DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Resource management evaluation of container runtimes

Lennart Espe

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Resource management evaluation of container runtimes

Untersuchung des Ressourcenmanagements von Containerlaufzeitumgebungen

Author:	Lennart Espe
Supervisor:	Prof. Dr. Michael Gerndt
Advisor:	Anshul Jindal, M.Sc.
Submission Date:	31.07.2019

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 31.07.2019 Lennart Espe

Acknowledgments

I would first like to thank my supervisor, Prof. Dr. Gerndt, as well as my advisor, M.Sc. Anshul Jindal, for their help and guidance.

In addition, I thank my friends and family for their support.

Abstract

As containers are getting more prevalent in modern software engineering due to cloud computing and resource sharing, the need to analyse the performance of the different solutions for resource management on host systems increases. New frameworks are built on top of container technology that profit from the small footprint and use specific facets of an implementation as a way to enable novel feature sets like rapid startup times and minimalistic resource consumption.

This thesis analyses three different implementations of the Container Runtime Interface, namely *containerd* as the industry standard, *CRI-O* as the reference implementation and *gVisor* as a high-security alternative to the aforementioned. It dives deep into the techniques behind container and process scheduling, I/O and disk limits, and establishes an objective measurement for CRI implementation performance in reference to resource management.

Contents

Acknowledgments iii

Abstract iv

1 Introduction 1

2 Fundamentals 3
2.1 Container runtimes . . . 3
2.2 Runtime architecture and implementation . . . 4
2.3 Namespaces . . . 5
2.3.1 Namespace creation . . . 5
2.3.2 `net` . . . 6
2.3.3 `user` . . . 7
2.3.4 `mount` . . . 7
2.3.5 `pid` . . . 8
2.3.6 `uts` . . . 9
2.3.7 `cgroup` . . . 9
2.4 Control groups . . . 9
2.4.1 cpu, cpuset, cpuacct . . . 9
2.4.2 memory . . . 12
2.4.3 blkio . . . 13
2.4.4 net_cls, net_prio . . . 14
2.5 runc and OCI . . . 14
2.5.1 OCI bundles . . . 15
2.5.2 OCI images . . . 16

3 Related work 17

4 Existing implementations 18
4.1 containerd . . . 18
4.1.1 Execution model . . . 19
4.2 CRI-O . . . 19
4.2.1 Execution model . . . 19

4.2.2 The Container Runtime Interface 20
4.3 gVisor 21
4.3.1 runsc 22

5 Benchmarking tool **23**
5.1 Requirements 23
5.2 Implementation 24
5.2.1 Package structure 24
5.2.2 Machine setup 29
5.3 Issues faced and challenges 29

6 Performance and benchmarks **31**
6.1 General performance 32
6.1.1 CPU 32
6.1.2 Memory 33
6.1.3 Disk I/O 36
6.2 Container operations 38
6.3 Scalability 39
6.4 Resource limits and quotas 42
6.4.1 CPU quotas 42
6.5 Summary 44

7 Conclusion **46**

8 Future work **47**

List of Figures **48**

List of Tables **50**

Bibliography **51**

1 Introduction

Cloud computing as an infrastructure model has become increasingly popular among a wide range of industries. Container technology has played a leading role in this change. The demand for an in-depth evaluation of the performance of container-based workloads is rising. The widely-used cluster orchestrator Kubernetes supports every container runtime that implements the standardised Container Runtime Interface [18]. While the cluster administrator is allowed to freely choose between the different implementations, an objective evaluation of container runtime performance metrics in reference to resource management is missing.

Resource management is a popular tool used in computing clusters shared between multiple persons, teams or companies to enable a fair-use model that satisfies the demands of all cluster tenants. It is also used to enable resource efficient, dense deployment models like FaaS[1] and other serverless architectures. But as these new deployment models flourish, they also have specific performance requirements on the numerous facets of a container runtime. To give an example, FaaS infrastructure demands low startup times for the underlying container runtime since the time until a response to an incoming request is critical to the quality of the service.

Previous research has shown that the performance overhead of popular container runtimes is very low, hence container runtimes are more resource-efficient than traditional virtual machines [31]. This leads to container runtimes being a main driver behind energy reduction in cloud datacentres due to the lowered amount of required machines for the same amount of work. This leads to container technology being a major cost reducer in the datacenter space. But while virtual machines are more resource-intensive, they still have a reputation of being more secure than containers [2].

As container runtimes continue their rise to the top, they must also take on the guarantee of a secure computing environment. In the past there have been several security issues in the Docker engine and runc, allowing a bad actor to escape the container sandbox as a privileged user [19]. There are several container runtimes tackling these issues, first and foremost *gVisor* by Google [14] and *Kata Containers* by OpenStack [21]. Resource management is a key factor to meet these security demands since gVisor implements a user-space kernel therefore taking over major responsibilities

[1] Functions-as-a-Service, a service that abstracts away any underlying infrastructure of an application deployment.

from the operating system and Kata Containers uses the virtualisation capabilities of a modern CPU.

This thesis explores three existing container and system runtime implementations, especially *containerd*, the CRI-compatible container runtime implementation used by Docker and some distributions of Kubernetes. It has been investigated how exactly these container runtimes use kernel features and functionality like process scheduling, memory allocation and I/O interaction to satisfy the limit requirements.

Especially interesting in the context of resource limiting is the question, how exactly can resources like CPU time be limited without hurting the overall performance of an application too much.

To evaluate the performance of containers in a reliable fashion, a benchmarking toolkit has been developed. This toolkit called Touchstone tests the numerous facets of resource management in container runtimes.

The benchmarking tool measures scalability (e.g. *does the runtime in general behave differently with 1 instance versus 50 instances?*), resource overuse (e.g. *what happens if the application continuously requests more resources than defined?*) and general performance (e.g. *how fast does a single container start?*).

This Bachelor thesis focuses on learning about the resource management techniques used by container runtimes and how their implementation may shape application performance. If a container runtime has unique properties related to resource management, these differences will be highlighted and a relation to the suited scope of applications established.

Chapter 2 defines a common terminology for the rapid-expanding space of containerised computing to reduce ambiguity. It explores the OS-level components of resource management in container runtimes. The goal is to be able to differ between higher- and lower level components in a consistent fashion.

Chapter 3 discusses previous research in container runtime performance relating to I/O and memory throughput issues, resource usage and general behavioural differences.

Chapter 4 dives deep into the implementation details of the three container and system runtimes containerd, CRI-O and gVisor.

Chapter 5 examines the architecture and implementation of the benchmarking tool described above, especially challenges encountered along the way and architectural idiosyncrasies.

Chapter 6 presents the results generated by the benchmarking tool and give reasoning for the findings derived from the benchmarking data. Chapter 7 concludes the results of this thesis and chapter 8 gives an outlook on what could be examined in future research and which facets of container runtimes require more investigation.

2 Fundamentals

This chapter will define some terms specific to containerised computing. The most important concept is the Linux container, which is basically a logically isolated virtual environment. Containers are started and managed using a container runtime, which uses operating system capabilities to enable environment isolation. Additionally, the container runtime sets up the container environment on startup according to a container specification. This specification is packaged in combination with the container filesystem layers in a container image. The container image is a template for containers that can be distributed and published on the internet.

To be able to differ between software solutions like containerd and runc, the label *container runtime* will be used for software that enables running containers as a high-level task while low level OCI-compatible engines [38] will be called *system runtimes*. A major difference between container and system runtimes is the dependency on OS-specific abstractions in system runtimes while container runtimes can be ported very easily.

2.1 Container runtimes

Linux containers are often compared to virtual machines but they are fundamentally different. While in virtual machines you virtualise the hardware, containers just limit access to host resources therefore virtualising the environment instead of the system. A major difference is that containers are transparent to the host system, while virtual machines are not – at least without contextual virtual machine introspection. In addition to being transparent they are also more performant than virtual machines due to the lower virtualisation workload. The only exception to the performance declarative are solutions like Kata Containers [21], that run containerised processes in lightweight virtual machines for security reasons.

In operating systems based on the Linux kernel the isolation between containers and the host system is typically done using Namespaces and Control Groups. Namespaces isolate different processes from each other while Control Groups control access to the resources in the namespace.

2.2 Runtime architecture and implementation

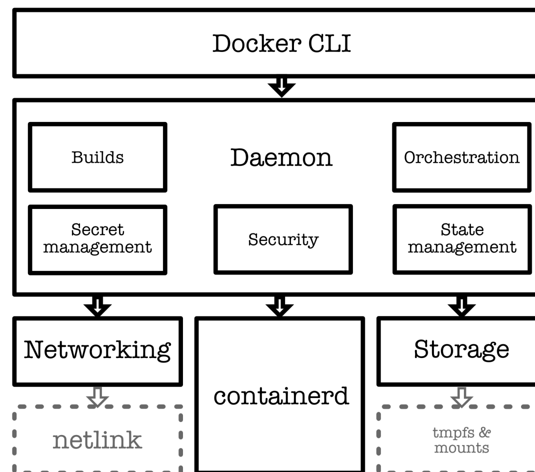

Figure 2.1: Architecture of Docker [9]

Since most of the container infrastructure tools are either written in C or Go, basic knowledge of both languages is required to fully grasp their functionality. Most toolchains use a mix of both languages, for example runc uses C in parts of libcontainer, a container management library, and wraps libcontainer in a Go binary with a command-line interface [49]. Many larger tools like the Docker engine are composed of a multitude of subsystems. Docker itself is made up of the `docker` command as the primary tool for user interaction on top, the `dockerd` daemon underneath and on the lowest level `containerd`. This decoupled architecture can be seen in 2.1.

The command line interface provides a user-friendly abstraction over the container operations happening on the lower levels of the Docker engine and directly interacts with the Docker daemon. It enables a user who does not mind about implementation specifics or peculiarities of control groups and namespaces to setup complex networks, storage bindings and resource limits. It talks to the Docker daemon using a HTTP RPC API [11].

The Docker daemon manages features such as volumes, mounts, networking endpoints and network drivers. It also provides the well-known `build` command that allows the creation of container images using a simple build script, the so-called Dockerfile. The build command uses image snapshotting to generate images layers and executes each Dockerfile command in its own container. The orchestration component of Docker is called Swarm, a distributed cluster orchestrator build on top of the Raft consensus algorithm [12]. Each Docker instance supports this special swarm mode

that enables multiple computing nodes to behave like a single Docker node similar to Kubernetes. Docker also supports security features like content trust signature verification to limit the runtime to running signed images. This makes it possible to monitor the distribution chain from the creator of a container image to the runtime, a critical feature for privacy- or security conscious users. It also provides utilities like log management and container state management, e.g., pause and resume [11].

All lower-level container management is delegated by the Docker daemon to the container runtime containerd, that was originally built as an internal subsystem of Docker and moved into a separate project later on [9].

2.3 Namespaces

Namespaces are the fundamental Linux feature that enable the functionality and separation that containers provide. They can be created using the `clone` and `unshare` system calls. There are different types of namespaces for specific groups of resources or configuration, like filesystem mounts, user mappings, networking or resource limiting [35].

2.3.1 Namespace creation

The `fork` system call creates an exact copy of the calling process. The child process has a different process ID, parent process ID and own copies of the open file descriptors. On first sight the `clone` system call is very similar to `fork`. But instead of creating a new execution context for the forked process, cloned processes keep a share of its parent execution context. To give an example, the parent process of a cloned child can access the child's stack. The developer can also specify which parts of the parents context should be shared. Typically, `clone` is used for multi-threaded applications [7].

The `unshare` system call allows to *unshare* parts of the current execution contexts with other process, e.g. changing from a common user namespace to a separated one [51]. The `setns` system call is the counterpart to `unshare`. It allows a process to join an existing namespace [42].

The calling order of the namespace operations is very relevant for the functionality of the container subsystem. For example, a process does not have privileges to administer a network namespace that has been created while residing in a different user namespace. Typically, the process initialisation is therefore split into multiple steps. As a first step, the user and group mappings for the container process are created. In the second step the user namespace is unshared and in a third step all other namespaces are unshared. In the last step, configuration steps like mounts, networking and routing are taken care of [49].

```
// Clone process with new namespaces
const int flags = CLONE_NEWNS
                | CLONE_NEWCGROUP
                | CLONE_NEWPID
                | CLONE_NEWUSER
                | CLONE_NEWIPC
                | CLONE_NEWNET
                | CLONE_NEWUTS;
if ((cfg.pid = clone(exec,
       malloc(STACK_SIZE) + STACK_SIZE,
       SIGCHLD|flags, &cfg)) < 0) {
       returrn -1;
}
```

Figure 2.2: Creating a new thread in a different namespace

2.3.2 `net`

Network namespaces provide isolation between groups of network devices, routing tables, protocol stacks and so on. Physical network devices can be bound to exactly one network namespace. Virtual network devices can be used to connect two network namespaces as a tunnel or as a bridge to a physical network device [36].

These virtual devices can be created by hand using the `ip` tool in Linux or by directly communicating with the Kernel using the `netlink` system interface. The typical procedure is to create all networking components beforehand and link in all endpoints after the container is up and running. Inter-container networking itself is a very complex topic and often handled separately from low level runtime environments like *runc*. A basic networking setup can be seen in figure 2.3. The basic setup can be configured by creating a virtual ethernet pair and assigning one of them to the container network namespace. After adding fitting routes via `iptables` and setting up network-address translation for incoming packets on the host system, the container is able to communicate with the outside world. In some cases it is useful to have multiple containers join a common network namespace, especially when they are tightly coupled like a web server and a sidecar, that grabs metrics from this web server. When joining a common network namespace, these containers share the `loopback` interface and are able to communicate with each other using `localhost`.

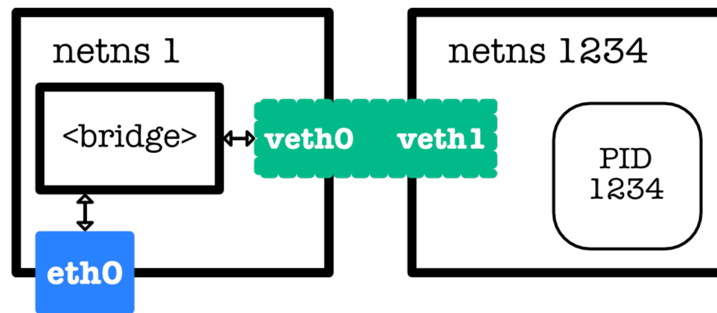

Figure 2.3: Usage of network namespaces to link container networks

2.3.3 `user`

User namespaces provide isolation between security groups. They can be nested and are used to give a non-privileged user specific privileges inside the given execution context. They are often paired with user and group ID mappings. This feature is especially useful when securing a process, which the system user may want to run as non-root due to security guidelines [52].

2.3.4 `mount`

Mount namespaces isolate the mount points of a process filesystem from the rest of the system. This is very useful when mounting the rootfs of a container or binding specific directories to a different path [34]. Mount namespaces are often used in combination with the system calls `chroot` and `pivot_root`.

The Linux system call `chroot` changes the root directory of the active process. It has been deemed insecure due to the superuser being able to escape a chroot restriction very easily [6].

An alternative to `chroot` is the system call `pivot_root`. Instead of changing the active root directory, the current root directory is *pivoted*. It is moved to a different directory and a child node of the old root is used as the new root. The old root directory is then unmounted. This technique is more complex and requires a variety of file system interactions as seen in figure 2.4. In the context of container runtimes `pivot_root` is preferred because it is non-invertible from inside the container and cannot be escaped without further system exploitation. Historically `pivot_root`'s main use has been for booting a Linux system using `initrd` [40].

```
// Create temporary rootfs
char rootfs[] = "/tmp/rootfs";
mkdir(rootfs);
mount("my-rootfs", rootfs, NULL, MS_BIND | MS_PRIVATE, NULL);
// Move current root to oldroot, make rootfs new fs root
char inner_rootfs[] = "/tmp/rootfs/oldroot";
mkdir(inner_rootfs);
pivot_root(rootfs, inner_rootfs);
// Set process root to fs root
chdir("/");
// Remove old root
char old_root[] = "/oldroot";
umount2(old_root, MNT_DETACH);
rmdir(old_root);
```

Figure 2.4: Performing `pivot_root` for process root isolation

```
// Mount procfs
if (mount("proc", "/proc", "proc",
    MS_NOEXEC | MS_NOSUID | MS_NODEV, NULL))
    panic("failed␣to␣mount␣/proc");
```

Figure 2.5: Mounting `proc` in a container

2.3.5 `pid`

A PID namespace allocates a local process hierarchy. The initial process in a process hierarchy always has the ID 1. In the case of `clone`, the cloned child has the PID 1. When using `unshare`, the first child created after calling unshare has PID 1. When the initial process with PID 1 terminates, all processes in the PID namespace are terminated as well [39].

When using runc, the `/proc` directory is re-mounted such that the container process can not see any running process outside of its namespace as demonstrated in listing 2.5 [49].

2.3.6 `uts`

A UTS namespace allows to set a separated host and domain name. These changes to the system identifiers do not change any identifiers in the host system. Container-specific hostnames can be very useful for service discovery and networking [35].

2.3.7 `cgroup`

Control Groups namespaces exist to provide a namespace-specific abstraction over the control groups system interface. They provide an own isolated subtree in the control groups hierarchy such that no containerised process may modify or access a control group that resides outside of its scope [4]. Control Groups are thoroughly discussed in the following subsection.

2.4 Control groups

This section explores control groups, the Linux kernel feature that enables important container capabilities related to resource usage and limits. There are currently two implementations of the control group system called `cgroup` and `cgroup2` [5]. This thesis is limited to `cgroup` since `cgroup2` is not used by `runc`. The main difference between version 1 and 2 is the unified file system hierarchy. While in version 1 each subsystem has its own hierarchy, in version 2 all controllers of a given control group sit in the same directory. This facilitates better control of operations that are performed by multiple subsystem [24, 5].

While using `cgroup` an indirect disk write, that is cached in memory, may be written to disk with significantly higher speeds than the `blkio` control group permits to, the unified hierarchy in `cgroup2` allows to account resources across controllers. In this case the indirect write can be limited by the `blkio` controller.

When using systemd `cgroup` may be accessed using the virtual filesystem mounted by default in `/sys/fs/cgroup`.

2.4.1 cpu, cpuset, cpuacct

The `cpu` controller is used to define the relative shares and ceiling time to be considered by the CFS[1] when scheduling processes that are assigned to this control group. Tasks can be added and removed dynamically. Furthermore the controller supports changes to the control groups limits and properties while a process is assigned either directly

[1] Completely Fair Scheduler, the default process scheduler in the Linux kernel.

```
# Set interval to 100ms
echo 100000 > /sys/fs/cgroup/cpu/lowprio/cpu.cfs_period_us
# Set max share to 1ms
echo 1000 > /sys/fs/cgroup/cpu/lowprio/cpu.cfs_quota_us
# Get task stats
cat /sys/fs/cgroup/cpu/lowprio/cpu.stats
nr_periods 5906
nr_throttled 4321
throttled_time 643372847465
```

Figure 2.6: `cgroup/cpu` ceiling enforcement

by a `tasks` entry or indirectly by assignment of a parent [5]. The dynamic assignment of resources facilitates more flexibility when scheduling containers.

Relative shares allow to perform weighted scheduling on groups of processes. For example, an administration tool may create a control group for each user of a shared system with `cpu.shares` set to 1024 and assign all processes run by this user to the corresponding control group. Therefore the tool can ensure that each user has the same quality of service while using the system. If a privileged user would exist, the tool may decide to assign the doubled amount to the user's control group such that the privileged user can use twice as much CPU resources as an unprivileged user [23].

In contrast to relative shares ceiling enforcement is performed using CPU time measured in microseconds. This ceiling is set by writing to `cpu.cfs_quota_us` and `cpu.cfs_period_us` [22].

If an administrator tool runs a low-priority batch job, it may decide to limit the maximum amount of CPU time available to this process. By writing 100000 to `cpu.cfs_period_us` and 1000 to `cpu.cfs_quota_us`, the batch process gets to run at maximum one millisecond each $100ms$. Ceiling enforcement is one of the most important tools in shared environments such that a single user may not use all of the computing power of a workstation. Using `cpu.stat` the administrator tool may analyze how often and how much the process group has been throttled as seen in figure 2.6.

As discussed before, the number of shares is specified in relative values. This is demonstrated in figure 2.7 where the minimum number of shares is specified in relative metrics. The shown assignment leads to tasks in group A being assigned only half of CPU shares relative to tasks in group B.

The `cpuset` controller allows to assign processes to specific CPU and memory nodes. This is especially useful for very large NUMA machines with hundreds of CPU and memory nodes. In Non-Uniform Memory Access machines each CPU has its own

```
# in cgroup A
echo 1024 > /sys/fs/cgroup/cpu/mygroupA/cpu.shares
# in cgroup B
echo 2048 > /sys/fs/cgroup/cpu/mygroupB/cpu.shares
```

Figure 2.7: cgroup/cpu share assignment

```
# Assign cgroup 'singlecore' to a single CPU and memory node
echo 0 > /sys/fs/cgroup/cpusets/singlecore/cpuset.cpus
echo 0 > /sys/fs/cgroup/cpusets/singlecore/cpuset.mems
```

Figure 2.8: cgroup/cpusets node assignment

memory node while being while being able to access the memory nodes of other CPU nodes. To make computations on these architectures faster, node affinity is extraordinarily important. cpusets can make a process run on a compute node with a local memory node. It also includes the possibility to assign a CPU or memory node exclusively, thus no other sibling cpuset may assign an overlapping subset [26].

Exclusive resource assignment is required if the system user wants to give strong performance guarantees to the process user. This comes in handy when managing multiple compute-bound processes, e.g. in high-performance computing. When running low-profile batch jobs, an administration tool may want to run these processes only on an exclusive CPU node and can perform this task using cpusets as seen in figure 2.8.

The cpuacct controller allows to group processes together and analyse their total CPU consumption. This makes usage inspection of specific process groups possible. All values reported by cpuacct are measured in nanoseconds as illustrated in figure 2.9 [25].

```
# retrieve cpuacct.usage
cat /sys/fs/cgroup/cpuacct/mygroupA/cpuacct.usage
# >> 129370653
# >> about 0.129 seconds
```

Figure 2.9: cgroup/cpuacct usage stats

```
# Set max memory usage to 128 MB
echo 128M > /sys/fs/cgroup/memory/smallmem/memory.limit_in_bytes
# Read memory limit
cat /sys/fs/cgroup/memory/smallmem/memory.usage_in_bytes
2949120
# About 2.297 MiB
```

Figure 2.10: `cgroup/memory` memory limits

2.4.2 memory

The `memory` controller provides management access to the systems memory resources. It allows to set limits for `memory` and `memory+swap` resources. Moreover, it makes the extraction of limit statistics possible, e.g. how often memory limits have been hit. These limits are set similar to the `cpu` controller using a virtual file system as illustrated in figure 2.10. The memory controller can be very useful to

- limit memory-hungry applications from hogging to much resources
- limit the amount of memory available to each user on a time-sharing system

The controller also supports the usage of soft limits. Normally a process that is only limited by a hard memory limit may consume as much memory as it wants up to the hard limit. Soft limits enable the sharing of memory up to the point when memory resources become scarce and the processes are limited to their soft limit. The enforcement of soft memory limits takes a long period of time due to the slower reclaiming of memory regions [27].

The core of the memory controller is based around counting pages allocated on a per cgroup basis. Per cgroup only anonymous pages[2] and cache pages are accounted for. Anonymous pages are counted when they are swapped in and unaccounted when they are unmapped. Cache pages are unaccounted when they are removed from the internal address space tree. Shared pages are accounted to the cgroup that first touched the shared page.

The memory controller also allows to administer the out-of-memory management system for the group of processes. It is possible to deactivate the OOM killer entirely. After deactivation of the OOM killer, processes that would normally be killed because they requested memory that cannot be allocated for them, cannot use `malloc` anymore until they have freed allocated pages.

[2] Anonymous pages are virtual memory mappings without any associated file.

```
# Limit sda throughput to 1 MB/s read/write
echo "8:0␣1048576" > .../blkio/mygroupA/blkio.throttle.write_bps_device
echo "8:0␣1048576" > .../blkio/mygroupA/blkio.throttle.read_bps_device
```

Figure 2.11: `cgroup/blkio` throttling

```
# Give group B double the allocated IOPS
echo 500 > .../blkio/mygroupA/blkio.weights
echo 1000 > .../blkio/mygroupB/blkio.weights
```

Figure 2.12: `cgroup/blkio` weighting

Like the `cpu` controller, the memory controller supports on-the-fly reassignment of limits.

2.4.3 blkio

The `blkio` controller manages I/O control policies in the kernel storage tree. It supports two different types of controlling I/O. On the one hand there is the throttling mode that limits the number of bytes or operations that can be performed on a specific device per second. On the other hand there is the weighted mode, that modifies the scheduling of the I/O operations similar to the `cpu.shares` in the `cpu` controller. I/O upper limits can be specified in bytes per second (`bps`) and I/O operations per second (`iops`) as seen in figure 2.11. If both limits are set, they are both considered by the I/O subsystem when determining the limits on a per-cgroup basis.

By default throttling operates on a flat tree, such that no control group respects I/O limits of a parent group. While throttling supports tree hierarchies, the feature has to be activated using the `sane_behavior` flag. This flag has not been available to the public outside of a Linux kernel development environment.

At the moment, throttling has its limits especially in respect to writing files. Because it is not possible to associate indirect write operations on files with the `blkio` control group it originated from (due to `memory` and `blkio` control groups operating independently), throttling for writes only works for direct writes as specified by `O_DIRECT` flag. Throttling for reads works on direct and indirect read operations.

The other supported control option is proportional I/O operation weighting. Linux schedules disk I/O operations using the Complete Fair Queuing-Scheduler. The scheduling weights used for process groups can be redistributed using the `blkio` control group as seen in figure 2.12. The weight can also be specified per I/O device by

using the blkio.device_weight parameter.

2.4.4 net_cls, net_prio

```
mkdir /sys/fs/cgroup/net_cls/eduroam
/bin/echo 0x100001 > /sys/fs/cgroup/net_cls/eduroam/net_cls.classid
tc qdisc add dev eth0 root handle 10: htb
tc class add dev eth0 parent 10: classid 10:1 htb rate 100kbit
tc filter add dev eth0 parent 10: protocol ip prio 10 handle 1: cgroup
```

Figure 2.13: cgroup/net_cls traffic shaping

net_cls allows to tag network packets that are routed from or to a specific process in the control group. This feature can be used in combination with the traffic control tool tc to shape the amount of traffic that can be send and received by processes in this control group. Other advanced actions can be taken by the iptable tool using the created network packets tag. For example, using the aforementioned tools an administrator may limit the upstream of a process to 100 KBit per second as demonstrated in figure 2.13 [28].

net_prio provides an interface to prioritise network traffic by using network interface handles. The controller facilitates setting the default network interface used by processes in this control group. The prioritisation can be overridden by the process itself via the SO_PRIORITY socket option [29].

2.5 runc and OCI

runc is the lowest runtime layer of containerd that explicitly handles containers. It was used as the reference implementation when drafting the OCI runtime specification [38]. Internally, runc uses libcontainer to interact with OS-level components.

The Open Containers Initiative has defined two standards, the *image-spec* for OCI images and the *runtime-spec* for system runtimes [38, 37]. The typical job sequence would be that a container runtime downloads such as containerd downloads an OCI image, unpacks it and prepares an OCI bundle, a container specification including the root filesystem, on the local disk. Then a system runtime like runc is able to create a running instance from this container specification. OCI images can be created using a number of tools, for example the famous docker build command or various standalone tools like kaniko, a replacement for the docker build command that does

not require privileged system access [20]. After a successful build, they are usually pushed and published to a public or private container registry.

2.5.1 OCI bundles

An OCI bundle consists of two major components, a `config.json` at the root level and the `rootfs` container filesystem. The `config.json` specifies [38]

- the required OCI version (`.ociVersion`)
- user configuration and mappings (`.process.user`)
- environment variables (`.process.env`) and working directory (`.process.cwd`)
- process capabilities (`.process.capabilities`) and Linux security features like
  - SECCOMP (`.linux.seccomp`)
  - SELinux (`.process.selinuxLabel`)
  - AppArmor (`.process.apparmorProfile`)
- root filesystem path (`root.path`) and mounts (`.mounts`)
- resource management (`.linux.resources`)

The runtime user has also the option to supply a range of POSIX-compatible hooks for pre-start, post-start and post-stop. Beside being able to parse the `config.json`, an OCI runtime has to implement among other commands the following operations [38].

- `create`, to create a new container from an OCI bundle
- `kill`, to signal a container to stop
- `delete`, to delete a stopped containers
- `state`, to retrieve the state of a container

Each of these operations is directly invoked by executing the runtime binary with specified parameters, a major difference to higher-level runtimes that mostly either use HTTP REST endpoints or gRPC in a client-server architecture.

```
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
  "config": {
    "mediaType": "application/vnd.docker.container.image.v1+json",
    "size": 3410,
    "digest": "sha256:4c108a37...
  },
  "layers": [
    {
      "mediaType": "application/vnd.docker.image.rootfs.diff.tar.gzip",
      "size": 26684508,
      "digest": "sha256:5b733921...
    },
    ...
```


Figure 2.14: `ubuntu:latest` OCI image manifest

2.5.2 OCI images

OCI images on the other hand consist of multiple filesystem layers that are layered onto each other using an overlay or snapshot filesystem. The main components of an OCI image is the manifest, storing basic information like intermediate layers and image size, and the layer manifest. Each image manifest is platform and operating-system specific. This is the reason why there often are different manifests for the same image source (for example one for `windows` and one for `linux`).

In addition to the image manifests, the OCI image specification also specifies the OCI image layout, a directory structure for storing OCI image manifests. It is itself described by an `index.json`, that specifies among other things the mapping between system platform and image manifest [37].

3 Related work

There has been some research related to the runtime overhead of Docker in the context of resource-critical mobile edge computing. Avino et al. have found that the CPU usage of the Docker process is constant regardless of the CPU cycle consumption of the containerized process [1].

Casalicchio et al. have benchmarked the Docker Engine in regards to different implementation details, mainly CPU- and I/O intensive workloads. They have found that monitoring disk I/O performance in Docker can be rather complicated since there are no tools available. An interesting fact they have discovered is that when the amount of requested CPU cycles by the container is over 80% the overhead of the Docker Engine shrinks from 10% down to 5%, concluding that the overhead of the Docker engine does not grow linearly with each container instance [3].

Kunal Kushwaha from the NTT Open-Source Center has looked at the performance of VM-based and OS-level container runtimes such as runc, kata-containers and higher level runtimes such as containerd and CRI-O. In his analysis he claims that containerd performs better in comparison to CRI-O and Docker due to the different file system driver interface design. One of his discoveries was the very low container startup latency of CRI-O in comparison to containerd. In all other test cases CRI-O performed worse than containerd, but starting a container using CRI-O was 5 times faster. This low startup latency also transferred to the kata-containers test environment, where CRI-O was 3 times as fast as containerd [32]. Kushwaha mainly focused on operations performance and runtime overhead, while this thesis also explores benchmarking of the scalability and resource management capabilities of container runtimes. He also compares containerd with Docker, which will not be done in this thesis due to the different abstraction layers these tools operate on.

4 Existing implementations

This chapter explores three different runtimes for containers, namely containerd, CRI-O and gVisor. containerd and CRI-O operate on the abstraction level of the Container Runtime Interface and are categorized according to the terminology defined in chapter 2 as container runtimes. gVisor is a system runtime that can be plugged in as a replacement for runc, which is used by both containerd and CRI-O.

4.1 containerd

containerd is the default runtime used by the Docker engine [9]. It is often called the industry-standard because of its high level of adoption. containerd runs on top of `runc`, the reference implementation of the OCI runtime specification [38] that runs containers specified in the OCI image format [37].

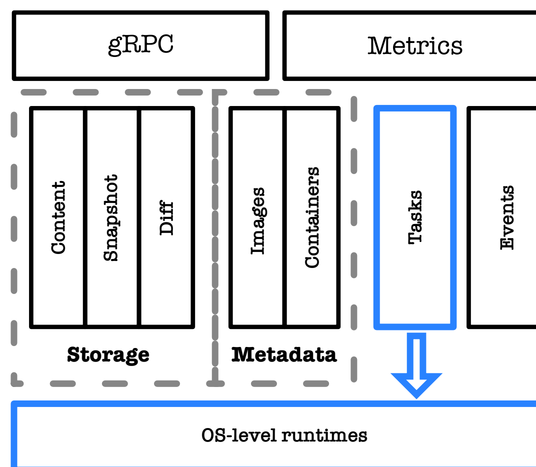

Figure 4.1: Architecture of containerd [43]

The architecture of containerd, which can be seen in figure 4.1, is rather complex. In the following section the intersection between the operating system and the high-level runtime will be discussed.

4.1.1 Execution model

containerd is a high-level container runtime. It is responsible for storing and managing images as well as snapshots and starting and stopping containers by delegating execution tasks to the system runtime.

The daemon itself works in a very linear fashion, working through the dependency graph of a task and finally running it using the given system runtime [46]. To start a new containerised process, containerd has to

1. create a new `Container` from a given OCI image
2. create a new `Task` in the `Container` context
3. start the `Task`, at this point runc takes over and starts executing the OCI bundle supplied by containerd

containerd also provides a CRI-compatible gRPC endpoint that is enabled by default. This was originally an out-of-tree plugin, but later merged into the main project. When using this endpoint abstractions specific to containerd are hidden from the client and the user can operate directly on CRI-specific abstractions.

4.2 CRI-O

CRI-O is a container runtime built to provide a bridge between OCI runtimes and the high-level Container Runtime Interface developed by Kubernetes. It is based on an older version of the Docker architecture that was built around graph drivers, which will be discussed in the execution model section [53].

It is mainly developed by RedHat and the default runtime for OpenShift, a popular Kubernetes distribution among enterprises [33].

4.2.1 Execution model

The project scope of CRI-O is limited to satisfy the CRI, therefore the container runtime does not provide tools for non-programmatic user interaction nor does it support building and publishing OCI images [47]. There exist a few tools like `crictl` that make direct interaction over the command line possible but these were crafted for testing purposes rather than real-world use due to the direct utilisation of CRI abstractions.

Implementing any of the aforementioned concepts like image publishings would lead away from the main goal to provide Kubernetes with an interface to interact as directly as possible with OCI containers [47].

As a consequence the typical lifecycle interaction with CRI-O is very similar to using `containerd` over the CRI endpoint. Major differences in internal container handling to containerd do not exist since runc is the default system runtime when running CRI-O.

As mentioned above CRI-O is built similar to an older design of the Docker runtime using graph drivers, an out-of-tree storage plugin system. The concept of graph drivers was born out of need to port Docker to other Linux distributions than Ubuntu. At the time this architecture decision was made, Ubuntu was the only Linux distribution that shipped with `aufs`, an overlay filesystem used by Docker. Graph drivers provide an abstraction over the creation, deletion and other operations on filesystem layers. It is described by the core maintainer of Docker, Michael Crosby, as "unnecessarily complex" and "hard to maintain". Today Docker and containerd use a simpler storage model built on top of snapshot filesystems that operate on filesystem blocks instead of layers [10].

4.2.2 The Container Runtime Interface

The Container Runtime Interface is a standardised interface for Kubernetes plugins that execute and watch over containers. It was created as an attempt to stabilise the communication interface between a `kubelet` and the host container runtime while not abstracting too much of the underlying execution layer. It is based on gRPC, a cross-language library for remote procedure calls using Protocol Buffers [41]. In contrast to the Kubernetes ecosystem CRI uses an imperative interface with the aim of higher development velocity for maintainers of CRI-compatible runtime systems [18].

The interface makes no assumption about the low-level implementation of containers as long as the monitoring, logging, image and lifecycle management capabilities are provided. This makes is possible to implement plugins that do not use typical Linux (or Windows) containers, but also virtual machines with hypervisors.

The gRPC interface consists of two major services. The Runtime service is responsible for running containers in pod sandboxes and direct interaction like starting processes in these containers, port forwarding, shell access and metrics extraction [48]. In the context of an application a pod sandbox is similar to an application-specific logical host. Containers in a shared pod sandbox can directly interact with each other using `localhost`.

The Image service main responsibility is to download images from a remote source and manage them locally. It also provides information about the filesystem that is used to store images. The typical sequence of actions when running a container using CRI consists of

1. pulling the image from a registry (`ImageService.PullImage`)

2. creating a pod sandbox (`RuntimeService.RunPodSandbox`)
3. creating a container in the sandbox (`RuntimeService.CreateContainer`)
4. starting the container (`RuntimeService.StartContainer`)
5. waiting for it to finish and remove the container (`RuntimeService.RemoveContainer`)
6. removing the sandbox (`RuntimeService.RemovePodSandbox`)

Every CRI remote procedure call is executed synchronously and blocks until the action is executed [18].

4.3 gVisor

Typical runtimes like `runc` limit system access using different capability models integrated into the Linux kernel like AppArmor, SELinux and SECCOMP as seen in figure 4.4. If a bad actor has access to a limited set of system calls and manages to exploit a vulnerability in the Linux kernel, it may be able to escape the sandbox and gain privileged system access. This is a major risk when running untrusted code like a provider of FaaS infrastructure does. When running a container in a VMM-based runtime like Kata Containers, a hypervisor will be started that virtualises the underlying system hardware the container is running on as seen in figure 4.2 [21].

This approach has a very large overhead due to the resources required to provide a fully virtualised set of hardware. In this setup, system calls travel from the application to the guest kernel. The major positive effect is the additional obstacle to escape the virtual machine to be able to wreak havoc on the host system.

The gVisor approach to container isolation can be classified as middle ground between a typical container runtime and a fully virtualised runtime built on top of a VM hypervisor. The developers of gVisor have taken the idea of a guest kernel from the VMM-based runtimes and implemented a user-space kernel called *Sentry*, that is instantiated once for every OCI pod sandbox, and *Gofer*, which manages container filesystem access and is instantiated once for every OCI container in a pod sandbox. This approach is a middle ground between the default control groups / namespaces approach and the VMM model as seen in figure 4.3. To reduce overhead, it does not virtualise the system hardware and leaves scheduling and memory management to the host kernel. This split of responsibilities impacts performance on applications loaded with system calls negatively and also leads to possible incompatibilities between some workloads and gVisor due to specific Linux kernel behaviour or unimplemented Linux system calls [14].

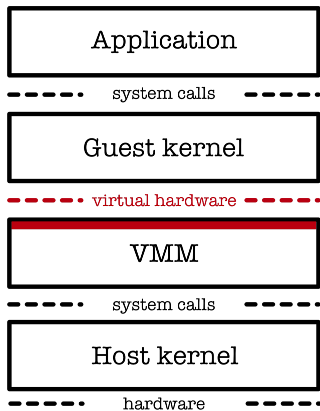

Figure 4.2: VMM [14]

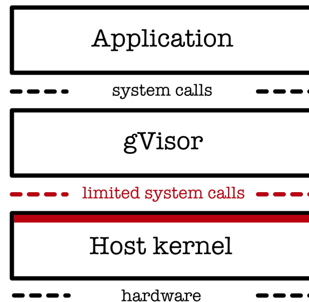

Figure 4.3: gVisor [14]

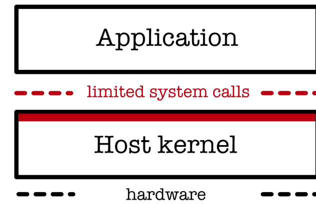

Figure 4.4: Default [14]

4.3.1 runsc

gVisor is implemented as an OCI-compatible runtime that can be easily plugged into well-known container tools like containerd, Docker and CRI-O. Using a containerd plugin it is also possible to switch between the default runtime and gVisor using annotations. This allows for running normal and highly-secure workloads side-by-side in a single containerd instance [14]. runsc, the gvisor-containerd-shim, Sentry and Gofer are completely implemented in Go and support at the time of writing this thesis most Linux system calls, except POSIX and System V message queues, signal file descriptors and some minor system calls. The user-space kernel implementation aims to be compatible with Linux 4.4. To control resource limits, gVisor utilises the host `cgroup` subsystem. This may influence the performance of the system runtime due to performing Kernel checks under the control group settings like CPU time [16].

At the time of writing gVisor did not support container-scoped resource limits and dynamic resource limits updates [15].

5 Benchmarking tool

This chapter describes the development of the benchmarking tool called *Touchstone*. The name refers to small stones that have been used in the past to determine the quality of gold. The tool has been published under the MIT license on GitHub [50].

The tool will be used in the next chapter to extract various performance benchmarks related to container runtimes from the testing system.

5.1 Requirements

As described in chapter 1, the main motivation behind the development of Touchstone was to improve container runtime evaluation. The tool should extract meaningful benchmarking data during one or multiple runs and export it in an easy-to-use format. During execution there are multiple facets to test, including CPU, memory and block I/O performance as well as general interface performance and behaviour under load. All of these facets have to be regarded by the tool to give a broad classification of runtime components. Additionally, the tool has to be as objective as possible regarding the specifics of any container runtime.

Furthermore, it should be easy to use new CRI runtime and OCI handler implementations in a plug-and-play style. Enabling these components should be performed either via a minor change in the code base or preferably by using a slightly tweaked configuration.

The configuration and user experience of the tool should feel familiar for anyone who knows other tooling from the container ecosystem. Configuration should be done using one or multiple YAML files that declare which tests to run, how often to run them and the scaling magnitude – e.g., how many containers should be run at the same time.

The internal testing framework should be able to start and stop containers as well as extract metrics and logs during a run. Specific tests like performance evaluations could be run inside the container using existing tools like sysbench [30].

The benchmark output should be easily readable and visualisable. Auto-generated graphics should help to extract meaningful predicates from the performance data.

```yaml
cri: ["containerd", "crio"]
oci: ["runc", "runsc"]
filter:
- performance
runs: 100
output: performance.json
```

Figure 5.1: Configuration YAML for performance testing

5.2 Implementation

Since most of the software in the container space has been written in Go, it made sense to implement Touchstone in Go due to its good integration with gRPC. Another benefit of implementing the tool in Go was the existing set of Go open-source tools that use the Container Runtime Interface. These existing projects were used as an inspiration and reference during implementation. For example, `crictl` from the `cri-tools` Kubernetes Incubator project implements a command line tool for interacting with runtimes using the CRI [8].

The implementation of the testing framework itself is relative simple and rather stripped-down. Benchmarks are run in a testing matrix that tests each configuration pair consisting of container runtime (CRI runtime) and system runtime (OCI handler) once. The tests are identified by a hierarchical naming scheme consisting of `context.component.facet` – like `performance.cpu.total` for a CPU performance benchmark. Tests can be filtered by prefix before being fed to the benchmark matrix. All of the configuration listed above is done using a YAML file similar to the one in figure 5.1. Each of these configuration files defines an output file path. The output of each configuration run is a JSON file containing all data generated from the benchmark executions. The user may choose one or more of these configuration files, each of them is run separately. This suits the concept of benchmarking suites, where running a folder generates a list of output files, each testing a specific facet of a container runtime implementation. After all benchmarks have been run, an index of the test results is generated and injected together with the results into an HTML file.

5.2.1 Package structure

`pkg/cmd`

The package `cmd` contains all code regarding the command line interface. Touchstone only provides four commands to the enduser – `version`, `command`, `list` and `index`. The

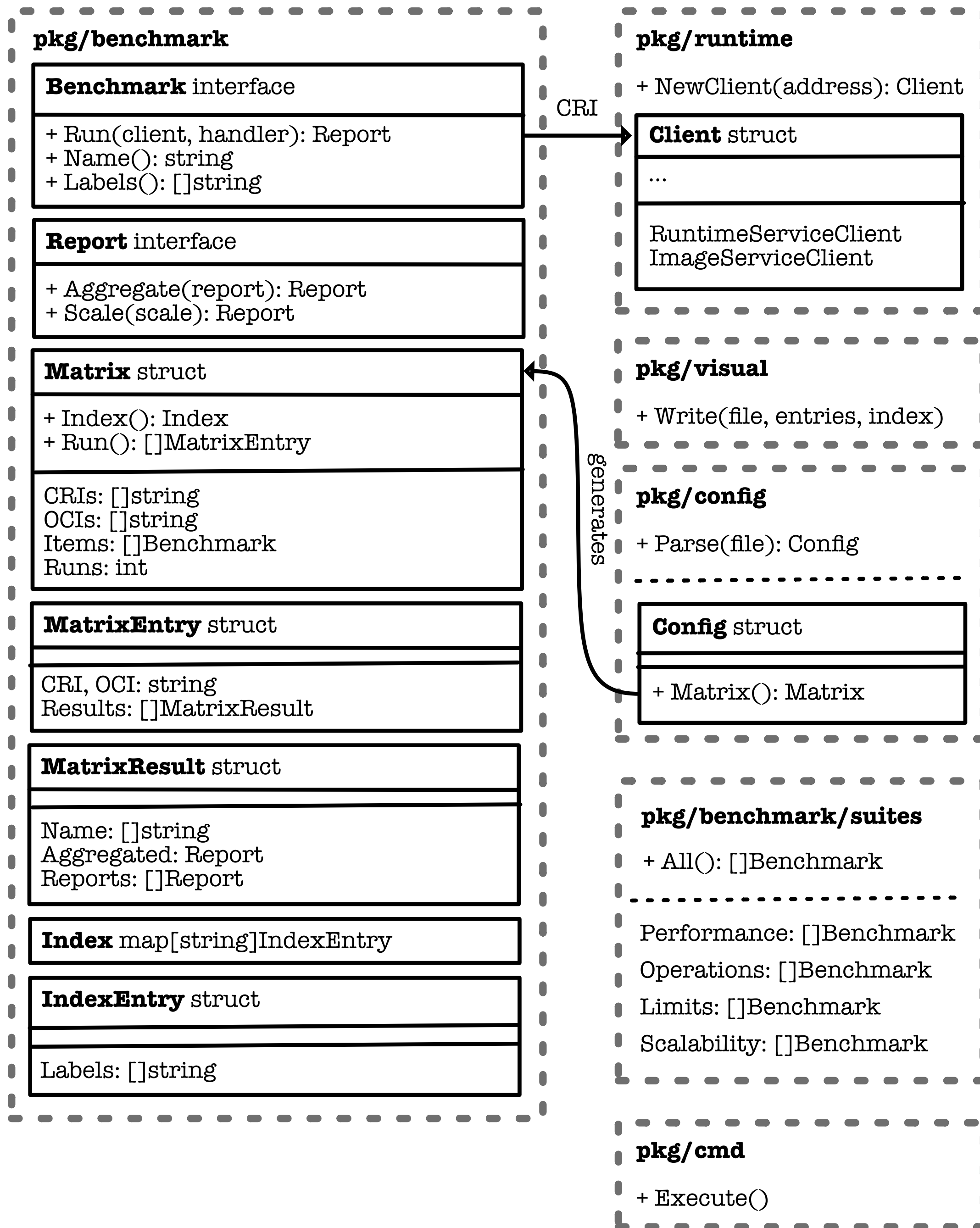

Figure 5.2: Touchstone package overview

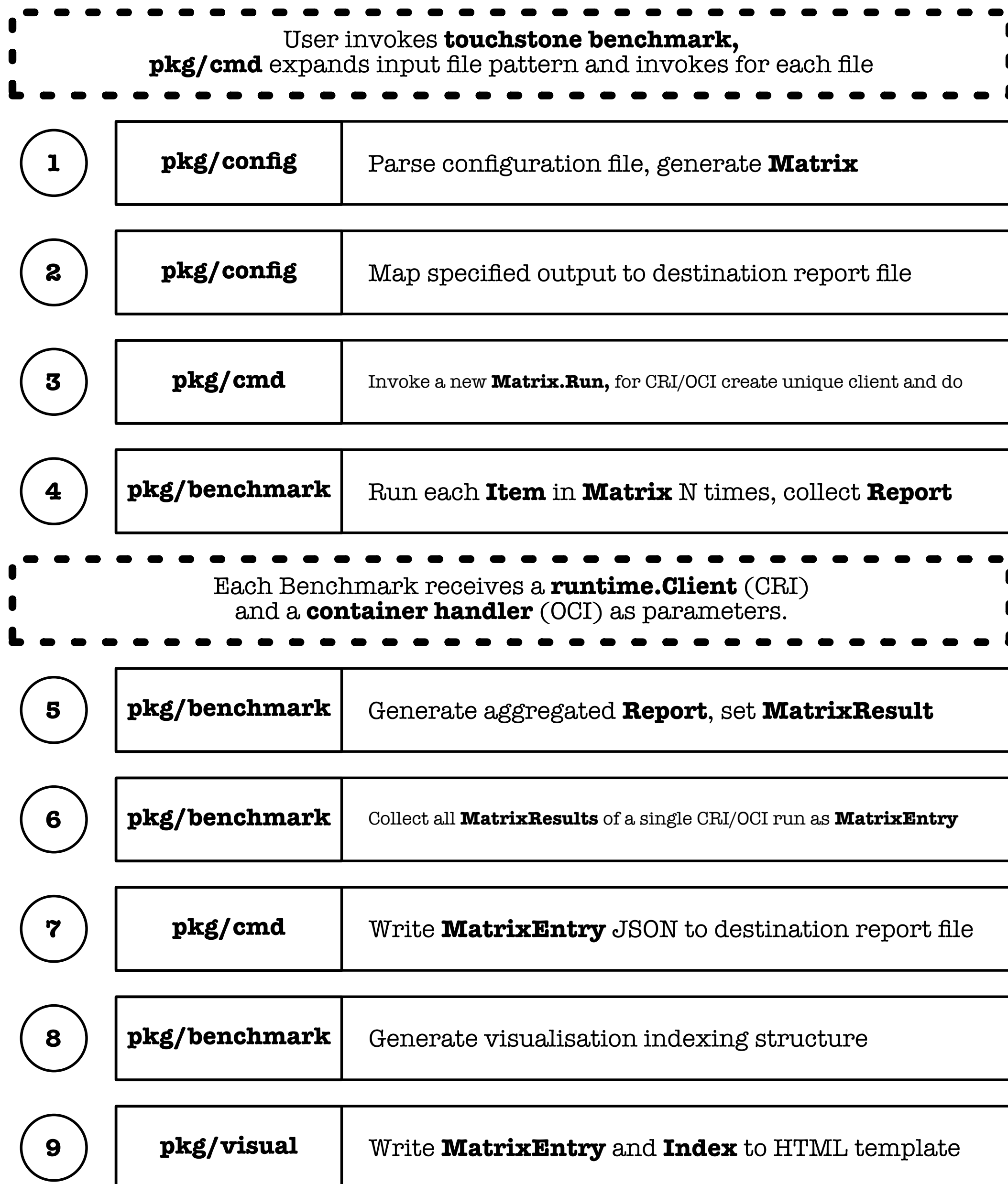

Figure 5.3: Touchstone control flow

implementation of the user interface is based on the popular cobra framework used by Docker, Kubernetes and many more [13]. The `version` command returns the version of the tool itself and all CRI implementations it was able to talk to. The `benchmark` command runs the benchmarks specified by the `-f` input files and spits out the results into the `-d` destination folder. It also generates a visualisation webpage and stores it in the destination folder. The `list` command lists all available benchmarking suites while the `index` command generates an index of a list of benchmarking suites. It can be used to preview the results generated by the `benchmark` command.

`pkg/config`

The `pkg/config` packages implements a YAML-parsable `Config` structure. Each configuration defines an output file to write the benchmark results to, a list of filters specifying which benchmarks to run, basic benchmark parameters like runs per benchmark and benchmark scale as well as which container (CRI) and system (OCI) runtime to use. Each CRI name is automatically expanded to a canonical UNIX socket endpoint using the pattern `unix:///var/run/[CRI]/[CRI].sock`. Both CRI-O and containerd use socket endpoints of that form. In the case of a user wanting to include a new runtime in the benchmarks, the runtime has to expose this endpoint either directly or via a symbolic link.

`pkg/benchmark`

The `benchmark` package provides a simple benchmarking framework. It defines the types `Matrix`, `Benchmark` and `Report` as well as `Index` and some helper structures. The heavyweight in this package is the `Matrix`, which specifies similar to a build matrix a list of targets to benchmark. A `Matrix` has a slice of `Benchmarks` that are invoked for each combination of CRI runtime and OCI handler. For each CRI runtime a new `runtime.Client` is instantiated as seen in figure 5.3 in step 3.

When a `Benchmark` is run, a `Report` is generated. Each `Report` has to be summable by using `Aggregate` and scalable by using `Scale`. The only structure implementing the `Report` interface is the `ValueReport` type. A `ValueReport` is a hash-map that is summable by key. All these reports are collected together with the report aggregate and the benchmark name in a `MatrixResult` as seen in figure 5.3 in step 5. After evaluating all benchmarks of a CRI/OCI combination, the `MatrixResults` are collected in a `MatrixEntry`. Both `MatrixResult` and `MatrixEntry` are annotated structures, such that they can be easily marshalled into a JSON string. When all CRI/OCI combinations are run, the array of `MatrixEntry` is written to the destination file specified in the YAML configuration and the `Index` of the `Matrix` is generated. An `Index` stores which

benchmark contains which report keys and persists across different configuration runs. The `MatrixEntry` items are collected in an even larger list that persists during different configuration runs, too.

`pkg/benchmark/suites`

The `suites` package implements the different benchmarks. They can be grouped into four categories.

The `limits` suite attempts to benchmark the performance of containers, whose resources have been limited. In this context the behaviour of gVisor is especially interesting due to the fact that the hypervisor is included in the control group accounting. These benchmarks may be of interest to users who run a lot of small, hard-limited containers.

The `operations` suite tests general container runtime operation performance. This includes measuring latencies during creation and destruction as well as container launch time. The benchmarks are relevant when evaluating runtimes in the context of small, rapid executions of a process – for example, serverless computing.

The `performance` suite measures the performance of containers running using a specific runtime. These measurements are done using the popular sysbench benchmarking tool [30]. They give an impression of the impact of the container and system runtime resource consumption.

The `scalability` suite quantifies the scalability of a runtime in relation to container instances. In this suite, the relation between the metrics is very relevant. For example, a runtime may perform slower in operations when running only a few containers but perform faster in comparison when running 50 instances.

`pkg/runtime`

The `runtime` package contains a wrapper for the default Container Runtime Interface gRPC client. It includes minor helper functions to simplify the creation of containers and sandboxes as well as pulling images. The `Client` implements basic container management functionality for pulling images, starting and stopping pod sandboxes, starting and stopping containers as well as extracting logs and container state.

`pkg/visual`

As described in the implementation introduction, the `visual` package uses HTML templating from `text/template`, Bootstrap [45] and ChartJS [44] to provide an appealing and understandable visualisation of the generated datasets.

5.2.2 Machine setup

To run all these benchmark on the same machine, a collection of container tools is required including

- containerd (`v1.2.0-621-g04e7747e`)
- CRI-O (`1.15.1-dev`)
- runc (`1.0.0-rc8+dev`)
- runsc (`release-20190529.1-194-gf10862696c85`)
- CNI (`v0.7.5`)
- crictl (`1.15.0-2-g4f465cf`)

All of these tools were compiled from scratch on the testing machine to minify the risk of running into incompatibilities. They were also run on a relatively recent Linux kernel `4.19.0-5-amd64` under Debian Buster on a 2,9 GHz Intel Core i5.

5.3 Issues faced and challenges

A major issue during the implementation of the tool has been dependency management. The typical style to refer to project dependencies has been to put them in a `vendor` folder in the project itself. Since Go 1.12, there exists a new dependency model implementation called `go mod`. As of writing this thesis Kubernetes did not support the dependency model. Early implementations of the tool still used the `k8s.io/kubernetes/pkg/kubelet/remote` package to interact with any CRI implementations. In the most recent versions of Kubernetes the gRPC definition and Go implementation of the Container Runtime Interface has been moved out of the Kubernetes main repository and into a subproject.

This collision of in-tree and out-of-tree dependencies made initial implementations unreliable, leading to multiple bugs – e.g., the tool could only talk to containerd despite using the standard CRI interface. After stripping the implementation of any references of Kubernetes `remote` package and implementing an own client wrapper and vendoring the gRPC version of CRI-O instead of the latest one, the tool was able to talk to both CRI implementations.

Another problem was the setup of the container runtimes itself. containerd as well as CRI-O are typically used in the context of Kubernetes and not installed from source and configured by hand. Since these container runtimes rely on the Container Networking Interface for networking configuration, the on-machine CNI had to be configured.

This included the installation of a collection of standard CNI plugins as well as the configuration of the CNI itself.

A major challenge has been to find valuable benchmarking metrics. A benchmarking tool can be very well written, but useless due to inconsistent metrics or data without significance. It still has to be assumed that some of the presented datasets favour one implementation over the other due to some implementation specifics in the benchmarking tool itself. Trimming benchmarks to fit into the toolset is also not easy. A good example for this issue is the problem of deciding where to stop testing scalability. Should the tool at maximum spawn 10, 25, 50, 100, 500 or even 1000? Where can the tool observe new behaviour that has not been observed before a specific stepping stone? These configuration specifics had to be tested extensively.

6 Performance and benchmarks

The following data points and graphics have been generated solely by the Touchstone tool. All benchmarks have been executed between 5 and 20 times depending on the workload of the benchmark itself. Due to the type of benchmarks ran and especially their vulnerability to strong outliers, only the median of the generated dataset is shown in the followings graphs and tables. All tests have been executed in a virtual machine with 2 CPU cores and 8 GiB of RAM on an Intel Core i5-6267U CPU. These conditions are very similar to a typical cloud virtual machine with the exception of the CPU itself because CPUs for servers tend to have more cache, additional instructions and other features.

6.1 General performance

6.1.1 CPU

The CPU benchmark measures the time needed to compute all prime numbers up to a certain maximum and therefore does not use a lot of system calls. Figure 6.1 shows that containerd has a lower computational overhead compared to CRI-O, the same applies to runc in comparison to runsc.

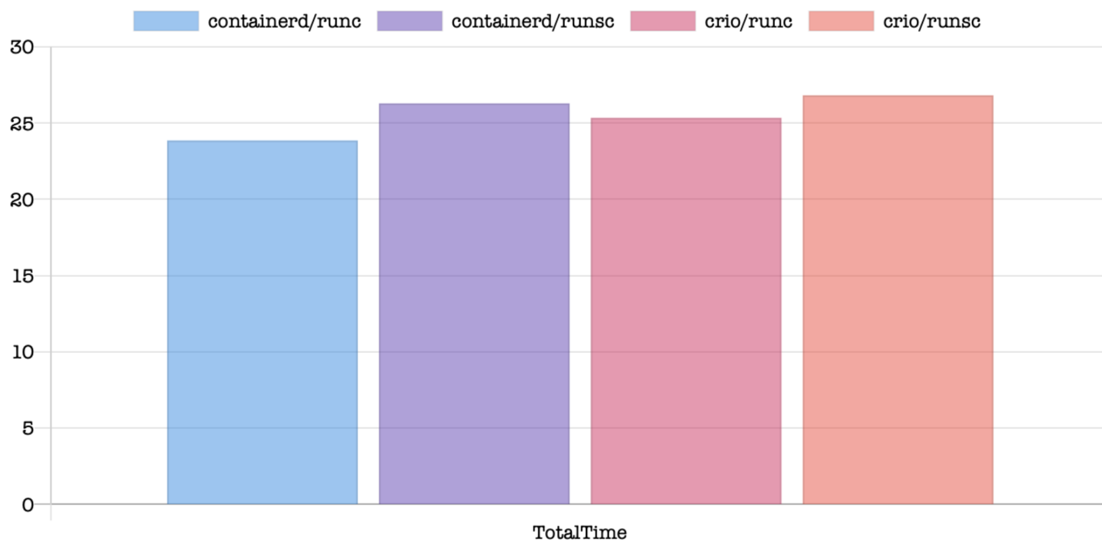

Figure 6.1: `performance.cpu.time`, 20 runs, measurement in seconds

Metric	containerd/runc	containerd/runsc	crio/runc	crio/runsc
`TotalTime`	23.8667s	26.2972s	25.3505s	26.8266s

Table 6.1: `performance.cpu.time`, 20 runs

6.1.2 Memory

When looking at total time spent in the memory benchmark, containerd is the clear winner. A subtle difference between runc and runsc can also be seen with runc as the leader.

Total time taken

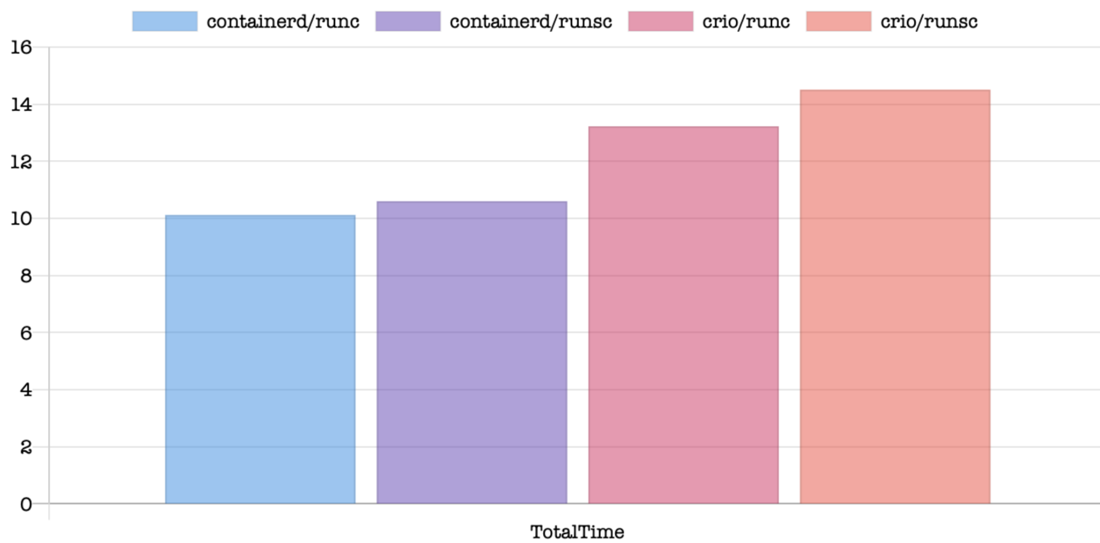

Figure 6.2: `performance.memory.total`, 10 runs, measurement in seconds

Metric	containerd/runc	containerd/runsc	crio/runc	crio/runsc
`TotalTime`	10.1209s	10.5991s	13.2264s	14.5114s

Table 6.2: `performance.memory.total`, 10 runs

Minimum and average latency

In this benchmark it can be observed that `runsc` introduces an additional overhead of a factor between 1.48x and 1.72x when accessing memory. Additionally, it seems that containerd performs on average better when accessing memory.

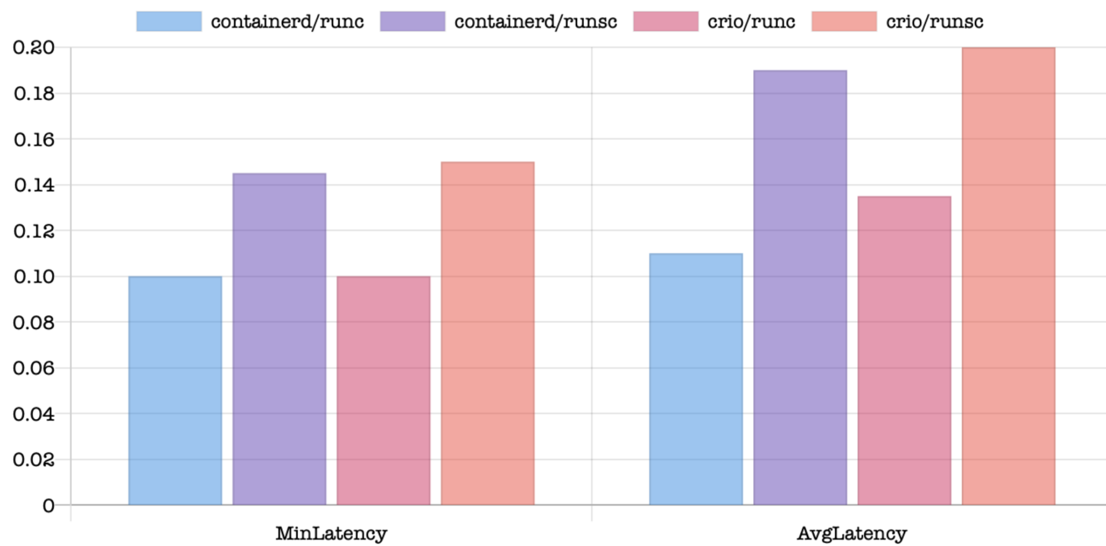

Figure 6.3: `performance.memory.minavglatency`, 10 runs, measurement in milliseconds

Metric	containerd/runc	containerd/runsc	crio/runc	crio/runsc
`AvgLatency`	0.1100ms	0.1900ms	0.1350ms	0.2000ms
`MinLatency`	0.1000ms	0.1450ms	0.1000ms	0.1500ms

Table 6.3: `performance.memory.minavglatency`, 10 runs

Maximum latency

The maximum latency benchmark is in stark contrast to the average latency, because `containerd/runc` has a major latency spike in comparison to `containerd|runsc`, `crio/runc` and `crio/runsc`. The benchmark was repeated with a reordered test execution and the same effect has been observed. It seems that using the `containerd/runsc` runtime setup leads to higher maximum latency.

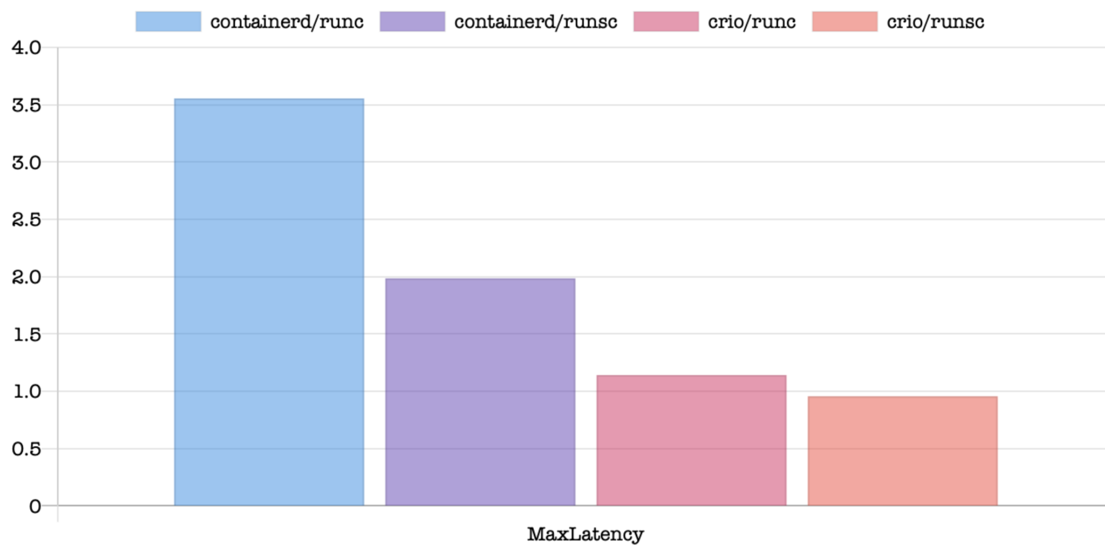

Figure 6.4: `performance.memory.maxlatency`, 10 runs, measurement in milliseconds

Metric	containerd/runc	containerd/runsc	crio/runc	crio/runsc
`MaxLatency`	3.5550ms	1.9850ms	1.1400ms	0.9550ms

Table 6.4: `performance.memory.maxlatency`, 10 runs

6.1.3 Disk I/O

Read throughput

In the sysbench `fileio` read benchmarks a large difference in performance between `runc` and `runsc` can be detected. gVisor introduces massive delays in workloads loaded with file operations like this synthetic test. This is due to an inefficient implementation of the virtual file system provided by Gofer and has been improved in a recent gVisor version [17].

The difference between `containerd` and `crio` when using `runc` is very minimal, even though `containerd` is slightly faster in almost all cases.

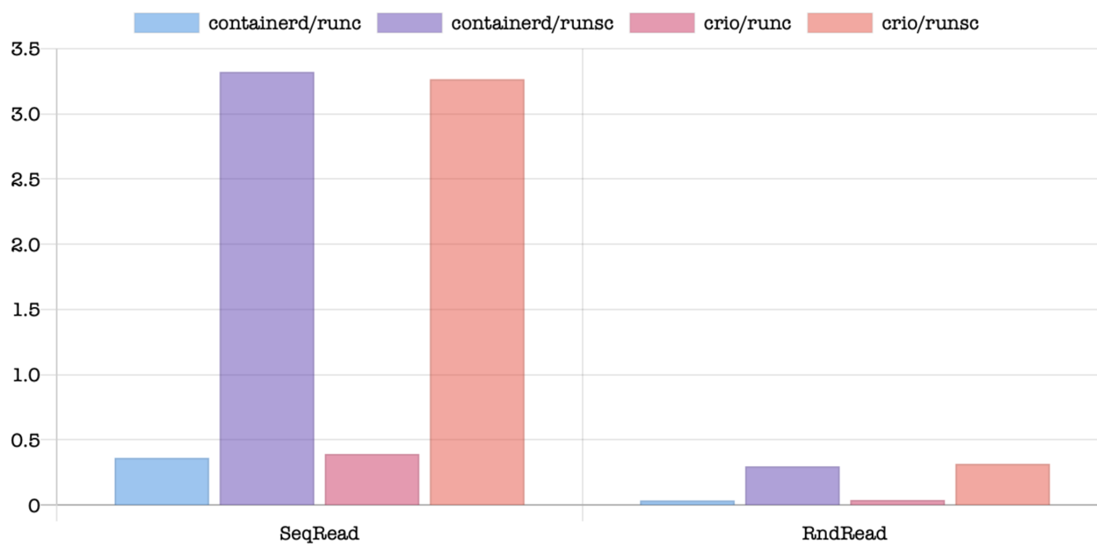

Figure 6.5: `performance.disk.read`, 10 runs, measurement in seconds

Metric	containerd/runc	containerd/runsc	crio/runc	crio/runsc
`RndRead`	0.0354s	0.2973s	0.0393s	0.3159s
`SeqRead`	0.3628s	3.3224s	0.3917s	3.2670s

Table 6.5: `performance.disk.read`, 10 runs

Write throughput

When writing to disk, the differences between runc and runsc stand out far less pronounced than in the read benchmarks. Therefore, the underlying hardware and operating system are, up to a certain degree of course, the cause for the smaller difference in write speeds. In contrast to the read benchmarks, CRI-O performs slightly better than containerd in all cases.

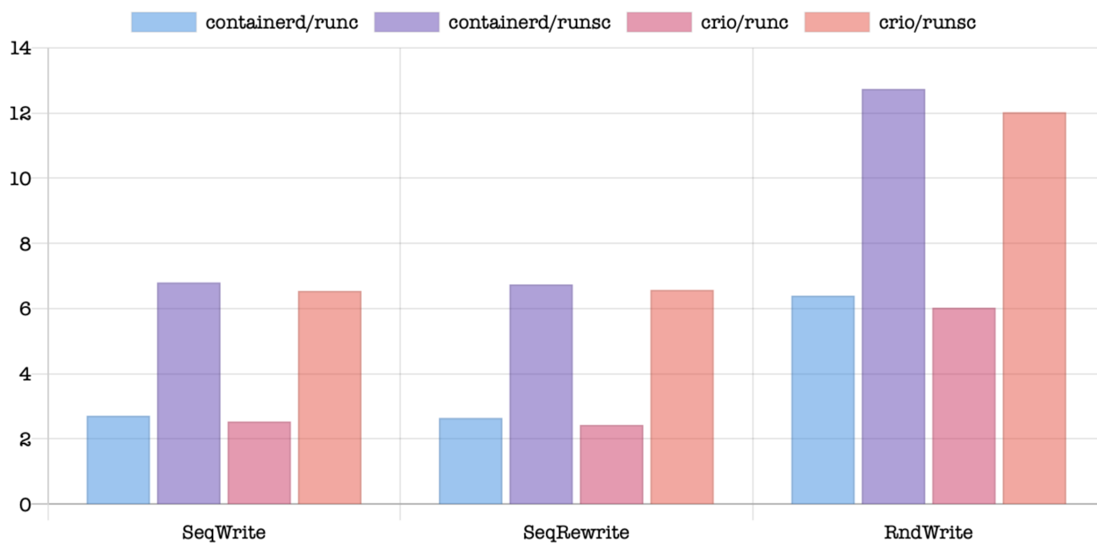

Figure 6.6: `performance.disk.write`, 10 runs, measurement in seconds

Metric	containerd/runc	containerd/runsc	crio/runc	crio/runsc
`RndWrite`	6.3971s	12.7412s	6.0281s	12.0289s
`SeqRewrite`	2.6437s	6.7412s	2.4302s	6.5751s
`SeqWrite`	2.7127s	6.8039s	2.5370s	6.5424s

Table 6.6: `performance.disk.write`, 10 runs

6.2 Container operations

In figure 6.7 it can be seen, that containerd performs slightly faster when creating a container than CRI-O. This head start is diminished by the higher run latency, in total containerd still performs worse than CRI-O. It is also visible that runsc operates almost always faster than runc.

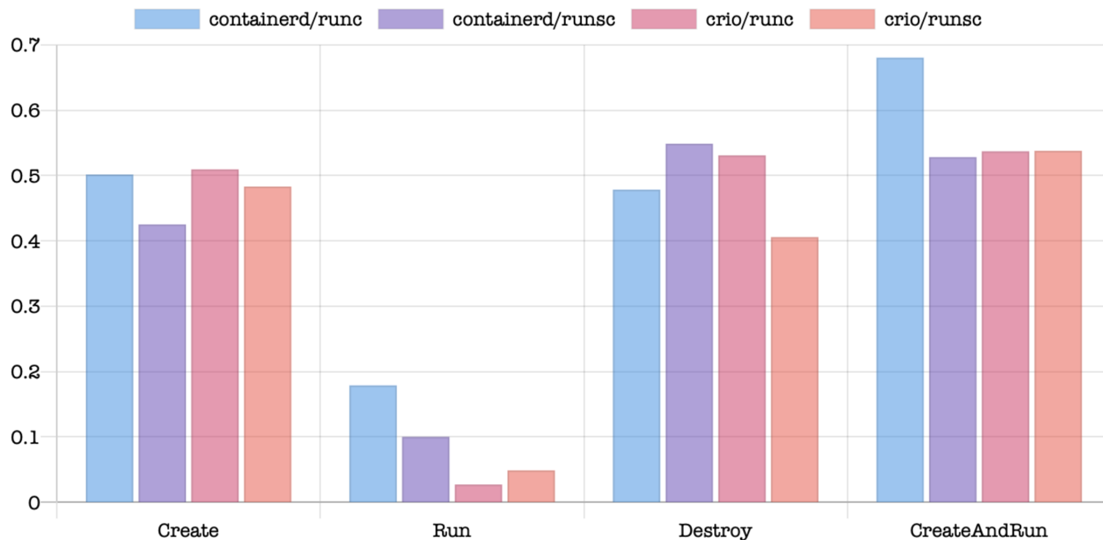

Figure 6.7: `operations.container.lifecycle`, 20 runs, measurement in seconds

Metric	containerd/runc	containerd/runsc	crio/runc	crio/runsc
`Create`	0.5017s	0.4254s	0.5097s	0.4832s
`CreateAndRun`	0.6807s	0.5284s	0.5372s	0.5380s
`Destroy`	0.4786s	0.5488s	0.5310s	0.4059s
`Run`	0.1790s	0.1001s	0.0272s	0.0489s

Table 6.7: `operations.container.lifecycle`, 20 runs

6.3 Scalability

The scalability benchmarks tests the performance of starting, running and destroying a specific amount of long-running containers.

5 containers

With 5 containers there is not much of a difference to the section 6.2, `containerd/runsc` performs best overall while `crio/runsc` falls behind.

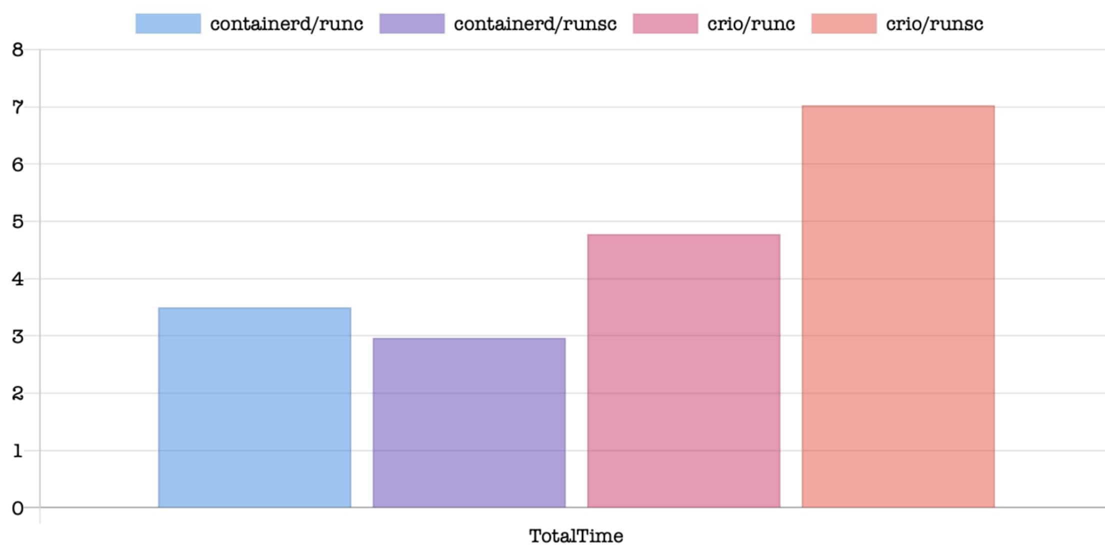

Figure 6.8: `scalability.runtime.5`, 10 runs, measurement in seconds

Metric	containerd/runc	containerd/runsc	crio/runc	crio/runsc
`TotalTime`	3.4959s	2.9636s	4.7769s	7.0276s

Table 6.8: `scalability.runtime.5`, 10 runs

10 containers

The trend seen in the 5 container test continues and becomes even more pronounced. We can see that the required time for starting and stopping 10 containers is in a linear relation to the container count.

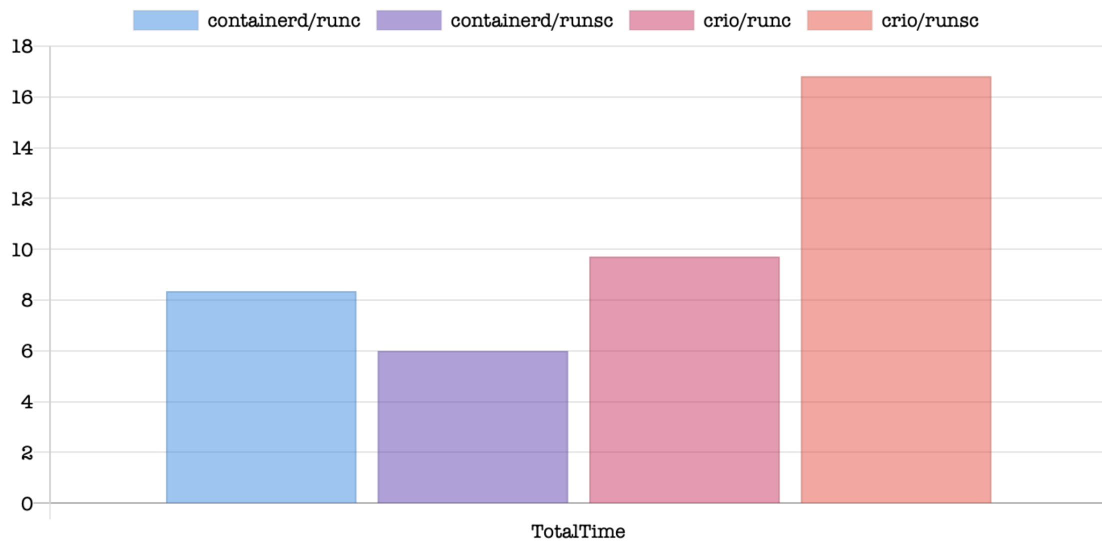

Figure 6.9: `scalability.runtime.10`, 10 runs, measurement in seconds

Metric	containerd/runc	containerd/runsc	crio/runc	crio/runsc
`TotalTime`	8.3489s	5.9964s	9.7095s	16.8144s

Table 6.9: `scalability.runtime.10`, 10 runs

50 containers

Interestingly, the linear trend discontinues and `containerd/runc` comes out ahead in this benchmark. Especially `crio/runsc` is very slow, this may be due to the larger runtime overhead.

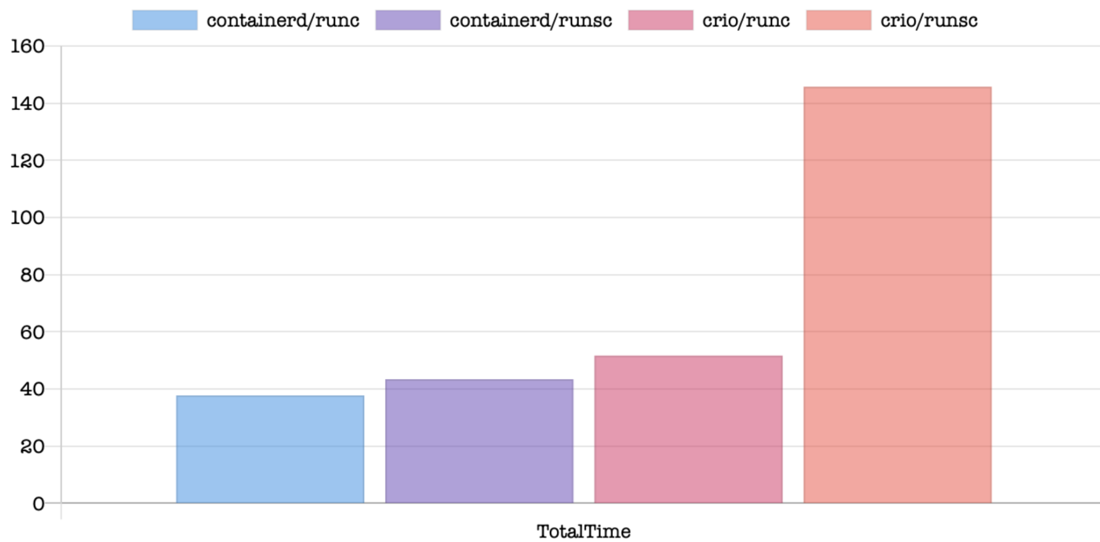

Figure 6.10: `scalability.runtime.50`, 10 runs, measurement in seconds

Metric	containerd/runc	containerd/runsc	crio/runc	crio/runsc
`TotalTime`	37.6949s	43.3633s	51.6614s	145.7691s

Table 6.10: `scalability.runtime.50`, 10 runs

6.4 Resource limits and quotas

6.4.1 CPU quotas

The following two benchmarks both run the same sysbench CPU test used in section 6.1.1 with the addition of limited CPU performance.

Hard CPU limits

The hard CPU limits assigns `cfs_quota_us = 1000` and `cfs_period_us = 10000`. It can be seen that the overhead of runsc hits the performance much harder in this benchmark compared to `performance.cpu.time`. This behaviour can be explained by the fact that gVisor adds both Sentry and Gofer to the control group. Since these components require CPU time, too, and are run in user space instead of kernel space, they count towards the control group CPU account [15]. Due to a bug the `containerd/runsc` setup ignored the control group limits and was therefore excluded from the benchmark.

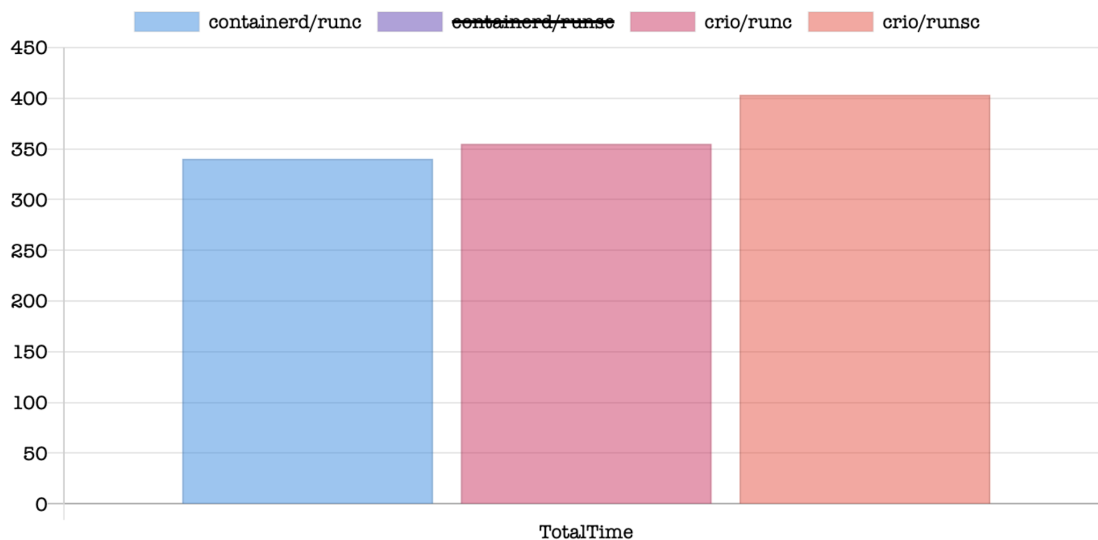

Figure 6.11: `limits.cpu.time`, 10 runs, measurement in seconds

Metric	containerd/runc	containerd/runsc	crio/runc	crio/runsc
`TotalTime`	340.3340s	-	354.9812s	403.4028s

Table 6.11: `limits.cpu.time`, 10 runs

Scaling CPU limits

The scaling benchmarks shines light into how workloads behave that are scaled from half of the allocatable CPU time up to full allowance over the course of 10 seconds. Since both gVisor does not support updating resource limits during the execution of a container, runsc had to be excluded from this benchmarks. It can be seen that the performance difference is very small, most likely due to the low logical overhead between CRI runtime and OCI handler – calls to `RuntimeService.UpdateContainerResources` can be directly translated to `runc update`.

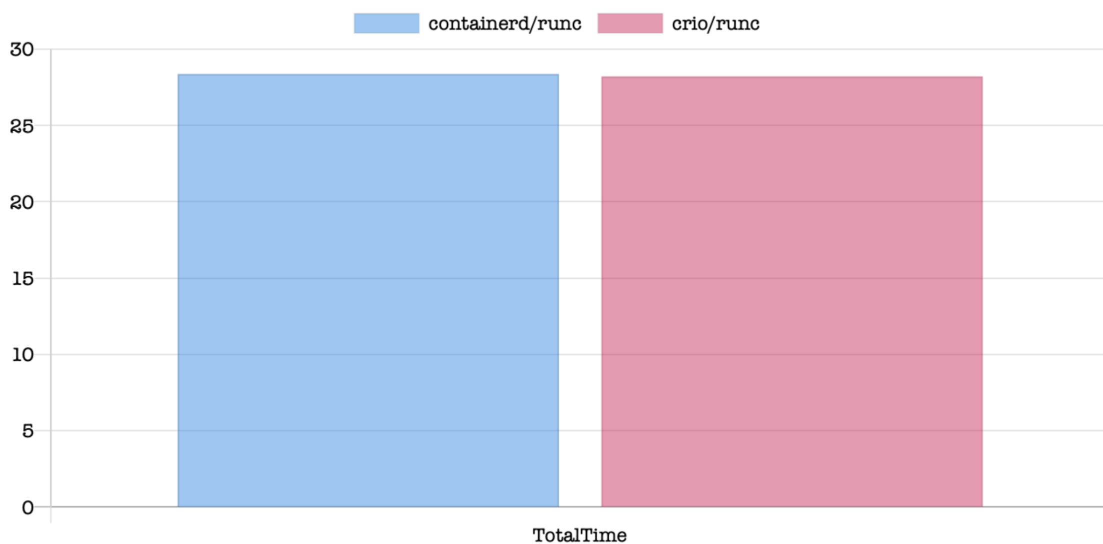

Figure 6.12: `limits.cpu.scaling`, 10 runs, measurement in seconds

Metric	containerd/runc	containerd/runsc	crio/runc	crio/runsc
`TotalTime`	28.3804s	-	28.2205s	-

Table 6.12: `limits.cpu.scaling`, 10 runs

6.5 Summary

In conclusion containerd performs better when running a container without limits due to the lower runtime overhead, although CRI-O is faster when running a container that has been created beforehand. The lead of containerd continues in memory performance. Disk performance gives a different picture. Although containerd leads in read performance, the superiority is broken by CRI-O in file read performance.

When comparing runc and runsc alias gVisor, runc comes ahead as the clear leader. This is not very surprising due to the Gofer virtual file system. It should be noted that runsc may perform on the same level as runc when running workloads that do not perform a lot of system calls, e.g., the `performance.cpu.time` benchmark. In every other benchmark runsc did worse than runc except the `operations.container.lifecycle`. It should be noted that runsc performs worse than runc even in compute bound workloads when running in a rigourously limited environment like `limits.cpu.time` [15].

The `containerd/runc` setup performs best for I/O heavy workloads like databases and web servers, while `crio/runc` is a solid alternative in any case. The usage of gVisor should be limited to systems where security is vitally important and has to be achieved with limited loss in raw computing power. Especially running very small containers with strong resource limitation may run significantly slower in gVisor compared to runc.

Metric	containerd/runc	containerd/runsc	crio/runc	crio/runsc
`CPUTime`	**23.8667s**	26.2972s	25.3505s	26.8266s
`HardLimit`	**340.3340s**	-	354.9812s	403.4028s
`ScalingLimit`	**28.3804s**	-	28.2205s	-
`MemMinLatency`	**0.1000ms**	0.1450ms	**0.1000ms**	0.1500ms
`MemAvgLatency`	**0.1100ms**	0.1900ms	0.1350ms	0.2000ms
`MemMaxLatency`	3.5550ms	1.9850ms	1.1400ms	**0.9550ms**
`FileRndRead`	**0.0354s**	0.2973s	0.0393s	0.3159s
`FileSeqRead`	**0.3628s**	3.3224s	0.3917s	3.2670s
`FileRndWrite`	6.3971s	12.7412s	**6.0281s**	12.0289s
`FileSeqRewrite`	2.6437s	6.7412s	**2.4302s**	6.5751s
`FileSeqWrite`	2.7127s	6.8039s	**2.5370s**	6.5424s
`OpCreate`	0.5017s	**0.4254s**	0.5097s	0.4832s
`OpRun`	0.1790s	0.1001s	**0.0272s**	0.0489s
`OpCreateAndRun`	0.6807s	**0.5284s**	0.5372s	0.5380s
`OpDestroy`	0.4786s	0.5488s	0.5310s	**0.4059s**
`Scalability5`	3.4959s	**2.9636s**	4.7769s	7.0276s
`Scalability10`	8.3489s	**5.9964s**	9.7095s	16.8144s
`Scalability50`	**37.6949s**	43.3633s	51.6614s	145.7691s

Table 6.13: Overview of benchmark results

7 Conclusion

This thesis has explored how container runtimes manage containers, provide isolation and limit resources. The functionality and inner working of namespaces and control groups have been investigated as well as the pros- and cons of the utility of user-space kernels in containerised environments.

Additionally, a benchmarking tool has been developed to evaluate container runtimes independent of any implementation detail using the popular Container Runtime Interface. The benchmarking tool is highly configurable and suited for continued development in the future. The tool makes it possible to include additional container runtimes easily, implement new kinds of benchmarks in a timely manner and access performance results without deep knowledge of additional tools by providing a high-quality visualisation that can be shared on the web.

The benchmarking tool has been applied to popular container runtimes and differences in the execution behaviour of these tools have been highlighted. These behavioural differences have been put in an application-oriented context including basic guidance in how to choose the right container runtime for a specific sets of requirements.

8 Future work

Of course, there is a lot of potential for future work. The benchmarking tool could be improved to benchmark all operations of a CRI implementation, including performance of extracting logs, pausing/resuming and more. It could also be tested on more CRI implementations, including Kata Containers and others, to find implementations that provide security guarantees comparable to gVisor.

Due to the fact that the CRI is currently missing the ability to set and update `blkio` control group properties, controlling CFQ properties using the Container Runtime Interface is impossible. In the case of support by a newer release of the CRI API specification that includes these resource controls, block I/O limits could also be benchmarked using the tool.

Another big task would be to improve the visualisation. At the moment it only shows the median values of each benchmarked metric. The visualisation could implement a switch between different index values per benchmark like minimum, maximum, average and median. A feature could be introduced that automatically annotates the results with human-interpretable facts like relative speed difference between two implementations.

Furthermore, the virtual machine is not standardised. The preferred way to distribute such a benchmarking environment would be a prebuilt virtual machine image, where the different CRI implementations can be installed on. This would also solve the dependency incompatibility problem, though it would make the integration of other CRI runtime harder. Another possible method of integrating standardisation could be the automatic creation of a cloud VM running the tool with preinstalled and preconfigured runtimes similar to tools like kops, which allows to create a Kubernetes cluster from the command line. This integration would also make cross-machine benchmarks possible – running Touchstone on multiple different machines and aggregating the results afterwards.

During the timeframe this thesis was written, a new version of gVisor has been released with a rewritten virtual file system [17]. The Touchstone benchmarks, especially for disk I/O, should be recreated with this newer runsc version.

List of Figures

2.1 Architecture of Docker [9] 4
2.2 Creating a new thread in a different namespace 6
2.3 Usage of network namespaces to link container networks 7
2.4 Performing `pivot_root` for process root isolation 8
2.5 Mounting `proc` in a container 8
2.6 `cgroup/cpu` ceiling enforcement 10
2.7 `cgroup/cpu` share assignment 11
2.8 `cgroup/cpusets` node assignment 11
2.9 `cgroup/cpuacct` usage stats 11
2.10 `cgroup/memory` memory limits 12
2.11 `cgroup/blkio` throttling 13
2.12 `cgroup/blkio` weighting 13
2.13 `cgroup/net_cls` traffic shaping 14
2.14 `ubuntu:latest` OCI image manifest 16

4.1 Architecture of containerd [43] 18
4.2 VMM [14] 22
4.3 gVisor [14] 22
4.4 Default [14] 22

5.1 Configuration YAML for performance testing 24
5.2 Touchstone package overview 25
5.3 Touchstone control flow 26

6.1 `performance.cpu.time`, 20 runs, measurement in seconds 32
6.2 `performance.memory.total`, 10 runs, measurement in seconds 33
6.3 `performance.memory.minavglatency`, 10 runs, measurement in milliseconds 34
6.4 `performance.memory.maxlatency`, 10 runs, measurement in milliseconds 35
6.5 `performance.disk.read`, 10 runs, measurement in seconds 36
6.6 `performance.disk.write`, 10 runs, measurement in seconds 37
6.7 `operations.container.lifecycle`, 20 runs, measurement in seconds . 38
6.8 `scalability.runtime.5`, 10 runs, measurement in seconds 39

6.9 `scalability.runtime.10`, 10 runs, measurement in seconds 40
6.10 `scalability.runtime.50`, 10 runs, measurement in seconds 41
6.11 `limits.cpu.time`, 10 runs, measurement in seconds 42
6.12 `limits.cpu.scaling`, 10 runs, measurement in seconds 43

List of Tables

6.1 `performance.cpu.time`, 20 runs 32
6.2 `performance.memory.total`, 10 runs 33
6.3 `performance.memory.minavglatency`, 10 runs 34
6.4 `performance.memory.maxlatency`, 10 runs 35
6.5 `performance.disk.read`, 10 runs 36
6.6 `performance.disk.write`, 10 runs 37
6.7 `operations.container.lifecycle`, 20 runs 38
6.8 `scalability.runtime.5`, 10 runs 39
6.9 `scalability.runtime.10`, 10 runs 40
6.10 `scalability.runtime.50`, 10 runs 41
6.11 `limits.cpu.time`, 10 runs 42
6.12 `limits.cpu.scaling`, 10 runs 43
6.13 Overview of benchmark results 45

Bibliography

[1] G. Avino, M. Malinverno, F. Malandrino, C. Casetti, and C.-F. Chiasserini. "Characterizing docker overhead in mobile edge computing scenarios." In: *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*. ACM. 2017, pp. 30–35.

[2] T. Bui. "Analysis of docker security." In: *arXiv preprint arXiv:1501.02967* (2015).

[3] E. Casalicchio and V. Perciballi. "Measuring docker performance: What a mess!!!" In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ACM. 2017, pp. 11–16.

[4] *cgroup_namespaces(7)*. `http://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html`. Accessed on 03.06.2019 16:36. The Linux man-pages project, 2019.

[5] *cgroups(7)*. `http://man7.org/linux/man-pages/man7/cgroups.7.html`. Accessed on 12.06.2019 12:10. The Linux man-pages project, 2019.

[6] *chroot(2)*. `http://man7.org/linux/man-pages/man2/chroot.2.html`. Accessed on 12.06.2019 13:50. The Linux man-pages project, 2019.

[7] *clone(2)*. `http://man7.org/linux/man-pages/man2/clone.2.html`. Accessed on 12.06.2019 12:12. The Linux man-pages project, 2019.

[8] *CRI Tools*. `https://github.com/kubernetes-sigs/cri-tools`. Accessed on 19.07.2019 17:17. Kubernetes SIGS.

[9] M. Crosby. *What is containerd?* `https://blog.docker.com/2017/08/what-is-containerd-runtime/`. Accessed on 17.06.2019 18:57. Docker Inc., 2017.

[10] M. Crosby. *Where are containerd's graph drivers?* `https://blog.mobyproject.org/where-are-containerds-graph-drivers-145fc9b7255`. Accessed on 03.07.2019 17:56. Docker Inc.

[11] *Docker Engine API Docs*. `https://docs.docker.com/engine/api/v1.24/`. Accessed on 22.07.2019 16:15. Docker Inc.

[12] Docker Inc. *Raft consensus in swarm mode*. `https://docs.docker.com/engine/swarm/raft/`. Accessed on 17.06.2019 18:50. Docker Inc., 2019.

[13] S. Francia. *cobra*. https://github.com/spf13/cobra. Accessed on 24.07.2019 15:09.

[14] *gVisor Architecture Guide*. https://gvisor.dev/docs/architecture_guide/. Accessed on 10.07.2019 20:07. Google LLC.

[15] *gVisor control group implementation*. https://github.com/google/gvisor/commit/29cd05a7c66ee8061c0e5cf8e94c4e507dcf33e0. Accessed on 24.07.2019 17:01. Google LLC.

[16] *gVisor source repository*. https://github.com/google/gvisor. Accessed on 10.07.2019 20:10. Google LLC.

[17] *gVisor VFS v2*. https://github.com/google/gvisor/pull/447. Accessed on 24.07.2019 17:02. Google LLC.

[18] Y.-J. Hong. *Introducing Container Runtime Interface in Kubernetes*. https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/. Accessed on 03.07.2019 17:59. Google LLC.

[19] A. Iwaniuk and B. Poplawski. *CVE-2019-5736*. https://nvd.nist.gov/vuln/detail/CVE-2019-5736. Accessed on 19.07.2019 15:03. 2019.

[20] *Kaniko*. https://github.com/GoogleContainerTools/kaniko. Accessed on 19.07.2019 14:15. Google LLC.

[21] *Kata Containers*. https://katacontainers.io. Accessed on 19.07.2019 14:05. OpenStack Foundation.

[22] Kernel Development Community. *CFS Bandwidth Control*. https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt. Accessed on 24.05.2019 13:24. The Linux Kernel Organization, Inc., 2019.

[23] Kernel Development Community. *CFS Scheduler*. https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt. Accessed on 24.05.2019 13:24. The Linux Kernel Organization, Inc., 2019.

[24] Kernel Development Community. *Control Group v2*. https://www.kernel.org/doc/Documentation/cgroup-v2.txt. Accessed on 24.05.2019 14:27. The Linux Kernel Organization, Inc., 2019.

[25] Kernel Development Community. *CPU Accounting Controller*. https://www.kernel.org/doc/Documentation/cgroup-v1/cpuacct.txt. Accessed on 24.05.2019 13:24. The Linux Kernel Organization, Inc., 2019.

[26] Kernel Development Community. *CPUSETS*. https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt. Accessed on 24.05.2019 13:24. The Linux Kernel Organization, Inc., 2019.

[27] Kernel Development Community. *Memory Resource Controller*. https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt. Accessed on 24.05.2019 16:01. The Linux Kernel Organization, Inc., 2019.

[28] Kernel Development Community. *net_cls Controller*. https://www.kernel.org/doc/Documentation/cgroup-v1/net_cls.txt. Accessed on 12.06.2019 13:44. The Linux Kernel Organization, Inc., 2019.

[29] Kernel Development Community. *net_prio Controller*. https://www.kernel.org/doc/Documentation/cgroup-v1/net_prio.txt. Accessed on 12.06.2019 13:43. The Linux Kernel Organization, Inc., 2019.

[30] A. Kopytov. *Sysbench*. https://github.com/akopytov/sysbench. Accessed on 22.07.2019 14:54.

[31] Z. Kozhirbayev and R. O. Sinnott. "A performance comparison of container-based technologies for the cloud." In: *Future Generation Computer Systems* 68 (2017), pp. 175–182.

[32] K. Kushwaha. *How Container Runtimes matter in Kubernetes?* https://events.linuxfoundation.org/wp-content/uploads/2017/11/How-Container-Runtime-Matters-in-Kubernetes_-OSS-Kunal-Kushwaha.pdf. Accessed on 6.06.2019 16:47. NTT OSS Center, 2017.

[33] S. McCarty. *Red Hat OpenShift Container Platform 4 now defaults to CRI-O as underlying container engine*. https://www.redhat.com/en/blog/red-hat-openshift-container-platform-4-now-defaults-cri-o-underlying-container-engine. Accessed on 22.07.2019 16:14. RedHat Inc.

[34] *mount_namespaces(7)*. http://man7.org/linux/man-pages/man7/mount_namespaces.7.html. Accessed on 03.06.2019 16:38. The Linux man-pages project, 2019.

[35] *namespaces(7)*. http://man7.org/linux/man-pages/man7/namespaces.7.html. Accessed on 03.06.2019 16:34. The Linux man-pages project, 2019.

[36] *network_namespaces(7)*. http://man7.org/linux/man-pages/man7/network_namespaces.7.html. Accessed on 3.06.2019 16:24. The Linux man-pages project, 2019.

[37] Open Container Initiative. *OCI Image specification*. https://github.com/opencontainers/image-spec/blob/db4d6de99a2adf83a672147d5f05a2e039e68ab6/. Accessed on 30.06.2019 12:52. The Linux Foundation, Inc., 2019.

[38] Open Container Initiative. *OCI Runtime specification*. https://github.com/opencontainers/runtime-spec/tree/74b670efb921f9008dcdfc96145133e5b66cca5c/. Accessed on 30.06.2019 12:54. The Linux Foundation, Inc., 2019.

[39] *pid_namespaces(7)*. http://man7.org/linux/man-pages/man7/pid_namespaces.7.html. Accessed on 03.06.2019 16:36. The Linux man-pages project, 2019.

[40] *pivot_root(2)*. http://man7.org/linux/man-pages/man2/pivot_root.2.html. Accessed on 12.06.2019 13:50. The Linux man-pages project, 2019.

[41] *Protocol Buffers*. https://developers.google.com/protocol-buffers/. Accessed on 19.07.2019 14:07. Google LLC.

[42] *setns(2)*. http://man7.org/linux/man-pages/man2/setns.2.html. Accessed on 12.06.2019 12:12. The Linux man-pages project, 2019.

[43] Stephen Day. *containerd Architecture Overview*. https://github.com/containerd/containerd. Accessed on 6.06.2019 14:51. The containerd Authors, 2019.

[44] T. C. team. *ChartJS library*. https://www.chartjs.org. Accessed on 19.07.2019 14:12.

[45] The Bootstrap team. *Bootstrap UI Toolkit*. https://getbootstrap.com. Accessed on 19.07.2019 14:11.

[46] The containerd authors. *containerd – Getting started*. https://containerd.io/docs/getting-started/. Accessed on 22.07.2019 19:00.

[47] The CRI-O authors. *CRI-O Project README*. https://github.com/cri-o/cri-o/tree/3e4eed9ddb67dacbcc38add0c021e23b255c49cd. Accessed on 03.07.2019 16:29. The Linux Foundation, Inc., 2019.

[48] The Kubernetes authors. *CRI-API*. https://github.com/kubernetes/cri-api. Accessed on 10.07.2019 20:12.

[49] The runc authors. *runc*. https://github.com/opencontainers/runc. Accessed on 12.06.2019 13:54.

[50] *Touchstone Benchmarking Tool*. https://github.com/lnsp/touchstone. Accessed on 19.07.2019 14:10.

[51] *unshare(2)*. http://man7.org/linux/man-pages/man2/unshare.2.html. Accessed on 12.06.2019 12:12. The Linux man-pages project, 2019.

[52] *user_namespaces(7)*. http://man7.org/linux/man-pages/man7/user_namespaces.7.html. Accessed on 03.06.2019 16:30. The Linux man-pages project, 2019.

[53] S. Walli. *Demystifying the Open Container Initiative (OCI) Specifications*. https://blog.docker.com/2017/07/demystifying-open-container-initiative-oci-specifications/. Accessed on 30.06.2019 15:50. Docker, Inc.