Understanding Container Isolation: An Investigation of Performance Implications of Container Runtimes

Naman Bhaia
University of Washington
Tacoma, Washington, USA
nbhaia@uw.edu

Ling-Hong Hung
University of Washington
Tacoma, Washington, USA
lhhung@uw.edu

Robert Cordingly
University of Washington
Tacoma, Washington, USA
rcording@uw.edu

Wes Lloyd
University of Washington
Tacoma, Washington, USA
wlloyd@uw.edu

ABSTRACT

Containers have become a popular method of deploying and managing applications due to their lightweight nature, scalability, and portability. However, as container technologies evolve, it is important to understand new implications for performance and resource isolation. In this paper, we compare the performance of the runc, crun, and runsc container runtimes standalone and in parallel. We found that all runtimes struggled to isolate identical memory operations of concurrent containers. Runsc had higher overhead and lower performance than runc or crun, but less performance loss when scaling to 40 concurrent containers.

CCS CONCEPTS

• **Software and its engineering → Cloud computing**; • **General and reference → Performance**.

KEYWORDS

Containers, Performance, Isolation, Resource Management, Virtualization, Multi-Tenancy, Open Container Initiative, Sandboxing

ACM Reference Format:
Naman Bhaia, Ling-Hong Hung, Robert Cordingly, and Wes Lloyd. 2023. Understanding Container Isolation: An Investigation of Performance Implications of Container Runtimes. In *24th ACM/IFIP International Middleware Conference, December 11–15, 2023, Bologna, Italy*. ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Containerization is a software technology that enables the creation and management of lightweight, portable, and isolated computing environments. It can be seen as a way to package and distribute applications with all the dependencies and configuration needed to run them. This enables easy deployment and consistent runs across different computing environments, such as development machines, testing servers, and production clusters.

Containers share the host operating system and provide an isolated environment for the application without emulating the entire computer system as a Virtual Machine (VM) does. This makes containers more lightweight and efficient, allowing more containers to run on the same host compared to VMs, while providing improved security and isolation compared to running applications directly on a shared host. Containers have become an important part of modern software development and deployment, owing to their support for faster and more reliable application deployment across various platforms and environments [18].

One major benefit of containerization is the isolation it provides to various processes. This isolation should ensure that containerized applications should not be able to observe the characteristics of co-hosted containers. Containerization is faced with the challenge of side channel attacks. A side channel is a vulnerability where an "attacker" gains access to sensitive information or resources by exploiting weaknesses or unintended interactions between the container and the underlying host system [1]. For example, if an attacker identifies high CPU use they could run CPU-bound tasks on the host to starve the CPU to impact a victim's container.

1.1 Research Questions

This paper investigates the following research questions: What is the degree of performance isolation provided by current container runtimes? Do some runtimes provide better isolation when containers compete for identical resources simultaneously (e.g., CPU, memory)?

For VMs with precise resource quotas (e.g. for CPU and memory), performance of identical workloads run concurrently across multiple VMs on the same host should not degrade. In this paper, we investigate the extent of resource isolation provided by different container runtimes for running containerized workloads concurrently on the same host. Ideally containers should provide performance isolation similar to VMs when sharing a host.

1.2 Contributions

This paper provides the following research contributions:

(1) Benchmarking Suite: We developed the Container Parallel Test Suite (CoPTS) benchmarking suite to automate scheduling parallel benchmark runs across different container runtimes. The tool orchestrated experiments for the paper.

Middleware 2023, December 11–15, 2023, Bologna, Italy

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

(2) Analysis of CPU resource utilization of various benchmarks using the Container Profiler tool [11, 12]
(3) Quantitative comparison of the performance isolation provided by the runc, runsc, and crun container runtimes

2 BACKGROUND AND RELATED WORK

2.1 Container Runtime Performance Overhead

As cloud technologies evolve, so do the number of virtualization methods. In [17], Roberto et al. compared hypervisor-based VMs and containers. Their study investigated overhead introduced by virtualization compared to a non-virtualized environment. Hypervisors abstract the hardware and this virtualization of hardware adds overhead. Containers typically share a host kernel and therefore introduce overhead at the OS level. Roberto et al. compared overhead of Docker, LXC, KVM, and OSv vs. a non-virtualized environment. They evaluated memory, CPU, disk I/O, and network performance. They found that benchmarks run in containers performed better than VMs while introducing minimal overhead providing a viable alternative to virtualization.

Prior research on containerization has focused on comparing containers to VMs and non-virtualized hosts while considering their design differences. In this paper, we compare multiple container runtimes in terms of performance and isolation. In [21], performance of runc, gVisor, and Kata containers was compared, but the performance implications of running many container instances in parallel was not investigated.

In [5], Everarts de Velp et al. compared various container runtimes, images, storage drivers, and container managers to report the combination(s) with the best performance. They profiled container startup time and disk I/O performance using isolated runs of various container technology combinations. In our paper, we complement this work by benchmarking other resources including memory and CPU in isolation and parallel.

Han et al. profiled resource contention using CPU and memory benchmarks run on up to 48 co-located VMs in parallel to observe performance implications [10]. In contrasting our results with 40 parallel containers to Han's running 48 parallel VMs, we conclude that containers afforded less performance degradation for y-cruncher, but potentially more for sysbech prime number generation as discussed later in section 4.

2.2 Sandboxing in gVisor to improve isolation

Prior work has compared gVisor's network and file access time to that of more traditional containers like Docker [23]. This effort examined the impact of having a separate application kernel for typical system calls and briefly described the benefits it provides. The design for having two isolation layers is to improve isolation between containers but leads to increased time for disk I/O and system calls [20]. In[23], Young et al. found that gVisor was at least 2.2x time slower at making system calls when compared to traditional containers and reading small files from the tempfs was found to be 11x slower on gVisor.

Previous comparisons by Anjali et al. found that gVisor's architecture led to considerable duplication of functionality [2]. While gVisor's sentry handles the majority of the system calls, it still emulates all steps that other Linux containers take. Thus, the design of gVisor is more complicated as it uses multiple implementations of the same functionality. The paper also compared how LXC uses the Linux kernel directly, and how Firecracker supports reduced kernel functionality to improve performance [15].

In this paper, we compare Docker runtime (runc) alternatives. We examine their ability to provide resource isolation while not sacrificing performance when hosting parallel workloads.

3 METHODS

3.1 Container Parallel Test Suite

To support orchestration of running benchmarks concurrently using different container runtimes, for this paper we developed the Container Parallel Test Suite (CoPTS). CoPTS provides an extensible benchmarking suite implemented using Bash and Python to automate orchestration of parallel benchmark runs across different container runtimes. CoPTS supports:
(a) running multiple benchmarks including Bonnie++, Linpack, Noploop, Stream, Sysbench, Unixbench, and Y-Cruncher.
(b) execution of benchmarks on OCI compatible runtimes including runc (Docker), runsc (gVisor), and crun runtimes.
(c) configurable options to specify: the benchmark to test, the number of processes to create, the number of containers to launch and run sequentially per process, and the number of benchmark runs per container. Parallel testing is achieved by having multiple processes create and execute containers in parallel on the same host.
The design of CoPTS is extensible so that new benchmarks or container runtimes can be added. CoPTS aggregates key performance indicators generated by benchmarks (e.g. runtime or throughput) into a tabular format to enhance readability. For each run, CoPTS generates 'x+1' CSV files: • 1 file from each of the 'x' parallel processes aggregating all benchmark iterations within it • 1 file that aggregates the runs from all the parallel process. For example, to orchestrate running the STREAM benchmark, CoPTS can be configured to create 40 processes which each launch a container 20 times sequentially. Then for each container CoPTS runs STREAM 10 times. CoPTS then generates 40 files summarizing output from each process to aggregate results from (20x10=200) runs. A final output file is then produced to aggregate results from all 40 processes to capture results for all (40x20=8,000) runs. The number of processes, containers, and runs are all configurable. Additionally, each container instance launched by CoPTS was limited to use 2 vCPUs and 4GB of memory.

3.1.1 ***Container Runtimes.*** This paper investigates the isolation and performance of alternate container runtimes. Container runtimes were selected based on their adoption in the industry, release dates, and compatibility with Docker. Docker was made open source in 2013 and has since been widely adopted and crowned the de facto representative for containerization. For this reason, this paper uses Docker and runc as the baseline for our performance comparisons. We compare performance and isolation of runc with runsc and crun (2019). We chose these container runtimes because they differ in their design philosophy. Runsc's design is focused on providing better isolation, while crun's design is focused on allowing for easy adoption and lower performance overheads. Container technologies used in this paper are further described below.

- Docker: Docker is an open-source container technology that provides interfaces to create and control containers. Docker employs runc as its default container runtime. runc is an OCI-compliant, high-performance, and low-level container orchestrator. Runc manages container creation while serving as an intermediate layer to conceal system call intricacies. Given runc's dependence on Linux kernel features such as control groups for resource allocation, and namespaces for isolation during container creation, the host kernel is shared across all containers. Sharing the host kernel without proper resource isolation can allow one container to utilize resources beyond the limits set and impact the throughput of other co-hosted containers [7, 14]. In this paper, the Docker runtime (runc) serves as a baseline for comparison. Performance and isolation attributes of alternate container runtimes are compared against runc to glean the advantages and drawbacks of each.
- gVisor: gVisor is a container engine developed by Google which implements a runtime called runsc. runsc is an open container initiative (OCI) runtime which has been designed to provide better isolation between container runs[8]. Runsc prevents attackers from gaining host access even if they have only compromised a single subsystem. Runsc introduces an application-level kernel (the sentry) that provides a layer of abstraction to the host kernel. This prevents the user from obtaining direct access to the host's kernel to eliminate unfiltered system calls[23]. Runsc also implements rule-based execution of system calls for added security. gVisor's sandboxing approach adds overhead to container operations like system calls [2].
- Crun: provides a fast and low-memory footprint container runtime compliant with the OCI specifications. Many container runtimes like runc and runsc are written in Go. The motivation to develop crun was to provide a container runtime implemented in C that is efficient. As a result, crun's compiled binaries are 50 times smaller than runc [4, 19]. Crun is open-source and currently maintained by Red Hat as part of their larger containers eco-system (Containers, 2021) [3].

Resource Type	Benchmark
Memory	stream, sysbench-memory, y-cruncher
CPU	linpack, noploop, sysbench-cpu, y-cruncher

Table 1: Benchmarks and the primary resources they stress

3.1.2 ***Benchmarks.*** This paper compares container runtimes by executing common benchmarks in parallel. Benchmarks that exercise core system resources (e.g. CPU and memory) are chosen to investigate performance and isolation characteristics of container runtimes. For each benchmark, we've configured CoPTS to execute an increasing number of parallel runs from 1 to 40 stepping up by 10. We executed the following benchmarks:

- B1 - Linpack: provides a collection of Fortran subroutines that performs lower-upper (LU) factorization to solve the matrix expression A*X = B. This calculation is performed in a loop and since it is a throughput-based benchmark, the number of loops can differ impacting the runtime. Linpack returns output in MegaFLOPS (floating point operations per second) [6]. We configured matrix A to be 600x600 in size.
- B2 - Noploop: provides a simple CPU clock speed benchmark which records the runtime of performing a fixed number of No-Operation (NOP) instructions. Dividing the number of NOPs by the time taken provides the observed clock speed of the container. Noploop is advantageous because it is simple to implement, minimizes cache-miss based variations, stall cycles, and branch mispredictions [9]. We set Noploop to 6 billion NOPs.
- B3 - Stream: System memory performance is measured using Stream which employs uncomplicated vector kernel operations. Stream measures throughput for four distinct operations: copy, scale, add, and triad reporting aggregate throughput [16].
- B4 - Sysbench-CPU: provides a popular benchmark based on LuaJIT which can profile multiple resources. We leveraged sysbench to generate 20 million prime numbers to stress the CPU [13].
- B5 - Sysbench-memory: was used to stress memory by writing 100GB in 1KB blocks to measure memory write throughput.
- B6 - Y-cruncher: computes Pi and other constants to trillions of digits as a deterministic workload to generate CPU and memory stress [22]. We calculated pi to 100 million digits.

3.2 Experiments

We conducted a series of experiments to profile isolation properties of container runtimes. The experiments examined performance using CPU and memory benchmarks, and also the isolation capabilities of container runtimes for those two resources.

- E1 – CPU Utilization Performance Test
  This experiment compares CPU performance of container runtimes by running CPU benchmarks across all runtimes. The benchmarks for this experiment were: Linpack (B1), Noploop (B2), and Sysbench CPU (B4).
- E2 – Memory Utilization Performance Test
  This experiment compares the memory performance of container runtimes by running memory benchmarks across all container runtimes. The benchmarks for this experiment were: Stream (B3), Sysbench Memory (B5), and Y-Cruncher (B6).
- E3 - CPU and Memory Resource Isolation Test
  This experiment ran benchmarks on multiple container instances in parallel on a shared host while scaling up the number of concurrent runs (1, 10, 20, 30, 40) to observe the resource utilization. We then compare the performance of the runtimes in parallel and in isolation (E1 and E2). The percentage change in performance suggests the isolation offered by different runtimes.

To ensure the test results were not affected by any other applications running on the same host, we performed our experiments on an AWS EC2 c5d.metal instance running Ubuntu Server 22.04 LTS which provided an isolated cloud server. This machine features 96 vCPUs with 192 GiB of memory.

One limitation of our experiments is that we cannot differentiate the amount of performance degradation resulting from the choice of container runtime versus the amount of degradation caused by resource contention of CPU hyper-threading. Our experimental configurations enabled benchmarking the same workloads against different container runtimes, while the Linux scheduler governed the executions. To ensure fairness in our evaluations, our experiments subjected all three container runtimes to the identical host environment, aiming to isolate the performance discrepancies attributable solely to the container runtime choice.

4 RESULTS

For the paper, we ran Linpack, Noploop, Stream, Sysbench CPU, Sysbench Memory, and Y-Cruncher. We first profiled benchmark resource utilization using the Container Profiler on a c5.xlarge AWS EC2 instance to ensure benchmarks had distinct resource utilization profiles. The c5.xlarge instance features 4 vCPUs and 8 GiB of memory. The Container Profiler is a Linux-based tool that enables resource utilization profiling of scripts and container-based tasks. It collects metrics related to CPU, memory, disk, and network utilization at the VM, container, and process levels [11, 12].

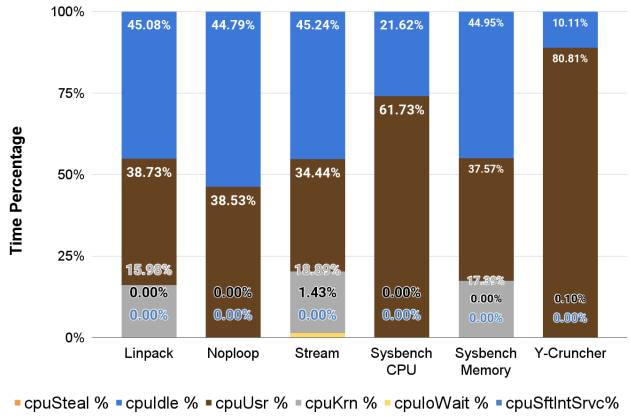

Figure 1: CPU Profile for various Benchmarks

In figure 1 we can see that other than Noploop, our benchmarks had unique resource utilization and therefore provide a complementary set of benchmarks. Noploop exhibited similar resource utilization to Linpack but we implement it in our comparison as it is light-weight and unique in that it provides an estimation of effective CPU clock speed.

B1 - Linpack

We observed the following results from running Linpack across our container runtimes:

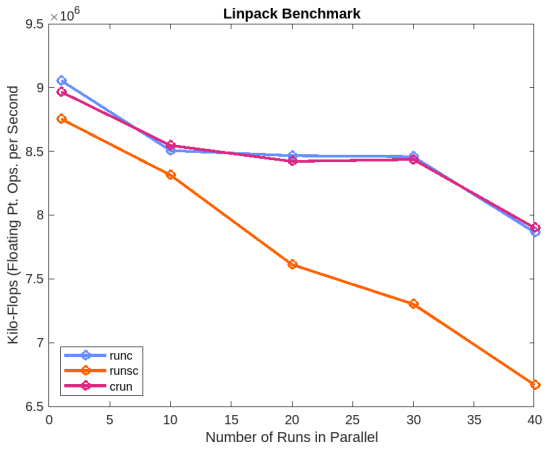

Figure 2: Linpack Benchmark Results

Based on Figure 2 and Table 2, we can see that runsc performs poorly relative to runc when scaling up concurrent runs (1, 10, 20, 30, 40). runsc's performance loss at 40 concurrent runs vs. 1 isolated run was 2x greater than runc. Similarly, on comparing the performance of runc and crun, we found that their throughput and performance loss when scaling up was almost identical. For Linpack we observed the following ordering of container CPU isolation: crun > runc > runsc.

Container Runtime	Throughput		Throughput Loss
	Isolated Run	40 Parallel Runs	
runc	1.00x	0.87x	~ -13%
runsc	0.97x	0.74x	~ -24%
crun	0.99x	0.87x	~ -12%

Table 2: Linpack performance comparison for an isolated run and 40 concurrent runs. Results are normalized to the isolated (1 run) performance of runc (x=9,054,862 KFLOPS) as a baseline. (Higher is better)

B2 - Noploop

We observed the following results when running Noploop across our container runtimes:

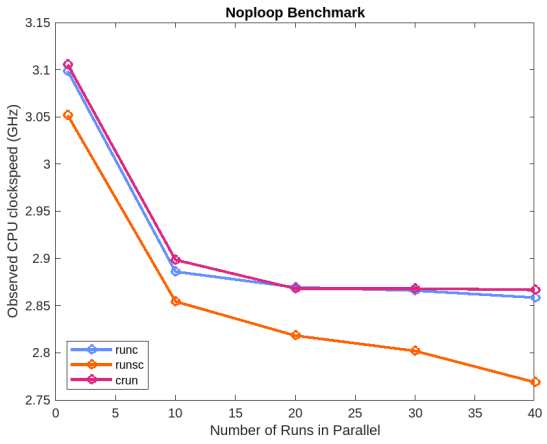

Figure 3: Noploop Benchmark Results

Container Runtime	Estimated CPU Clock Speed (Ghz)		Throughput Loss
	Isolated Run	40 Parallel Runs	
runc	3.1 GHz	2.85 GHz	~ -8%
runsc	3.04 GHz	2.82 GHz	~ -7.5%
crun	3.1 GHz	2.85 GHz	~ -7.9%

Table 3: Noploop performance comparison for an isolated run and 40 concurrent runs. (Higher is better)

Based on Figure 3 and Table 3, for Noploop, we see that runsc performs slightly worse than runc and crun but the clock speed degradation when scaling from 1 to 40 concurrent runs is fairly similar (runc > crun > runsc). Running 40 concurrent container instances on the same host had the effect of degrading clock speed by approximately 8%. Noploop performance overall was similar across our container runtimes and therefore did not inform us about which container runtimes provided better CPU isolation.

B3 - Stream

The results for running Stream across our container runtimes for the copy operation are shown in table 4 and figure 4. For Stream, runsc

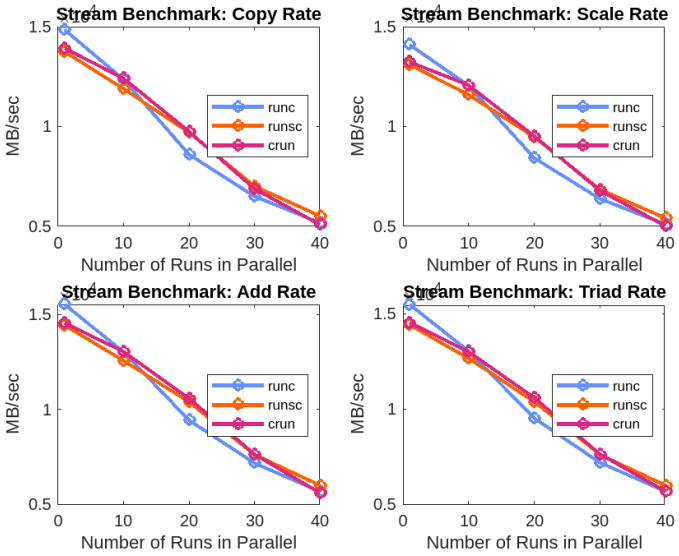

Figure 4: Stream Benchmark Results

Container Runtime	Throughput (MB/sec)		Throughput Loss
	Isolated Run	40 Parallel Runs	
runc	1.00x	0.35x	~ -65%
runsc	0.92x	0.37x	~ -60%
crun	0.94x	0.34x	~ -63%

Table 4: Stream throughput copy comparison for an isolated run and 40 concurrent runs. Results are normalized to the isolated (1 run) throughput of runc (x=14,843 MB/sec) as a baseline. (Higher is better)

performed poorly vs. runc and crun when scaling up the number of concurrent runs. Runsc's performance loss with 40 concurrent runs vs. 1 isolated run, however, was less than runc. Results for scale, add, and triad operations were similar to copy. For stream we infer the following order for memory isolation: runsc > crun > runc.

B4 - Sysbench CPU

We observed the following results when running sysbench-CPU across our container runtimes:

Container Runtime	Avg. Event Exec Time in sec		Runtime Increase
	Isolated Run	40 Parallel Runs	
runc	1.00x	1.51x	~ +51%
runsc	1.02x	1.54x	~ +50%
crun	1.02x	1.25x	~ +25%

Table 5: Sysbench Prime Number Generation runtime comparison for an isolated run and 40 concurrent runs. Results are normalized to the isolated (1 run) runtime of runc (x=150.34 seconds) as a baseline. (Lower is better)

Based on figure 5 and table 5, for an isolated run, runsc and crun performed poorly compared to runc. However, for the 40 concurrent runs, crun outperformed runsc and runc. For Sysbench CPU, we infer the following order for CPU isolation: crun > runc > runsc.

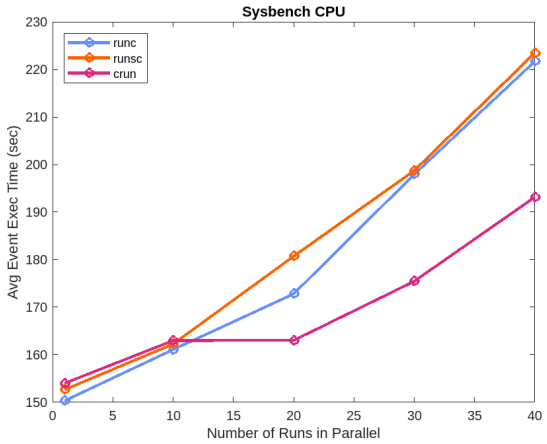

Figure 5: Sysbench-CPU Benchmark Results

B5 - Sysbench Memory

We observed the following results when running sysbench-memory across our container runtimes:

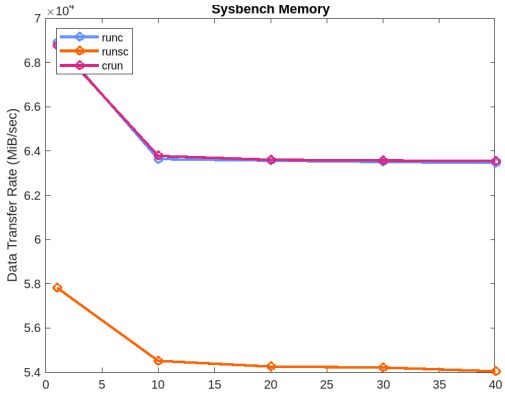

Figure 6: Sysbench-memory Benchmark Results

Container Runtime	Data Transfer Rate in MiB/sec		Throughput Loss
	Isolated Run	40 Parallel Runs	
runc	1.00x	0.92x	~ -8%
runsc	0.84x	0.78x	~ -6%
crun	1.00x	0.92x	~ -7%

Table 6: Sysbench Memory throughput comparison for an isolated run and 40 concurrent runs. Results are normalized to the isolated (1 run) throughput of runc (x=68,903 MiB/sec) as a baseline. (Higher is better)

Runsc performed poorly compared to runc and crun across all configurations as shown in figure 6 and table 6. Runsc, however, had slightly less performance loss at 40 concurrent runs vs. runc and crun. We observed the following order of memory isolation: runsc > crun > runc.

B6 - Y Cruncher

After running the Y-Cruncher benchmark across our container runtimes, we observed the results shown in table 7 and figure 7.

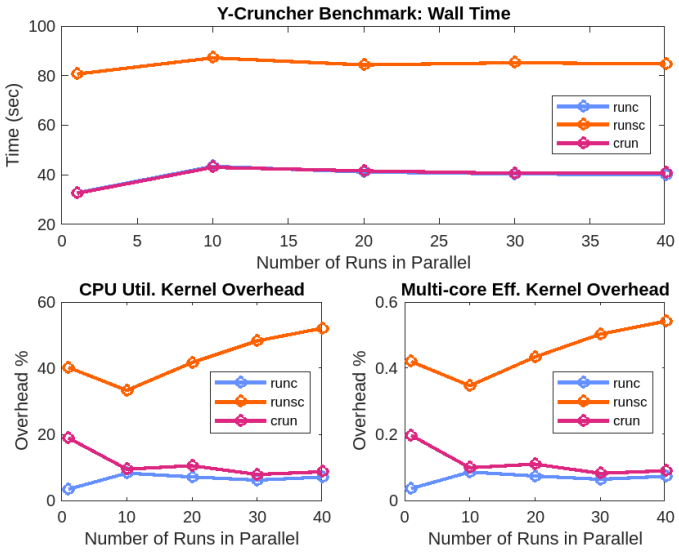

Figure 7: Y-Cruncher Benchmark Results

Container Runtime	Execution Time in sec		Runtime Increase
	Isolated Run	40 Parallel Runs	
runc	1.00x	1.22x	~ +22%
runsc	2.46x	2.59x	~ +5%
crun	0.99x	1.24x	~ +25%

Table 7: Y-Cruncher runtime comparison for an isolated run and 40 concurrent runs. Results are normalized to the isolated (1 run) runtime of runc (x=32.77 sec) as a baseline. (Lower is better)

For y-cruncher, runsc performed twice as poorly as runc and crun. Though net throughput across different configurations was poor for gVisor (runsc), the performance loss when scaling up to 40 concurrent runs was nearly five times less than runc and crun. For y-cruncher we infer the following order for memory isolation: runsc > runc > crun.

5 CONCLUSION

From our experiments, we observed performance of CPU and memory benchmarks on runsc, the container runtime employed by gvisor, were consistently less than runc and crun. For all benchmarks, runsc had lower throughput and/or longer runtime. Worst case performance was for y-cruncher where runtime in isolation was 2.46x, and in parallel 2.59x compared to runtime in isolation on runc. Y-cruncher performance with 40 parallel instances only degraded 5% on runsc suggesting another bottleneck other than parallel instances restricted performance.

The throughput of stream copy operations dropped over 60% for all container runtimes suggesting container isolation is poor for this memory operation. Prime number generation runtime with sysbench increased over 50% on runc and runsc, whereas crun increased only 25% when scaling up from 1 to 40 parallel instances. Here optimizations provided by crun were very helpful to deliver better runtime. Overall crun had less performance degradation compared to runc, the default container runtime, for all benchmarks except y-cruncher. Crun featured a performance loss less than runc of: 1% linpack, .01% noploop, 2% stream, 26% sysbench-cpu, and 1% sysbench-mem.

If containerization offered perfect isolation of resources then we should not see ***any*** loss of throughput or increase in runtime for benchmarks run in parallel vs. isolation. However, we observed performance degradation for every benchmark.

REFERENCES

[1] Hussain AlJahdali, Abdulaziz Albatli, Peter Garraghan, Paul Townend, Lydia Lau, and Jie Xu. 2014. Multi-tenancy in Cloud Computing. In *2014 IEEE 8th International Symposium on Service Oriented System Engineering*. 344–351. https://doi.org/10.1109/SOSE.2014.50

[2] Anjali, Tyler Caraza-Harter, and Michael M. Swift. 2020. Blending Containers and Virtual Machines: A Study of Firecracker and GVisor. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Lausanne, Switzerland) *(VEE '20)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/3381052.3381315

[3] Fredrik Björklund. 2021. A comparison between native and secure runtimes : Using Podman to compare crun and Kata Containers. , 26 pages.

[4] Giuseppe Scrivano Dan Walsh, Valentin Rothberg. 2020. An introduction to crun, a fast and low-memory footprint container runtime. https://www.redhat.com/sysadmin/introduction-crun.

[5] Guillaume Everarts de Velp, Etienne Rivière, and Ramin Sadre. 2021. Understanding the Performance of Container Execution Environments. In *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds* (Delft, Netherlands) *(WOC'20)*. Association for Computing Machinery, New York, NY, USA, 37–42. https://doi.org/10.1145/3429885.3429967

[6] Jack Dongarra, Piotr Luszczek, and Antoine Petitet. 2003. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience* 15 (08 2003), 803–820. https://doi.org/10.1002/cpe.728

[7] Lennart Espe, Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. 2020. Performance Evaluation of Container Runtimes.. In *CLOSER*. 273–281.

[8] Google. 2013. gVisor. https://github.com/google/gvisor.

[9] Brendan Gregg. 2014. Noploop Benchmark. https://www.brendangregg.com/blog/2014-04-26/the-noploop-cpu-benchmark.html.

[10] Xinlei Han, Raymond Schooley, Delvin Mackenzie, Olaf David, and Wes J. Lloyd. 2020. Characterizing Public Cloud Resource Contention to Support Virtual Machine Co-residency Prediction. In *2020 IEEE International Conference on Cloud Engineering (IC2E)*. 162–172. https://doi.org/10.1109/IC2E48712.2020.00024

[11] Varik Hoang, Ling-Hong Hung, David Perez, Huazeng Deng, Raymond Schooley, Niharika Arumilli, Ka Yee Yeung, and Wes Lloyd. 2023. Container Profiler: Profiling resource utilization of containerized big data pipelines. *GigaScience* 12 (08 2023), giad069. https://doi.org/10.1093/gigascience/giad069

[12] Varik Hoang, Ling-Hong Hung, David Perez, Huazeng Deng, Raymond Schooley, Niharika Arumilli, Ka Yee Yeung, and Wes Lloyd. 2023. Container Profiler Source Code. https://github.com/wlloyduw/ContainerProfiler/.

[13] Alexey Kopytov. 2017. Sysbench Benchmark. github.com/akopytov/sysbench.

[14] Rakesh Kumar and B Thangaraju. 2020. Performance Analysis Between RunC and Kata Container Runtime. In *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. 1–4. https://doi.org/10.1109/CONECCT50063.2020.9198653

[15] Ilias Mavridis and Helen Karatza. 2021. Orchestrated sandboxed containers, unikernels, and virtual machines for isolation-enhanced multitenant workloads and serverless computing in cloud. *Concurrency and Computation: Practice and Experience* 35 (05 2021). https://doi.org/10.1002/cpe.6365

[16] John D McCalpin et al. 1995. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter* 2, 19-25 (1995).

[17] Roberto Morabito, Jimmy Kjällman, and Miika Komu. 2015. Hypervisors vs. Lightweight Virtualization: A Performance Comparison. In *2015 IEEE International Conference on Cloud Engineering*. 386–393. https://doi.org/10.1109/IC2E.2015.74

[18] Claus Pahl. 2015. Containerization and the PaaS Cloud. *IEEE Cloud Computing* 2, 3 (2015), 24–31. https://doi.org/10.1109/MCC.2015.51

[19] Redhat. 2023. crun Source Code. https://github.com/containers/crun.

[20] Xu Wang. 2018. Kata Containers and gVisor: a Quantitative Comparison. https://www.openstack.org/summit/berlin-2018/summit-schedule/events/22097/kata-containers-and-gvisor-a-quantitative-comparison.

[21] Xingyu Wang, Junzhao Du, and Hui Liu. 2022. Performance and isolation analysis of RunC, gVisor and Kata Containers runtimes. *Cluster Computing* 25 (04 2022), 1–17. https://doi.org/10.1007/s10586-021-03517-8

[22] Alexander J. Yee. 2023. Y-cruncher Doc. numberworld.org/y-cruncher/.

[23] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. USENIX, Renton, WA.