Synopsys Users Group

Containerize Your Chip Development Environment Using Docker

Chris Drake

Google
Mountain View, CA
google.com

Abstract

A modern SoC development environment requires a complex web of application dependencies. OS distributions come with a variety of configuration files, system packages, and libraries. Architectural modeling and hardware development require open source tools, simulation frameworks, and proprietary CAD tools. In addition, Android/IOS targets have software development kits. The cost of maintaining consistent and repeatable development operations can be formidable with several versions of dependencies. Virtualization solutions are one way of dealing with this complexity, however they are a heavyweight solution that lead to performance problems, and system management complications. Docker is an open source tool that achieves many of the same goals with a smaller footprint. It combines a method for OS and package management with Linux container technology to create a hermetically-sealed development environment which can be distributed as a file. In this paper we suggest a methodology for how to use Docker for development of complex hardware IP and its corresponding software stack.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

As semiconductor manufacturing technology continues relentlessly on a path of innovation, chip makers are cramming more separately-developed IP blocks onto their SoCs with each new generation. Given the challenge to verify such complex systems, the demand for developer productivity is higher than ever. And yet in the year 2015, highly competent and professional logic design and verification engineers sometimes waste time chasing bugs caused by using different versions of Perl on their development machines. Furthermore, it can take literally years for the operations team to upgrade from an ancient version of SUSE Linux Enterprise due to the challenge of disrupting in-flight operations with system dependency issues.

Chip design is hard enough without having to actively manage the version of all dependencies of all tools installed on all machines used by developers spread across the globe in an increasingly distributed and international workforce. This is the so-called “dependency matrix of hell” that is so familiar to system administrators.

In this paper we will describe how to tackle this problem using Docker: a recently popular tool that has emerged from the world of software development operations (devops) and continuous integration. Even though its marketing copy describes how to “compose web applications from microservices”, one can use it to package up a Linux-based hardware design and verification environment as an image file that can be shared by all members of the team on any machine with the Docker runtime installed. This image file can be configured as any version of any Linux distribution. For example, you can run CentOS 6 applications seamlessly on an Ubuntu Trusty 14.04 desktop distribution.

We will describe how you can use Docker for development of complex hardware IP and its corresponding software stack. By containerizing all tool dependencies, you will be able to perform continuous integration and versioning not only on the project source code, but also on the development environment itself, including core OS libraries.

2. Docker Fundamentals

Docker is an open source software package that allows users to package an application with all of its dependencies into a standardized unit for software development [4]. It combines the following Linux technologies into a cohesive whole:

- Linux containers (LXC)
- Control Groups (cgroups)
- Chroot jails
- Versioned filesystem

Linux containers are an operating system-level virtualization method for running multiple isolated Linux systems on a single control host [11]. Control Groups (cgroups) are a Linux kernel feature that limits, accounts for, and isolates the resources (cpu, mem, disk, I/O) of a collection of processes [2]. A chroot (change root) jail is a way to isolate the filesystem of a process from other processes on the same system. The combination of these three technologies makes for a lightweight virtual machine that runs at native performance levels, but can be isolated and controlled by the user.

In addition to using the latest Linux virtualization technology, Docker utilizes a multi-layered, unification filesystem that is useful for performing exact version control of the VM's filesystem contents. Whenever a user performs an OS package management action, it is atomic, and the filesystem adds another layer that can be saved and replayed at a later time or on a different host.

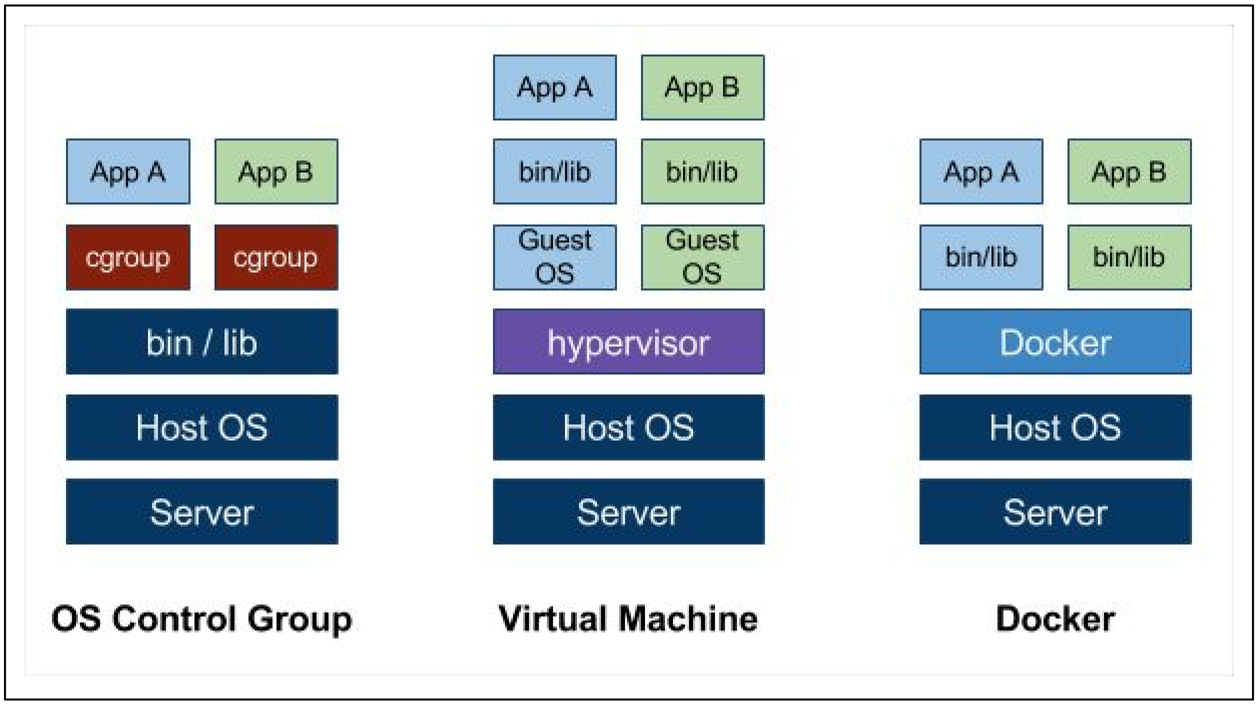

Figure 1: Comparison of cgroups, VMs, and Docker

Docker's feature set is often confused with other, related virtualization technologies. Figure 1 shows how Docker's usage model differs from bare Linux cgroups and full system virtualization (i.e. a "virtual machine"). Control groups merely isolate resources of processes that share the same host; system files (such as `/bin`, `/usr/bin`, `/lib`, `/usr/lib`, etc) are shared between processes. Virtual machines require users to boot a complete guest operating system on top of a hypervisor before executing virtualized applications on top of virtualized hardware. Docker, on the other hand, is a thin runtime installed on the host operating system that can execute parallel, virtualized applications each in their own sandbox, called a "container".

For developers, this means that as long as the machine has a recent Linux kernel and has Docker installed, a user can create a Docker container with a complete development environment with all system and application dependencies. Any system changes only apply within that container, and filesystem modifications can be saved and versioned. This technology does not apply only to system packages such as GCC, GNU make, and Perl, however. It also applies to EDA applications such as Synopsys VCS-MX or DesignCompiler, or a development environment such as Android SDK.

In the next section we explain the details of how to create a Docker image, and distribute it to internal and external users in the form of a Docker "development shell".

3. Development Shell Methodology

First we need to define some important, Docker-specific terminology. An "image" is a set of filesystem layers that are ultimately composed into a single file with a globally unique SHA-256 hash value. A "container" is a process that is attached to an instance of the image.

In this section, we will describe how first to create an image that contains an EDA development environment, and execute the container as a "development shell" that can be used interactively by an engineer.

3.1 Creating an Image

To create a Docker image, first write a script called a "Dockerfile". This is essentially a recipe for how to construct the image, step-by-step. Figure 2 shows an example Dockerfile that will install Synopsys VCS-MX on top of a CentOS-6 base image.

```
FROM centos:6
RUN yum install -y bc gcc-c++ perl tar tcsh
RUN yum install -y compat-libtiff3.i686 compat-libtiff3.x86_64 \
       fontconfig.i686 fontconfig.x86_64 glibc-devel.i686 glibc-devel.x86_64 \
       libjpeg-turbo.i686 libjpeg-turbo.x86_64 libmng.i686 libmng.x86_64 \
       libpng12.i686 libpng12.x86_64 libstdc++-devel.i686 libstdc++-devel.x86_64 \
       libSM.i686 libSM.x86_64 libX11-devel.i686 libX11-devel.x86_64 \
       libXScrnSaver.i686 libXScrnSaver.x86_64 libXext.i686 libXext.x86_64 \
       libXft.i686 libXft.x86_64 libXi.i686 libXi.x86_64 \
       libXrandr.i686 libXrandr.x86_64 libXrender.i686 libXrender.x86_64 \
       ncurses-devel.i686 ncurses-devel.x86_64

COPY vcs-mx_vK-2015.09-SP1_common.spf vcs-mx_vK-2015.09-SP1_amd64.spf \
     vcs-mx_vK-2015.09-SP1_linux.spf vcs-mx_vK-2015.09-SP1_SI32.tar \
     SynopsysInstaller_v3.2.run /root/

ENV SYNOPSYS /usr/local/cad/synopsys
ENV VCS_VERSION K-2015.09-SP1
ENV VCS_HOME $SYNOPSYS/vcs-mx/$VCS_VERSION
RUN ./SynopsysInstaller_v3.2.run -dir .
RUN yes | ./installer -install_as_root -batch_installer -source . -target $SYNOPSYS
ENV PATH $VCS_HOME/bin:$PATH
```

Figure 2: Example Dockerfile to Install VCS-MX K-2015.09-SP1

To create a Docker image, create a temporary directory, then copy the contents from Figure 2 into a file called "Dockerfile". Then run "`docker build -t synopsys.vcs-mx:K-2015.09-SP1 .`". The build command will start a container with a private filesystem. By executing the actions in the script, Docker will build the image step-by-step. It will further tag this image with the name "`synopsys.vcs-mx:K-2015.09-SP1`"

The "`FROM centos:6`" line is the conventional starting point. It tells Docker to find and download the contents of version "`6`" of a repository named "`centos`" from https://hub.docker.com. There are several other repository hosting solutions, for example: https://quay.io (from CoreOS), or https://gcr.io (Google Container Registry).

Lines two and three use the RedHat "`yum`" package management utility to update the system and install dependencies. First it installs system tools such as GCC's C++ compiler and Perl—`yum` automatically takes care of installing their dependencies. Next it installs several system libraries that are required by Synopsys VCS-MX.

After installing dependencies, the script copies the VCS-MX installation files onto the container's filesystem. These files can be obtained by FTP transfer from Synopsys' servers. The rest of the script sets environment variables and executes the Synopsys installer script for VCS-MX.
Once the docker build command has finished, if we run "`docker images`", we should see the output from Figure 3:

```
$ docker images
REPOSITORY       TAG            IMAGE ID       CREATED       VIRTUAL SIZE
synopsys.vcs-mx  K-2015.09-SP1  5ad3e2c0135a 5 minutes ago  9.224 GB
```

Figure 3: Docker Images Output

To start a Docker container, execute "`docker run -it synopsys.vcs-mx:K-2015.09-SP1 bash`". This launches an interactive Bash script that runs inside the container we just defined.

Notice that in this particular example, the size of the image created by Docker is 9.224 GB. This is a combination of CentOS base files and the installation of VCS-MX itself. One way to avoid having such a large image is to install VCS-MX (and/or any other standalone tool) to a network-attached storage, and mount that volume as a disk when starting the container. The "`docker run`" command supports the ability to mount external volumes using the "`-v VOLUME`" option. The tradeoff is that the smaller image will contain all the system files and library dependencies, but since the VCS-MX installation is external it is not always guaranteed to exist and be accessible.

3.2 Saving and Loading the Image as a File

Now that we have created a Docker image with all the necessary development files, we want to save it as a file and distribute it to an engineering team. There are two ways of doing this:

1. Pushing and pulling the image to a Docker repository
2. Using `save/load` commands to archive the image as a tar file

The first method is the most popular method for distributing open source Docker images for reuse. All Docker images available from https://hub.docker.com have been stored using the "docker push" command, and may be retrieved using the "docker pull" command. The "`FROM`" step in our Dockerfile from the previous section implicitly did a "docker pull" to fetch the contents of the base image from DockerHub.

If we do not want to distribute the Docker image to the world, there are private Docker repository services available. For example, both quay.io and gcr.io allow one to host a repository that is protected by OAuth2 and two-factor authentication.

Despite the apparent convenience of pushing and pulling the image like a Git repository, it is sometimes easier and more secure to simply save the image as a file. To do this, simply execute the "`docker save -o <tarfile>`" command to an appropriate tar file location (for example, a team shared location). Similarly, the "`docker load -i <tarfile>`" command will load the tar file into a development machine's local image repository.

3.3 Creating a Devshell Script

The previous section described how to start with a base image, and install some dependencies on top of it, using just the "`docker run`" command. But inside that default container there are significant limitations. The shell is started as user "root", and no external files are accessible—not even one's own project source code. Also, for GUI debugging, X11 forwarding is not enabled from the container to the host.

To remedy these limitations, we recommend creating a light-weight "devshell" script that does the following:

1. Collect user and group information from `/etc/passwd` and `/etc/groups`.
2. Load the correct base image from its canonical location.
3. Execute a new "`docker build`" step that inherits from the base image and adds user/group info, and produces a devshell image.
4. Set the working directory to the top of the project repository (using "`WORKDIR`" Dockerfile command).
5. Change user ID to whoever executed the script (using "`USER`" Dockerfile command).
6. Launch a new container (using "`docker run`") from the devshell image, using host networking, and mounting any necessary external volumes, including the local X11 socket.

See https://github.com/cjdrake/AES/blob/master/devsh for an example implementation written in Python.

Figure 4 shows the generic flow for creating a containerized build environment from start to finish. The first two steps of creating the base image and adding application files are performed in the first "docker build" from the Dockerfile in Section 3. The last step invokes your devshell script to customize the environment for a particular engineer's user and group id, and development platform information.

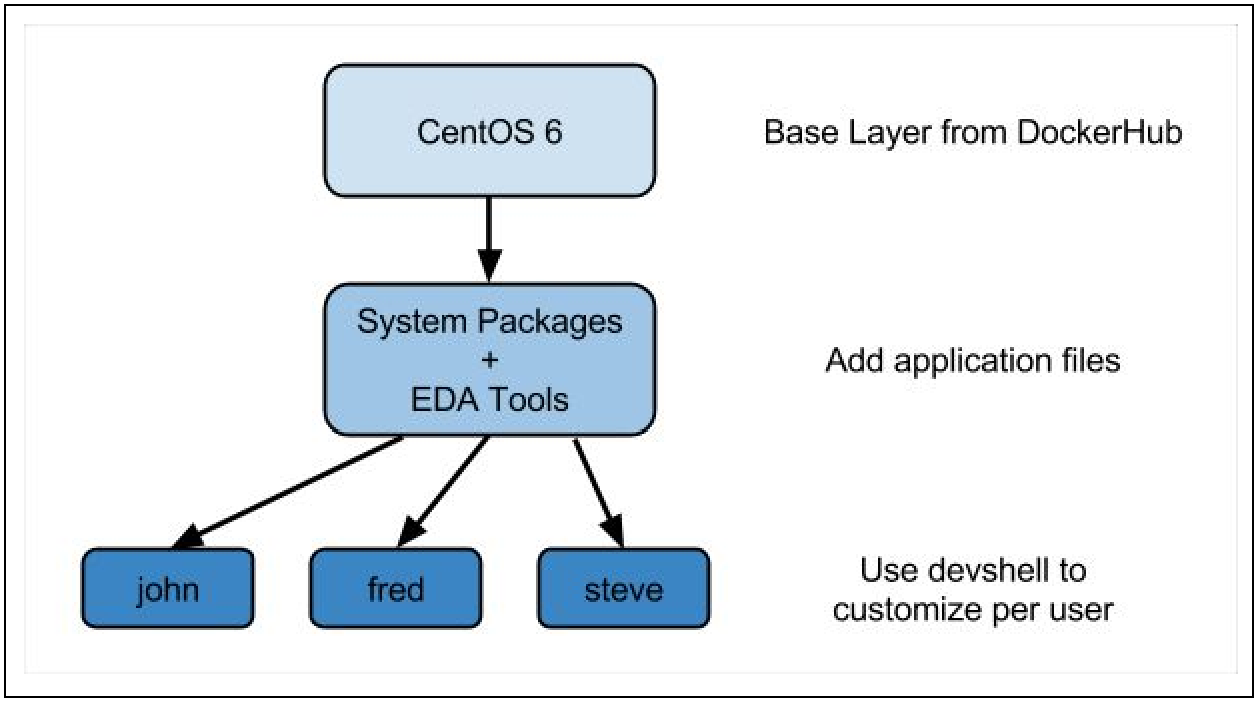
CentOS 6
Base Layer from DockerHub
System Packages + EDA Tools
Add application files
john
fred
steve
Use devshell to customize per user

Figure 4: Basic Development Image Flow

3.4 Performance Considerations

We ran extensive simulations at the architectural and RTL level to test out the performance of this methodology. We performed our experiments using a desktop machine running a variant of Ubuntu Trusty OS, with 12 Intel Xeon E5-1650 v2 3.50GHz processors and 32GB of RAM. We did not restrict Docker with any cgroup limitations on CPU/memory/IO, which means the container had access to full system resources.

First, since the devshell methodology imposes the overhead of 1) building a base image, and 2) building a devshell image, we measure how much time is typical for these actions. We used the Dockerfile from Figure 2 for "Base CentOS-6 plus system dependencies". The results are summarized in Table 1. Note that the Docker build step is highly dependent on the performance of the `yum` package manager, and network speed, so these results may vary. Also, these build times are *one-time costs*. That is, once the base or devshell image is built, it can be reused with zero execution overhead.

Image	Build Time (s)
Base CentOS-6 plus system dependencies	320
Devshell layer	20

Table 1: Docker Image Build Times

Next, we measured the simulation performance of Synopsys VCS-MX natively on the host machine, and using the Docker devshell. We used a simple, open source AES implementation and testbench [20], and executed the compile, elaboration, and simulation using the 64-bit version of VCS-MX K-2015.09-SP1. This testbench has two input parameters: NUM_BLOCKS and NUM_VECS. Varying the number of blocks changes the size of the design, and the corresponding elaboration time. Varying the number of vectors changes how long the test simulation runs after elaboration is finished. The results are the average of three runs.

Parameters	Host Runtime (s)	Docker Runtime (s)

NUM_BLOCKS=10
NUM_VECS=12,000
	56.65	56.59

NUM_BLOCKS=10
NUM_VECS=120,000
	548.88	548.67

Table2: Simulation Performance Comparison

The results show that executing a heavyweight process inside a Docker container has the same performance as executing it on the host. The difference was not statistically significant.

There are two other performance considerations: devshell launch time, and X11 forwarding. We found that after Docker had cached the devshell image, the amount of time it took to load was negligible. Similarly, we did not observe any noticeable slowdown of X11 GUI applications when running EDA applications from within the devshell sharing the host desktop's X11 socket.

4. Related Work

Docker was one of the hottest technologies of 2014, but it is not unique in its domain. In this section we will briefly overview a few similar tools for application containerization, and also discuss Docker's role in the broader domain of high-performance computing.

Though Docker has emerged as the clear leader in the area of open source, lightweight containerization tools, there are several alternatives that provide similar functionality.

In 2011, Microsoft demonstrated an application virtualization and sandboxing tool called Drawbridge [5]. This year, CoreOS publically disagreed with some of Docker's implementation details, and implemented a similar tool called “Rocket” [3]. Also recently, Canonical announced “LXD”, a Linux container hypervisor solution that focuses on security features [1]. A company called “Parallels” also offers an extensive virtualization solution for desktop applications [14].

One very important alternative to using Docker for development is a tool called Vagrant. Vagrant uses Oracle Virtualbox [13] to create sandboxed development environments that can be saved and reused by team members [15]. This is exactly the use case we were targeting in this paper. Vagrant's reliance on Virtualbox makes it a more heavyweight solution than Docker. First, in order to get decent performance, you need to enable hardware virtualization in the BIOS settings of your development machine. Second, the Virtualbox image files are much bigger than Docker images for the same content because of the requirement of having a guest OS instead of sharing the host's kernel. Third, there is more administrative overhead required to manage a full virtual machine, and the brunt of that overhead is born by the developer. For example, the unmanaged guest OS might be considered a security hazard operating inside a corporate network. Furthermore, Virtualbox is difficult to incorporate into a traditional HPC grid computing environment.

In this paper we have focused on the interactive application of a development container. A very important consideration is whether we can use containers for batch execution of parallel jobs in a high-performance computing (HPC) system. Google recently published details of the Borg system for large-scale cluster management [16]. They also recently open sourced a tool called Kubernetes that was inspired by the architecture of Borg [7]. Borg uses a proprietary Linux container implementation to achieve global-scale distributed computing. Kubernetes uses Docker directly as its containerization solution. In the area of conventional HPC, IBM LSF supports a Docker plugin [12], Univa GridEngine announced Docker support in 2015, and the US Department of Energy presented at SLURM User Group 2015 the details of using Docker with SLURM [9]. Lastly, the popular devops and continuous integration tool Jenkins supports several plugins for building and using Docker images as part of your development flow [10].

5. Conclusions

In this paper, we described some of the common problems experienced by chip design and verification teams working in a complex, Linux-based development environment. As it turns out, the dependency “matrix of hell” is not unique; web developers have been familiar with these issues for quite some time, and have engineered very elegant solutions for them. This paper described how to use Docker to simplify devops for your engineering teams. With Docker, teams can package all the operating system files, application files, and other dependencies together into an image that can be version controlled and continuously integrated along with the rest of the team’s project source code.

By reducing the application dependency requirements from a cornucopia of dependencies to a single Docker container runtime and a single chroot file, you cut all the IT-induced fat from your development timeline.

Driven by innovations in virtualization and Linux containerization, the devops community has raced ahead of conventional high performance computing. As more businesses migrate their IT operations to the cloud, this trend will only accelerate. The so-called warehouse-scale computer of the future will employ multiple levels of virtualization to efficiently utilize available resources in an environmentally conscious way. Fortunately, the chip design and verification community can leverage this technology to improve productivity as well. Perhaps this may also enable a future of safe, reproducible, and efficient, EDA in the cloud.

Never spend precious time triaging failures related to what version of Perl is on your system, and never fret about whether you have the right version of GCC. Just wrap your development environment in a Docker container, and get to work!

6. References

1. Canonical LXD: http://www.ubuntu.com/cloud/lxd
2. J. Corbet, "Process Containers", http://lwn.net/Articles/236038, 2007
3. CoreOS Rocket: https://coreos.com/blog/rocket
4. Docker: https://www.docker.com
5. Drawbridge, Microsoft Research, http://research.microsoft.com/en-us/projects/drawbridge
6. C. Drake, SystemVerilog AES implementation and testbench, https://github.com/cjdrake/AES
7. Google Kubernetes: http://kubernetes.io
8. IBM LSF: http://www-03.ibm.com/systems/platformcomputing/products/lsf
9. Jacobson, Botts, Canon, "Never Port Your Code Again: Docker Deployment with SLURM", SLURM User's Group 2015.
10. Jenkins: http://jenkins-ci.org
11. Linux Containers Project: https://linuxcontainers.org
12. B. McMillan, C. Chen, "High Performance Docking", Technical White Paper, IBM Computing, 2014.
13. Oracle Virtual Box: https://virtualbox.org
14. Parallels: http://www.parallels.com
15. Vagrant: https://www.vagrantup.com
16. A. Verma, et al, "Large-scale Cluster Management at Google with Borg", EuroSys 2015.