Next-Generation Embedded Development Tools and Technologies – Virtualisation and Automation

Undergraduate Thesis

Submitted in partial fulfillment of the requirements of
BITS F421T Thesis

By

Richi Dubey
ID No. 2017A7TS0099G

Under the supervision of:

Prof. Marko Bertogna

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, GOA CAMPUS
June 2021

Declaration of Authorship

I, Richi Dubey, declare that this Undergraduate Thesis titled, 'Next-Generation Embedded Development Tools and Technologies – Virtualisation and Automation' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 15-06-2021

Certificate

This is to certify that the thesis entitled, "*Next-Generation Embedded Development Tools and Technologies – Virtualisation and Automation*" and submitted by Richi DUBEY ID No. 2017A7TS0099G in partial fulfillment of the requirements of BITS F421T Thesis embodies the work done by him under my supervision.

Supervisor
Prof. Marko BERTOGNA
Full Professor,
University of Modena
Date:

Co-Supervisor
Dr. Marco SOLIERI
Post Doctoral Researcher,
University of Modena
Date:

"Mindfulness and a never-say-die attitude can do wonders."

Richi Dubey

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI, GOA CAMPUS

Abstract

Bachelor of Engineering

Next-Generation Embedded Development Tools and Technologies – Virtualisation and Automation

by Richi DUBEY

There is an increasing interest in embedded systems. From hospitals to automobiles and avionics to bionics, we see them every day. Due to the huge demand for better, faster, and smaller embedded devices, tools and technologies must be developed that can increase their efficiency, make them safer, and conform to the standards through which they are certified. The aim of this thesis is to explore such tools and technologies. We focused our study on three important areas: The Configuration of Jailhouse Hypervisor, Yocto build system, and the Workload-Automation framework. We have then extended this study to provide a Proof of Concept benchmark analysis of the JH-supported workload using Workload-Automation. Tests for the PoC are carried out using Zynq UltraScale+ MPSoC ZCU102.

Acknowledgements

I could not have done any of this work without the help of the amazing community at **HiPeRT Lab** and I would remain forever grateful for all their help. I would specifically like to thank the following people:

Thanks to *Prof. Marko Bertogna* (UNIMORE), for believing in me from the start and for always being so supportive of my work.

Thanks to *Dr. Marco Solieri* (UNIMORE), for answering all my doubts and helping me navigate through this journey.

Thanks to *Luca Miccio and Angelo Ruocco* (UNIMORE), for endlessly helping me with all my questions.

Thanks to *Vijay Kumar Banerjee* (UCCS), for his invaluable reviews.

Thanks to *Prof. Vinayak Naik* (BITS Pilani), for being my on-campus supervisor and for having faith in me.

Contents

List of Figures

Abbreviations

HDD	**H**ard **D**isk **D**rive
JH	**J**ailhouse **H**ypervisor
NIC	**N**etwork **I**nterface **C**ontroller
OS	**O**perating **S**ystem
QEMU	**Q**uick **EMU**lator
RTOS	**R**eal **T**ime **O**perating **S**ystem
SSH	**S**ecure **S**hell **P**rotocol
VM	**V**irtual **M**achine
WA	**W**orkload **A**utomation

Dedicated to my parents, and to my unbelievably cute nephew, Medhu

Chapter 1

Hypervisors

1.1 Introduction to Hypervisors

A hypervisor is a software (or hardware) that allows the creation and execution of Virtual Machines (VMs). A VM is an environment that runs like a computer within a computer. Hypervisors allow the main host Operating System to share its resources(virtually) among VMs. The resources that can be shared are memory (RAM), processing (access to cores), and storage (access to secondary storage blocks) among many others. Hypervisors can be classified into two categories:

1. Native or Bare Metal Hypervisors: The hypervisor runs directly on the hardware (as an OS) and manages all the guest OSs as VM. Examples of such Hypervisors: KVM, Microsoft Hyper-V, Xen.
2. Hosted Hypervisor: This runs as an application software on a guest OS. So the OS is booted first and then Hypervisor is run as software by the OS. Some examples of such Hypervisors are Oracle VirtualBox, QEMU, Jailhouse.

1.2 Need of Hypervisors in a real-time system

Real-Time Systems rely on predictability and execution of tasks without buffer delays. In a safety-critical real-time system, the use of high-performance multicore platforms is challenging because shared hardware resources, such as cache and memory controllers, can cause extremely high timing variations. The timing unpredictability is a serious problem in both the automotive and aviation industries. For example, Bosch, a major automotive supplier, recently announced "predictability on high-performance platforms" as a major industrial challenge for which the industry is actively seeking solutions from the research community[2]. In aviation, the problem

was dubbed as a "one-out-of-m" problem because the current industry practice is to disable all but one core as recommended by the Federal Aviation Administration (FAA) for certification, which requires evidence of bounded interference[11]. By partitioning the resources between OSs, Hypervisors ensures that an application/OS runs with its own set of resources (computing, storage, memory among others) and faces no interference from other cores/tasks/OSs hence ensuring predictability. This allows for precise calculation of WCET and leads to better usage of resources.

1.3 Pros vs Cons of using Hypervisors

First, let's talk about the pros since there are many. Hypervisors allow for better usage of resources and give administrators the ability to allocate as many resources as needed by an application/OS running in VM (dynamic allocation during run time). Separation of resources also allows for easier debugging in case of system failure since each VM can be separated out and monitored safely without affecting other VMs behavior.

The possible downsides of using Hypervisors can be underutilizing resources when allocation is done poorly(mostly when done statically). Security is also an issue, since Hypervisors provide a centralized system, they become the target of various cyber-attacks [3].

1.4 QEMU

QEMU is an open-source machine emulator and virtualizer.

It uses dynamic translation to run Operating Systems and programs made for a specific machine (ex. Aarch64 board) on a different machine (ex. Ubuntu running on Intel i7).

It is used to emulate a machine's behavior on running an application on the machine. It is useful when access to a real machine is not possible (due to cost or other issues).

1.4.1 Basic Example: Using QEMU

We demonstrate the use of QEMU by running Alpine Linux as a VM on QEMU. Alpine Linux is a security-oriented lightweight Linux distribution [9].

Step 1: Getting the tools

Download QEMU for your operating system by downloading it from the official source [6] and Alpine Linux from its official source [5]

If you are using Ubuntu and an x86_64 system, the command would be:

```
sudo apt-get install qemu
curl -O https://dl-cdn.alpinelinux.org/alpine/v3.14/releases/x86_64/alpine-standard-3.14.0-x86_64.iso
```

Step 2: Preparing virtual HDD

Create a 16G virtual hard disk by running the command below:

```
qemu-img create -f qcow2 alpine.qcow2 16G
```

This command creates a hard disk drive (HDD) named *alpine.qcow2* that appears to be 16G in size to the guest (VM which is Alpine OS in our case), but the actual size is always the size of all the sectors that the guest writes(So, the size could go up to 16 GB). This virtual HDD can also be exposed as a block device to the local system (on which this virtual HDD is created) using qemu-nbd (optional step).

Step 3: Booting up the Alpine OS VM using virtual HDD and the fetched OS image

To execute VM on QEMU, run the following command:

```
qemu-system-x86_64 \
    -enable-kvm \
    -m 2048 \
    -nic user,model=virtio \
    -drive file=alpine.qcow2,media=disk,if=virtio \
    -cdrom alpine-standard-3.8.0-x86_64.iso \
    -sdl
```

The explanation of the command and its options are:

- *qemu-system-x86_64* is the executable qemu program for x86_64 architecture machine. (So, if your machine has an ARM architecture, the equivalent command would be qemu-system-arm).
- *enable-kvm* enables the Kernel Virtual Machine subsystem that allows the use of hardware-accelerated virtualization on Linux hosts. KVM lets us turn Linux into a type-1 bare-metal hypervisor that allows a host machine to run multiple, isolated VMS. Every VM is run as a regular Linux process, scheduled by the standard Linux scheduler with dedicated virtual hardware - like a network card, graphics adapter, access to processors, memory, disks [15].
- *m 2048* specifies that 2G (2048M) of RAM is being provided to the VM.

- *nic user, model = virtio* adds a virtual network interface controller (NIC), using a virtual LAN emulated by qemu. This is one of the most straightforward ways to provide internet access to a guest, another way is using the option *-nic tap* which would allow the guest to do networking using the host's NIC. *model = virtio* specifies a special virtuo NIC model that is used by the virtio kernel module in the guest to provide faster networking. Learn more about the virtio module on the KVM webpage here [14]
- *drive file=alpine.qcow2,media=disk,if=virtio* attaches the virtual HDD that we generated to the guest. It will show up as /dev/vda on the guest.
- *cdrom alpine-standard-3.8.0-x86_64.iso* connects the virtual CD drive to guest and loads the Alpine OS source that we had fetched earlier.
- *sdl* specifies the graphical configuration. SDL backend is one of the simplest graphical backends which provides a GUI for the guest on a newly opened window on host. Learn about SDL from its website here [1].

This would start the Alpine OS installation from the .iso image in a new window. Once the installation finishes (the files are installed on the virtual HDD), the command has to be re-run without the *cdrom* option. This would boot up Alpine OS using QEMU.

1.5 Jailhouse

Jailhouse is a partitioning Hypervisor based on Linux. It is able to run bare-metal applications or (adapted) operating systems besides Linux. Jailhouse is all about static partitioning, and it doesn't provide many features one expects from a virtual machine. There is no overcommitting of resources, VM scheduling, or device emulation. Jailhouse actually focuses on two main things: being small and simple and allowing guests (called "inmates") to execute with nearly zero latencies.

1.5.1 Jailhouse cells and inmates

JH runs on top of Linux and allows a developer to split the multi-core system into multiple cells - where one cell executes on one or more cores. Each cell has a set of resources assigned to it - core, PCI devices, memory region, etc. The job of JH is to manage resources for the cells, provide isolation and support for inter-cell communication. The "root" cell is the Linux host that started JH and non-root cells are any other cell.

Guests that run on a cell are called inmates. JH tries to run inmates with near-zero latencies. Various open-source OSs can be run as an inmate - Linux, FreeRTOS, Zephyr to name a few. An

inmate can also be bare-metal applications - that is written in C language by using the structure definition provided by the JH library.

1.5.2 Jailhouse configurations

Jailhouse is run by booting Linux first and then enabling Jailhouse. Jailhouse uses Linux to bootstrap itself, and once that is done, it runs on its own on the hardware. Jailhouse creates cells that do not interfere with each other.

Besides each cell requiring a configuration file of its own, the Jailhouse needs one configuration file for the entire system.

System-wide configuration file for jailhouse can be created by running (only for x86 systems) :

jailhouse config create sysconfig.c

The config for non-root cells is created using the C data structures provided by JH. There are open-source examples available, that can be referenced to create such configurations [7],[8].

So, future work would be to provide better documentation for creating JH non-root cells. This thesis is a stepping stone in that direction.

1.6 Hypervised: A Jailhouse library

This is a private library built inside HiPeRT Lab. The library is for virtual machines and bare metal softwares. It includes benchmarks for internal testing and memory profiling. It is based on the 'inmates' subdirectory of Jailhouse. It is used for creating custom inmates and helps in cross-compiling them faster than what would be possible with a complete Jailhouse source.

Chapter 2

Remote Systems

2.1 Introduction to Remote Systems

Remote systems are workstations, servers, or embedded boards (or any computing device) that are connected to a network and can be accessed remotely.

Connecting a device to a network (making it a remote device) allows access to it by users from all across the world. Remote access saves time by allowing quick deployment and remote debugging.

2.2 SSH

Secure Shell Protocol (SSH) is a cryptographic network protocol to operate network devices over an unsecured network (Ex: IP).

SSH is used for remote logging and remote command execution on systems. SSH provides a secure channel of communication between SSH servers and SSH clients that are connected over an unsecured network.

2.2.1 Setting up your own SSH Server

Step 1: Install OpenSSH on the machine that has to be set up as the SSH server.

```
sudo apt-get upgrade
sudo apt-get install openssh-client
sudo apt-get install openssh-server
```

openssh-client should also be installed on all the machines that will be used as SSH clients.

Step 2: To check if OpenSSH installed correctly and is working properly, connect to 'localhost' from the SSH server:

```
ssh localhost
```

So by running this command, the machine uses SSH to connect to itself. Once it is connected, type 'exit' to end the session.

Step 3: Connecting from clients on the same network.

Find out the IP address of the SSH server by running 'ifconfig' on the machine that is set up as the server.

```
ifconfig
```

Then connect from an SSH client on the network by running the following command on the client machine:

```
ssh username@ip-address-of-server
```

where *username* is the name of the user account that was used while setting up SSH on the server and '*ip-address-of-server*' is the output of 'ifconfig' - at the appropriate entry (eth for ethernet connection, wlan for wifi connection).

Step 4: Setting up ssh server for accepting connections from any client on the internet (outside the local network).

Due to NAT (Network Address Translation), every device on a local connection has its own private IP address, but all the devices have the same public IP address. This creates confusion when accepting an incoming connection from a device outside the network - since every device inside the network has the same IP, which device is the intended device for the connection is not known.

To overcome this issue, 'port forwarding' has to be set up on the router of the network that hosts the SSH server.

In general, SSH uses port 22. Port forwarding tells the router to forward requests made from a particular port from any device outside the network to a particular device inside the network.

The process of setting up port forwarding is specific to each router, but general steps are:

- Log in to the router's admin page
- Navigate to the page for adding a service (sometimes SSH is named as the default option)
- Enter the port number where the request will be made - 22 in our case

- Input the private IP address of the SSH server machine (that we found earlier by using 'ifconfig')
- Save the settings and you're done!

2.2.2 Accessing SSH Server from SSH Clients

To access an SSH Server from your machine, a user needs to generate a public-private key pair and the public key has to be added to the list of allowed users on the SSH server's configuration.

Step 1: Create a public-private key on the SSH Client machine.

Run the following command on the client machine:

```
mkdir ~/.ssh
chmod 700 ~/.ssh
ssh-keygen -t rsa
```

The command prompts for an optional passphrase for the public key (*id_rsa.pub*). This passphrase protects the key while it is stored in the drive of the client machine.

Step 2: Copy the public key file to SSH Server and authorize the client

Copy the public key (*id_rsa.pub*) file to the SSH server and add it to the list of authorized keys:

```
cat id_rsa.pub >> authorized_keys
```

Make sure that the SSH connection is successful by running the following command from the client machine:

```
ssh <username>@<host>
```

where *username* is the name of the user account of the SSH server that was used to set up the OpenSSH tool and host is the public IP address of the SSH Server. This command would also prompt for the passphrase of the key, that was generated in the last step:

```
Enter passphrase for key '/home/<user>/.ssh/id_rsa':
```

2.3 FTP

FTP or File Transfer Protocol, like the name suggests, is a protocol to exchange files between a server and a client over TCP, Transmission Control Protocol - which is a reliable communication protocol built on top of unreliable IP (Internet Protocol) [12]; Most website on the internet use TCP to form connection-oriented connections with the host.

Users can work with FTP using a simple command line interface, or with a dedicated graphical user interface. Most web browsers can also serve as FTP clients.

2.4 TFTP

TFTP or Trivial File Transfer Protocol is a lightweight protocol that is used for getting a file or putting a file onto a remote system. TFTP is a less secure version of FTP, generally used to boot embedded systems, configure LAN systems, perform firmware updates on routers etc.

2.4.1 Setting up a TFTP server

Step 1: Installing required packages

```
sudo apt-get install xinetd tftpd tftp
```

Step 2: Configuration

Create a file */etc/xinetd.d/tftp* and put the following entries in it:

```
service tftp
{
protocol        = udp
port            = 69
socket_type     = dgram
wait            = yes
user            = nobody
server          = /usr/sbin/in.tftpd
server_args     = /tftpboot
disable         = no
}
```

Create a folder **/tftpboot** (the value of the paramter server_args), and give it read, execute, write permission by all (only for testing, this rule should be used in production system):

```
sudo chmod -R 777 /tftpboot
sudo chown -R current_user /tftpboot
```

Step 3: Restart the service

```
sudo service xinetd restart
```

This sets up the TFTP server.

2.4.2 Fetching files from a TFTP server

Example command:

```
tftp -r rdubey/jailhouse_files/rootfs.cpio -g 172.19.0.3
```

Here the argument *rdubey/jailhouse_files/rootfs.cpio* should be replaced by the location of the file that needs to be fetched and the parameter *172.19.0.3* by the IP address of the TFTP server.

Chapter 3

Build Systems

3.1 Introduction to Build Systems

Build systems are a set of tools that are used to compile and convert files and other assets into a software. In the current context of Embedded Systems, Build systems are used to build portable OSs with small memory footprints for specific architectures.

3.2 Yocto Build System

The Yocto build system [17] allows developers to create custom Linux-based OSs independent of the architecture. It provides flexible set of tools and a worldwide community of embedded system devlopers that share technologies, software-stack and best-practices.

3.2.1 Yocto layers, recipes

Recipes in Yocto describe how to fetch, configure, compile and package applications and images.

Layers are sets of recipes that match a common purpose (Ex. building and installing the JH hypervisor).

3.2.2 Yocto build system flow

The Yocto Project is not intended to be used as a limited set of layers or tools, but instead, it provides a common base of tools and layers already provided, on top of which custom and specific layers are added, which depend on the target and user use case requirement.

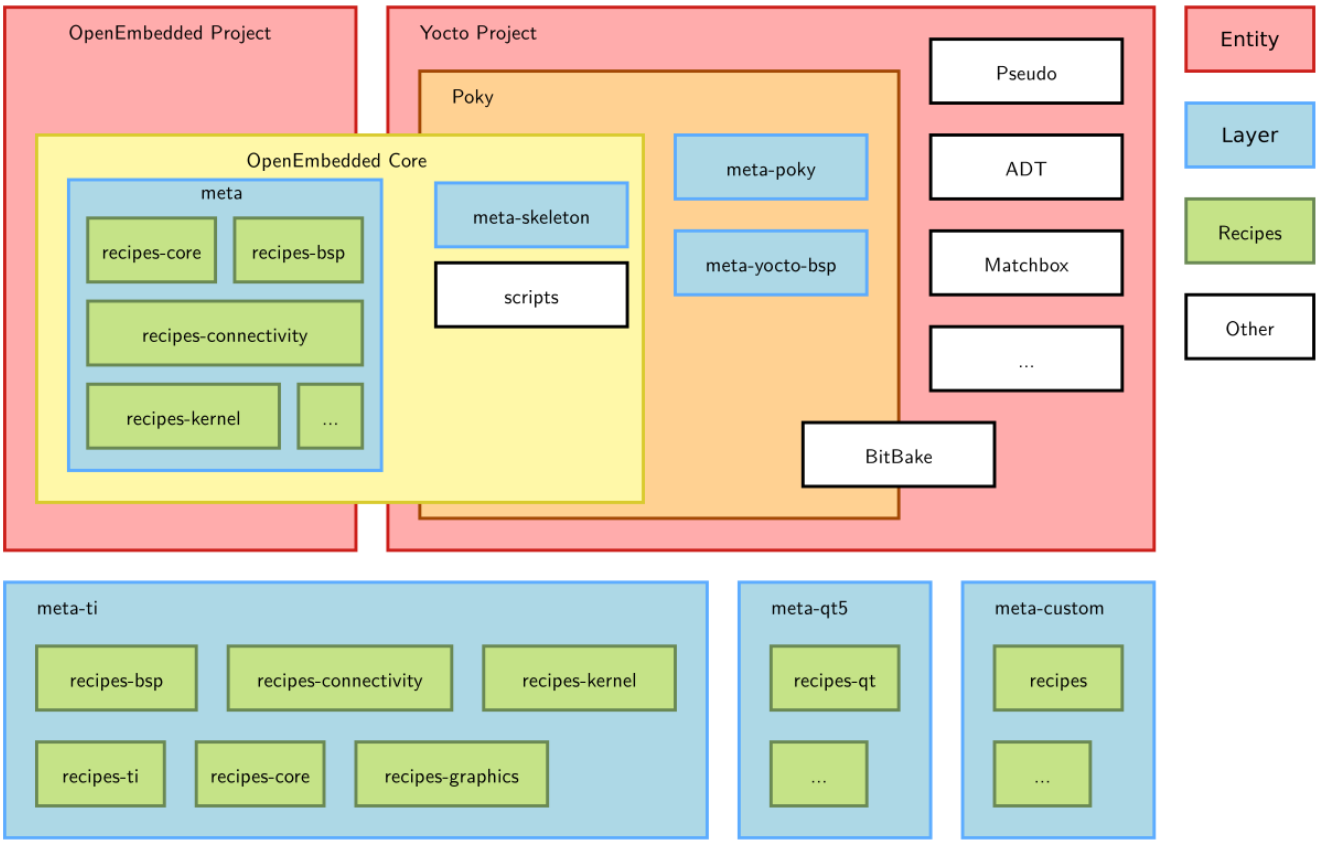

FIGURE 3.1: Yocto Project Components

The main required element is *Poky*, the reference system which has *OpenEmbedded-Core*. All other layers are optional but might be important. *OpenEmbedded-Core* is a set of recipes, layers, and classes that are shared between all *OpenEmbedded* based systems.

OpenEmbedded-Core supports QEMU emulated machines for various system architectures.

3.3 Writing a recipe in Yocto Build System

Recipes instruct how to handle an application. It is a set of instructions to retrieve, patch, compile, install and generate binary packages for an application and defines the dependencies required for the application. Recipes are parsed by the Bitbake build engine.

A recipe contains Tasks, which are functions that can be run (configure, compile, install, etc.).

Predefined variables in Yocto recipes:

PN - package name, as specified in the recipe file name
PV - package version, as specified in the recipe file name
PR - package revision, defaults to r0.

For the recipe file *autostart-jailhouse_1.0.bb*, the values of the variables PN, PV and PR would be autostart-jailhouse, 1.0 and r0 respectively.

3.3.1 Default Tasks and Example Recipe

Following is the list of defaults tasks having predefined functionality (that can be overwritten):

do_fetch, do_unpack, do_patch, do_configure, do_compile, do_install, do_package, do_rootfs.

Example Recipe:

```
 #This recipe copies the file hello to the /bin location
do_install() {
    install -d ${D}${bindir}
    install -m 0755 hello ${D}${bindir}
}
```

More information about the default directory location is present at [10].

3.3.2 Writing a recipe that autostarts Jailhouse on the board

From: sources/meta-hipert/recipes-extended/autostart-jailhouse/autostart-jailhouse_1.0.bb:

```
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/COPYING.MIT;\

FILESEXTRAPATHS_append := "${THISDIR}/files:"
SRC_URI += "file://autostart.sh"
SRC_URI += "file://autostart.service"

do_install() {
    install -d ${D}${sysconfdir}/init.d
    install -d ${D}${sysconfdir}/rcS.d
    install -d ${D}${sysconfdir}/rc5.d
    install -d ${D}${sbindir}

    install -m 0755 ${WORKDIR}/autostart.sh      ${D}${sysconfdir}/init.d/
    install -m 0755 ${WORKDIR}/autostart.service  ${D}${sbindir}/

    ln -sf ../init.d/autostart.sh      ${D}${sysconfdir}/rc5.d/S90run-script
}
```

This recipe basically the JH autostart script (that enables Jailhouse and create a root cell) in the rc5 directory: the directory that includes all the script which should be started at runtime when

the system is booted at run level 5. In Linux, run levels are operational levels that describe the state of the system with respect to what services are available. The Zynq UltraScale+ MPSoC ZCU106 present in the HiPeRT lab boots in run level 5.

3.3.3 Writing a recipe that installs Hypervised on the board

From: sources/meta-hipert/recipes-extended/hypervised/hypervised.inc:

```
SUMMARY = "Hypervised"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COREBASE}/meta/COPYING.MIT;"

S ="${WORKDIR}/git"
B = "${S}"

require hypervised-defs.inc

do_configure(){
        cd ${S}
        git submodule update --init --recursive
}

do_compile(){
        cd ${S}
        oe_runmake \
                ARCH=${HV_ARCH} \
                CROSS_COMPILE=${TARGET_PREFIX}
}

do_install() {
        install -d ${D}${HV_DATADIR}
        install ${S}/build/src/bench/${HV_ARCH}/*.bin ${D}${HV_DATADIR}/
}

# Enable use of menuconfig directly from bitbake and also within the devshell
OE_TERMINAL_EXPORTS += "HOST_EXTRACFLAGS HOSTLDFLAGS TERMINFO"
HOST_EXTRACFLAGS = "${BUILD_CFLAGS} ${BUILD_LDFLAGS}"
HOSTLDFLAGS = "${BUILD_LDFLAGS}"
TERMINFO = "${STAGING_DATADIR_NATIVE}/terminfo"
do_devshell[depends] += "ncurses-native:do_populate_sysroot"

KCONFIG_CONFIG_COMMAND ??= "menuconfig"
python do_menuconfig() {
    import shutil

    try:
        mtime = os.path.getmtime("hypervised/.config")
        shutil.copy("hypervised/.config", "hypervised/.config.orig")
    except OSError:
        mtime = 0

    oe_terminal("${SHELL} -c \"make ARCH=%s %s; \
    if [ \$? -ne 0 ]; then echo 'Command failed.'; \
```

```
    printf 'Press any key to continue... '; \
    read r; fi\"" % (  d.getVar('HV_ARCH'), \
    d.getVar('KCONFIG_CONFIG_COMMAND')), d.getVar('PN') \
    + ' Configuration', d )

    try:
        newmtime = os.path.getmtime("hypervised/.config")
    except OSError:
        newmtime = 0

    if newmtime > mtime:
        bb.note("Configuration changed, recompile will be forced")
        bb.build.write_taint('do_compile', d)
}
do_menuconfig[depends] += "ncurses-native:do_populate_sysroot"
do_menuconfig[nostamp] = "1"
do_menuconfig[dirs] = "${B}"
addtask do_menuconfig before do_compile
addtask do_configure before do_menuconfig

DEPENDS = "bison-native"
FILES_${PN} += " ${HV_DATADIR}"
```

This recipe fetches the hypervised library from the private GitLab repository of HiPeRT lab and sets up the settings to configure the recipe using menuconfig. Once configured, it installs the inmates generated by hypervised at /usr/share/hypervised directory on the board (more precisely, the rootfs that gets generated for the board).

Chapter 4

Automating Workloads

4.1 Introduction to Automating Workloads

Automating Workload means using software to schedule and manage specific tasks (which are called workloads). This can be used from a host computer (that is easily accessible) to perform tasks on a remote embedded board that can be continents away! This allows for efficient use of resources; time saved on reconfiguring the board every time a new workload has to be tested and saves the physical travel needed to access the board physically, hence saving both time and money.

4.2 Workload-Automation (WA) framework

The Workload-Automation (WA) is a framework by ARM "for executing workloads and collecting measurements on Android and Linux devices. WA includes automation for nearly 40 workloads and supports some common instrumentation (ftrace, hwmon) along with a number of output formats." [16]

4.3 Running a workload using WA

To run a workload, pass it as a parameter to the run command in WA:

```
wa run hackbench
```

But the target system has to specified first, by editing the file

```
.workload_automation/config.yaml
```

and setting correct values to:

```
device_config:
        host: '172.19.253.142'
        username: 'root'
        password: ''
        # ...
```

also, the device type (Android/Linux) has to be specified in the same file, by editing the field:

```
# This setting defines what specific Device subclass will be used to
# interact with the connected device. Obviously, this must match your
# setup.
device: generic_linux
```

A sample output of the run looks like this:

```
wa run hackbench -d all_wa_output/01_May_hackbench_0
INFO     Creating output directory.
INFO     Initializing run
INFO     Initializing execution context
INFO     Connecting to target
INFO     Setting up target
WARNING  No CGroups controller available
INFO     Generating jobs
INFO         Loading job wk1 (hackbench) [1]
INFO     Installing instruments
INFO     Installing output processors
INFO     Starting run
INFO     Initializing run
INFO         Initializing job wk1 (hackbench) [1]
INFO     Running job wk1
INFO         Configuring augmentations
INFO         Configuring target for job wk1 (hackbench) [1]
INFO         Setting up job wk1 (hackbench) [1]
INFO         Running job wk1 (hackbench) [1]
INFO         Processing output for job wk1 (hackbench) [1]
INFO             Processing using "csv"
INFO         Tearing down job wk1 (hackbench) [1]
INFO         Completing job wk1
INFO     Job completed with status OK
INFO     Run completed
INFO         Finalizing job wk1 (hackbench) [1]
INFO     Finalizing run
INFO         Processing using "csv"
INFO         Processing using "status"
INFO         Status available in all_wa_output/01_May_hackbench_0/status.txt
INFO     Done.
INFO     Run duration: 3 seconds
INFO     Ran a total of 1 iterations: 1 OK
INFO     Results can be found in all_wa_output/01_May_hackbench_0
```

When the run finishes, the result is present in the directory

```
all_wa_output/01_May_hackbench_0
```

(i.e. the directory specified as a parameter following -d). The result is saved in two different file formats (result.json and results.csv).

The result for the above command looks like (from the file results.csv):

```
id,workload,iteration,metric,value,units
wk1,hackbench,1,total_groups,10,groups
wk1,hackbench,1,total_fd,40,file_descriptors
wk1,hackbench,1,total_messages,100,messages
wk1,hackbench,1,total_bytes,100,bytes
wk1,hackbench,1,test_time,0.606,seconds
wk1,hackbench,1,execution_time,1.0232908725738525,seconds
```

To understand the result, consider the definition of the hackbench benchmark test available for Linux distributions:

"Hackbench is both a benchmark and a stress test for the Linux kernel scheduler. It's main job is to create a specified number of pairs of schedulable entities (either threads or traditional processes) which communicate via either sockets or pipes and time how long it takes for each pair to send data back and forth."[13]

So, the result tells about the number of messages transferred between the group, and the number of file descriptors used, along with total bytes of data used in communication and finally the time it took.

WA runs benchmarks by sending the file via SCP/SFTP and it uses ssh to authenticate the connection. The run.log file generated in the output directory gives a detailed log of the entire order of flow of execution.

4.4 Adding a new workload in WA

To create a basic workload template file, execute the following command on the host computer:

```
wa create workload -k basic hello_world
```

The workload can be of following types: 'basic', 'apk', 'revent', 'apkrevent', 'uiauto', 'apkuiauto'. Except 'basic' all other types corresponds to a workload meant to be run on an android device. More information about the workloads currently available in the framework is present at WA docs. [4]

The workload is created and can be seen in the file:

```
/home/{$USERNAME}/.workload_automation/plugins/hello_world/__init__.py


from wa import Parameter, Workload


class hello_world(Workload):

    name = 'hello_world'
    description = "This is an placeholder description"

    parameters = [
        # Workload parameters go here e.g.
        Parameter('example_parameter', kind=int, allowed_values=[1,2,3],
                  default=1, override=True, mandatory=False,
                  description='This is an example parameter')
    ]

    def __init__(self, target, **kwargs):
        super(hello_world, self).__init__(target, **kwargs)
        # Define any additional attributes required for the workload

    def init_resources(self, resolver):
        super(hello_world, self).init_resources(resolver)
        # This method may be used to perform early resource discovery and
        # initialization. This is invoked during the initial loading stage and
        # before the device is ready, so cannot be used for any device-dependent
        # initialization. This method is invoked before the workload instance is
        # validated.

    def initialize(self, context):
        super(hello_world, self).initialize(context)
        # This method should be used to perform once-per-run initialization of a
        # workload instance.

    def validate(self):
        super(hello_world, self).validate()
        # Validate inter-parameter assumptions etc

    def setup(self, context):
        super(hello_world, self).setup(context)
        # Perform any necessary setup before starting the workload

    def run(self, context):
        super(hello_world, self).run(context)
        # Perform the main functionality of the workload

    def extract_results(self, context):
        super(hello_world, self).extract_results(context)
        # Extract results on the target

    def update_output(self, context):
        super(hello_world, self).update_output(context)
        # Update the output within the specified execution context with the
```

```
        # metrics and artifacts form this workload iteration.

    def teardown(self, context):
        super(hello_world, self).teardown(context)
        # Perform any final clean up for the Workload.
```

Most of the functions are self-explanatory (and the comments help in understanding). 'Workload' is the base class that all the workloads inherit. Explanation of the methods in the base class:

- init_resources: May (optional) be overridden to dynamically discover resources for the workload. This method executes when the device has not been initialized, so only those resources should be initialized using this method that does not depend on the device to resolve.

- validate: Used to validate assumptions about the workload (presence of files, checking environment variables, etc.). Raise a wa.WorkloadError if any assumption turns out to be false.

- initialize: Defined with @once_per_instance, hence this function is executed only once for all the resources/instances of the workload. Runs after initialization of the device so can be used to perform device-dependent initialization.

- run: This method should perform the task that has to be measured. When this method exits, the task is assumed to be complete.

- extract_results: This method is invoked after the execution of a task has finished and this should be used to extract the results from the target.

A sample workload (hello_world) is given below that accepts a parameter num_times to measure the number of times "hello world" has to be printed in a text file called hi.txt in the target system. Eventually, the result of the 'time' command is pulled (moved) back from the target to the host.

os is imported to support path functions (to get the working directory in the target)

```
import os
from wa import Parameter, Workload


class HelloWorld(Workload):

    name = 'hello_world'
    description = "Simple hello world workload"

    parameters = [
        # Workload parameters go here e.g.
        Parameter('num_times', kind=int, allowed_values=[1,2,3],
                  default=1,
                  description='Number of Hello World executions')
    ]
```

```
def __init__(self, target, **kwargs):
    super(HelloWorld, self).__init__(target, **kwargs)
    # Define any additional attributes required for the workload

def init_resources(self, resolver):
    super(HelloWorld, self).init_resources(resolver)
    # This method may be used to perform early resource discovery and
    # initialization. This is invoked during the initial loading stage and
    # before the device is ready, so cannot be used for any device-dependent
    # initialization. This method is invoked before the workload instance is
    # validated.

def initialize(self, context):
    super(HelloWorld, self).initialize(context)
    # This method should be used to perform once-per-run initialization of a
    # workload instance.

def validate(self):
    super(HelloWorld, self).validate()
    # Validate inter-parameter assumptions etc

def setup(self, context):
    super(HelloWorld, self).setup(context)
    # Perform any necessary setup before starting the workload

    devpath = self.target.path  # os.path equivalent for the target
    self.target_infile = devpath.join(self.target.working_directory, 'hi.txt')
    self.target_outfile = devpath.join(self.target.working_directory, 'outfile')

def run(self, context):
    super(HelloWorld, self).run(context)
    # Perform the main functionality of the workload
    cmd = 'time (echo Hello World >> {}) >>{}'

    for i in range(0,self.num_times):
        self.target.execute(cmd.format(self.target_infile,self.target_outfile))

def extract_results(self, context):
    super(HelloWorld, self).extract_results(context)
    # Extract results on the target
    self.host_outfile = os.path.join(context.output_directory, 'timing_results')
    self.target.pull(self.target_outfile, self.host_outfile)
    context.add_artifact('helloworld-results',self.host_outfile, kind='raw')

def update_output(self, context):
    super(HelloWorld, self).update_output(context)
    # Update the output within the specified execution context with the
    # metrics and artifacts form this workload iteration.
    content = iter(open(self.host_outfile).read().strip().split())
    for value, metric in zip(content, content):
        mins, secs = map(float, value[:-1].split('m'))
        context.add_metric(metric, secs + 60 * mins, 'seconds')

def teardown(self, context):
    super(HelloWorld, self).teardown(context)
```

```
        # Perform any final clean up for the Workload.
```

4.5 Proof Of Concept: Automating workload on the Jailhouse hypervisor using WA

Setup for the hackbench workload (included by default) is as follows:

Yes

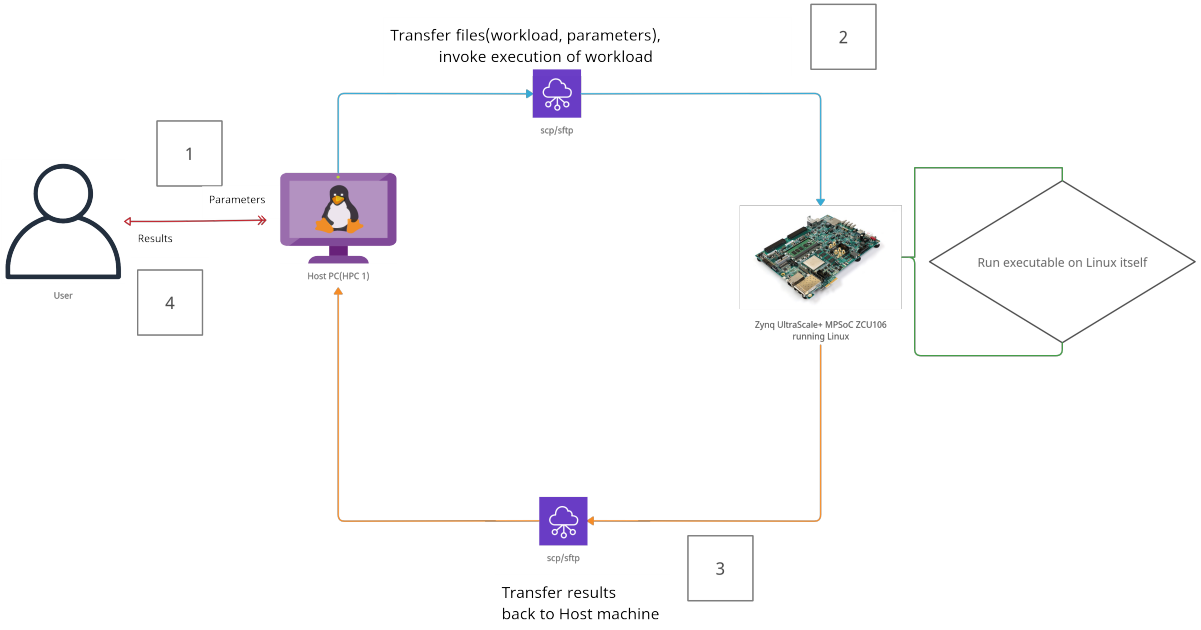

FIGURE 4.1: Hackbench Flow

Proposed setup for testing benchmarks on the Jailhouse Hypervisor using WA:

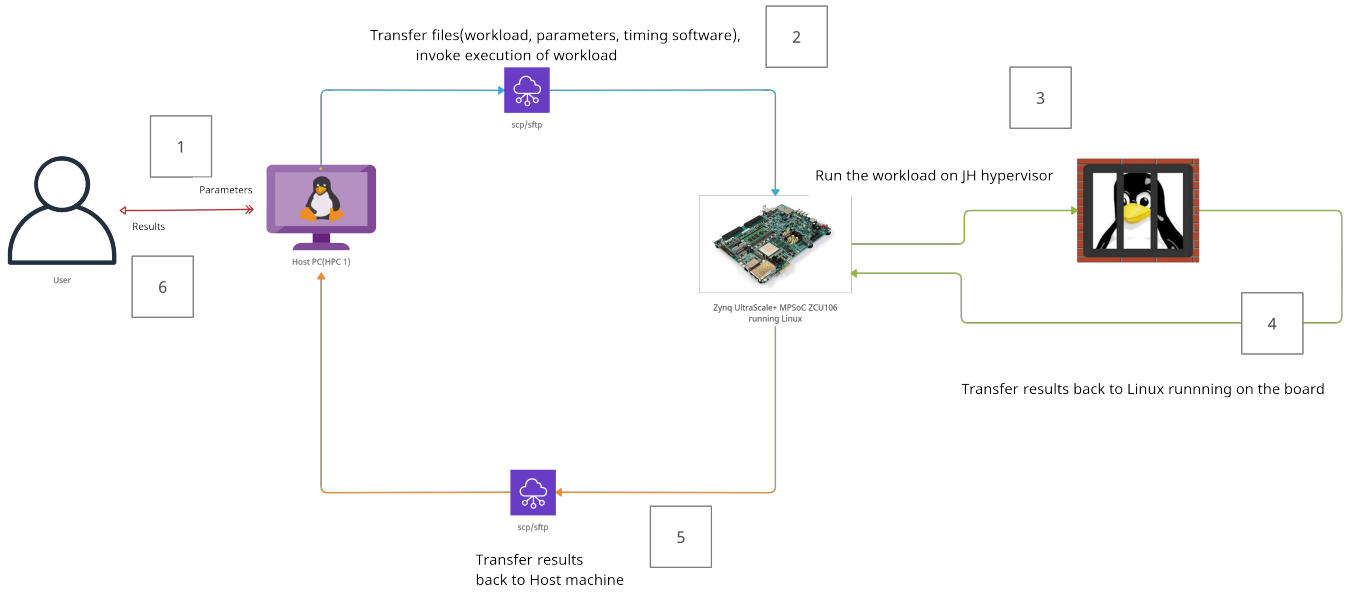

FIGURE 4.2: PoC WA with Jailhouse Flow

4.5.1 Execution Flow of the Proposed Proof of Concept

The important components in the PoC are:

- User - with remote access to the Host, connected via SSH
- Host Machine - running WA, connected to the board via SCP (Secure copy protocol).
- Embedded Board - having JH enabled and connected to Host Machine via SSH.

Steps for the execution:

1. User selects from a list of JH supported workloads provided in Workload-Automation tool.
2. The selected workload is loaded onto the board via SCP using WA.
3. Non-root cell configuration specific to the board architecture and the chosen workload is loaded on the board using WA. The definition of the cell configuration is provided as part of the workload in WA.
4. JH runs on the board and when it finishes, the result is stored in a .txt format on the board.

5. Results are transferred from the board back to the Host machine using SCP - this step is performed by WA in the *extract_results* step.

6. In the final *update_output* step, the output is stored in the required format - .json/.csv format and presented to the user.

Bibliography

[1] *About SDL*. URL: https://www.libsdl.org/.

[2] Waqar Ali and Heechul Yun. *RT-Gang: Real-Time Gang Scheduling Framework for Safety-Critical Systems*. 2019. arXiv: 1903.00999 [cs.DC].

[3] Nancy Arya, Mukesh Gidwani, and Shailendra Kumar Gupta. "Hypervisor Security - A Major Concern". In: *International Journal of Information and Computation Technology* (2013), pp. 533–538.

[4] *Developer Information - Workload Types*. URL: https://workload-automation.readthedocs.io/en/latest/developer_information.html#workload-types.

[5] *Download Alpine Linux. Small. Simple. Secure.* URL: https://alpinelinux.org/downloads/.

[6] *Download QEMU*. URL: https://www.qemu.org/download/.

[7] Siemens. *Jailhouse Inmate demo for zcu102*. URL: https://github.com/siemens/jailhouse/blob/master/configs/arm64/zynqmp-zcu102-inmate-demo.c.

[8] Siemens. *Jailhouse Linux demo for zcu102*. URL: https://github.com/siemens/jailhouse/blob/master/configs/arm64/zynqmp-zcu102-linux-demo.c.

[9] *Small. Simple. Secure.* URL: https://alpinelinux.org/.

[10] *Standard Filesystem Paths - Yocto*. URL: http://git.yoctoproject.org/cgit.cgi/poky/plain/meta/conf/bitbake.conf?h=purple.

[11] Certification Authorities Software Team. *CAST-32A: Multi-core Processors*. Tech. rep. November 2016.

[12] *Transmission Control Protocol*. 2021. URL: https://en.wikipedia.org/wiki/Transmission_Control_Protocol.

[13] *Ubuntu Manpage: hackbench - scheduler benchmark/stress test*. URL: http://manpages.ubuntu.com/manpages/xenial/man8/hackbench.8.html.

[14] *Using virtio_net For The Guest NIC*. URL: https://www.linux-kvm.org/page/Using_VirtIO_NIC.

[15] *What is KVM?* URL: https://www.redhat.com/en/topics/virtualization/what-is-KVM.

[16] *Workload-Automation*. URL: https://github.com/ARM-software/workload-automation.

[17] *Yocto Project*. URL: https://www.yoctoproject.org/.