Build-Time Optimization

1. Introduction

This document describes the project undertaken by PalC Networks for the build-time optimization of a source-code base involving the combination of multiple versions of kernel-source code-bases, 3rd party system libraries, NOS-code and Software Development Kit of ASIC.

The customer is one of a billion dollar company who designs, develops, and manufactures wired and wireless network equipment and develops the software for network management, policy, analytics, security etc.

1.1. Architecture of the source-code base under consideration

The source-code consists of multiple components: the basic Linux-kernel, the control-plane, data-plane code, tool-chains and third party packages. All of these will be built one after the other for specified hardware target/controller and the complete time taken to produce a final binary image is a sum total of all the previous mentioned components. This project aims at instrumenting the current build time by measuring the time taken by the individual sub-components and reducing this total build-time through means mentioned in the later sections.

The overall requirements can be categorized into the following four divisions and the subsequent sections explain in detail about the same.

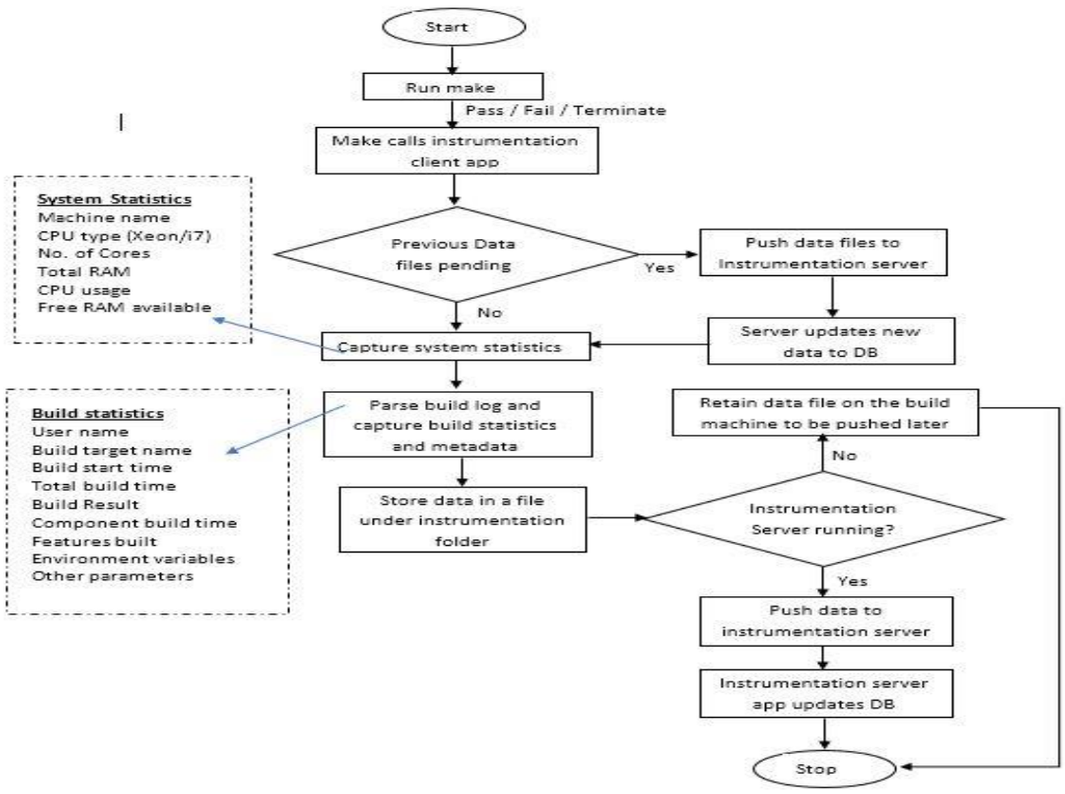
1. Instrumentation and profiling
2. Dockerization of the build environment
3. Distributed build environment using Linux distCC
4. Conanizing the third-party packages and Jfrog artifact repository management

1.2. Instrumentation and profiling

The objective of this sub-step was to capture the build time of individual components of the source code base including various components of the control-plane and data-plane code, kernel, third party packages and system-libraries.

A wrapper shell-script was developed on top of the existing linux makefile, which makes use of the starting system timestamp, ending timestamp for each sub-module component compilation, takes the difference, which denotes the overall time taken for that particular module. The captured timing information was then sent to a server and represented in a graphical form with appropriate filters to narrow down the displayed information.

Overall flow for the instrumentation and profiling part of the execution has been captured in the below mentioned flow diagram:



1.3. Dockerization of the build environment

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package up an application with all of the parts it needs, such as libraries and other dependencies, and ship it all out as one package. By doing so, the developer can rest assured that the application will run on any other Linux machine regardless of any customized settings that machine might have that could differ from the machine used for writing and testing the code.

Rather than creating a virtual operating system, Docker allows applications to use the same Linux kernel as the system that they are running on and only requires applications is shipped with things not already running on the host computer. This gives a significant performance boost and reduces the size of the application.

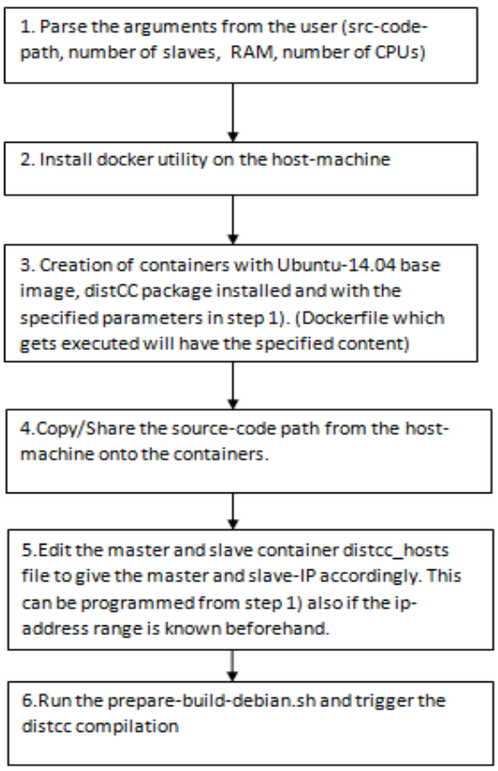
Implementation

- Set up a Docker installation on the host machine.
- Creation of multiple Docker containers with Ubuntu-14.04 image.
- Establish communication between the docker containers
- Copy/share the source code onto the containers from the host place.
- Install DistCC on the containers for parallel computation
- Configure the Master-Slave details on the containers
- Trigger the build on the container using DistCC command for parallel building
- All the above steps to be executed using a Dockerfile created on the host machine and having a top-level shell script, which automates the whole process.

The top-level shell script which prepares the environment for parallel compilation has been explained in the below section. This script reads the contents from a config.sh file which will have the parameters such as source-code path in the host-machine, number of CPUs, number of master-slave docker containers, RAM for each container etc.

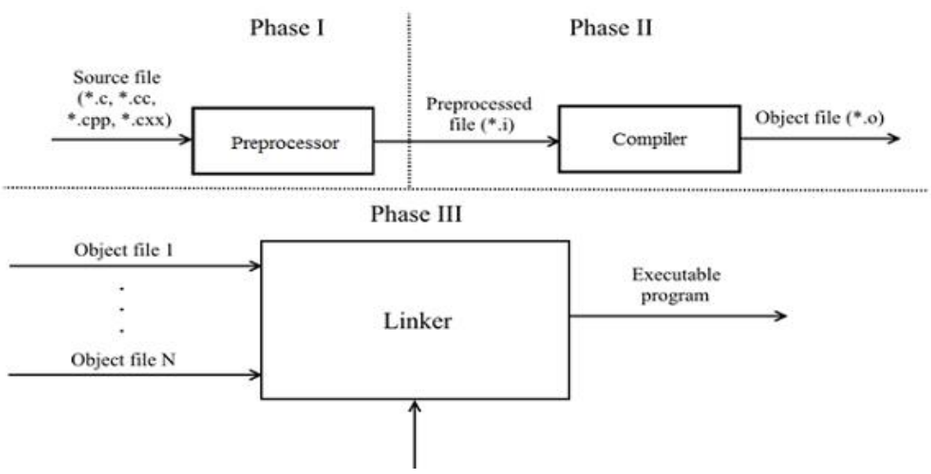
The overall process is summarized in the below diagram. Prepare-build-debian.sh is the in-house script,

which comes along with the source-code and sets up all the basic packages required to build the source code. The end product of the project had the container with prepare-build script integrated onto it and submitted to a private repository itself, which was then used to setup distCC and build the source-code.



1.4. Distributed build environment using distCC

The main phases of the translation of the C/C++ program into an executable involve the following steps:



1. The source-code file comes to the preprocessor, which makes the content substitution of the corresponding #include header files and expands the macros.
2. The preprocessed files come to the compiler and are converted to the specific object file.
3. The linker links all the object files together and provides static libraries, forms the final executable program.

So theoretically, the program consists of multiple translation units (*.c, *.cpp) which can be pre-processed or compiled independently from each other. In addition, each translation unit does not have any information about the other unit and only during the third stage; linker is going to link the different object files. When using a standard approach, a new file will get to the compiler for preprocessing and compiling. As each translation unit is self-sufficient, then a good way of speeding up is to parallelize phase 1 and 2 (above mentioned translation phases), processing simultaneously N files at a time. The below mentioned approach related to parallel compilation exploits that.

Distcc is a program to distribute builds of C, C++, Objective C or Objective C++ code across several machines on a network to speed up building. It should always generate the same results as a local build, is simple to install and use, and is usually much faster than a local compile. Further, one can use it together with native Arch build tools such as makepkg. In the distcc, the nodes take the role of either master or slave. The master is the computer, which initiates the compilation, and the slave accepts compilation requests sent by the master. One can setup multiple slave systems or just a single one.

Implementation:

- Distcc needs to be called in place of the compiler. We can export CC=distcc for the compilers we want to replace with it.
- Slave-Node Configuration
    - Each client/slave node needs to run the distcc daemon and needs to allow connections from the master host on the distcc port (3632). The daemon can be started manually at boot time by adding it system service.

- Master Node Configuration
    - On the master machine/container, we need to specify the list of slave IP-addresses that are participating in this cluster for compilation. There are two ways of achieving this. We can add the hostnames or IP addresses of our cluster to the file ~/.distcc/hosts, or you can export the variable DISTCC_HOSTS delimited by whitespace.

Once the master-slave node cluster is setup as mentioned in the above workflow, the only difference lies in the way make command is issued where we have to specify the number of jobs. The general guideline that is followed is the number of jobs should be approximately twice the number of CPUs available. The 3-basic models with which distcc can be deployed are as below:

1. Master and slave dockers inside the same host-machine.
2. Master and slave in different host-machines
3. Master and slave dockers present in two different host-machines in which case the communication between the dockers need to be established using GRE/VxLAN tunnels.

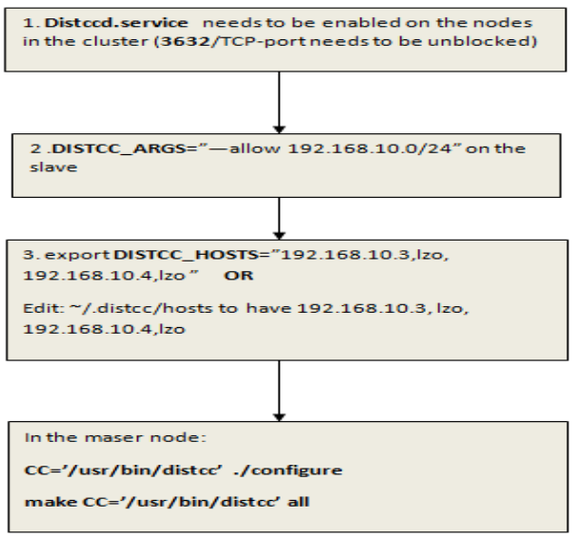
The following tools can be used to track the compilation progress.

1. **distccmon-text**

   A console based monitoring which is useful if the master docker is being accessed via SSH. When the compile job is running, the below command should be executed where N represents the number of seconds at which you would like to refresh the log.

2. **distccmon-gnome**

   A graphical representation of the above information can be got through the above tool



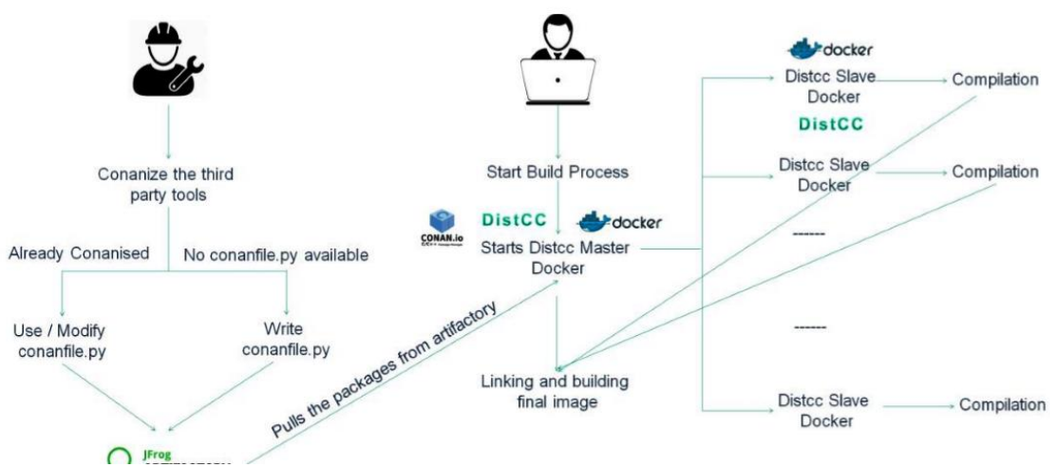
1.5. Conanizing the third-party packages

Conan is a Free Open Source Software with the permissive MIT license Conan is a portable package manager, intended for C and C++ developers, but it is able to manage builds from source, dependencies, and pre-compiled binaries for any language. Conan is a decentralized package manager with client-server architecture. This means that clients can fetch packages from, as well as upload packages to, different servers ("remotes"), similar to the "git" push-pull model to/from git remotes. On a high level, the servers are just package storage. They do not build nor create the packages. The client creates the packages, and if binaries are built from sources, that compilation is done by the client application.

One of the most powerful features of Conan is that it can manage per-compiled binaries for packages. To define a package, referenced by its name, version, user and channel, a package recipe is needed. Such a package recipe is a conanfile.py python script that defines how the package is built from sources, what the final binary artifacts are, the package dependencies, etc.

The Conan installation and Conan client configuration are done on the host/build machine and the built binaries are uploaded to the Jfrog artifactory. The developer instead of rebuilding the object-files/binaries from the scratch can fetch these from the Jfrog artifactory during the next build-time.

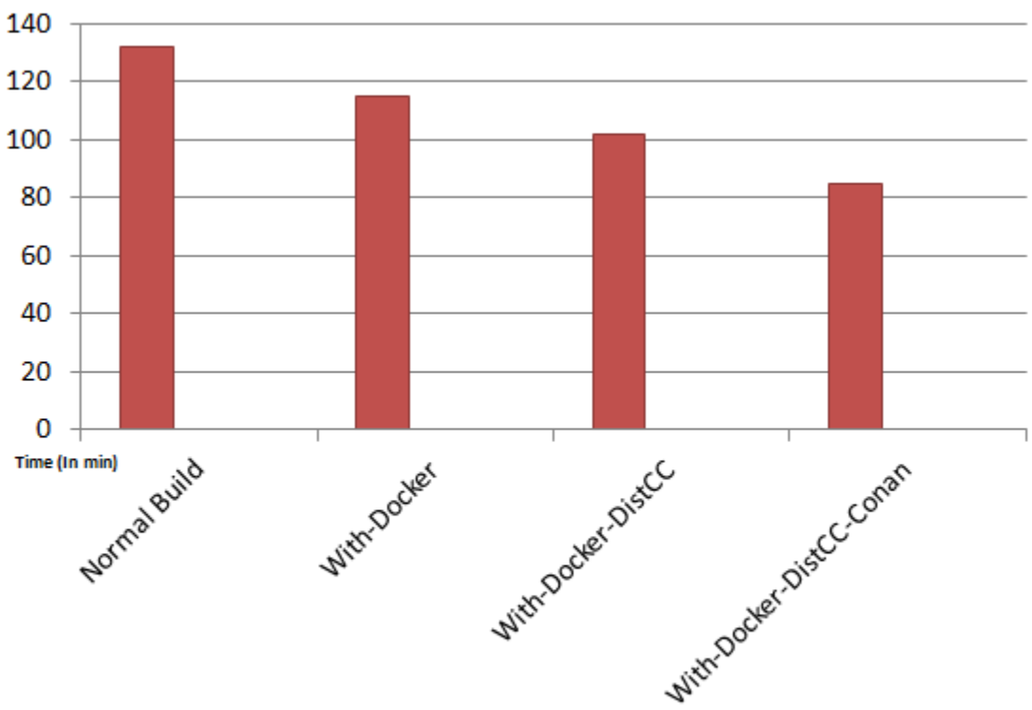
A pictorial representation of the overall workflow has been depicted below:



1.6. Optimization achieved

The source code-base under consideration was taking around 132 minutes for the complete build process with the 4GB RAM, 8-core CPU machine. Post the incorporation of Dockerization, Distcc and Conanization processes mentioned in the previous sections, the build time came down to 85 minutes.

There was an overall build-time reduction of 35% for the specified code-base. Depending on the number of third-party linux packages, which can be canonized in the existing source-code, the performance can be improved to even greater extent. The below image gives a pictorial representation of the results:



2. GLOSSARY

NOS Network Operating System

CPU Central Processing Unit

RAM Random Access Memory