University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

REPORT 1

Minimizing Dockerfiles

Scientific Adviser:
Prof. Răzvan Deaconescu

Author:
Andrei Petrea

Bucharest, 2025

I want first and foremost to thank my advisor, Prof. Răzvan Deaconescu, for his guidance and support throughout this project. His expertise and insights have been invaluable in shaping my understanding of the subject matter and in helping me navigate the challenges I faced during this endeavor.

I also want to thank my friends and family without whom these 4 years would have been so much harder.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

"But ... it works on my machine..." is a phrase no developer wants to hear. It is a phrase that highlights both the frustration of software engineers and the complexity of the software development life cycle. The most common cause for this issue arising lies in the mismatch between the developer's local environment and the production environment where the code is run [13]. On traditional deployments on a physical server or virtual machine, it is up to the developer to ensure that all the necessary configurations are met for the application to run, which can be a tedious process, especially when the the number of dependencies is high or when we want to upgrade to a newer version.

The solution? ... Using containers. Containers are software that package up code, runtime, libraries, config files, everything needed for an application to run [11] which can then be executed across different machines without additional setup. This, alongside faster boot times, lower resource usage and ease of scale, especially when coupled with orchestration tools like Kubernetes, made containers grow in popularity, even surpassing virtual machines in some cases [18].

However, the comparison with virtual machines, where containers are usually much better, often paints the picture that these metrics are irrelevant when it pertains to them, one such metric being size of the container itself. This is not true. The size of the container is directly proportional to the number of binaries and packages that are installed and as such reducing the size has the following benefits [7]:

- **Security** - smaller the size, smaller the attack surface a hacker has to work with
- **Performance and Efficiency** - smaller images are faster to deploy and, in general, use fewer system resources
- **Maintainability** - smaller images have fewer dependencies, making it easer to maintain and update them

As such, the goal of my project is to create a tool than can strip a container of all its unnecessary files and packages, leaving only the files needed for the application to run properly and export it as a *Dockerfile*, the recipe used to create the container.

Chapter 2

Background

2.1 A brief history of containers

The idea of containerization is not a new one. The concept has it's roots since the late 70s, with *chroot*, a Unix command that allows a process to change its root directory, effectively isolating it from the rest of the system [3], creating so called *chroot jails*. Over the decades, this concept grew and evolved, with the introduction of *FreeBSD jails* and *Solaris Zones* around the turn of the millennium adding support for multiple isolated environments within the same OS instance [3].

The next major milestone happened in 2008 with the introduction of *LXC* (Linux Containers), adding kernel-level support for containers, by leveraging two Linux kernel components: *cgroups*, which provides ways to group processes in order to better manage resources and *namespaces*, which provide isolation.[3]

In 2013, the launch of *Docker* radically changed the landscape of containerization by providing a developer-friendly way to create, manage and deploy containers. Docker introduced the concept of container images, files which store the data needed for the container to run, *Docker Hub*, a public repository for sharing container images, and a powerful command-line interface for managing containers [3]. In the years that followed, multiple platforms sprung up such as *Apptainer* and *Podman*, which are both open-source alternatives to Docker, but none of them managed to gain the same level of popularity. As such, Docker became synonymous with containerization, holding an overwhelming market share of *87.85%*[1] thus cementing themselves as the de facto standard in the industry.

2.2 The anatomy of a container

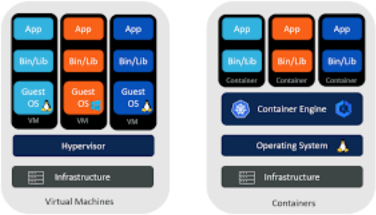

Figure 2.1: Containers vs VMs

As opposed to virtual machines, which run a full operating system on top of a hypervisor, containers share the host operating system and run as isolated processes making them much more lightweight and efficient by only having to package the application and its dependencies into a so-called container image. Currently, there are over 10000 container images available on Docker Hub, ranging from simple *hello-world* apps, to *databases* and even full-blown Linux distributions like *Ubuntu*.

Now, how do we create our own container image? By creating a *Dockerfile*, a simple text file that contains all the necessary steps to build the image.

```
FROM python:3.10.0-slim
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
CMD ["python", "adapter.py"]
```

Listing 2.1: Sample Dockerfile

The format of a Dockerfile is quite simple, with each line representing a command that will be executed in order, the specifications for what each command does can be found in the official documentation[1] As each command is executed, a new layer is applied on top of the previous one, the container image being represented as a stack of these layers. A layer represents a change to the base image layer such as adding or removing files, installing packages or changing environment variables.

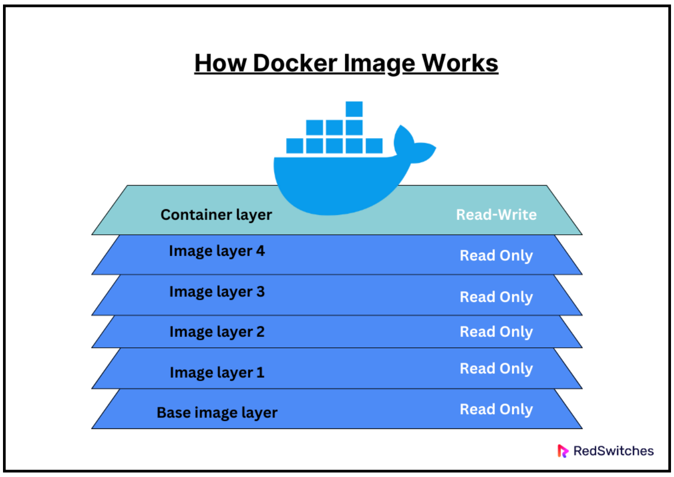

Figure 2.2: Container layers

All Dockerfiles require a base image, which is specified by the keyword *FROM*. The base image can be represented by either an existing image, or if you want to create your own, you can use ***FROM** scratch* to start from an empty filesystem. Additionally, Docker supports building the container image in multiple stages, by using the *FROM* command multiple times in the same Dockerfile.

```
    FROM alpine:latest AS builder
    RUN apk --no-cache add build-base

    FROM builder AS build1
```

[1] https://docs.docker.com/reference/dockerfile/

```
COPY source1.cpp source.cpp
RUN g++ -o /binary source.cpp

FROM builder AS build2
COPY source2.cpp source.cpp
RUN g++ -o /binary source.cpp
```

Listing 2.2: Sample Multi-stage Dockerfile

This coupled with the *scratch* base image serves as the mechanism that allows for the creation of minimal container images, by defining first a builder image and then copying the required files to the empty image.

2.3 Related work

Before proceeding further, it is important to acknowledge the work done by my peers in the field of container image minimization.

The earliest mention of building minimal container images is in a blog post dated 4th of July 2014 by Adriaan de Jonge on the site Xebia [5] and is corroborated by a 2015-02-03[1] München talk by Brian Harrington [9] which describe using *tar* to create the *scratch* image and then copying the required files to it, or by using tools like *buildroot*[2] or *debootstrap*[3] to create a minimal Linux distribution and then copying the required files to it.

In 2016, the first Alpine Linux image was published, which had a compressed size of 1.86MB[4] as it was built using *musl* and *busybox*, two lightweight alternatives to the standard C library and coreutils respectively. Around the same time, the *Distroless* philosophy i.e images containing only the application and its dependencies, was introduced by Google, with the first of these images being build using the open source tool *Bazel* for languages such as Java, Go and Python. Recently, Canonical, the company behind Ubuntu, has embraced distroless with their *Chiseled*[5] ubuntu images.

However, none of these tackle the fundamental issue of using minimal container images, being that the developer has to manually determine the dependencies of the application, which can be a arduous task, especially for large applications. The only project that I could find that is tangentially related to this is *dockershrink* by developer Raghav Dua [6]. This tool utilizes Artificial Intelligence in order reduce the size of the container image by generating a new Dockerfile updated to use slimmer base images and removing unnecessary files. [6] However, this tool is still in beta and it only works for NodeJS applications, as well as having to provide it with a OpenAI API key in order to access the full functionality of the tool. Additionally, it does not utilize the *scratch* image, which means that the final container image generated by the shrunk Dockerfile is not the most minimal possible.

[1] not sure if it's the 2nd of March or 3rd of February as I could not find this presentation and the only mention of this is in his GitHub repository [9]

[2] Buildroot

[3] debootstrap

[4] Oldest Alpine Linux image on Docker Hub

[5] Chiseled

Chapter 3

Motivation and Objectives

The main advantages of minimal containers are their enhanced security and small image size. Their improved security comes from the inherent properties of being minimal, meaning:

- **minimized attack surface** - by having only the required dependencies for the application to run, the only attack vector a hacker has is the application itself and not other components
- **clear dependency tree** - with only the required dependencies, it is easier to identify and audit them in case of a vulnerability being discovered, [2]

Additionally, their small size means that they are faster to deploy and use fewer system resources like *CPU* and *memory*, which is especially important in a cloud environment where the cost is directly proportional to the resources used and shaving a couple of seconds off the deployment time can lead to hours, even days given the size of the cluster. [12].

By creating a tool that can automatically detect an application's dependencies and create the Dockerfile which produces the minimal container for that app, we can save developers the time and effort of having to do it themselves, which can be tiresome and frustrating process.

Chapter 4

Use Cases

A real world example for the need to generate these minimal Dockerfiles and the reason that spawned this project is *Unikraft*[1].

Unikraft was envisioned as a faster and more secure alternative to running applications in containers or virtual machines, by leveraging the power of ultra-lightweight virtual machines knows as unikernels [17].

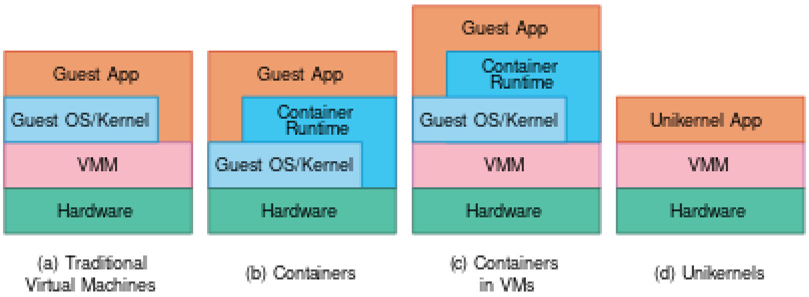

Figure 4.1: High-level comparison of the software components of traditional VMs (a), containers (b), containers in VMs (c) with unikernels (d).

Unikernels are specialized operating systems that are built as single-address space binary objects, meaning that they don't have user space-kernel space separation. They combine both the complete execution isolation and hardware access typical of VMs with the size of containers, which makes them ideal for cloud deployments [17].

Unikraft provides the tools needed to build, run and manage unikernels, in the form of a command-line program called *kraft*[2]. It features an interface similar to that of Docker's, with it being able to run pre-build unikernels or create new ones from a file called *Kraftfile*[3].

```
spec: v0.6
runtime: base:latest
rootfs: ./Dockerfile
cmd: ["/helloworld"]
```

Listing 4.1: Sample Kraftfile

[1] https://unikraft.org/

[2] https://unikraft.org/docs/cli

[3] also *kraft.yaml* or *kraft.yml* for legacy support

Kraftfiles are written in YAML[1], which follows an attribute-value structure. For my thesis, the only relevant attribute is *rootfs*, which describes the source from which the unikernel's root filesystem is built. Although optional [16], since most applications do require one, it is usually specified. As unikernels operate in resource-constrained specialized environments where efficiency is crucial, so do their filesystems need to strike the balance between providing the necessary storage requirements and also adhering to their lightweight philosophy.

There are many ways to define a root filesystem, one of which being highlighted in the Kraftfile example above, where a Dockerfile is used to build a container whose filesystem is later extracted to be used by the resulting unikernel. Thus, the Dockerfile should be as minimal as possible. So far, this process was done manually[2], which explains the rather low number of applications that have been ported to Unikraft[3], which need constant maintaining as different versions of the same app may require different Dockerfiles.

[1] https://yaml.org/

[2] https://unikraft.org/docs/contributing/adding-to-the-app-catalog

[3] https://github.com/unikraft/catalog

Chapter 5

Building Blocks

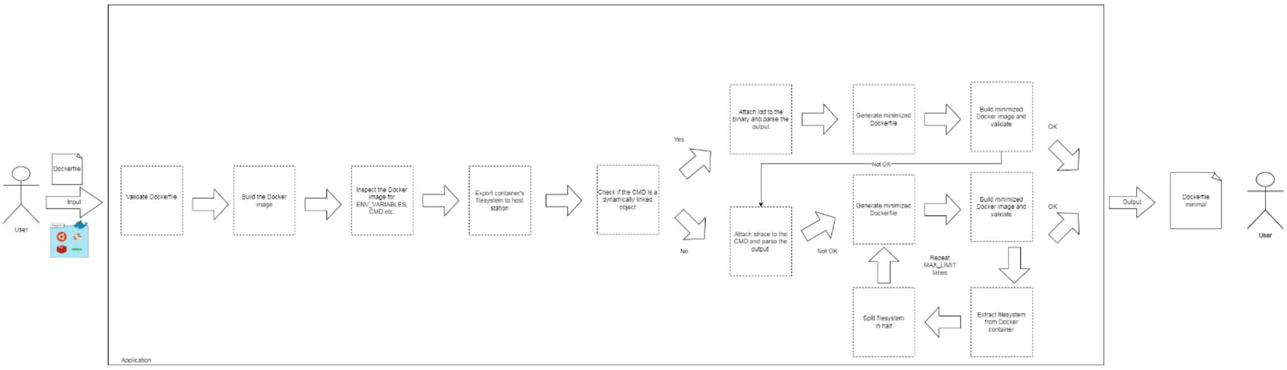
User
Dockerfile
Input
Validate Dockerfile
Build the Docker image
Inspect the Docker image for ENV_VARIABLES, CMD etc.
Export container's filesystem to host station
Check if the CMD is a dynamically linked object
Yes
No
Attach ldd to the binary and parse the output
Generate minimized Dockerfile
Build minimized Docker image and validate
OK
Not OK
Attach strace to the CMD and parse the output
Not OK
Generate minimized Dockerfile
Build minimized Docker image and validate
OK
Repeat MAX_LIMIT times
Split filesystem in half
Extract filesystem from Docker container
Output
Dockerfile minimal
User
Application

Figure 5.1: High-level architecture of the application

My solution is a command-line tool that automates the process of determining the runtime dependencies of a given application and generating the minimal Dockerfile.

For finding the runtime dependencies, I will be following the steps outlined here [15] and here [14] and will be a three-pronged approach, consisting of:

- **Static analysis** - inspecting the executable file for any linked libraries.
- **Dynamic analysis** - running the application and tracing its system calls [1].
- **Brute force** - a fail-safe incase the other two fail to yield any results.

The handling of the interaction between the program and Docker is done through the the *Docker SDK*[2], which provides a programmatic way to interact with containers. It is available both for Python and Go, both very popular languages.

Go is a statically-typed, compiled language known for its simplicity and efficiency. It was built by Google in 2007 for use in networking and infrastructure services, being designed to be efficient, readable and high-performing. [4]

Python is a high-level, interpreted language known for its simplicity and readability. It was created by Guido van Rossum in the late 1980s and has since become one of the most popular programming languages in the world. [8]

[1] system calls, *syscalls*, are the mechanism used by applications to request services from the kernel, like I/O, spawning processes, etc.

[2] https://docs.docker.com/reference/api/engine/sdk/

Although both languages are capable of achieving the same results, Go's performance superiority over Python, coupled with its concurrency and low-level capabilities, make it the clear choice for this project. The choice of using Go is also motivated by the fact that Docker itself is written in Go [10], which means that the SDK is better integrated with the rest of the Docker ecosystem and has better performance than the Python SDK.

Chapter 6

Architecture Overview

6.1 Static Analysis

Static analysis is the process of analyzing a program's code without executing it. In our context, we will be using it to identify the libraries that the application is dynamically linked using the *ldd* command.

```
~ > ldd /usr/bin/man
    linux-vdso.so.1 (0x00007ffe831f5000)
    libmandb-2.9.1.so => /usr/lib/man-db/libmandb-2.9.1.so (0x00007f068b572000)
    libman-2.9.1.so => /usr/lib/man-db/libman-2.9.1.so (0x00007f068b52f000)
    libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00007f068b502000)
    libpipeline.so.1 => /lib/x86_64-linux-gnu/libpipeline.so.1 (0x00007f068b4f1000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f068b2ff000)
    libgdbm.so.6 => /lib/x86_64-linux-gnu/libgdbm.so.6 (0x00007f068b2ef000)
    libseccomp.so.2 => /lib/x86_64-linux-gnu/libseccomp.so.2 (0x00007f068b2cb000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f068b59b000)
```

Listing 6.1: ldd command

The output of the command is a list of shared libraries that the application depends on, along with their paths. This approach works well for simple applications but it quickly becomes insufficient for most production applications.

6.2 Dynamic Analysis

Dynamic analysis is the process of analyzing a program's behavior during its execution. Our use case is to identify the other files that the application accesses during its execution, as well any spawned processes and their dependencies. This is done using the *strace* command, which uses the *ptrace* system call to trace the system calls made by a process.

```
~ > strace -fe execve,openat echo "Hello, World\!"
execve("/usr/bin/echo", ["echo", "Hello, World!"], 0x7fff43b04ce8 /* 36 vars */) = 0
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC)
    = 3
Hello, World!
+++ exited with 0 +++
```

Listing 6.2: strace command

For our scenario, we are interested in the *openat*, which means that the application is trying to open a file, and *execve*, which means that the application is trying to execute another program. The *-f* flag is used to allow strace to also trace the child processes spawned by the application, thus ensuring that we do not omit any dependencies. Additionally, it can trace processes that are already running using the *-p* flag, which will be used to trace the process running inside the container, this however needing root privilege since the process inside the container is in another namespace.

6.3 Brute Force

The brute force approach is a last resort method that is used should both the static and dynamic analysis fail in generating the minimal Dockerfile. Using the Docker SDK, we can extract the filesystem from the container and use it for this step. Given that in the containerized environment, the application is running normally and in a *FROM scratch* environment it is not (which does not contain any files), therefore there exists a point where removing a file from the filesystem causes the application to fail. Since removing files one at a time and building a new container each time is not feasible, we have to attempt a more efficient approach, that being a *binary search.*

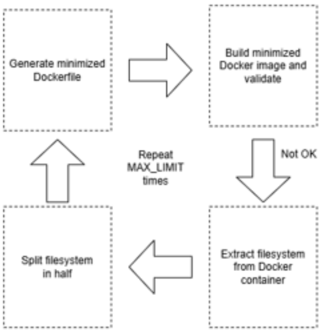
Generate minimized Dockerfile
Build minimized Docker image and validate
Repeat MAX_LIMIT times
Not OK
Split filesystem in half
Extract filesystem from Docker container

Figure 6.1: Brute force approach to finding the minimal Dockerfile

We start by splitting in half the filesystem into two buckets - *used* and *unused.* The decision of which files land in what bucket will be left to a coin toss in order to ensure fairness. We then build the Dockerfile with the files in the *used* bucket and run the application. If it fails, then we split the *unused* bucket in half and repeat the process. If the app succeeds, we repeat the cycle by splitting the shrunken filesystem in half again and follow the same steps. This will repeat at max *MAX_ LIMIT* times, a predefined limit we impose on the number of iterations. This sifting process is a greedy one and thus will not always generate the minimal Dockerfile, but it will offer a good approximation of it, especially if the limit is set to a high value.

Chapter 7

Status and Planned Work

7.1 Status

I have yet to implement the solution, but I have already done the research into the feasibility and, if all goes according to plan, there should be not major roadblocks in the implementation.

7.2 Planned Work

Implement the solution described a priori.

Bibliography

[1] 6sense. Market share of docker. https://6sense.com/tech/containerization/docker-market-share.

[2] 8grams Tech. Distroless: Using minimal container image for kubernetes workload. https://blog.8grams.tech/distroless-using-minimal-container-image-for-kubernetes-workload, February 2024.

[3] Chris Aubuchon. The shortlist: History of containers. https://cycle.io/blog/2024/07/shortlist-history-of-containers.

[4] William Boyd. What is go? an intro to google's go programming language (aka golang). https://www.pluralsight.com/resources/blog/cloud/what-is-go-an-intro-to-googles-go-programming-language-aka-golang.

[5] Adriaan de Jonge. Create the smallest possible docker container. https://xebia.com/blog/create-the-smallest-possible-docker-container/.

[6] Raghav Dua. dockershrink. https://github.com/duaraghav8/dockershrink.

[7] Ashan Fernando. Why it's important to keep your containers small and simple. https://hackernoon.com/why-its-important-to-keep-your-containers-small-and-simple-618ced7343a5.

[8] Python Software Foundation. What is python? executive summary. https://www.python.org/doc/essays/blurb/.

[9] Brian Harrington. Minimal containers 101. https://github.com/brianredbeard/minimal_containers.

[10] Docker Inc. The underlying technology. https://docs.docker.com/get-started/docker-overview/#the-underlying-technology.

[11] Docker Inc. Use containers to build, share and run your applications. https://www.docker.com/resources/what-container/.

[12] Ogo Ozotta. Small is beautiful: How container size impacts deployment and resource usage. https://www.fullstack.com/labs/resources/blog/small-is-beautiful-how-container-size-impacts-deployment-and-resource-usage.

[13] J. A. Pardo. But... it works on my machine.... https://medium.com/@josetecangas/but-it-works-on-my-machine-cc8cca80660c.

[14] Adam Rehn. Identifying application runtime dependencies. https://unrealcontainers.com/blog/identifying-application-runtime-dependencies.

[15] Unikraft. Adding applications to the catalog. https://unikraft.org/docs/contributing/adding-to-the-app-catalog.

[16] Unikraft. Concepts. https://unikraft.org/docs/concepts.

[17] Unikraft. Filesystems. https://unikraft.org/docs/cli/filesystem.

[18] Lionel Sujay Vailshery. Adoption rate of container technologies in organizations worldwide from 2016 to 2021, by development stage. https://www.statista.com/statistics/1104543/worldwide-container-technology-use/, February 2024.