Performance Evaluation of Container Runtimes

Lennart Espe, Anshul Jindal[a], Vladimir Podolskiy[b] and Michael Gerndt
Chair of Computer Architecture and Parallel Systems, TU Munich, Garching, Germany

Abstract: The co-location of containers on the same host leads to significant performance concerns in the multi-tenant environment such as the cloud. These concerns are raised due to an attempt to maximize host resource utilization by increasing the number of containers running on the same host. The selection of a *container runtime* becomes critical in the case of strict performance requirements. In the scope of the study, two commonly used runtimes were evaluated: **containerd** (industry standard) and **CRI-O** (reference implementation of the CRI) on two different Open Container Initiative (OCI) runtimes: **runc** (default for most container runtimes) and **gVisor** (highly secure alternative to runc). Evaluation aspects of container runtimes include the performance of running containers, the performance of container runtime operations, and scalability. A tool called **Touch-Stone** was developed to address these evaluation aspects. The tool uses the CRI standard and is pluggable into any Kubernetes-compatible container runtime. Performance results demonstrate the better performance of containerd in terms of CPU usage, memory latency and scalability aspects, whereas file system operations (in particular, write operations) are performed more efficiently by CRI-O.

1 INTRODUCTION

Cloud computing as an infrastructure model has become an industry standard. Container technology plays a leading role in this change allowing to efficiently share host resources among multiple tenants. Hence, the demand for the performance evaluation of container workloads has risen. The widely-used container orchestrator Kubernetes supports every container runtime that implements the standardized Container Runtime Interface (CRI) (Hong, 2019). Although the cluster administrator can choose any CRI implementation, a guiding evaluation of runtime performance is missing.

Resource management is the most important task in multi-tenant environments such as the cloud. It aims at resource efficiency in dense deployment models like FaaS (Functions-as-a-Service, a service that abstracts away any underlying infrastructure of an application deployment) and other serverless architectures. As new deployment models are being introduced to the market, more requirements on specific facets of container runtime performance pop up. For example, Function-as-a-Service infrastructure de-

mands for low start-up times of the underlying container runtime. Past studies have shown that the performance overhead of wide-spread container runtimes is low and they are more resource-efficient than traditional virtual machines (Kozhirbayev and Sinnott, 2017). Combined with the ability to co-locate containers, this resource-efficiency leads to container runtimes being the main driver behind energy and cost reduction in the container-based computing environments. Still, though virtual machines are more resource-intensive, they offer a higher degree of security and performance isolation (Bui, 2015).

In this study, two commonly used runtimes are evaluated: **containerd** and **CRI-O** on two different Open Container Initiative (OCI) runtimes: **runc** (default for most container runtimes) and **gVisor** (highly secure alternative to runc). The evaluation tool **TouchStone** is developed to evaluate various performance aspects of resource management in container runtimes in a reliable manner. TouchStone measures how well the runtime's performance scales with the increase in the number of containers, how resource overuse is coped with as well as general runtime performance.

Section 2 provides the basic background knowledge for the paper. Section 3 delves into the imple-

[a] https://orcid.org/0000-0002-7773-5342

[b] https://orcid.org/0000-0002-2775-3630

mentation details of the container runtimes studies. Section 4 discusses related works. Section 5 examines the architecture and implementation of Touch-Stone evaluation tool. Section 6 provides evaluation results. Section 7 summarizes and discusses the results. Section 8 concludes the work.

2 BACKGROUND

2.1 Evaluated Container Runtimes

A container runtime is a software that runs the containers and manages the container images on a deployment node. Specific container runtimes are discussed in the following subsection. This subsection explores two container runtimes: containerd and CRI-O.

2.1.1 Containerd

Containerd is the default runtime used by the Docker engine (Crosby, 2017). It is often being referred to as an industry standard because of its wide adoption. Underneath, this runtime uses *runc*, the reference implementation of the OCI runtime specification. *containerd* stores and manages images and snapshots; it starts and stops containers by delegating execution tasks to the OCI runtime (The containerd authors,). To start a new containerized process, containerd has to undertake the following actions: 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 provides CRI-compatible gRPC endpoint enabled by default. When using this endpoint, abstractions specific to containerd are hidden from the client so that the user can operate on CRI-level abstractions.

2.1.2 CRI-O

Container Runtime Interface (CRI) is a standardized interface for Kubernetes plugins that execute and watch over containers. It was created as an attempt to stabilize the communication interface between *kubelet* and the host container runtime. It is based on gRPC, a cross-language library for remote procedure calls using Protocol Buffers.

CRI-O is a container runtime built to provide a bridge between OCI runtimes and the high-level Kubernetes CRI. It is based on an older version of the Docker architecture that was built around graph drivers (Walli,). It is mainly developed by RedHat, and it serves as a default runtime for OpenShift, a popular Kubernetes distribution for enterprise (Mc-Carty,). The typical life cycle of the interaction with CRI-O is similar to *containerd* over the CRI endpoint. Major differences in internal container handling to containerd do not exist since runc is the default OCI runtime when running CRI-O.

2.2 Evaluated OCI Runtimes

The Open Container Initiative is a Linux Foundation project to design open standards for operating-system-level virtualization, most importantly Linux containers. The Open Containers Initiative defines two standards – the *image-spec* for OCI images and the *runtime-spec* for OCI runtimes[1,2]. The typical job sequence would be that a container runtime (e.g. containerd) downloads an OCI image, unpacks it and prepares an OCI bundle (a container specification including the root filesystem) on the local disk. After that, a OCI 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 *docker build* command. After the successful build, these images are usually pushed and published to a public or private container registry. This subsection explores two container runtimes: runc and gVisor (runsc).

2.2.1 Runc

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 (Open Container Initiative, 2019). Internally, runc uses libcontainer to interact with OS-level components.

2.2.2 Gvisor (Runsc)

Typical OCI runtimes like *runc* limit system access using different capability models integrated into the Linux kernel like AppArmor, SELinux and SEC-COMP. If an adversary has access to a limited set of system calls and manages to exploit a vulnerability in the Linux kernel, then he may be able to escape the sandbox and gain privileged 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 and it will virtualize the

---

[1]github.com/opencontainers/image-spec/

[2]github.com/opencontainers/runtime-spec/

underlying system hardware the container is running on (kat, 2019); this approach has a significant performance overhead.

gVisor's approach to container isolation is in between that of a typical container runtime and of a fully virtualized runtime. The idea of a guest kernel of VMM-based runtimes served as an inspiration for a user-space kernel *Sentry* (which uses `ptrace` for syscall interception) and for container file-system access manager *Gofer*. Both of them are 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. To reduce overhead, it does not virtualize system hardware and leaves scheduling and memory management to the host kernel. This split of responsibilities negatively impacts performance for applications loaded with system calls and leads to potential incompatibilities between some workloads and gVisor due to specific Linux kernel behaviour or unimplemented Linux system calls (LLC, 2019).

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 (LLC, 2019).

3 RELATED WORK

The literature on container runtime evaluation is scarce with the most of the related research only partially being devoted to the evaluation of container runtime overhead.

Avino et al. performed the research in the context of resource-critical mobile edge computing, and have discovered that the CPU usage of the Docker process is constant regardless of the CPU cycle consumption of the containerized process (Avino et al., 2017). Casalicchio et al. have benchmarked Docker Engine for CPU- and I/O-intensive workloads with the results reporting on the reduction in overhead of Docker Engine from 10% down to 5% when the amount of requested CPU cycles by the container is over 80% (Casalicchio and Perciballi, 2017). Kozhirbayev et al. compared Docker to rivaling Flockport containers (based on LXC) showing that I/O and system calls were responsible for most performance overheads for both container implementations with Docker memory performance being slightly better than that of Flockport (Kozhirbayev and Sinnott, 2017). Morabito et al. also identify the disk performance as the major bottleneck both for container-based and hypervisor-

based virtualization although no significant overheads in other performance metrics were detected (Morabito et al., 2015).

The study by Xavier et al. went further by discovering that the near-native performance overhead of containers (LXC, Linux-VServer, and Open VZ) is acceptable for HPC applications (Xavier et al., 2013). The same argument holds for Docker in data-intensive HPC applications (Chung et al., 2016). Lee et al. evaluated performance of containers under the hood of production serverless computing environments by AWS, Microsoft, Google and IBM concluding that Amazon Lambda performs better in terms of CPU, network bandwidth, and I/O throughput (Lee et al., 2018). K. Kushwaha's study on the performance of VM-based and OS-level container runtimes such as runc, kata-containers, also containerd and CRI-O concluded that containerd performs better in comparison to CRI-O and Docker due to different file system driver interface design though CRI-O has very low container start-up latency in comparison to containerd (Kushwaha, 2017).

4 TouchStone

4.1 Overview of the Tool

This section describes the evaluation tool **Touch-Stone**(the name refers to small stones that have been used in the past to determine the quality of gold) which was implemented to conduct tests for this study. The tool is published under the MIT license on GitHub. TouchStone is implemented in Go due to available set of open-source tools that use CRI are implemented in Go.

The main motivation for the development of the tool is to improve container runtime evaluation by collecting meaningful evaluation data during multiple runs and exporting it. TouchStone allows to evaluate CPU, memory and block I/O performance as well as container operations performance and scalability of the performance under varying load. TouchStone runs evaluations in a testing matrix that tests each configuration pair consisting of container runtime (CRI runtime) and OCI runtime once. The tests are identified by a hierarchical naming scheme `context.component.facet`, for instance, `performance.cpu.total` in a CPU performance benchmark. All of the configuration listed above is done using a YAML file. After all benchmarks have been run, an index of the test results is generated and injected together with the results into an an easily readable HTML report.

pkg/benchmark

Benchmark interface

+ Run(client, handler): Report
+ Name(): string
+ Labels(): []string

Report interface

+ Aggregate(report): Report
+ Scale(scale): Report

Matrix struct

+ Index(): Index
+ Run(): []MatrixEntry

CRIs: []string
OCIs: []string
Items: []Benchmark
Runs: int

MatrixEntry struct

CRI, OCI: string
Results: []MatrixResult

MatrixResult struct

Name: []string
Aggregated: Report
Reports: []Report

Index map[string]IndexEntry

IndexEntry struct

Labels: []string

pkg/runtime

+ NewClient(address): Client

Client struct

...

RuntimeServiceClient
ImageServiceClient

pkg/visual

+ Write(file, entries, index)

pkg/config

+ Parse(file): Config

Config struct

+ Matrix(): Matrix

pkg/benchmark/suites

+ All(): []Benchmark

Performance: []Benchmark
Operations: []Benchmark
Limits: []Benchmark
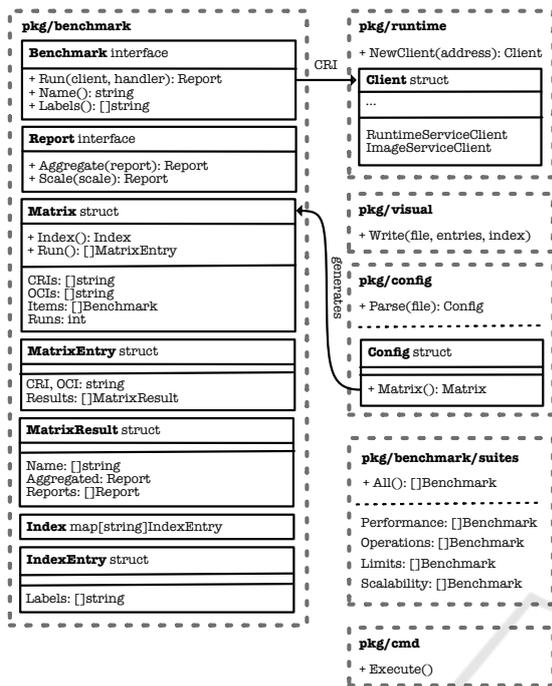Scalability: []Benchmark

pkg/cmd

+ Execute()

CRI

generates

Figure 1: Touchstone Package Overview.

4.2 Architecture

TouchStone consists of multiple packages as shown in the Figure 1. Following subsections provide an overview for each of them.

4.2.1 *pkg/cmd*

This package implements the command line interface (CLI). TouchStone provides four commands to the user – version, benchmark, list and index. The implementation of the user interface is based on the Cobra framework used by e.g. Docker and Kubernetes. version command returns the version of the tool and all CRI implementations it is able to talk to. benchmark command runs the specified benchmark. list command lists all benchmarking suites while the index command generates an index to preview the results generated by the benchmark command.

4.2.2 *pkg/config*

pkg/config implements YAML-parsable configuration 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 OCI runtime to use. In case the user wants to include a new

runtime in benchmarks, the runtime has to expose this endpoint either directly or via symbolic link.

4.2.3 *pkg/benchmark*

This package provides benchmarking framework and defines types Matrix, Benchmark and Report as well as Index and some helpers. Matrix specifies a list of targets to benchmark and has a slice of Benchmarks that are invoked for each combination of CRI runtime and OCI runtime. For each CRI runtime a new runtime.Client is instantiated.

After Benchmark finishes, a report is generated. All the reports are collected together with the report aggregate and the benchmark name in a MatrixResult. When all CRI/OCI combinations benchmarks are finished, the array of results is written to the destination file specified in the YAML configuration and the Index of the Matrix is generated. Index stores benchmark-report combinations and persists across different configuration runs.

4.2.4 *pkg/benchmark/suites*

Benchmarks implemented in this package can be grouped into four categories:

- **Limits.** Suite attempts to benchmark performance of containers with capped resources. Behaviour of gVisor on such benchmark is especially interesting due to the hypervisor being included in the control group accounting.

- **Operations.** Suite tests common container runtime operation performance including latencies during creation and destruction as well as container launch time.

- **Performance.** Suite collects conventional performance metrics of containers running using a specific runtime with measurements provided by **sysbench**[3].

- **Scalability.** Suite quantifies how scalable is the performance of a given runtime in relation to the amount container instances.

4.2.5 *pkg/runtime*

This package contains a wrapper for the default CRI gRPC client. It includes helpers to simplify creation of containers and sandboxes as well as pulling images.

---

[3]github.com/akopytov/sysbench

4.2.6 *pkg/visual*

It uses HTML templating from `text/template`, Bootstrap and Chart.js to provide a visualization of the collected results.

5 EXPERIMENTAL RESULTS

5.1 Test Setting

To run all the benchmarks on the same machine, a collection of container tools is used:

- 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 mitigate the incompatibility risk. They were run on a linux kernel `4.19.0-5-amd64` under Debian Buster. All benchmarks have been executed 5 to 20 times depending on the benchmark workload. Due to the type of benchmarks ran and especially their vulnerability to strong outliers, only the median of the generated dataset is shown in resulting graphs and tables. All tests have been executed in a virtual machine with 2 CPU cores (2,9 GHz Intel Core i5-6267U CPU) and 8 GiB of RAM. These conditions are similar to a typical cloud virtual machine though server CPUs tend to have more cache, additional instructions and other features.

5.2 General Performance

5.2.1 CPU

The CPU benchmark measures the time needed to compute all prime numbers up to a certain maximum, thus it should not use a lot of system calls. Figure 3 shows that containerd has a lower computational overhead compared to CRI-O, and the same applies to runc in comparison to runsc.

5.2.2 Memory

When looking at the 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 introducing higher performance (see Figure 2a).

However, when looking at the minimum and average latency, 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 as seen in Figure 2a.

The maximum latency benchmark is in stark contrast to the average latency as seen in Figure 2c, because `containerd/runc` has a 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. Apparently, this effect is rather a `containerd/runsc` runtime feature than an outlier.

5.2.3 Disk I/O

Sysbench's `fileio` read benchmarks reveals large difference in performance between `runc` and `runsc` as shown in Figure 4a. gVisor introduces massive delays in workloads loaded with file operations like this synthetic test. The observed performance drop is due to an inefficient implementation of the virtual file system provided by Gofer, which has been improved in a recent gVisor version[4]. The difference between `containerd` and `crio` when using `runc` is negligible, even though `containerd` is slightly faster in almost all the cases.

When writing to disk, the differences between `runc` and `runsc` stand out far less pronounced than in the read benchmarks as shown in Figure 4b. Therefore, the underlying hardware and operating system are, up to a certain degree, the cause for smaller variation in write speed. In contrast to read benchmarks, CRI-O performs slightly better than containerd in all cases.

5.3 Container Operations

Figure 5 points at a slightly better performance of containerd when creating a container (in comparison to CRI-O). This head start is diminished by the higher run latency such that in total containerd performs worse than CRI-O. It is also visible that runsc operates almost always faster than runc.

5.4 Scalability of Runtime Performance

Scalability benchmarks test the performance of starting, running and destroying an increasing amount of long-running containers.

---

[4]github.com/google/gvisor/pull/447
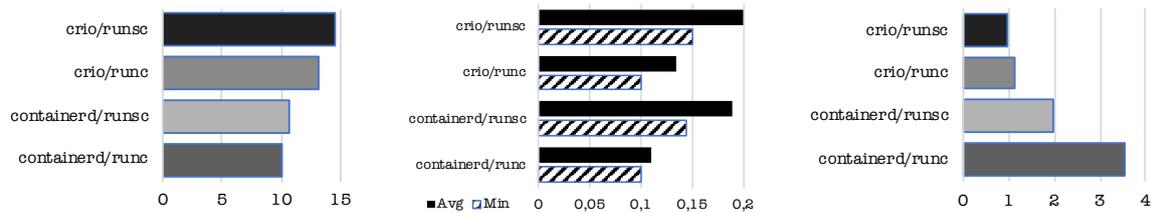
(a) Total Time and Measurements Are in Seconds.

(b) Minimum Avg. Latency and Measurements Are in Milliseconds.

(c) Maximum Latency and Measurements Are in Milliseconds.

Figure 2: `performance.memory` Benchmark for Different Aggregations with 10 Runs.

Table 1: Overview of the Evaluation Results.

| Metric	cd/runc	cd/runsc	crio/runc	crio/runsc
CPUTime(s)	**23.86**	26.29	25.35	26.82
HardLimit(s)	**340.33**	-	354.98	403.40
ScalingLimit(s)	**28.38**	-	28.22	-
MemTime	**10.12**	10.6	13.23	14.51
MemMinLatency(ms)	**0.10**	0.14	**0.10**	0.15
MemAvgLatency(ms)	**0.11**	0.19	0.13	0.20
MemMaxLatency(ms)	3.55	1.98	1.14	**0.95**
FileRndRead(s)	**0.03**	0.29	0.04	0.31
FileSeqRead(s)	**0.36**	3.32	0.39	3.26
FileRndWrite(s)	6.39	12.74	**6.03**	12.03
FileSeqRewrite(s)	2.64	6.74	**2.43**	6.57
FileSeqWrite(s)	2.71	6.80	**2.54**	6.54
OpCreate(s)	0.50	**0.42**	0.51	0.48
OpRun(s)	0.18	0.10	**0.03**	0.05
OpCreateAndRun(s)	0.68	**0.53**	0.54	0.54
OpDestroy(s)	0.48	0.55	0.53	**0.40**
Scalability5(s)	3.49	**2.96**	4.77	7.03
Scalability10(s)	8.35	**5.99**	9.71	16.81
Scalability50(s)	**37.69**	43.36	51.66	145.77
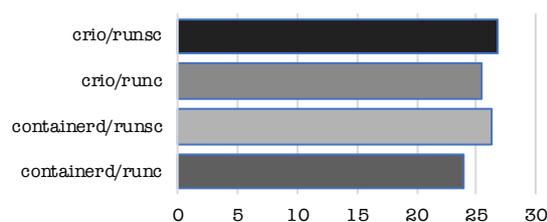
Figure 3: `performance.cpu.time`, 20 Runs, Measurement in Seconds.

With 5 containers there is not much of a difference to the section 5.3; `containerd/runsc` performs best overall while `crio/runsc` falls behind.

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

The linear trend discontinues with 50 containers and `containerd/runc` comes out ahead in this

benchmark. `crio/runsc` demonstrates significant slowdown on 50 containers, which might be due to the larger runtime overhead.

5.5 Resource Limits and Quotas

This benchmark focused at testing containers with the CPU Hard and Scaling limits using the same sysbench CPU test from subsection 5.2.1, but with capped CPU resource. Capping configuration was set to `cfs_quota_us` = 1000 and `cfs_period_us` = 10000.

Table 1 showing the CPU Hard Limit shows 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
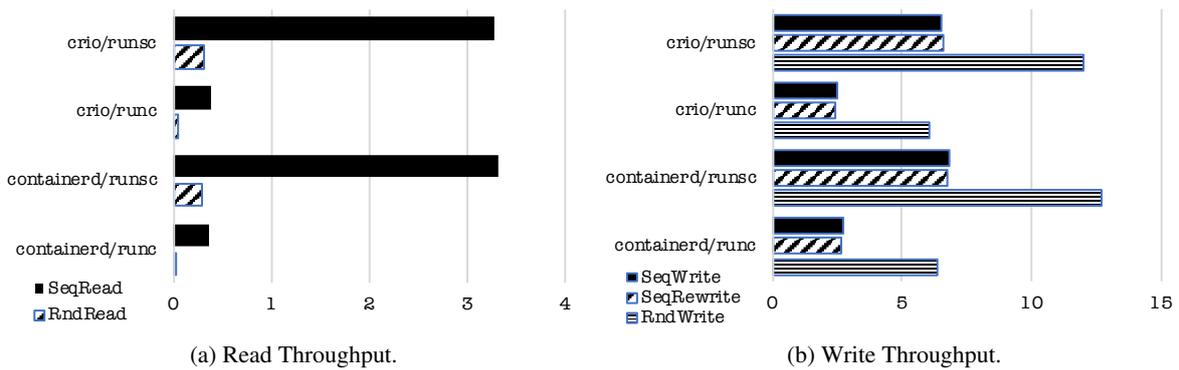
(a) Read Throughput.

(b) Write Throughput.

Figure 4: `performance.disk` Benchmark for Read and Write with 10 Runs and All the Measurements Are in Seconds.
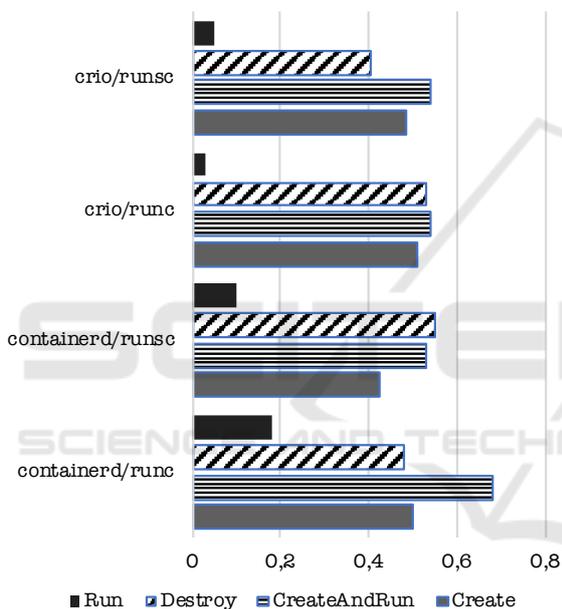

Figure 5: `operations.container.lifecycle`, 20 Runs, Measurement in Seconds.

account. Due to a runsc bug, the `containerd/runsc` setup ignored the control group limits and hence was excluded from the benchmark.

The scaling benchmark 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 gVisor does not support updating resource limits during the execution of a container, runsc had to be excluded from these tests. It can be seen that the performance difference is very small as shown in Table 1 Scaling Limit, most likely due to the low logical overhead between CRI runtime and OCI runtime

6 DISCUSSION

Table 1 shows the summary results of all the benchmarks run in the scope of the study with the highlighted ones being the best for that particular benchmark. To summarize, containerd performs better when running a container without limits due to 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 takes an upper hand in read performance, its 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 surprising due to Gofer virtual file system being the bottleneck. 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`. The `containerd/runc` setup performs best for I/O-heavy workloads like databases and web servers, while `crio/runc` is a solid starter pack in almost 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. Running very small containers with strong resource capping may be significantly slower in gVisor compared to runc.
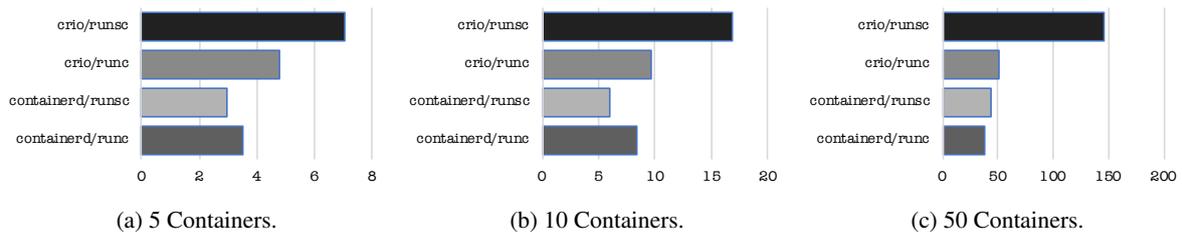
Figure 6: `scalability.runtime` Benchmark, 10 Runs for Different Number of Containers, All the Measurements in Seconds.

7 CONCLUSION

The developed evaluation tool TouchStone has been applied to wide-spread container and OCI runtimes such as containerd, CRI-O and runc and gVisor to determine performance implications of various evaluated aspects such as scaling the number of containers started in the runtime. The evaluation results highlighted `crio/runc` setup as the best starting option almost for any use-case, whereas `containerd/runc` turned out to excel at I/O-heavy workloads. Runtimes setups including `runsc` might provide better security, but their use must be carefully considered when performance is important. Prospective directions of future work include automatic deployment of a Kubernetes cluster with the container runtime picked specifically for given types of workloads and the support for cross-machine evaluations, i.e. running Touchstone on multiple machines and aggregating the results from such distributed testing environment.

ACKNOWLEDGEMENTS

REFERENCES

(2019). Kata containers. https://katacontainers.io. Accessed on 19.07.2019 14:05.

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

Bui, T. (2015). Analysis of docker security. *arXiv preprint arXiv:1501.02967*.

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

Chung, M. T., Quang-Hung, N., Nguyen, M., and Thoai, N. (2016). Using docker in high performance computing applications. In *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, pages 52–57.

Crosby, M. (2017). What is containerd? https://blog.docker.com/2017/08/what-is-containerd-runtime/. Accessed on 17.06.2019 18:57.

Hong, Y.-J. (2019). 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.

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

Kushwaha, K. (2017). 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.

Lee, H., Satyam, K., and Fox, G. (2018). Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 442–450.

LLC, G. (2019). gvisor architecture guide. https://gvisor.dev/docs/architecture_guide/. Accessed on 10.07.2019 20:07.

McCarty, S. 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.

Morabito, R., Kjällman, J., and Komu, M. (2015). Hypervisors vs. lightweight virtualization: A performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, pages 386–393.

Open Container Initiative (2019). Oci runtime specification. https://github.com/opencontainers/runtime-spec/tree/74b670efb921f9008dcdfc96145133e5b66cca5c/. Accessed on 30.06.2019 12:54.

The containerd authors. containerd – getting started.

https://containerd.io/docs/getting-started/. Accessed on 22.07.2019 19:00.

Walli, S. 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.

Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T., and De Rose, C. A. F. (2013). Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240.