Analysing Performance of Docker-slim

Erfan Mehraban *erfan.mehraban@gmail.com*

Abstract

Containerized applications are lightweight virtualization environments that run applications that have been packaged with their resources and configuration information. This ensures that they can be deployed across a wide variety of virtualization platforms. Containers are easy to create, but their ease is often a double-edged sword, encouraging the packaging of applications that shouldn't be together, as well as the inclusion of unnecessary components. These practices needlessly increase the container's size, sometimes by an order of magnitude. Additionally, they decrease overall security because each component, regardless of whether it is necessary or not, may introduce its own security risks.
In this report, I study `docker-slim`, which is an open sourced software used to reduce the size of images. Also, slim images were benchmarked against the original image to see how much memory and processing overhead they had.

1 Introduction

`docker-slim` is a tool that simplifies and optimizes the experience of working with containers for developers. `docker-slim` reduces the size and increases the security of developer containers. Using various analysis techniques, Docker-slim optimizes and secures containers by understanding the developer's application. Container's attack surface will be reduced since docker-slim discard what application don't need. in general `docker-slim` inject some sensors to support multiple monitors like `ptrace`, `fanotify`, etc to running original fat image. then by probing running container like executing command or sending http request montoring file acess and process runs. after recording these file accesses and syscalls and applying some heuristics like keeping ssl files or shells, it build slim image and generate other reports and additional artifacts. so the final image contains only files which has been accessed.

2 Process

General process to build and test slim image contain 3 step:

1. I started by looking for a suitable benchmarking tool. A benchmark is the process of evaluating an application's performance by executing a computer program, a set of programs, or some other operation. This is done normally by executing a number of standard tests and trials against it. Benchmark tools generate end-to-end tests covering the entire functionality of a target software. As a result, a benchmark test is performed on a software enforce runtime program to cover mostly all code, especially concurrency management components.

2. As mentioned, `docker-slim` sensors rely on file access to determine which dependency is bloated and which one is necessary. As a result, using a light configuration of a benchmark tool as a `docker-slim` probe is a smart idea. Simulating a production environment by loading software heavily in a test environment is most effective. Therefore, `docker-slim` sensors can find and track all components of production software. In this step, a slim image is created.

3. After the slim image has been built, two versions of `docker-compose` files are written to apply high pressure to the service with the same benchmark tool, one for the original image and one for the

slim image. `Dstat` captures CPU and memory usage simultaneously. `Dstat` is a tool that is used to retrieve information or statistics form components of the system such as network connections, IO devices, or CPU, etc. `Dstat` outputs aggregates after the benchmark is completed and renders charts based on that. Host environment specifications are presented in Table 1.

It is important to note that it was not a wise idea to use unit test or manual test as the probe. First of all, unit tests only test internal components and functionality, which does not guarantee their use by the user under production conditions. Second, manual tests cannot guarantee full coverage because it was time consuming to write every scenario for general images.

In running benchmarks, these two metrics of the system are mostly measured:

Memory RAM usage plus Swap reserved by all process in system. Hence, the key metric here is the change in memory usage, not the total amount of memory accessed.

CPU usage measured in percent of all core usage (which is equal to the sum user usage and waited tasks)

All codes including benchmarking apps and analysis scripts, can be accessed at `https://github.com/erfan-mehraban/docker-slim-experiment`.

2.1 Redis

Redis is an in-memory data structure store, used as a distributed, in-memory key–value database, cache and message broker, with optional durability.

First attempt was to use `docker-compose` to start up 2 containers on the same network, one for the primary redis database and another with the same image to benchmark redis. It was done in order to prevent injected sensors from detecting redis benchmark dependencies as a primary redis artifact. In early stages and as an experimental feature, `docker-compose` is supported only for reading properties of containers from `docker-compose` files, not for running all services in compose file in both probe and target containers at the same time and especially on the same network.
In the end, after trying several parameters and arguments, I gave up using this feature and used the exec command to run redis-benchmark from the main redis server container. So build script changed to use internal `exec` command to run benchmark alongside server.
Benchmarking original and slim version of redis image doesn't show any significant difference in using memory or CPU.

2.2 Nginx

Nginx is a web server that can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache.

Default `docker-slim` http probe running GET / with HTTP and then HTTPS on every exposed port. Nginx exposes port 80 and serves the default welcome page. As a result, testing the default page load is sufficient for testing nginx and benchmarking nginx performance does not cover any of its special features.

For the Nginx benchmark, requesting the default page 1000 times concurrently will put a heavy load on the web server. CPU usage statistics don't show any overhead, but memory usage shows a small difference of about 10% in the initial load. However, After 8 seconds, the usage of both images converges.

Table 1: Host system environment

OS	Ubuntu 18.04.6 LTS
CPU	12th Gen Intel Core i5-12400
Memory	16GiB RAM DIMM1 DDR4 Synchronous 2666 MT/s 64 bits (+ 2GiB Linux swap)

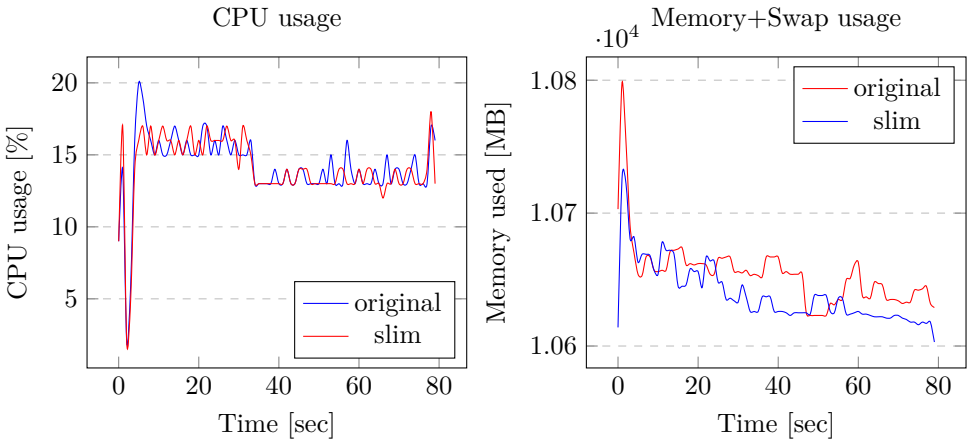

Figure 1: System CPU and memory usage for redis

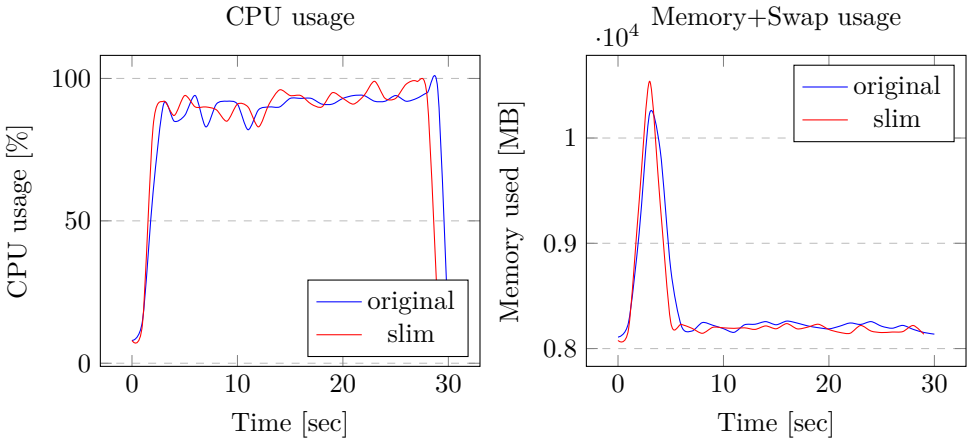

Figure 2: System CPU and memory usage for nginx

2.3 PostgreSQL

PostgreSQL, also known as Postgres, is a free and open-source relational database management system emphasizing extensibility and SQL compliance.

`pgbench` is a benchmarking tool written for Postgres. It puts a configurable load on the database and compares results to ensure consistency. By default, `pgbench` tests a scenario that is loosely based on TPC-B, involving five SELECT, UPDATE, and INSERT commands per transaction.
Postgres stores persisted database records on local files. `docker-slim` allows you to mount a directory and exclude it during the build process of the minified image. However, contrary to this option, `docker-slim` doesn't behave as expected and doesn't exclude the data directory. In order to follow up this issue on the project, I opened an issue in the `docker-slim` repository . Therefore, as a result of this problem, the built image does not shrink, but instead grows. Another bug in the artifact sensor causes the main binary file of Postgres software to be deleted, so benchmarking the slim image is not possible.

2.4 MongoDB

MongoDB is a source-available cross-platform document-oriented database program. Classified as a NoSQL database program, MongoDB uses JSON-like documents with optional schemas.

`mongo-perf` is a tool to test and benchmark MongoDB, and the project has been open sourced under the official MongoDB group. Since the official Docker file was outdated, I opened a merge request to update it . After changing the docker file, I build it locally and choose a wide range of test cases. The same problem is observed with all test cases as with the Postgres bug discussed in the last section. Upon `mongo-perf` completing its job, a USR1 signal was sent to `docker-slim`, which terminated sensors and built the image. Therefore, benchmarking and the built process probe were identical. While both images used the same CPU, the memory usage of the original image was around 5% higher.

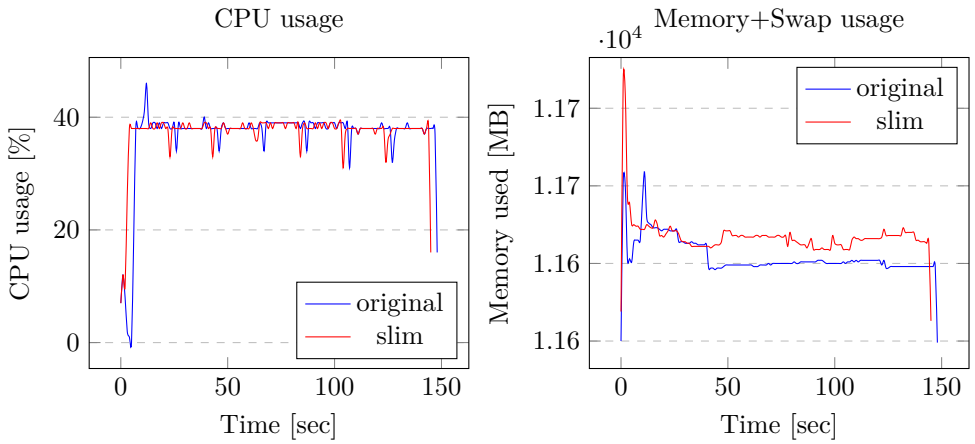

Figure 3: System CPU and memory usage for mongo

2.5 memcached

Memcached is a general-purpose distributed memory-caching system.

To profile `memcached`, memtier_benchmark was used which is a benchmarking tool for the redis group and supports both binary and text protocol of memcached. This tool's default configuration was sufficient for probing. There was no noticeable difference in CPU usage, but the performance of the slim image in memory usage was 4% higher than the original.

3 Result

Overview of results is shown in table 2. In summary, `docker-slim` correctly minifies images for simple applications which don't store data in local storage, but there is a tiny overhead in memory usage. In general, this tool can be used for production, but sufficient tests must be performed to ensure that slim images are operating correctly.

4 Related works

Another tool for debloating container is `cimplifier` which has small difference in sensor implementation.(Rastogi et al., 2017) Also there is some tools for debloating software dependencies before containerization or in the process of compiling. (Bruce et al., 2020)

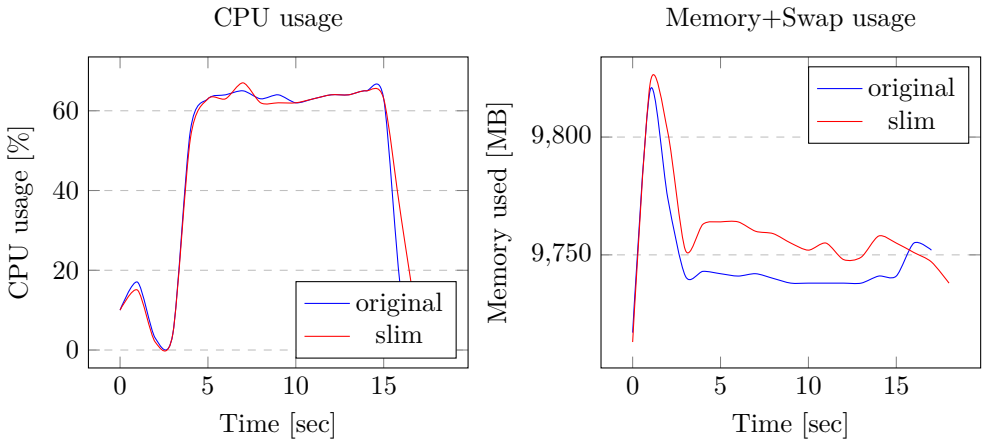

Figure 4: System CPU and memory usage for memcached

Table 2: Overview of results

Image name	Minified By	Original Size	Minified Size	Memory Overhead
`memcached:1.6.15`	11.54x	89 MB	7.7 MB	4%
`postgres:14.3-alpine`	0.91x	189 MB	207 MB	-
`redis:7.0.0`	3.5x	117 MB	33 MB	Nothing significant
`nginx:1.22.0`	13x	142 MB	11 MB	10% at initial
`mongo:5.0.8`	2.83x	690 MB	243 MB	5%

References

Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. JShrink: in-depth investigation into debloating modern java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, nov 2020. doi: 10.1145/3368089.3409738. URL `https://doi.org/10.1145%2F3368089.3409738`.

Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, aug 2017. doi: 10.1145/3106237.3106271. URL `https://doi.org/10.1145%2F3106237.3106271`.