AUTOMATED FRAMEWORK FOR VALIDATION OF EMBEDDED LINUX BUILD SYSTEM IN CONTINUOUS INTEGRATION ENVIRONMENT

M. Junaid Arshad[1*], *Ali Anwar*[2], *Amjad Hussain*[1]

[1]Department of Computer Science and Engineering, University of Engineering and Technology, Lahore, Pakistan
[2]Mentor Graphics, Floors 5-6-7-8, Ali Tower, MM Alam Road
105-B2 Gulberg-III, Lahore, Pakistan

Abstract

Availability of open source build systems like OpenEmbedded can simplify the usage of embedded Linux by allowing creation of highly customizable Linux distributions in time efficient fashion. Testing matrix becomes huge if build system supports various host operating systems and target platforms. In this paper we present a comprehensive model for executing unattended tests covering both the host side build system testing and target side runtime testing including benchmarking of an embedded Linux. This model facilitates testing for “Agile distributed development” in continuous Integration environment. Utilizing this model enabled us to identify some critical issues that would have gone undetected otherwise.

Keywords: Test Automation Framework; Embedded Linux Build System Verification; Automated Open Source Benchmarking; Continuous Integration of Embedded Linux

I. Introduction

Linux has been enjoying popularity in embedded domain for quite some time now [1], [2]. Developments of frameworks that build embedded Linux distributions like OpenEmbedded have also played a major role in gaining this popularity by simplifying the development process [3]. Also, major industrial players are still collaborating to simplify development of Embedded Linux distribution, Yocto project for instance [4]. There are a number of companies that make use of such open-source frameworks, perform proprietary value addition and roll out of their customized embedded Linux distributions [5], [6]. However, due to nature of application of embedded Linux [7], the importance of testing of build system and overall embedded distribution cannot be ignored. Moreover, if the Embedded Linux Build System has to provide support for a large matrix of host operating systems and embedded platforms, manual testing becomes impractical. Similarly, execution of runtime functionality tests and performance benchmarking becomes daunting task if newer versions are released quite often, which is nothing abnormal. Situation further aggravates if more than one embedded platform is to

[1] Corresponding Author E-mail: junaidarshad@uet.edu.pk
Phone no.: 0092-42-99029260;

be supported as all runtime tests are to be repeated for every supported platform. Apart from the human effort required to test all the possible configurations, scenarios and platforms the chance of error or missing out anything becomes more plausible.

We faced similar issues in testing Mentor Embedded Linux (MEL) while working as part of agile teams. MEL includes OpenEmbedded based build system called System Builder and it supports large number of platforms based on PowerPC, ARM and MIPS architectures. We had to ensure that System Builder is able to build images for every supported platform with supported configurations. We also needed to ensure that System Builder works seamlessly on all supported host operating systems that included RHEL 5.4, Ubuntu 8.04, Ubuntu 10.04 and CentOS 5.x meaning that whole testing effort for one host OS had to be repeated for every supported host OS.

We also had to test that System Builder generated images work as desired by running them on target platform, verify overall functionality by execution of open-source test suits like Linux Test Project (LTP). Benchmarking of various important embedded system components like CPU through Dhrystone, memory through LMBench, of network stack through IPerf and of file system through IOZone was also required. Here again, execution of images, test suites and benchmarks had to be repeated for every supported target platform.

To address above issues we have come up with an automated testing framework. We have sub-divided the testing effort into two parts host side and target side testing.

Host side testing involves building Linux kernel with different set of configurations which is usually controlled using “defconfig/.config” file, root file-system with different set of applications packaged together and it may or may not include u-boot images created for different boot setups. It could also include building with BOM, GCOV support or Binary to source tagging being enabled.

Target side testing involves booting up of kernel compiled with different configurations, testing device drivers, applications packaged within the file system which even includes testing of packages like BusyBox using test suites or custom tailored set of test case. In additions to functional testing target side testing also includes the execution of propriety or open source benchmarks to cover performance testing.

The remainder of the paper is organized as follows: Section 2 discusses issues that were faced during testing in the absence of framework being presented and utilization of open-source and proprietary tools along-with software components that were developed in-house to cope with the challenges. Section 3 explains build verification testing. Host side testing is explained in sections 4 and 5 followed by explanation of target side testing in section 6. Section 7 talks about role of Jenkins server in our automated framework. Result analysis is done in section 8 and limitation of our automation framework and future work has been discussed in sections 9 and 10 respectively. In section 11 we have presented our conclusion and references are provided at the end.

2. Requirements for an Automated Testing Framework

While performing automated testing of such build systems and generated images on the target platforms, issues typically encountered are mentioned below. Solutions devised to address these issues are also discussed.

2.1 Automation using Continuous Integration (CI) tools

In agile development process, to ensure timely detection of issues,   continuous testing is required. Continuous Integration tools help us meet this challenge. There are number of tools available like Jenkins [11], Hudson [12] and CruiseControl [12] etc. In our case we have used Jenkins [11]. We have used Jenkins setup to accomplish this but proposed framework would also be applicable if used with any of the other CI tools, and not just Jenkins.

Jenkins

Jenkins [11] is an open source CI tool. It is written in Java and is server-base system. Jenkins provides you the facility to check out the source from a Git, CVS or Subversion controlled repositories. After checking out the source it performs a set of instruction to execute a complete build. These set of instructions are called Jenkins script or simply build script. The build script is used to create a final installer and after which rest of the testing is carried out on that installer by another Jenkins job.

Jenkins Server

Jenkins Server is responsible for managing the administrative tasks which include triggering jobs fetching sources from repositories and executing the build script on appropriate node or slave configured while creating a Jenkins Job**.**

Jenkins server plays a central role and performs jobs synchronization. Figure 1 shows how the flow of testing is governed by Jenkins server. It keeps track of the jobs it executes. Each job execution is assigned a unique build tag. This build tag consists of job name and sequentially increasing execution number.

This unique build number is used internally by Jenkins to keep the jobs in synchronization. The same build tag is used by image transfer APIs while transferring the images to the TFTP and NFS servers. This build tag is also dumped in result database.

There arises such scenario where the execution of second job is not yet done and first job completes couple of times. The jenkins server creates a queue of jobs that needs to be executed in correspondence with each build and executes the second job in a sequential manner.

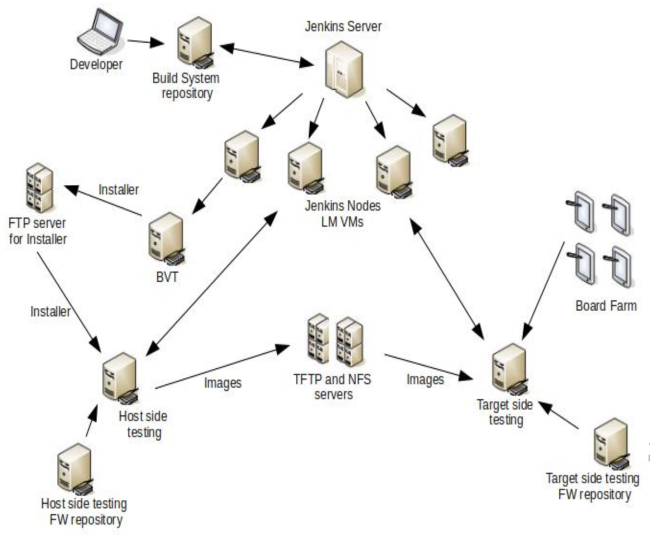


Figure 1: Flow of testing governed by Jenkins server

Jenkins Node/Slave

Jenkins Server is responsible for managing all the available machines on which the build script is required to be executed. All such machines are added to the list maintained by the Jenkins server and these machines are called Jenkins nodes or slaves.

Another advantage offered by CI tools is that they can be used to control and link self-contained components in an effective manner.

When and how the target side testing is to be triggered is an important question. Host side of the framework should be able to trigger the target side testing along with the information about the location of images.

2.2 Meeting Environmental dependencies using Virtual Machines

The process of automating certain test cases requires that the system used for executing the tests meets all the environmental dependencies. These dependencies could be regarded as the external elements necessary for the execution of test cases. Build system test cases have complex dependencies like availability of certain packages like python, python-pexpect, python-database etc.), external tool chain or some shared libraries.

The automatic initiation of complex environment necessary for tests execution could be achieved by creation of Virtual machine on the fly [8]. Lab manager is software which could be used to create the virtual machines on the fly and thus no permanent allocation of hardware for the framework is required.

Jenkins plug-in to manage Lab managers virtual machines

The Lab manager Virtual machines are added to the list of known slaves for Jenkins Server. There is a Jenkins Plug-in for managing Lab Managers virtual machines available which is used. This plug-in provides the support to power up the machine only when required and after the completion of Jenkins job the virtual machine is powered off hence resource management is effectively enhanced.

2.3 Board Farm for target testing

Non availability of hardware has always been an issue when it comes to testing of embedded system. So a Board Farm is required which follows proper queued reservation system to ensure the timely availability of hardware for testing purpose.

The board farm houses number of boards that can be accessed remotely for automated testing or direct user interaction [17].

The interface to the board farm needs to be handled through a set of APIs designed for such a purpose. Host side of testing framework needs to interact with the Board Farm database using those APIs. Following are some common python based in-house developed methods in our framework.

```
def get_avail_board_ids(self):
def checkout_target(self, id):
def checkin_target(self, id):
def power_reset_target(self, id):
def setup_telnet(self, id):
```

2.4 Transporting images to servers

The images generated by host side framework needs to be transferred to the NFS and TFTP servers for target side testing. A certain set of APIs is required which could take care of transfer process. This could be controlled through the configuration file containing the information regarding the data that needs to be transferred along with the destination.

```
def copy_bin(self):
def copy_dtb(self):
def copy_and_extract_fs(self):
def dir_chmod(self):
```

2.5 Parallel Execution

Compilation time of embedded Linux along with the root file system on typical dual core Host OS is roughly in hours. So there is a need to parallelize the tasks to reduce the overall time required for testing. Hence Jenkins job needs to be triggered in a way that it executes tests in such fashion.

2.6 Configurable Test Cases

To increase reusability and portability test cases need to have generic interface. Target specifics are supplied using configuration (*.cfg) files. The framework also contains a *.cfg file parser which extracts such data and provides this to assertion based test cases. In following example test to build BusyBox is kept generic but *.cfg file points for which target platform test needs to be executed.

```
# executing for PPC
MACHINE = mpc8536ds, TOOLCHAIN=PPC
```

```
testBusyBoxBuilding
#executing for arm target
MACHINE =dm37x, TOOLCHAIN=ARM
testBusyBoxBuilding
```

3. Build Verification Test (BVT)

In the framework multiple Jenkins jobs have been defined to perform different isolated tasks. Each job is configured not only to perform that task but also to trigger the next job. The first job in the chain is configured such that it observes source repository link for any change. This is a feature provided by the source control management part of the CI tool. Observation time period is configurable. As soon as any change is observed, requirements for the execution of Build Verification Test (BVT) are fulfilled, and BVT is run. The main purpose of Build Verification Test is to create an installer of build system which then can be installed on any supported host. Once the build system is installed it is used to build embedded Linux with specified kernel or file system configuration.

Utilization of Virtual Machines (VMs) and their management through Lab Manager simplifies the process. VMs check out source of a proprietary Build System i.e. Mentor Embedded System Builder from configurable repository link and performs BVT. On successful completion, installer for Platform Development Kit (PDK) is generated and copied over to FTP server otherwise failure notice is sent to configurable email address. In both cases, execution log is available at Jenkins web interface. Last task in this job is to trigger next job with the installer location for testing of generated PDK as displayed in Figure 2.

3.1 Importance of BVT with respect to testing under process

The purpose of build verification test is not only to create an installer of a build system for embedded Linux but it also acts as a pre check to insure that the build system created is in a state to be tested. For a build system to be in a state to be tested requires that at least embedded Linux kernel and minimal image of file system can be build without any issue. If we are facing any issue while building simply configured kernel and minimal root file system that it does not make sense to proceed with the complex testing.

4. Triggering Host Side Testing

This job is triggered with a parameter i.e. location from where the installer of product under test could be fetched. Testing a proprietary build system also includes the testing of installer produced by the first job.

Installer testing could be skipped however a script to install PDK is required. As soon as the installed location is identified the host side testing is initiated.

Figure 2 explains the flow of testing. First we checkout the testing scripts. Then we start installer testing and try to install the PDK, failure in doing so triggers an e-mail and no further testing is carried out. Once we have installed PDK we carry out host side testing and transfer the generated images to the appropriate servers for the target side testing.

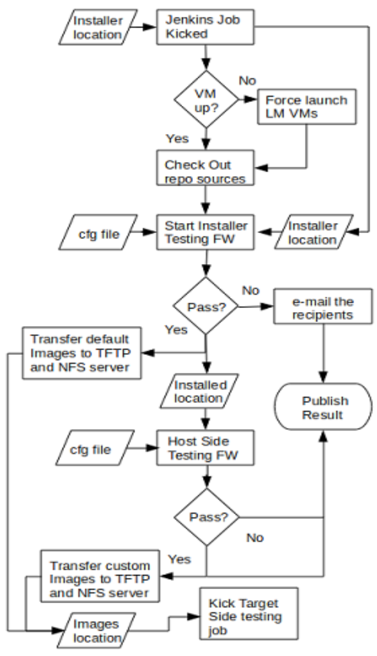


Figure 2: Triggering host side testing via Jenkins job

5. Host Side Testing Framework in a CI Environment

Host side testing consists of Jenkins Job configured to execute assertion based framework and APIs to dump results in database. Figure 3 explains the architecture of the host side testing. Actual testing scripts controls the utility scripts and uses PyUnit based framework for assertions. The purpose of utility scripts is to communicate with the test database to dump the test results and APIs which are used to transfer kernel binaries, file system etc to the NFS and TFTP servers.

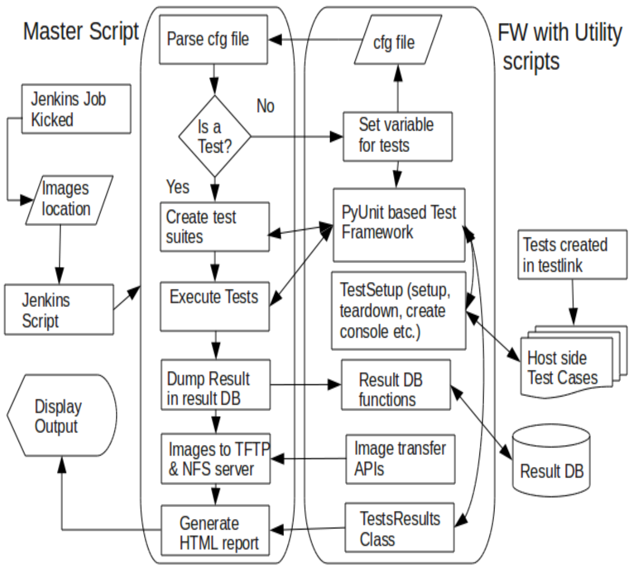
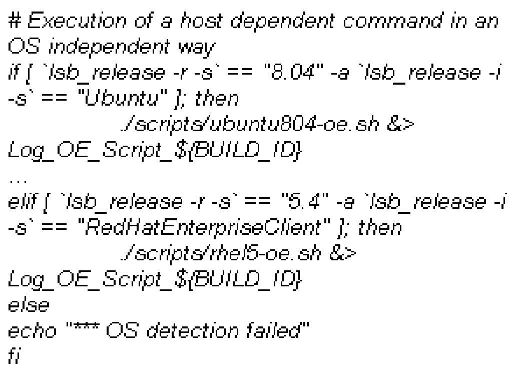


Figure 3: Host side testing

5.1 Jenkins script for host side

Jenkins script setups environment for framework execution which includes installation of cross tool-chain if it's not already part of the build system, along with the tool-chain updates if required.

The script is host independent i.e. it can be executed for testing on all supported host OS as shown in Figure 4.



```
# Execution of a host dependent command in an
OS independent way
if [ `lsb_release -r -s` == "8.04" -a `lsb_release -i
-s` == "Ubuntu" ]; then
                ./scripts/ubuntu804-oe.sh &>
Log_OE_Script_${BUILD_ID}
...
elif [ `lsb_release -r -s` == "5.4" -a `lsb_release -i
-s` == "RedHatEnterpriseClient" ]; then
                ./scripts/rhel5-oe.sh &>
Log_OE_Script_${BUILD_ID}
else
echo "*** OS detection failed"
fi
```

Figure 4: Jenkins script

Jenkins script executes the Master script which is a wrapper over the assertion framework used for actual testing.

5.2 Master Script

Master script is a wrapper over the assertion based framework. It performs four basic tasks which include configuration file parsing, creation and execution of test suites, dumping test results in Result Database and generation of HTML based report.

5.3 Assertion framework for host side test cases

Each test case corresponds to a unique test scenario described in Testlink [9]. In our case, testing framework has PyUnit [10] based scripts to verify different assertions. The test cases in the framework are kept generic and configurable so that they can be executed with appropriate input parameters for another target platform as explained earlier. Another important point is that these tests need to be executed on different OS so they are also Host independent. Each Test case contains a special *SetUp* and *Teardown* function in a similar way the unit tests are carried out normally.

5.4 Result database and APIs to dump the test results in that database

To dump the test results a proper database is used. These results could be used to analyze the project stability over a very long period of time for a specific test scenario.

Proper APIs are used to dump the result of individual test cases along with the additional info. Additional info contains the time taken by each test case, Build Tag of the job that executed the test case, host ip, host OS etc. The same set of APIs will also be used for target side test case result compilation.

```
def ASSERT_FALSE(self, test_name, test_id, location = None, result_string = " ",
additional_results = None):
def ASSERT_TRUE(self, test_name, test_id, location = None, result_string = " ",
additional_results = None):
```

5.5 APIs to transfer images to the TFTP and NFS server

The images go through the transfer process if the test case passes. The data that needs to be transferred is configurable. Configuration file are used for this purpose.

```
# Executing for PPC
TRANSPORT_IMAGES = MPC8572ds
 serverPath->/tftpboot/build_mpc8572d
```

6. Trigger the Target Side Testing

The Jenkins job used for target side testing is in the downstream of the job which executed the host side testing framework. This job is triggered with a predefined set of parameters which include the location of images transferred by the previous job to TFTP and NFS servers as explained in Figure 5.

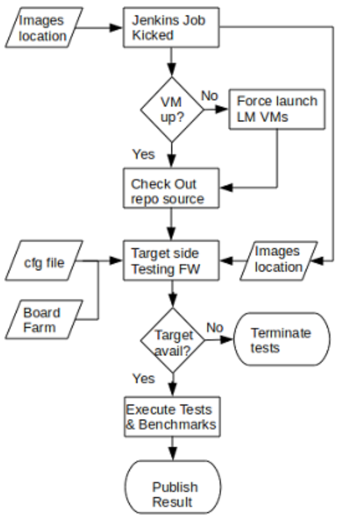


Figure 5: Triggering target side testing via Jenkins job

7. Target Side Testing Framework in a CI Environment

Figure 6 shows the testing of target side. The APIs are used to check out the target platform from the board farm configured with proper database to keep reservation record.

The test results are dumped into the same results database that is used by the host side of the framework. The assertion framework is able to communicate with the checked out board through serial as well as terminal connection. The APIs used to communicate with target either via serial or terminal connection is also the part of target side testing.

The u-boot or any other boot loaders environment is set according to the configuration in the *.cfg file. Once the boot environment is set we can boot target using kernel and file system created with different configurations.

The open source test suites and open source benchmarks are also executed as part of the target side testing.

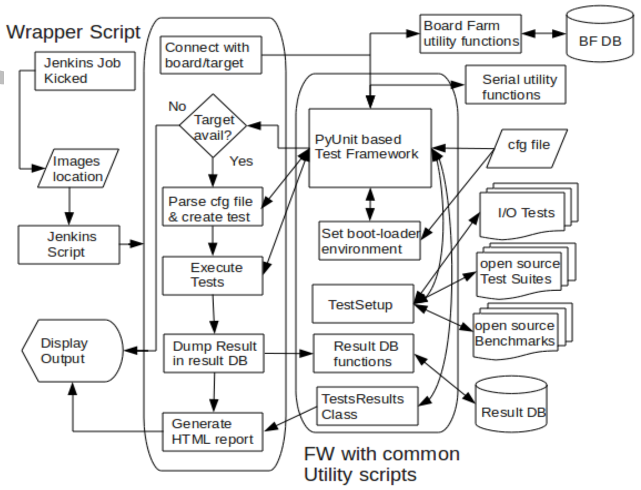


Figure 6: Target side testing

7.1 Configuration of Jenkins Job along with Lab Manager VM for target side

This job controls the Target side testing framework. It checks out the assertion framework along with APIs for board farm and Result database access. The job accepts the data being passed by the host side testing job and triggers the assertion framework with appropriate set of input. It is important to execute tests for the appropriate set of images produced by the same build that triggered this testing process.

7.2 Jenkins Script for target side

Jenkins script is used to execute the assertion framework. For target side testing the Jenkins script needs not to be host independent. You could execute the job on any host you are comfortable with.

7.3 Assertion framework for target side test cases

All the test cases are being executed via an assertion based framework. Target side of the testing framework is also written as PyUnit [10] based scripts to verify different assertions. The test cases in the framework are generic. The Framework uses the images generated by the host side framework for this particular iteration. The Framework is able to utilize different set of APIs that will enable it to reserve actual hardware target.

Another set of Utility functions are used enables the framework to communicate with the target via serial as well as telnet session. A configuration file is used to tune the test cases to be executed with desired parameters and on a proper hardware.

7.4 Board Farm with Queued reservation system

A board farm is used that allows the developer/tester to connect to a target via simple web based interface. With a database that keeps track of the actions being performed the reservation system is be able to handle the queries in appropriate manner. A request to check out a specific target is held in a queue unless specified otherwise.

7.5 APIs to communicate with Board Farm

The board farm is created in such a way that board could be checked out and used in a distributed environment using the web interface or directly communicating with the reservation system via set of APIs explained earlier. The APIs parse the database for the required target and report back with the list of available targets that matches the description. APIs also provide the facility to check out a target from the database if it is available and to check in the target etc.

7.6 Common Utility Functions/APIs to control the target

Two set of APIs are used which control the target through serial as well as telnet session. Serial connection is used in case the direct access to the board is available where as telnet session is used by the remote users in a distributed setup. These common Utility functions or APIs allow users to send command and get the output. Pexpect is used to verify the output for each command executed in both scenarios.

APIs for bootloader

One category to facilitate testing of u-boot, a set of function that executes and provide the output of the commands executed at u-boot terminal.

```
def send_command_and_get_output(self, command, expected, waittime):
def load_kernel(self):
def stop_auto_boot(self,console):
```

APIs to interact once kernel is up

Another category covers the functionality once the kernel is already up and running. These functions are used to execute certain tasks at the Linux console.

```
def send_command_and_get_output(self, command, expected, waittime):
```

8. Result Analysis

The presented automated framework performs efficient build system and run-time validation of embedded Linux in continuous integration environment. Major advantages gained by using this framework include comprehensive testing coverage, time saving, efficient hardware resource utilization, identification of subtle bugs and reduced mistakes due to minimized human involvement.

8.1 Increase in Test Coverage

While executing the test cases manually, repeating same tests on all supported hosts is a laborious task. But due to automation resource utilization was optimum so it was easy to repeat complete set of test cases on each supported host as well as supported target platforms.

The test coverage is directly proportional to the number of supported hosts OS and the target platforms. For a single platform executing tests in automated way could increase test coverage up to 75%.

8.2 Reduction in Testing effort measured in Man days

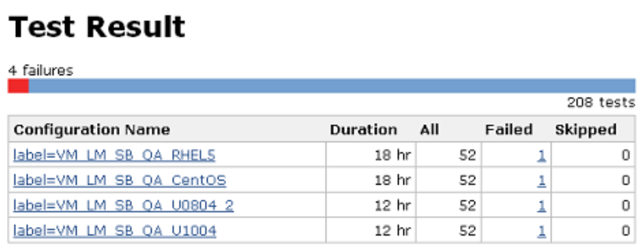
The reduction in testing time could be evaluated for the both sides individually.

Effort Reduced in Man Days for Host side testing

There is huge time saving since no human resource needs to be allocated for the execution of test cases. Testing is performed repeatedly during the entire development process to ensure the quality and timely detection of any issue.

Figure 7 explains each test execution cycle consumed around 12 to 18 hours for a single host OS intended for single target platform. If these tests would have been executed manually without any script the testing effort would have been huge.

The main advantage of the framework is that it does not require any human interference not even for deploying the images in actual targets hence allowing us to execute tests in fully automated fashion.


Test Result

4 failures

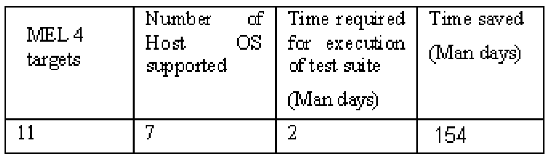
208 tests

Configuration Name	Duration	All	Failed	Skipped
label=VM_LM_SB_QA_RHEL5	18 hr	52	1	0
label=VM_LM_SB_QA_CentOS	18 hr	52	1	0
label=VM_LM_SB_QA_U0804_2	12 hr	52	1	0
label=VM_LM_SB_QA_U1004	12 hr	52	1	0

Figure 7: Testing matrix being displayed at Jenkins web interface

Following calculation in Figure 8 shows time saved while executing a single host side testing cycle on the Mentor Embedded Linux.

*Time saved in Man Days = (Number of Host OS * Number of Target platform supported * Avg. time required for execution of test suite)/8*



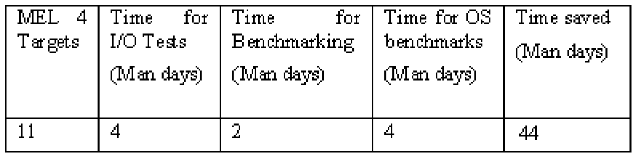
MEL 4 targets	Number of Host OS supported	Time required for execution of test suite (Man days)	Time saved (Man days)
11	7	2	154

Figure 8: Host side testing (time saved in Man days)

Effort Reduced in Man Days for Target side testing

The automated way of testing target side not only reduced the time required for testing but also enabled us to utilize the hardware resources efficiently. The APIs written to communicate with target and board farm made sure that target is checked out as long as it is required only. Following calculation in Figure 9 shows time saved while executing a single host side testing cycle on the Product.

*Time saved in Man Days = (Avg. time required for the execution of I/O tests + Avg. time required for the execution of Benchmarks + Avg. time required for execution of Open Source Test suites) * Number of Target platform supported / 8*



MEL 4 Targets	Time for I/O Tests (Man days)	Time for Benchmarking (Man days)	Time for OS benchmarks (Man days)	Time saved (Man days)
11	4	2	4	44

Figure 9: Target side testing (time saved in Man days)

8.3 Identification of subtle issues

The build systems are usually multi threaded and execute the build processes in parallel threads (Openembedded and MEL utilizes BitBake [14] for that purpose). But there could be build dependency of one binary on another. Hence issues caused by race conditions could be encountered while executing such a build system. As such issues are not guaranteed to occur on each execution so they are difficult to reproduce especially if manual approach is being used.

The probability of identification of such an issue is directly proportional to the numbers of times the build is executed. The probability can further be increased by specifying different number of threads and jobs to be created while each execution of build system. Example of such an intricate bug identified by the automated framework was a race condition in which a process tried to read a file which was yet being written by other process resulting in build failure.

9. Limitation

There are certain limitations of any automation framework so is the case with this framework. It could not be used for verification of such target side test cases which requires the physical interaction e.g. changing the HW switches to perform the target boot from an alternative flash, connecting target with SGMII interface instead of RGMII interface etc. On the other hand connecting multiple targets to meet such requirements appears more to be a waste of hardware resources.

10. Future Work

Sometimes it’s difficult to reproduce the reported issue and often takes a lot of time. As a proposed future work, the presented framework can be enhanced to reproduce the reported scenario by utilizing the data collected while testing e.g., the revision number of repository, target used for testing along with the environment configuration (*.cfg file). Implementing this feature would help the resource investigating the issue.

11. Conclusions

In this paper we presented the use of an automated framework for testing of embedded OS that simplifies the testing process greatly. The framework not only executes the host side testing but also covers the target side testing and thus is very useful for testing complete embedded Linux distributions. In nutshell, usage of this framework resulted in improved product quality.

References

[1] D. Geer, “Survey: Embedded Linux Ahead of the Pack”, IEEE Distributed Systems Online, Vol. 5, No. 10, pp. 3, 2004.

[2] Embedded Linux gets a boost in newly unified project, http://www.pcworld.com/businesscenter/article/221151/embedded_linux_gets_a_boost_in_newly_unified_project.html, 2011.

[3] C. Lauer. Building Embedded Linux Distributions with BitBake and OpenEmbedded. In Proceedings of the Free and Open Source Software Developers’ European Meeting (FOSDEM), Brussels, Belgium, Feb. 2005.

[4] Yocto Project, http://www.yoctoproject.org/, 2011.

[5] MontaVista Linux 6, http://mvista.com/product_detail_mvl6.php, 2011.

[6] Mentor Embedded Linux, http://www.mentor.com/embedded-software/linux/, 2011.

[7] Wind River Linux 4, http://www.windriver.com/products/linux/, 2011.

[8] Vander Burg, S. Dolstra, E "Automating System Tests Using Declarative Virtual Machines", Software Reliability Engineering (ISSRE), 2010.

[9] Testlink, http://testlink.sourceforge.net/docs/testLink.php, 2011.

[10] PyUnit, http://pyunit.sourceforge.net/, 2011

[11] Jenkins, http://jenkins-ci.org/m 2011.

[12] Hudson, http://hudson-ci.org/, 2011.

[13] CruiseControl, http://cruisecontrol.sourceforge.net/, 2011.

[14] Bitbake, http://bitbake.berlios.de/manual/, 2011.

[15] G. Tassey, The economic impacts of inadequate infrastructure for software testing, National Institute of Standards and Technology.

[16] A. Claudi and A. F. Dragoni, Lachesis: a testsuite for Linux based real-time systems, presented at 13th Real-Time Linux Workshop, Prague, Czech Republic, 2011.

[17] Shoeleather lab, http://www.shoeleather-lab.com/, 2011.