Daniele Jahier Pagliari 196046
Luca Nassi 194675
Mattias Gebrie Tsegaye 192516
Michele Di Giorgio 198400
Stefano Ricci 198402

February 2013
Politecnico di Torino
Computer Engineering

Operating Systems for Embedded Systems
LXC-Bench Project Report

1 Introduction

The LXC-Bench project is an initiative born to port LXC inside a Linux-based automotive embedded system and verify its real capabilities in this environment. The main goals are:

1. To measure the overhead introduced by LXC virtualization on a desktop machine running a general-purpose Linux operating system, deploying some tests belonging to Phoronix Test Suite (PTS).
2. To effectively install LXC and PTS on a real embedded system and measure performances and overheads in this environment.

In the following parts of this document we will report a short introduction of these two tools and the results of the tests.

2 Linux containers: LXC

Linux Container (LXC) is a virtualization mechanism for Linux systems. It is able to implement resource management, process management and isolation, exploiting kernel features such as *process control groups* (*cgroups*) or root directory change (*chroot*). Aiming at efficiency, it follows an opposite approach if compared to many other virtualization tools. In fact, instead of starting from an emulated hardware (completely isolated), and then trying to reduce overhead and improve performances, it uses already efficient mechanisms (such as chroot), and builds up isolation. LXC can be used to simulate a single application or an entire system, and can provide different degrees of isolation. For the following tests, we will use version 0.7.5-3.

2.1 Installation

On a system which supports apt-get and has standard Ubuntu repositories, LXC can be installed by simply typing the command:

```
sudo apt-get install lxc
```

This will also automatically install all the dependencies, plus some other recommended package. Dependencies:

- upstart-job

- libapparmor1 ($>= 2.6 - devel$)
- libc6 (>= 2.8),
- libcap2 (>= 2.10),
- bridge-utils, dnsmasq-base,
- iptables
- rsync

Recommended:

- debootstrap
- libcap2-bin
- cgroup-lite, cgroup-bin
- openssl

2.2 Files organization and position

After it is installed, LXC creates a number of files and directories. On Ubuntu, they are placed at the following paths:

- */usr/bin*: contains the various containers management tools which can be run from the command line.
- */usr/lib/lxc/lxc-init*: contains a minimal "init" process used by LXC when it only has to run an application rather than an entire system (i.e. when you type the *lxc-execute* command).
- */usr/lib/lxc/templates*: contains the template files to build ad-hoc containers for certain distributions (Ubuntu, Fedora, etc.)
- */var/lib/lxc*: stores the containers and their configuration files.
- */var/cache/lxc*: stores caches of distribution data, to speedup multiple container creations.
- */etc/init/lxc.conf* : startup program that looks inside */etc/lxc/auto/* to find symbolic links to configuration files for the containers which the user wants to start at boot. It runs only if the variable `LXC_AUTO` is set to true (default) inside the file */etc/default/lxc*.
- */etc/lxc/lxc-net.conf* : sets up a NAT bridge for containers. Run only if */etc/default/lxc* has `USE_LXC_BRIDGE` set to true (default).
- */etc/lxc/lxc-net.conf*: default configuration file, used when creating a new container if no configuration file is specified. It directs containers to use the NAT bridge created by *lxc-net.conf*.
- */usr/share/doc/lxc/examples*: contains examples of configuration files for containers.

2.3 Main commands and features

Here we list some of the main commands of LXC. We do not aim at providing a complete description of all the possible commands and options, but we prefer to concentrate on those which are useful for the purposes of the project. It is possible to refer to the *man* pages of the commands for an exhaustive list of all their possibilities.

Create a container

- `lxc-create -n name [-f config_file] [-t template]`: creates a new container object inside */var/lib/lxc/<name>*. The command takes a possible configuration file to specify the separation degree of the container, otherwise it uses the default isolation options. The *template* option triggers a script to setup a container adapt to run some predefined systems (Busybox, Debian, Fedora, Ubuntu or SSHD). It downloads from the net the required rootfs and binaries in order to make those systems runnable without any further configuration. The object which is created, can then be used with the different LXC commands.
- `lxc-destroy -n name [-f]`: destroy an existing container that can be specified with its name. The -f options stops the container if it is running.

Starting and stopping

- `lxc-start  -n name [command]`: start a container with the configuration provided when doing *lxc-create* for that *name*. If no *command* is specified (this is the way we used it), the container will run the default */sbin/init* of the system inside the container.
- `lxc-execute -n name command`: run a command in a container, previously configured with *lxc-create*. It is different from *lxc-start* because it does not run the container's */sbin/init*, but uses a minimal init program called *lxc-init*, as an intermediate process to launch the application. This program will attempt to mount */proc*, */dev/mqueue* and */dev/shm*, will execute the program specified on the *command* field, and will wait for it to finish executing. It will also forward all signals (like kill signals etc.) to the command. In the running container *lxc-init* will have PID 1, while the command will have PID 2. In few words, *lxc-start* is used to start en entire system inside a container, while *lxc-execute* is good for running single applications in a separate space. For our tests, we used the first option.
- `lxc-stop -n name`: kills all processes running inside the container. Should be used only if the processes are no longer accessible, otherwise it is enough to simply shutdown the system/exit the application to go back to the host environment.
- `lxc-kill --name=NAME signum`: Sends a signal to process with PID 1 inside the container. If the container was started with *lxc-execute* the signal is forwarded to the process with PID 2 (except some un-forwardable signals). The signum has to be a numeric value, signal names are not allowed.

Information

- `lxc-checkconfig`: is a script placed in */usr/bin/* when LXC is installed. It can be used to verify the host support for containers. It prints out a series of kernel features which are used by LXC, telling if they are enable or missing in the current host.

- `lxc-info [-n name]`: prints information on the state of a container.
- `lxc-list`: list all existing containers in the host, ordered by state (running, frozen, stopped).
- `lxc-ps [--name name] [ps option]` lists all processes belonging to the container specified by *name*. It is a script built on top of *ps* so it accepts all the options of that command.

Configuration

When running a *single application* application inside LXC, it is possible to decide what resources to isolate. The default configuration separates the PIDs, the SysV Inter Process Communication resources and the mount points. But it is also possible to separate the network stack and the entire rootfs.

It is not mandatory to create a container object before to start it. It is possible to create and start the container at the same time, providing the configuration file as parameter of the *lxc-execute*.

On the contrary, when running an *entire system* inside a container, everything must be isolated. It is still possible to re-use some files and folders of your host system by binding them to the correct mount point inside the container's rootfs.

Example of some *fstab* entries for the contained system:

```
/dev               /home/root/debian/rootfs/dev              none  bind  0  0
/etc/resolv.conf   /home/root/debian/rootfs/etc/resolv.conf  none  bind  0  0
```

The configuration file defines each resource's isolation. Each option of the file is in the *key* = *value* form. The # character is used for comments. Some of the possible options are:

- *lxc.arch*: specifies the architecture of the container. It is useful for containers running on 32bits on a 64bits host. Some of the valid options are: *x86, i686, x86_64, amd64*
- *lxc.network.type*: if the value is *empty* only the loopback interface will be created. *vlan* can be used to create a virtual LAN interface, linked with the host interface specified by *lxc.network.link*. The VLAN identifier can be specified in *lxc.network.vlan.id*. *phys* is used to physically link a host interface, specified in *lxc.network.link* to the container.
- *lxc.network.flags*: with value *up*, activates the interface.
- *lxc.network.link*: specifies the host interface used for real network traffic
- *lxc.network.name*: used to specify a name for the virtual interface. If not present, a default name is given, but it may be useful if the network configuration inside the container works for a particular name (such as *eth0*).
- *lxc.network.hwaddr*: specify the MAC address of the virtual interface. Also in this case, if the field is not present, a default MAC is given.
- *lxc.network.ipv4*: specify the IPv4 address for the interface, in the format *address/netmask*.
- *lxc.network.ipv4.gateway*: default gateway for the virtual interface.

- *lxc.network.ipv6* and *lxc.network.ipv6.gateway*: similar to the previous two but dedicated to IPv6 configurations
- *lxc.console*: specify a file where to write the console output for the container. If set to *none* the console will be disabled.
- *lxc.mount*: path to a file in the *fstab* format with the mount points for the container.
- *lxc.mount.entry*: specify a single mount point without using a file, in the format of a line in *fstab*.
- *lxc.rootfs*: root file system for the container (image file, directory or block device). If not specified, the container shares its root file system with the host.

There are many other options, to include additional configuration files, to specify scripts to be run in the various steps of the creation of the container, etc. All of them are described inside `man lxc.conf(5)`.

2.4 Examples

Run a single application inside a default container

This example shows how to create a default container named *def_container* and then run a bash shell inside it. The container will have the parameters specified inside the file */etc/lxc/lxc.conf*, which in this example is not touched after installation:

```
sudo lxc-create -n def_container
sudo lxc-execute -n def_container /bin/bash
```

The new shell's prompt should appear. Trying to type *ls* inside it will confirm that, in this case, the file system is completely shared between the host and the container. The process namespace, on the contrary, is completely isolated. In fact the result of a process listing (*ps -e*) inside the container is the following:

```
 ~$ ps -e
    1 pts/1    00:00:00 lxc-init
    2 pts/1    00:00:00 bash
   54 pts/1    00:00:00 ps
```

As explained in the previous sections, using the *lxc-execute* command, exploits a minimal "init" process, which is only used to start the executable specified as parameter. The container user can only see that process, which in his namespace has PID = 1 and is the father of all the processes in the system, plus the bash shell, and *ps* itself.

Boot Ubuntu inside a container

LXC comes with a series of templates to easily set-up a standard distribution system (Ubuntu, Fedora, Busy Box, etc.) inside a container. This feature exploits a tool called *debootstrap* to download a copy of the file system of these distributions inside whichever folder of the local machine. Then we need to type *lxc-start* to specify LXC that we don't want to use the minimal "lxc-init" but normally look for the init process of the contained system into its directories:

```
sudo lxc-create -t ubuntu -n ubuntu_container
sudo lxc-start -n ubuntu_container
```

The *lxc-create* command can take several minutes to complete. After starting the system, we will be prompted with a login screen. On Ubuntu, the default credentials are:

```
login: ubuntu
password: ubuntu
```

This time, “ls” and similar commands can clearly show that we are in a separated root file system (by default, the home directory of the “ubuntu” user is empty), and “ps -e” produces as a result:

```
~$ ps -e
  PID TTY          TIME CMD
    1 ?        00:00:00 init
  113 ?        00:00:00 upstart-udev-br
  138 ?        00:00:00 udevd
  149 ?        00:00:00 rsyslogd
  152 ?        00:00:00 upstart-socket-
  183 ?        00:00:00 dhclient3
  193 ?        00:00:00 ntpdate
  197 ?        00:00:00 lockfile-touch
  200 ?        00:00:00 ntpdate
  205 ?        00:00:00 sshd
  233 ?        00:00:00 getty
  236 ?        00:00:00 getty
  238 ?        00:00:00 getty
  240 ?        00:00:00 cron
  257 ?        00:00:00 ondemand
  263 ?        00:00:00 sleep
  267 ?        00:00:00 login
  269 ?        00:00:00 getty
  289 lxc/console 00:00:00 bash
  299 lxc/console 00:00:00 ps
```

which is a more complex set of processes with respect to the previous example, but still it is a completely different tree from the one of the host machine.

Finally, we can halt the contained system as we normally do in Ubuntu, by typing from container command line:

```
sudo shutdown -h now
```

3 Phoronix Test Suite

Phoronix Test Suite (PTS) is a testing and benchmarking platform which supports hundred of tests and suites. It can run on different operating systems and on systems with different architectures. Moreover it is able to automatically retrieve from the net packages and files needed to perform the tests. The version used for our purpose will be *Phoronix Test Suite 4.0*, the latest stable release.

3.1 Installation

On a Ubuntu-bases system installation is very straightforward, just type inside a terminal:

```
sudo apt-get install phoronix-test-suite
```

The only dependency is PHP5 (packages are commonly named: *php5-cli* or *php-cli* or *php*), but *apt-get* would take care of it.

Instead, if a standalone suite is needed, it is possible to download the file *phoronix-test-suite-4.0.1.tar.gz* from *http://www.phoronix-test-suite.com/?k=downloads*. Once extracted, PTS can run locally without creating any file outside of its folder.

Files organization and position

If it is installed, PTS files are located at:

- */usr/bin/phoronix-test-suite*: executable
- */usr/share/doc/phoronix-test-suite/*: documentation directory
- */usr/share/phoronix-test-suite/*: core files directory
- *~/.phoronix-test-suite/*: user specific files, such as configuration, installed tests and suites along with their results etc.

3.2 Main Commands and features

Here is reported a list of the main commands belonging to Phoronix Test Suite, these has been used to install, configure and run the tests whose results are reported in the following sections.

Installation

These commands are used to install tests, they retrieve from the net all needed software and permanently store it locally, so that tests can be run in another moment.

- `phoronix-test-suite install [test | suite]`: install a test or a suite in the proper directory inside *~/.phoronix-test-suite/*
- `phoronix-test-suite install-dependencies [test | suite]`: install all the packages needed to run a test or a suite, it uses the services of the operating system (such as *apt-get* on Ubuntu-like systems)

- `phoronix-test-suite make-download-cache`: make a backup copy of the source of all tests installed, this copy is not modified by successive operations
- `phoronix-test-suite remove-installed-test [test]`: uninstall selectively a test (but not its dependencies)

Interactive mode testing

The following commands are used to start tests (or suites) in interactive mode, this means that each possible configuration parameter will be prompted and require a specific choice from the user.

- `phoronix-test-suite benchmark [test | suite]`: start a test or a suite, it is downloaded and installed if not already present on the machine
- `phoronix-test-suite run [test | suite]`: start a test or a suite but requires that it is already installed

Batch mode testing

These commands are used to let the tests automatically run once started, without prompting anything except non-interactive information. For this purpose a configuration file, which specifies the different parameters of the tests, might be needed.

- `phoronix-test-suite default-benchmark [test | suite]`: works as *benchmark* except that it runs the tests in batch mode using the default configuration
- `phoronix-test-suite default-run [test | suite]`: works as *run* except that it runs the tests in batch mode using the default configuration
- `phoronix-test-suite batch-setup [test | suite]`: set up the batch mode configuration, it prompts some questions whose answers are used to create or update the configuration file (*˜/.phoronix-test-suite/user-config.xml*)
- `phoronix-test-suite batch-benchmark [test | suite]`: works as *benchmark* except that it runs the tests in batch mode using the configuration file created with *batch-setup*
- `phoronix-test-suite batch-run [test | suite]`: works as *run* except that it runs the tests in batch mode using the configuration file created with *batch-setup*

Information

These commands are used to obtain information about the different tests and suites and about the system PTS is running on.

- `phoronix-test-suite system-info`: prints information about the system, both hardware and software. Even more information can be shown using *detailed-system-info*
- `phoronix-test-suite info [test | suite]`: print information about the test or suite specified, such as the stressed device, used techniques, expected duration, dependencies etc

- `phoronix-test-suite list-available-tests`: print a list of all available tests for the machine which call this command. Instead, *list-available-suites* retrieve a list of all available suites
- `phoronix-test-suite list-installed-tests`: prints a list of all installed test. *list-installed-suites* have to be used for suites
- `phoronix-test-suite list-missing-dependencies`: prints all the packages which are missing in order to properly run the installed tests. To view installed packages that are used type *list-installed-dependencies* while *list-possible-dependencies* can be used to find all dependencies of installed tests.

Managing tests results

In the end, the following commands can be useful to manage the results obtained from the tests, to create a tidy documentation.

- `phoronix-test-suite list-saved-results`: list all saved test results found on the system
- `phoronix-test-suite extract-from-result-file [test_result]`: used to extract a single result out of a result file which contains many tests together
- `phoronix-test-suite merge-results [test_result]`: merge different set of test results in a single file
- `phoronix-test-suite result-file-to-pdf [test_result]`: output a pdf file which contains the specified test results along with information about the system
- `phoronix-test-suite result-file-to-txt [test_result]`: prints on the terminal (possibly redirected to a textual file) the specified test results along with system information

3.3 Examples

Run tests in batch mode

This example shows how to run the 5 tests we have selected for the following experiments (see next Sections), in batch-mode. These tests are:

- Cachebench
- C-Ray
- Dbench
- LAME MP3 encoder
- Stream

For a description of the reasons why we selected these tests and why, please refer to the dedicated section below.

First of all install the dependencies (in terms of software needed to run the test) on your system:

```
phoronix-test-suite install-dependencies pts/cachebench pts/dbench pts/c-ray
                                                           pts/encode-mp3 pts/stream
```

Now download and install the tests:

```
phoronix-test-suite install pts/cachebench pts/dbench pts/c-ray
                                                           pts/encode-mp3 pts/stream
```

This process can take some times, but is fully automatic. Tests are downloaded and permanently stored in *~/.phoronix-test-suite/installed-tests*

At this point we need to configure the batch-mode:

```
phoronix-test-suite batch-setup
```

It will prompt some questions, here they are along with my answers:

```
These are the default configuration options for when running the Phoronix Test
Suite in a batch mode (i.e. running phoronix-test-suite batch-benchmark universe)
Running in a batch mode is designed to be as autonomous as possible, except for
where you'd like any end-user interaction.

    Save test results when in batch mode (Y/n): y
    Open the web browser automatically when in batch mode (y/N): n
    Auto upload the results to OpenBenchmarking.org (Y/n): n
    Prompt for test identifier (Y/n): n
    Prompt for test description (Y/n): n
    Prompt for saved results file-name (Y/n): n
    Run all test options (Y/n): y

Batch settings saved.
```

Now you are ready to run all tests in a sequential and automatic way (batch-mode).

```
phoronix-test-suite batch-run pts/cachebench pts/dbench pts/c-ray
                                                          pts/encode-mp3 pts/stream
```

It will not prompt anything, just print messages about the status of the operations, and the results of a test, once it has completed. The first print contains the system-infos (both software and hardware), making it easy to collect this information together with the results of the tests.

To save the output, you can use a cut-and-paste at the end of all tests (if your terminal has a buffer long enough to keep all printed messages) or redirect the output to a file (typical Unix style: *command > output.txt*). However PTS will also save the result in a HTML file located inside *~/.phoronix-test-suite/tests-result*, in a folder default-named as the day/time at which you started the test. This page will contain the results in a graphical form, so to properly show it, it is necessary to have some additional files, such as CSS templates etc, which are located inside *~/.phoronix-test-suite/tests-result/pts-results-viewer* (and the relative paths must remain the same).

In the end, if you want, you can save the tests you have downloaded (only the compressed source), so that even if you uninstall or change them, you still have a backup copy inside the "download cache" directory (*~/.phoronix-test-suite/download-cache*):

```
phoronix-test-suite make-download-cache
```

4 Tests on desktop PCs

4.1 Goals

The main goal of this first part of the project is to estimate the overhead introduced by the LXC virtualization, if any, on a generic Linux system. Considering that the final target is an infotainement system, a set of relevant tests from the Phoronix Test Suite has been selected and run in three different conditions:

1. In the host system
2. In a LXC container
3. Concurrently, in the host system and in the container

The hardware used is composed by five general purpose PCs (and laptops) and is documented in the dedicated section below.

To run the tests in the first two conditions, the batch-mode of Phoronix Test Suite has been used, as shown in the example. In the third case, we had to start the tests exactly at the same time, to guarantee that the influence of the host on the container and vice versa was present for the whole duration of the test, or at least until the first of the two completed. To achieve this, we used the *at* command, scheduling one test at a time, in order to recover synchrony at the beginning of each of them. It is important to say that *at* was not installed on the container by default, but it can be retrieved as usual from *apt-get*. The command we used was:

```
at [time] -f [script]
```

where [time] is the scheduled starting instant for the job, and [script] is a file containing the commands to run just one test in batch mode. The system-time of the container and the host is **not independent**, the only thing that can differ is the time zone. Thus, changing the date and time in the container will also affect the host. This could be an important issue for the scope of the project, but by now it is helpful. In fact, scheduling the same starting-time in the host and in the container (subtracting the offset caused by the time-zone) will actually produce a synchronous start. This can be checked using the *date* command.

4.2 Selected tests

For this first session of tests, we selected, together with professor Violante, a restricted subset of the Phoronix Test Suite, aimed at analyzing the impact of LXC on all the main parts of the systems (CPU, memory, disks, etc.) but keeping the cost of the tests in terms of time and disk space as low as possible. The subset is reported in the following table:

Name	**Tested device**	**PTS mnemonic**
Cachebench	Processor's cache	pts/cachebench
Dbench	Disk	pts/dbench
C-Ray	Processor's FPU	pts/c-ray
LAME MP3 Encoding	Processor	pts/encode-mp3
Stream	RAM	pts/stream

4.3 Tested machines

	Machine A	**Machine B**	**Machine C**	**Machine D**	**Machine E**
Owner	Michele	Daniele	Luca	Stefano	Mattias
Software					
OS	Ubuntu 12.04	Linux Mint 13	Linux Mint 13	Ubuntu 12.04	Ubuntu 12.04
Kernel	3.2.0-33-generic-pae (i686)	3.2.0-23-generic-pae (i686)	3.2.0-34-generic (x86_64)	3.2.0-33-generic-pae (i686)	3.2.0-29-generic (x86_64)
Desktop	Unity 5.14.0	MATE 1.2	MATE 1.2	Unity 5.14.0	GNOME Shell 3.4.1
Display Server	X Server 1.11.3	X Server 1.11.3	X Server 1.11.3	X Server 1.11.3	X Server 1.11.3
Display Driver	intel 2.17.0	NVIDIA 295.40	fglrx 8.98.2		intel 2.17.0
OpenGL		2.1.2 NVIDIA 295.40	4.2.11762	3.3.11627	
Compiler	GCC 4.6	GCC 4.6	GCC 4.6	GCC 4.6	GCC 4.6
File-System	ext4	ext4	ext4	ext4	ext4
Screen Resolution	1024x600	1280x800	1600x900		1366x768
Hardware					
Product	ASUS EeePC 1001PXD	Acer Aspire 5630	Home-assembled workstation	HP-EliteBook-6930p	Acer Aspire 5750G
Processor	Intel Atom N455 @ 1.67GHz (2 Cores)	Intel Core 2 T5500 @ 1.67GHz (2 Cores)	AMD Phenom II X4 965 @ 3.40GHz (4 Cores)	Intel Core 2 Duo P8700 @ 2.53GHz (2 Cores)	Intel Core i3-2350M @ 2.29GHz(1 core)
Motherboard	ASUS 1001PXD	Acer Grapevine	ASUS M4A88TD-M EVO	HP 30DC	Acer JE50_HR, Chipset
Chipset	Intel N10 Family DMI Bridge	Intel Mobile 945GM/P-M/GMS + ICH7-M	AMD RS880	Intel Mobile 4 MCH + ICH9M-E	Intel 2nd Generation Core Family DRAM
Memory	1024MB	2048MB	4096MB	4096MB	4096 MB
Disk	250GB Hitachi HTS54322	120GB Seagate ST9120821A	750GB Seagate ST3750330AS + 1000GB Seagate ST31000528AS	320GB Western Digital WD3200BEKT-6	320GB Western Digital WD3200BPVT-2