Dockerust: A Fast, Safe and Resource-friendly Container Runtime

Youhan Wu, Zhaocheng Huang
Rice University

Abstract

Container runtimes are the underlying components of Docker, Podman and other container engines. It mostly utilized OS-level virtualization features to create and run a container process. Most of the popular runtimes are written in Go - as it is the implementation language of Docker. However, after investigation of Go-based runtimes, we argue that Go may be not the ideal language for this task due to its expensive and incomplete syscall support, as well as its heavyweight language runtime. What we need is a system language, not a language of the cloud. Since the problem mostly lies in the implementation language itself, we built a container runtime in Rust - a modern system language, to explore the potential of being a better alternative. We implemented a micro-runtime with key functionalities, including namespace setup, IPC, rootfs mount, etc., and ran an a set of benchmark testing against other popular runtimes. As illustrated in the evaluation part, our implementation, Dockerust, achieves 29X performance leverage and 45.4% less memory usage. It strikes the balance between performance, security and cost, making it the perfect fit for general-purpose scenarios.

1 Introduction

Container-based virtualization is really popular in recent years. Thanks to the advent of Docker, container technology has subverted the traditional systems development life cycle and refactored the way people build and deploy software. Docker introduces an underlying container runtime to help start, terminate container processes and multiplex system resources. It is later extracted out as a separate open source tool called *runc*. Runc has since become the most popular container runtime as the underlying support for many high-level container engines.

As a low level component, a container runtime talks directly to the host OS, and utilizes OS-level virtualization features, including namespaces, cgroups, VIF (virtual network interfaces), etc. This process involves a lot of syscalls, such as read/write, exec, fork and so on. However, many existing container runtimes like runc, gVisor are written in Go, which introduces overhead to syscalls and resources utilization [1] [2]. Besides, unlike other system languages, Go has only limited system call bindings, leaving a number of Linux ABI unimplemented. It is both unhandy and unsafe for developers to fill the implementation gap. [3] In addition to the incomplete and expensive syscall support, Go-based container runtimes suffer from a performance cost due to the language runtime. As we covered in the evaluation part, Go's GC and pre-allocated thread pool takes up a significant amount of the execution time. From these perspectives of view, Go might not be the best programming language for this task.

There do exist some alternative container runtime written in system languages with relatively lower overhead. In 2017, Red Hat introduced crun to replace runc. However, C is not recognized as a memory safe language. It can expose too many vulnerabilities due to its inherent defect. One simple memory-related vulnerability may cause huge economic losses to container users.

Since most of the issues fall in the implementation languages, we implemented an OCI-compliant runtime in Rust to explore the potential of being faster, safer and more resource-friendly. Compared with other popular runtime engines, Dockerust is faster than runc due to its built-in binding for raw system calls, and requires fewer system resources without GC. Moreover, it is much safer than crun with restrict compile-time type checking and data race detection. To sum up, Dockerust strikes the balance between performance, security and cost, which

makes it the perfect fit for general-purpose scenarios, and has an edge in resource-limited platforms, such as single-board computers.

We performed a benchmark analysis on the lifetime of the above mentioned runtimes from the performance and resource usage aspects. And to further confirm our conclusions, we created and started an accessible nginx server image respectively, and run a latency test against the server to compare the performance of the two runtimes.

2 Motivation

The biggest issue with runc is every time we use it to run a container, it has to call a C-based program to help it set up the container environment due to its lack of some necessary Linux syscall support. That is to say, some of virtualization-related Linux ABI bindings have not been implemented in Go's standard library. The control will go back to runc only after this helper program exits. This is non-acceptable as the foreign function call in Go is rather expensive according to dyu's experiment [4].

For other Go's implemented syscalls, our experiment shows that they take more time than the implementations in Rust. We wrote a benchmark program to compare the performance of syscall bindings from Rust and Go's standard library. To more details, we performed getppid for 1,000,000 times, mkdir/rmdir for 10,000 times, mount/unmount for 200,000 times, chroot for 200,000 times and dup for 200,000 times, all of which are quite common syscalls in initializing and running a container. Then we tested the two programs on Github codespace. We compiled the two programs using official compilers. We allowed default optimization of the Go compiler and set the build target of the Rust compiler to release for best performance. We tested each program 10 times and recorded the running time in each run. We also recorded the memory usage of Getppid in a separate run using Valgrind. The result is shown in Table 1 and 2 and Figure 1.

The result shows that the running time of the Rust program is 11.16% less than the Go program on average in the five scenarios, and the average memory usage of the Rust Getppid program is 5.13% less than the Go program. Therefore, we believe Rust has the potential of being faster and more memory-saving when executing tasks that require a lot of system calls, and we believe Rust will be suitable for our container runtime.

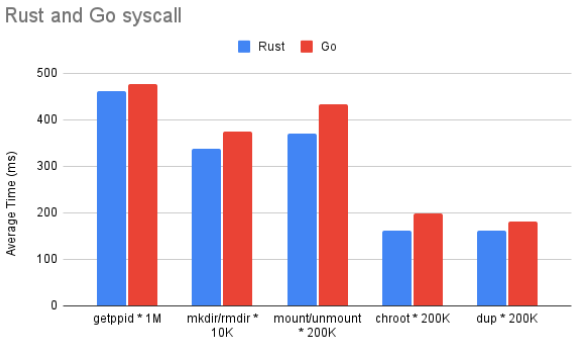

Figure 1: Running time of the two demo programs.

Memory Usage (B)	Average	Max	Min
Rust	2960.096386	7656	312
Go	3120.177778	62944	2448

Table 1: Memory usage of the two demo programs.

Average time (ms)	Rust	Go	Rust faster
getppid * 1M	461.5	477.6	3.37%
mkdir/rmdir * 10K	338.3	373.9	9.52%
mount/unmount * 200K	369.5	432.5	14.57%
chroot * 200K	161.5	198.5	18.64%
dup * 200K	162.8	180.3	9.71%

Table 2: Running time of the two demo programs.

3 Background

Docker is one of the most popular technologies for containerization. Instead of boosting the whole system environment in VMs, Docker utilizes the Linux kernel's resource isolation and constraint features to share a kernel across containers. Docker can create isolated environment for containers in different levels (e.g., network, PIDs, UIDs, IPC) with support of Linux namespaces [5]. Besides, it can control resource allocation and limitation among containers using cgroups.

Rust is a programming language that emphasizes performance and memory safety. Rust aims to be as efficient as C++. It doesn't perform garbage collection, and it has features like zero-cost abstractions and inline expansion to optimize the compiled code. Rust is designed to be memory safe. It does not permit null pointers, dangling pointers, or data races, and safety and validity of the underlying pointers is checked at compile time. Rust has been growing fast in recent years, and has been widely used in many large projects. It is reported that Rust is incorporated into the forthcoming 6.1 Linux kernel, which is an important milestone for Rust, and also a good signal for our project.

With official support from system level, and the brilliant features from the language level, a new container runtime written in Rust will be a better alternative to existing runtimes.

4 Methods and Plans

We concluded the minimal modules a runtime should include in the following sections. This project is not ready for production yet but is sufficient enough as a prototype to verify our assumptions.

4.1 Namespaces Setup

In Linux context, a container is just a forked process. It's nothing fancy about it, expect it is running in an isolated space. But how is it isolated from the rest of the system? The answer is namespace.

Namespaces are a feature of the Linux kernel that provide the logical isolation of resources for processes running in different sets of namespaces. There are different types of namespaces, including IPC, Network, Mount, PID, User, UTS, etc.

Linux provides the *CLONE(2)* syscall to fork a child process and allow it to create or enter new namespaces, as long as the namespace flags are specified.

4.2 Pipe-based IPC

Till now, we have successfully spawn a child form the parent process. Although the child is running in an isolated namespace, we still need a way to enable the communication between the child and parent. For example, the child process needs to notify the parent whether the environment is set up and if it is ready to execute the user-input commands. Likewise, the parent process should tell the child process whether it is terminated or a new data volume is mounted.

In the case of spawning a new thread, it is a very common way to create a in-memory channel to allow IPC as Rust has a good support for it. In our case, this is not possible, as it is cloning a new process with its own separate memory space. In our implementation, we used the anonymous pipe to establish a IPC channel. When the parent process runs the *PIPE(2)* syscall, a set of read/write fd is created. At the time of the clone call, by default, the child process will inherit a copy of the parent's file descriptor table, thus allowing the child process to read from or write to the pipe.

4.3 OverlayFS-based Storage Driver

Containers get their root filesystem from a container image file which is usually pulled from a registry (eg., Docker Hub). This file is, essentially, a TAR archive with the root filesystem and some image metadata. The image layers are stored in some special storage drivers that are optimized for space efficiency. Among these storage methods, Overlay2 is the most preferred one due to its performance advantages. Overlay2 is now the default storage driver for Docker and Podman, and has been in the Linux kernel since 3.18.

To enable OverlayFS support for a container, we can easily use *MOUNT(2)* syscall to create a mount point with the overlay filesystem type, and unpack the container image to a read-only directory that is associated with the mount point. Then we can see the image's rootfs in the mount point and any changes made will go to a separate associated directory.

4.4 Pivot rootfs

Since the image's rootfs is mounted, we are ready to switch the container process's rootfs from the host's rootfs to the mounted one through *PIVOT_ROOT(2)* syscall.

pivot_root performs a *chroot*-like action, but in a cleaner way - it allows user to switch to the new root

mount and unmount the the old root mount more easily. It takes two arguments: new_root and put_old. To switch to the image's rootfs, first, make a directory for put_old and then pivot into the intended rootfs, and finally, unmount the put_old. In such a manner, we can get rid of the dependency of the old root mount.

4.5 Virtual Network

As the child process enters a new network namespace, it cannot use the network devices on the host system directly. To connect the container to the Internet, we created a pair of veth devices, put one in the host network namespace, and put the other in the child network namespace. Then, we set up iptables rules on the host to allow forwarding, and set IP addresses and routes for the two veth devices. After that, the host and container can access each other through their IP addresses.

5 Protection Mechanism

The Dockerust is running in privileged mode, so extra protections must be applied to mitigate privilege escalation attacks.

5.1 Seal the executing program

A malicious container can overwrite the dockerust runtime binary and thus gain root-level code execution on the host.

When the user image is run, it must be the init process with PID equals 1 in its own procfs. In our implementation, when the user enters *dockerust run <image>*, dockerust creates a *dockerust init <image>* subprocess in a namespace that is already set up. Then, dockerust init the image, by calling *execv* syscall to overite itself with the user-provided image binary.

However, if an attacker pass */proc/self/exe* as the image url to *dockerust run <image>* command, the attacker will have root access to the */proc/[dockerust-pid]*. By using it as a reference to the dockerust binary and overwrite it, the attacker can achieve code execution on the host as root.

This vulnerability is known as CVE-2019-5736, with a severity score of 8.6 [6]. To prevent this attack, we followed LXC's patch commit [7] to create a temporary copy of the calling binary itself when it starts. To do this, We called *MEMFD_CREATE(2)* syscall to hide the Dockerust binary file from an anonymous, temporary file. Consequently, */proc/[runc-pid]/exe* now points to the temporary file, and the original binary can't be reached from within the container.

6 Evaluation and Results

We evaluated the total lifetime of the Docker and Dockerust container to see the performance of creating and destroying a container, and the round trip latency of HTTP request into both Docker and Dockerust nginx containers to see the performance when the container is running.

6.1 Lifetime performance

We used Docker shipped with Github codespace, and compiled Dockerust on Github codespace with release flag. We let Docker download the busybox container and saved the container to a tar image file to be used by Dockerust. Then we called the two programs to create a container from busybox image, enter the container, execute /bin/ls inside, output the result on screen, exit the container and destroy the container. We tested each program 10 times and recorded the running time in each run. We then recorded the memory usage in a separate run using Valgrind. We also recorded the time consumed by each procedure in a separate run using perf, and visualized two flame graphs on speedscope. The result is shown in Table 3 and 4 and Figure 2 and 3.

The result shows that the running time of Dockerust is 29 times less than Docker on average, and the memory usage of Dockerust is 45.4% less than Docker on average. That means Dockerust can greatly speed up the performance and use less memory resource for use cases when the procedure program is short and containers need to be created and run frequently.

In the flame graph, we noticed that the two most time consuming operations in Dockerust are write and wait syscalls, and both of them takes 23% in the total running time respectively. They happened at the second half of the program, and from their name we can guess they are related to writing the ls result on screen and waiting for child container process to exit. The most time consuming in Docker are finish_task_switch and futex_cleanup, and both of them takes 38% in the total running time respectively. They happened at the last 75% of the program, and from their name we can guess they are related to multithreading.

6.2 HTTP performance

In the same environment, we let Docker download the nginx container and saved the container to a tar

Running time (s)	Average	Max	Min
Dockerust	0.03	0.033	0.027
Docker	0.883	1.014	0.841

Table 3: Lifetime of Docker and Dockerust.

Memory Usage (B)	Average	Max	Min
Dockerust	30640.32877	88408	400
Docker	56131.66667	175272	520

Table 4: Memory usage of Docker and Dockerust.

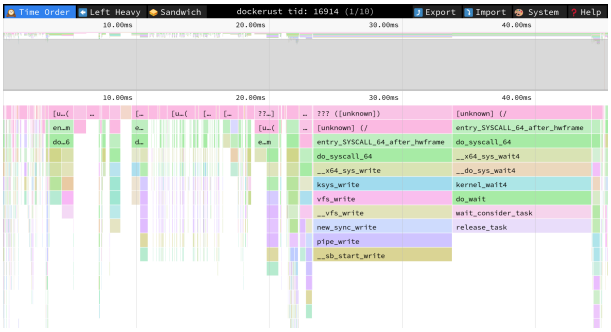

Figure 2: Perf flame graph of Dockerust.

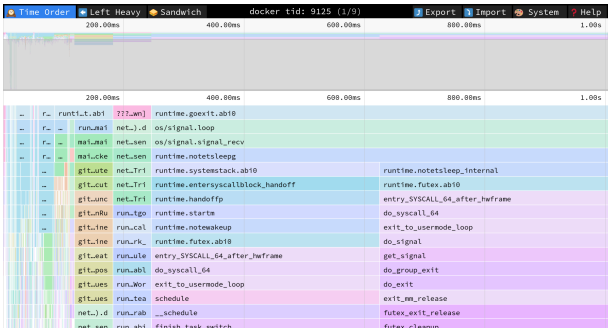

Figure 3: Perf flame graph of Docker.

RTT (ms)	Average	Max	Min
Dockerust	0.494	0.37	0.7
Docker	1.148	0.74	2.06

Table 5: Round trip time of HTTP GET to nginx instance in Docker and Dockerust.

image file to be used by Dockerust. Then we called the two programs to create a container from nginx image, enter the container, and start nginx. We used httping to request HTTP GET into the two containers 10 times and recorded the round trip time in each run. The result is shown in Table 5.

The result shows that the RTT to Dockerust is 56.9% less than Docker on average. That means Dockerust can greatly speed up the performance of services running inside the containers.

7 Discussion and Lessons Learned

We are trying to find out the reason behind the huge difference in performance between Docker and Dockerust. We only implemented key components of a container, including namespace, pipe IPC, OverlayFS storage, pivot rootfs and virtual network. The remaining part like detach mode, volume, cgroup setting and port mapping are left for users, and these features may take a reasonable time in Docker to set up, regardless whether the application really needs them. In the flame graph of Docker, we found that Docker spent last 75% time on multithreading. Since the content of the two images used by Docker and Dockerust are the same, we suspect Docker does more work in the child container process after the container is terminated. We also noticed that runtime.mallocgc of Docker took 24.06ms (2.4%), which is 80% compared to the total running lifetime of Dockerust. It shows that languages without garbage collection like Rust have advantage in running time performance when creating and destroying container.

8 Conclusion

We built a fast, safe and resource-friendly container runtime based on Rust. We utilized some safety features of Rust, and our evaluation on Docker and Dockerust shows that Dockerust runs faster and uses less memory. The flame graphs or Docker and Dockerust give us a good clue to explore the reason Dockerust is more efficient than Docker. In the future, we plan to implement more features to make it more usable for more people, and perform more benchmarks from performance, resource usage, and security aspects.

References

[1] Ethan G Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The true cost of containing: A {gVisor} case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[2] Emiliano Casalicchio and Vanessa Perciballi. Measuring docker performance: What a mess!!! In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 11–16, 2017.

[3] The Go Authors. go/sys/master/./unix/syscall_linux.go. *https://go.googlesource.com/sys/+/master/unix/syscall_linux.go*, 2022.

[4] dyu. dyu/ffi-overhead: comparing the c ffi (foreign function interface) overhead on various programming languages. *https://github.com/dyu/ffi-overhead*, 2021.

[5] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[6] NIST. Cve-2019-5736. *https://nvd.nist.gov/vuln/detail/CVE-2019-5736*, 2019.

[7] Serge Hallyn Christian Brauner, Alesa Sarai. Cve-2019-5736 (runc): rexec callers as memfd. *https://github.com/lxc/lxc/commit/6400238d08cdf1ca20d49bafb85f4e224348bf9d*, 2019.