portability seems to be a huge question mark in the container space. The OCI specification detailed CPU architecture agnostic containers to be possible, but the usage of high level language Python did not make the container portable. The advanced runtimes that made use of in between kernels or virtual machines do not provide any binary emulation or add any portability benefits. It remains to be seen if other high level programming languages such as JavaScript can be portable between x86 and ARM on containers. Maybe there is a need for new type of runtimes designed to be portable from the get go utilizing binary emulation or development environment that support creating ARM container on other processor architectures.

Memory overhead estimation showed that Kata runtime had the largest overhead but for the other runtimes the overhead was less than 80 MB. 7z benchmark also showed that depending on the workload the overhead can be non existent or even smaller when using a container. Is this is only related to 7z or are Kata and gVisor not able to use as much memory as runc and crun?

REFERENCES

- [1] *An overview of s6*. URL: <https://skarnet.org/software/s6/overview.html>.
- [2] Bobo Bauer, Jorge Nerin, and Stefani Seibold. *The proc FILESYSTEM*. URL: <https://www.kernel.org/doc/Documentation/filesystems/proc.txt>.
- [3] *cgroups namespace*. URL: <https://man7.org/linux/man-pages/man7/cgroups.7.html>.
- [4] Minh Thanh Chung et al. “Using Docker in high performance computing applications”. eng. In: *2016 IEEE 6th International Conference on Communications and Electronics, IEEE ICCE 2016*. IEEE, 2016, pp. 52–57. ISBN: 1509019316.
- [5] The FreeBSD community. *FreeBSD Handbook*. URL: <https://docs.freebsd.org/en/books/handbook/>.
- [6] *Containers(V10): Generic Process Containers*. URL: <https://lwn.net/Articles/236032/>.
- [7] *cpuset(7) — Linux manual page*. URL: <https://man7.org/linux/man-pages/man7/cpuset.7.html>.
- [8] *Dive github*. URL: <https://github.com/wagoodman/dive>.
- [9] Lennart Espe et al. “Performance Evaluation of Container Runtimes.” In: *CLOSER*. 2020, pp. 273–281.
- [10] Phel Estes. *runc: The little container engine that could*. URL: <https://opensource.com/life/16/8/runc-little-container-engine-could>.
- [11] Wes Felter et al. “An updated performance comparison of virtual machines and Linux containers”. eng. In: *2015 IEEE International Symposium on Performance Analysis and Software (ISPASS)*. IEEE International Symposium on Performance Analysis of Systems and Software-ISPASS. IEEE. NEW YORK: IEEE, 2015, pp. 171–172. ISBN: 9781479919567.
- [12] The Linux foundation. *Open Container Initiative*. URL: <https://opencontainers.org/>.
- [13] *free utility*. URL: <https://man7.org/linux/man-pages/man1/free.1.html>.
- [14] freebsd. URL: <https://docs.freebsd.org/44doc/papers/jail/jail-9.html>.
- [15] freebsd. URL: [https://www.freebsd.org/cgi/man.cgi?chroot\(8\)](https://www.freebsd.org/cgi/man.cgi?chroot(8)).
- [16] *Github page crun runtime*. URL: <https://github.com/containers/crun>.
- [17] google. *What are containers?* URL: <https://cloud.google.com/learn/what-are-containers>.
- [18] *gVisor github*. URL: <https://github.com/google/gVisor>.
- [19] Yu-Ju Hong. *Introducing Container Runtime Interface (CRI) in Kubernetes*. URL: <https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-Kubernetes/>.

- [20] *IPC namespace*. URL: https://man7.org/linux/man-pages/man7/ipc_namespaces.7.html.
- [21] *Kata containers*. URL: <https://github.com/Kata-containers>.
- [22] *Kata container github page on limitations*. URL: <https://github.com/Kata-containers/Kata-containers/blob/main/docs/Limitations.md>.
- [23] Jay Kruemcke. *Wind River Linux LTS 21 is no available*. URL: <https://www.linkedin.com/pulse/wind-river-linux-lts-21-now-available-jay-kruemcke>.
- [24] Rakesh Kumar and B Thangaraju. "Performance Analysis Between RunC and Kata Container Runtime". eng. In: *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. IEEE, 2020, pp. 1–4. ISBN: 1728168287.
- [25] Daniel Lezcano. *LXC github*. URL: https://github.com/lxc/lxc/tree/lxc_0_1_0.
- [26] Redis Ltd. *Introrduction to Redis*. URL: <https://Redis.io/topics/introduction>.
- [27] Redis Ltd. *Redis benchmark*. URL: <https://Redis.io/topics/benchmarks>.
- [28] Redis Ltd. *Redis commands*. URL: <https://Redis.io/commands/>.
- [29] *LZMA Benchmark Description*. URL: <https://www.7-cpu.com/>.
- [30] Valentina Manea et al. "Native Runtime Environment for Internet of Things". eng. In: *Advanced Computational Methods for Knowledge Engineering*. Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, pp. 381–390. ISBN: 3319179950.
- [31] *Manual mage for namespaces*. URL: <https://man7.org/linux/man-pages/man7/namespaces.7.html>.
- [32] *Manual page for seccomp*. URL: <https://man7.org/linux/man-pages/man2/seccomp.2.html>.
- [33] John Paul Martin, A Kandasamy, and K Chandrasekaran. "Exploring the support for high performance applications in the container runtime environment". eng. In: *Human-centric computing and information sciences 8.1 (2018)*, pp. 1–15. ISSN: 2192-1962.
- [34] Ilias Mavridis and Helen Karatza. "Orchestrated sandboxed containers, unikernels, and virtual machines for isolation-enhanced multitenant workloads and serverless computing in cloud". eng. In: *Concurrency and computation (2021)*. ISSN: 1532-0626.
- [35] *Merkle Directed Acyclic Graphs (DAGs)*. URL: <https://docs.ipfs.io/concepts/merkle-dag/>.
- [36] *Mount namespace*. URL: https://man7.org/linux/man-pages/man7/mount_namespaces.7.html.
- [37] *Network namespace*. URL: https://man7.org/linux/man-pages/man7/network_namespaces.7.html.
- [38] *Picotts github page*. URL: <https://github.com/naggety/picotts>.

- [39] *PID (process id) namespace*. URL: https://man7.org/linux/man-pages/man7/pid_namespaces.7.html.
- [40] Matthew Portnoy. *Virtualization Essentials*. eng. Newark: John Wiley and Sons, Incorporated, 2016. ISBN: 1119267722.
- [41] *RKT github page*. URL: <https://github.com/rkt/rkt>.
- [42] McCarty Scott. *A Practical Introduction to Container Terminology*. URL: <https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction#>.
- [43] Rohit Seth. *Containers: Introduction*.
- [44] *Seven Segment Optical Character Recognition*. URL: <https://www.unix-ag.uni-kl.de/~auerswal/ssocr/>.
- [45] sorcun. *Supported devices*. URL: https://openwrt.org/supported_devices.
- [46] Chris Swan. *Docker drops LXC as default execution environment*. URL: https://www.infoq.com/news/2014/03/docker_0_9/.
- [47] IBM cloud team. *Containers vs. Virtual Machines (VMs): What's the Difference?* URL: <https://www.ibm.com/cloud/blog/containers-vs-vms>.
- [48] LXC team. *LXC 1.0.0 realease announcement*. URL: https://linuxcontainers.org/lxc/news/2014_02_20_00_00.html.
- [49] *Time namespace*. URL: https://man7.org/linux/man-pages/man7/time_namespaces.7.html.
- [50] *User namespace*. URL: https://man7.org/linux/man-pages/man7/user_namespaces.7.html.
- [51] *UTS namespace*. URL: https://man7.org/linux/man-pages/man7/uts_namespaces.7.html.
- [52] Various. *Home assistant web page*. URL: <https://www.home-assistant.io/>.
- [53] Various. *Merkle Tree*. URL: https://en.wikipedia.org/wiki/Merkle_tree.
- [54] Various. *OCI Image Format Specifiation*. URL: <https://github.com/opencontainers/image-spec>.
- [55] Various. *OCI Runtime specification: Runtime and Lifecycle*. URL: <https://github.com/opencontainers/runtime-spec/blob/main/runtime.md>.
- [56] Various. *Sanxbox (computer security)*. URL: [https://en.wikipedia.org/wiki/Sandbox_\(computer_security\)](https://en.wikipedia.org/wiki/Sandbox_(computer_security)).
- [57] Various. *Virtualization*. URL: <https://en.wikipedia.org/wiki/Virtualization>.
- [58] *What is gVisor*. URL: <https://gvisor.dev/docs/>.
- [59] *Yocto home page*. URL: <https://www.yoctoproject.org/>.
- [60] Ethan G. Young et al. "The True Cost of Containing: A gVisor Case Study". In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, July 2019. URL: <https://www.usenix.org/conference/hotcloud19/presentation/young>.

APPENDIX A: APPENDIX

A.1 Redis benchmark ARM results raw version

Table A.1. ARM native results on Redis benchmark

test	Request/ second	Average latency (ms)	Minimum latency (ms)	50th per- centile (ms)	95th per- centile (ms)	99th per- centile (ms)	Maximum latency (ms)
PING_INLINE	15951.51	1.623	0.552	1.535	2.095	2.767	9.631
PING_MBULK	15775.36	1.637	0.536	1.543	2.095	2.759	8.991
SET	16113.44	1.613	0.68	1.511	2.111	2.775	10.783
GET	16007.68	1.612	0.464	1.519	2.095	2.591	8.751
INCR	15893.2	1.624	0.848	1.527	2.103	2.687	7.471
LPUSH	15898.25	1.63	0.648	1.527	2.111	2.687	8.551
RPUSH	15862.94	1.639	0.552	1.527	2.159	2.879	12.599
LPOP	15883.1	1.634	0.528	1.527	2.119	2.751	10.431
RPOP	15933.72	1.631	0.592	1.519	2.127	2.935	10.959
SADD	16189.09	1.602	0.504	1.503	2.087	2.735	8.735
HSET	15810.28	1.636	0.528	1.535	2.119	2.687	8.743
SPOP	15976.99	1.622	0.52	1.527	2.103	2.783	9.431
ZADD	15880.58	1.63	0.688	1.535	2.103	2.623	9.471
ZPOPMIN	16168.15	1.598	0.584	1.503	2.087	2.647	8.343
LPUSH (needed to benchmark LRANGE)	16095.28	1.614	0.512	1.511	2.103	2.759	10.079
LRANGE_100 (first 100 ele- ments)	11234.69	2.289	1.12	2.159	2.847	3.687	13.095
LRANGE_300 (first 300 ele- ments)	6191.95	4.06	1.992	3.927	5.143	6.311	16.783
LRANGE_500 (first 500 ele- ments)	4560.38	5.491	2.104	5.311	6.991	8.447	24.223
LRANGE_600 (first 600 ele- ments)	3989.63	6.259	2.344	6.087	8.007	9.583	17.759
MSET (10 keys)	16194.33	1.614	0.624	1.503	2.103	2.727	10.239

Table A.2. ARM runc results on Redis benchmark

test	Request/ second	Average latency (ms)	Minimum latency (ms)	50th per- centile (ms)	95th per- centile (ms)	99th per- centile (ms)	Maximum latency (ms)
PING_INLINE	14810.43	1.758	0.704	1.679	2.199	2.999	9.527
PING_MBULK	14553.92	1.784	0.52	1.719	2.239	2.967	8.663
SET	14695.08	1.771	0.504	1.687	2.239	2.927	11.063
GET	14577.26	1.786	0.496	1.711	2.279	3.039	7.687
INCR	14376.08	1.801	0.608	1.743	2.255	2.903	6.911
LPUSH	14440.43	1.806	0.768	1.719	2.271	3.199	8.631
RPUSH	14478.07	1.8	0.624	1.719	2.303	3.103	9.367
LPOP	14771.05	1.766	0.504	1.679	2.247	3.007	14.239
RPOP	14903.13	1.748	0.792	1.663	2.207	2.951	7.719
SADD	14688.6	1.769	0.504	1.703	2.207	2.975	9.231
HSET	14330.75	1.816	0.68	1.743	2.263	3.111	7.511
SPOP	14386.42	1.802	0.472	1.735	2.295	2.967	7.335
ZADD	14594.28	1.786	0.536	1.711	2.239	3.071	9.351
ZPOPMIN	14547.57	1.785	0.48	1.719	2.255	2.935	7.775
LPUSH (needed to benchmark LRANGE)	14537	1.792	0.472	1.711	2.279	3.103	8.871
LRANGE_100 (first 100 ele- ments)	10564.12	2.44	1.272	2.327	2.943	3.975	11.551
LRANGE_300 (first 300 ele- ments)	5226.03	4.802	2.24	4.719	6.223	7.559	13.887
LRANGE_500 (first 500 ele- ments)	4576.66	5.477	2.288	5.319	6.775	8.047	25.311
LRANGE_600 (first 600 ele- ments)	4125.07	6.066	2.76	5.903	7.527	8.959	21.119
MSET (10 keys)	14499.06	1.812	0.568	1.711	2.327	3.223	8.943

Table A.3. ARM crun results on Redis benchmark

test	Request/ second	Average latency (ms)	Minimum latency (ms)	50th per- centile (ms)	95th per- centile (ms)	99th per- centile (ms)	Maximum latency (ms)
PING_INLINE	14727.54	1.758	0.576	1.687	2.207	2.807	6.815
PING_MBULK	14583.64	1.774	0.6	1.711	2.231	2.831	6.919
SET	14367.82	1.808	0.72	1.735	2.247	3.079	7.343
GET	14817.01	1.755	0.552	1.663	2.255	2.967	10.039
INCR	14764.51	1.762	0.536	1.687	2.183	3.015	8.839
LPUSH	14444.61	1.805	0.528	1.719	2.255	3.079	10.015
RPUSH	14376.08	1.809	0.52	1.727	2.303	3.111	9.247
LPOP	14817.01	1.765	0.616	1.671	2.231	3.103	8.671
RPOP	14677.82	1.778	0.664	1.687	2.239	3.039	10.415
SADD	14471.78	1.798	0.6	1.735	2.263	3.071	7.375
HSET	14194.46	1.829	0.52	1.751	2.279	3.055	8.639
SPOP	14446.69	1.8	0.544	1.727	2.271	3.007	10.655
ZADD	14639.15	1.779	0.488	1.695	2.231	3.055	9.247
ZPOPMIN	14677.82	1.771	0.52	1.695	2.215	2.991	13.847
LPUSH (needed to benchmark LRANGE)	14652.01	1.774	0.52	1.687	2.279	2.991	8.023
LRANGE_100 (first 100 ele- ments)	10674.64	2.406	1.048	2.311	2.887	3.615	11.407
LRANGE_300 (first 300 ele- ments)	5195.61	4.829	2.232	4.719	6.247	7.615	15.983
LRANGE_500 (first 500 ele- ments)	4590.74	5.466	2.24	5.311	6.775	8.127	15.623
LRANGE_600 (first 600 ele- ments)	4021.39	6.206	2.56	6.039	7.647	8.991	17.263
MSET (10 keys)	14598.54	1.794	0.592	1.703	2.255	2.991	9.607

Table A.4. ARM Kata results on Redis benchmark

test	Request/ second	Average latency (ms)	Minimum latency (ms)	50th per- centile (ms)	95th per- centile (ms)	99th per- centile (ms)	Maximum latency (ms)
PING_INLINE	12072.92	3.687	0.736	3.335	6.095	9.543	12.911
PING_MBULK	13104.44	3.378	0.632	2.639	5.911	9.303	18.319
SET	12615.11	3.514	0.736	2.919	6.111	9.519	13.087
GET	14035.09	3.163	0.704	2.383	5.911	9.079	14.007
INCR	12884.94	3.44	0.664	2.703	6.119	9.439	13.047
LPUSH	13768.42	3.222	0.712	2.527	5.991	9.039	12.303
RPUSH	14403	3.075	0.648	2.439	5.887	8.895	16.895
LPOP	12465.72	3.566	0.688	2.839	6.271	9.727	23.567
RPOP	13422.82	3.312	0.608	2.591	6.095	9.047	20.271
SADD	13943.11	3.175	0.72	2.439	5.919	8.999	15.271
HSET	13294.34	3.335	0.688	2.623	6.055	9.703	14.991
SPOP	12948.34	3.421	0.632	2.743	5.967	9.439	13.743
ZADD	13603.59	3.26	0.648	2.575	5.975	9.095	12.567
ZPOPMIN	13798.81	3.211	0.704	2.455	5.935	9.199	15.391
LPUSH (needed to benchmark LRANGE)	13664.94	3.253	0.704	2.583	6.015	9.055	13.767
LRANGE_100 (first 100 ele- ments)	10130.69	3.758	0.744	2.815	8.031	10.767	17.231
LRANGE_300 (first 300 ele- ments)	4858.61	6.322	0.848	5.431	12.479	16.943	29.455
LRANGE_500 (first 500 ele- ments)	3676.88	8.145	0.912	7.143	15.335	21.503	36.479
LRANGE_600 (first 600 ele- ments)	3417.87	8.773	1.008	7.855	15.743	21.519	33.567
MSET (10 keys)	11117.29	4.024	0.688	3.687	6.751	10.279	13.631

A.2 Redis benchmark x86 results raw version

Table A.5. X86 native results on Redis benchmark

test	Request/ second	Average latency (ms)	Minimum latency (ms)	50th per- centile (ms)	95th per- centile (ms)	99th per- centile (ms)	Maximum latency (ms)
PING_INLINE	102249.49	0.251	0.096	0.239	0.367	0.471	0.919
PING_MBULK	102354.15	0.25	0.064	0.247	0.279	0.399	0.759
SET	103305.79	0.248	0.096	0.247	0.279	0.391	0.679
GET	102986.61	0.248	0.088	0.247	0.279	0.407	0.671
INCR	102354.15	0.25	0.096	0.247	0.279	0.383	0.975
LPUSH	103950.1	0.247	0.072	0.239	0.279	0.407	0.719
RPUSH	105263.16	0.244	0.064	0.239	0.279	0.399	0.735
LPOP	107874.87	0.24	0.064	0.231	0.279	0.407	2.151
RPOP	106951.88	0.241	0.088	0.239	0.279	0.423	0.695
SADD	103950.1	0.246	0.08	0.239	0.279	0.407	0.823
HSET	103842.16	0.247	0.08	0.247	0.279	0.391	0.615
SPOP	102880.66	0.248	0.096	0.247	0.279	0.407	1.151
ZADD	105042.02	0.245	0.088	0.239	0.279	0.399	0.815
ZPOPMIN	96432.02	0.265	0.088	0.255	0.359	0.423	1.311
LPUSH (needed to benchmark LRANGE)	104602.52	0.245	0.088	0.239	0.279	0.407	0.863
LRANGE_100 (first 100 ele- ments)	60459.49	0.423	0.256	0.423	0.479	0.647	1.375
LRANGE_300 (first 300 ele- ments)	25893.32	0.961	0.304	0.959	1.087	1.303	2.463
LRANGE_500 (first 500 ele- ments)	17658.48	1.409	0.304	1.399	1.615	1.935	4.919
LRANGE_600 (first 600 ele- ments)	14981.27	1.663	0.304	1.655	1.911	2.823	5.567
MSET (10 keys)	110132.16	0.242	0.088	0.239	0.279	0.479	0.911

Table A.6. X86 runc results on Redis benchmark

test	Request/ second	Average latency (ms)	Minimum latency (ms)	50th per- centile (ms)	95th per- centile (ms)	99th per- centile (ms)	Maximum latency (ms)
PING_INLINE	83752.09	0.305	0.088	0.303	0.367	0.455	0.815
PING_MBULK	84889.65	0.301	0.08	0.303	0.351	0.455	0.967
SET	84961.77	0.301	0.096	0.295	0.351	0.455	1.167
GET	83472.46	0.306	0.128	0.303	0.359	0.471	1.039
INCR	85034.02	0.301	0.104	0.287	0.359	0.447	1.007
LPUSH	84175.09	0.304	0.112	0.303	0.351	0.463	0.991
RPUSH	83892.62	0.305	0.112	0.295	0.351	0.455	0.743
LPOP	85251.49	0.301	0.104	0.295	0.351	0.479	0.959
RPOP	85106.38	0.301	0.12	0.295	0.359	0.527	0.999
SADD	84104.29	0.304	0.072	0.303	0.351	0.447	0.983
HSET	84033.61	0.305	0.096	0.295	0.351	0.455	0.823
SPOP	85324.23	0.3	0.112	0.295	0.351	0.439	0.767
ZADD	84530.86	0.303	0.096	0.295	0.359	0.471	0.999
ZPOPMIN	83333.33	0.306	0.072	0.303	0.351	0.455	0.871
LPUSH (needed to benchmark LRANGE)	84388.19	0.303	0.128	0.295	0.351	0.447	0.839
LRANGE_100 (first 100 ele- ments)	54945.05	0.464	0.28	0.463	0.527	0.687	1.383
LRANGE_300 (first 300 ele- ments)	23877.74	1.04	0.432	1.039	1.151	1.415	2.799
LRANGE_500 (first 500 ele- ments)	16380.02	1.515	0.424	1.511	1.711	1.927	4.719
LRANGE_600 (first 600 ele- ments)	14341.03	1.732	0.44	1.711	1.991	2.823	4.399
MSET (10 keys)	88339.23	0.293	0.088	0.279	0.351	0.447	1.223

Table A.7. X86 crun results on Redis benchmark

test	Request/ second	Average latency (ms)	Minimum latency (ms)	50th per- centile (ms)	95th per- centile (ms)	99th per- centile (ms)	Maximum latency (ms)
PING_INLINE	82781.46	0.309	0.104	0.303	0.367	0.447	0.951
PING_MBULK	81833.06	0.313	0.096	0.311	0.359	0.463	1.039
SET	82034.45	0.312	0.096	0.303	0.359	0.559	0.791
GET	83263.95	0.308	0.096	0.303	0.359	0.479	2.079
INCR	82850.04	0.309	0.112	0.303	0.351	0.471	1.239
LPUSH	87183.96	0.296	0.104	0.287	0.359	0.495	2.319
RPUSH	82850.04	0.309	0.096	0.303	0.359	0.471	0.983
LPOP	82101.8	0.312	0.112	0.311	0.367	0.455	0.663
RPOP	84245.99	0.306	0.096	0.295	0.359	0.471	0.751
SADD	83612.04	0.306	0.088	0.295	0.359	0.455	0.983
HSET	83194.67	0.308	0.088	0.303	0.367	0.463	0.663
SPOP	82169.27	0.311	0.112	0.311	0.359	0.455	1.007
ZADD	85034.02	0.302	0.096	0.295	0.359	0.439	0.807
ZPOPMIN	82304.52	0.311	0.12	0.303	0.367	0.463	1.063
LPUSH (needed to benchmark LRANGE)	82987.55	0.309	0.104	0.303	0.359	0.455	0.799
LRANGE_100 (first 100 ele- ments)	40453.08	0.625	0.336	0.623	0.703	0.895	3.255
LRANGE_300 (first 300 ele- ments)	15384.62	1.615	0.44	1.591	2.183	2.295	3.551
LRANGE_500 (first 500 ele- ments)	10979.36	2.253	0.44	2.239	2.623	3.159	5.527
LRANGE_600 (first 600 ele- ments)	9452.69	2.62	0.456	2.623	3.055	3.551	6.735
MSET (10 keys)	89445.44	0.298	0.096	0.287	0.359	0.495	1.159

Table A.8. X86 gVisor results on Redis benchmark

test	Request/ second	Average latency (ms)	Minimum latency (ms)	50th per- centile (ms)	95th per- centile (ms)	99th per- centile (ms)	Maximum latency (ms)
PING_INLINE	5662.83	4.627	1.664	4.031	6.663	8.183	141.567
PING_MBULK	5288.49	4.881	1.944	4.847	6.111	6.943	14.135
SET	6060.97	4.263	1.496	4.175	5.207	5.927	14.487
GET	5890.67	4.389	1.344	4.319	5.415	6.175	25.231
INCR	5641.11	4.578	0.968	4.551	5.703	6.583	9.799
LPUSH	5658.03	4.567	1.344	4.487	5.591	6.335	15.903
RPUSH	5644.62	4.579	1.328	4.487	5.567	6.399	27.935
LPOP	5643.02	4.576	1.64	4.487	5.551	6.239	11.263
RPOP	5651.63	4.571	0.84	4.495	5.575	6.359	10.903
SADD	5650.36	4.572	1.408	4.487	5.615	6.351	10.623
HSET	5492.69	4.736	1.152	4.559	5.727	6.743	138.623
SPOP	5753.08	4.491	1.608	4.439	5.551	6.455	10.847
ZADD	5639.84	4.58	1.984	4.503	5.623	6.327	10.351
ZPOPMIN	5659.63	4.571	1.32	4.487	5.623	6.423	28.079
LPUSH (needed to benchmark LRANGE)	5693.46	4.541	1.976	4.471	5.599	6.335	14.391
LRANGE_100 (first 100 ele- ments)	5039.81	5.101	1.808	4.991	6.143	6.799	12.903
LRANGE_300 (first 300 ele- ments)	3923.88	6.486	3.976	6.311	7.647	8.391	32.463
LRANGE_500 (first 500 ele- ments)	3088.61	8.236	5.264	7.279	10.575	24.319	177.663
LRANGE_600 (first 600 ele- ments)	2492.21	10.146	5.704	9.967	11.599	12.527	20.079
MSET (10 keys)	6297.63	4.114	1.184	4.007	5.087	6.047	36.351

Table A.9. X86 Kata results on Redis benchmark

test	Request/ second	Average latency (ms)	Minimum latency (ms)	50th per- centile (ms)	95th per- centile (ms)	99th per- centile (ms)	Maximum latency (ms)
PING_INLINE	54406.96	0.601	0.088	0.567	0.903	1.039	1.935
PING_MBULK	53705.69	0.58	0.144	0.543	0.879	1.023	1.855
SET	53022.27	0.607	0.12	0.519	1.023	1.439	2.655
GET	52742.62	0.516	0.2	0.471	0.719	0.943	2.015
INCR	51203.28	0.593	0.152	0.567	0.879	1.031	2.303
LPUSH	54644.81	0.587	0.192	0.487	0.975	1.351	2.631
RPUSH	54141.85	0.657	0.096	0.679	0.951	1.135	2.247
LPOP	54259.36	0.608	0.096	0.551	0.967	1.287	2.591
RPOP	53763.44	0.586	0.112	0.535	0.903	1.055	1.831
SADD	53475.94	0.514	0.104	0.463	0.799	0.943	1.727
HSET	53821.31	0.573	0.096	0.487	0.903	1.071	3.095
SPOP	54764.51	0.532	0.128	0.471	0.839	0.999	2.343
ZADD	50276.52	0.571	0.112	0.543	0.847	1.039	1.767
ZPOPMIN	53475.94	0.63	0.104	0.615	0.927	1.071	2.615
LPUSH (needed to benchmark LRANGE)	57471.27	0.544	0.096	0.463	0.919	1.287	2.423
LRANGE_100 (first 100 ele- ments)	34352.46	0.765	0.288	0.735	0.943	1.143	2.527
LRANGE_300 (first 300 ele- ments)	15372.79	1.651	0.504	1.591	2.007	2.535	8.167
LRANGE_500 (first 500 ele- ments)	11309.66	2.241	1.04	2.215	2.623	2.919	5.647
LRANGE_600 (first 600 ele- ments)	10130.69	2.493	0.88	2.447	2.935	3.335	9.687
MSET (10 keys)	46533.27	0.642	0.24	0.575	1.039	1.295	2.767