Lachesis: a testsuite for Linux based real-time systems

Andrea Claudi
Università Politecnica delle Marche, Department of Ingegneria dell'Informazione (DII)
Via Brecce Bianche, 60131 Ancona, Italy
a.claudi@univpm.it

Aldo Franco Dragoni
Università Politecnica delle Marche, Department of Ingegneria dell'Informazione (DII)
Via Brecce Bianche, 60131 Ancona, Italy
a.f.dragoni@univpm.it

Abstract

Testing is a key step in software development cycle. Error and bug fixing costs can significantly affect development costs without a full and comprehensive test on the system.

First efforts to introduce real-time features in the Linux kernel are now more than ten years old. Nevertheless, no comprehensive testsuites is able to assess the functionality or the conformance to the real-time operating systems standards of the Linux kernel and of real-time nanokernels that rely on it.

In this paper we propose Lachesis, an automated testsuite derived from the LTP (Linux Test Project) real-time tests. Lachesis is designed with portability and extensibility as main goals, and it can be used to test Linux, PREEMPT_RT, RTAI and Xenomai real-time features and performances. It provides some tests for SCHED_DEADLINE patch, too. Lachesis is now under active development, and more tests are planned to be added in the near future.

1 Introduction

Linux kernel is being increasingly used in a wide variety of contexts, from desktops to smartphones, from laptops to robots. The use of Linux is rapidly growing due to its reliability and robustness.

Thanks to these qualities, Linux is widely used in safety and mission critical systems, too. In these contexts, time is extremely important: these systems must meet their temporal requirements in every failure and fault scenario. A system where the correctness of an operation depends upon the time in which the operation is completed is called a real-time system.

Over the years various solutions have been proposed to adapt the Linux kernel to real-time requirements. Each of them has found extensive applications in production environments. Some of these solutions, like PREEMPT_RT [1] and, more recently, IRMOS [2] and SCHED_DEADLINE [3], are based on a real-time patch to the Linux kernel, which introduces or improves some essential features for a real-time operating system.

Other solutions use a real-time nanokernel in parallel with the Linux kernel. This is made possible through the use of an hardware abstraction layer on top of the hardware devices, that listens for interrupts and dispatches them to the kernel responsible for managing them. RTAI [4] and Xenomai [5] nanokernels, both built on the top of Adeos [6] hardware abstraction layer, belong to this category.

Although first efforts to enhance and introduce real-time features in the Linux kernel are now more than ten years old, nowadays there are no comprehensive testsuites able to assess the functionality or the conformance to the real-time operating systems standards for the Linux kernel and for the nanokernels that rely on it.

Properly testing the Linux kernel is not easy, since the code base is continuously growing and in rapid evolution. We need more efficient, effective and comprehensive test methods, able to ensure proper software behaviour in the wide range of situations where systems can be deployed. For example, tests are critical in an environment where a malfunctioning system can seriously damage machinery, structures or even human life.

Nowadays there are many automatic testsuites, covering an increasingly wide range of kernel features. Many of these testsuites make it possible to functionally test file systems and network stack, to evaluate efficiency in memory management and in communication between processes, and to assess standards compliance. Very few of them make functional testing on real-time features, and none of them test performances or conformance with real-time features.

1.1 Paper contributions

In this paper we propose Lachesis, an automated testsuite for Linux based real-time systems, derived from the LTP real-time tests. Lachesis main goals are:

- to provide extensive and comprehensive testing of real-time Linux kernel features
- to provide a common test environment for different Linux based real-time systems
- to provide a set of functional, regression, performance and stress test, either developing or porting them from other testsuites
- to design and experiment a series of build tests
- to minimize development time for new tests
- to make the testsuite extensible and portable

Several real-time tests were ported to Lachesis from other testsuites. In this paper we also detail porting procedures and results.

1.2 Paper structure

The rest of this paper is organized as follow: section 2 illustrates a taxonomy for testing methodologies; section 3 briefly introduces the most important testsuites for the Linux kernel and give some reasons why to choose LTP as start point to develop a new real-time testsuite; Section 4 illustrates Lachesis and the tests it includes; Section 5, finally concludes the paper.

2 Taxonomy of testing methodologies

In software engineering, testing is the process of validation, verification and reliability measurement that ensure the software to work as expected and to meet requirements.

The Linux kernel has been tested since its introduction. In the early stage of development tests were ad-hoc and very informal: every developer individually conducted tests on the portion of code he developed, with his own methodologies and techniques; frequently tests came after end-users bug reports, and were aimed at resolving the problem, identifying the section of code causing it.

Over the years kernel grew across many different architectures and platforms. Testing activities became increasingly difficult and costly in terms of time, but remained very critical for kernel reliability, robustness and stability.

For this reason different testing methodologies and techniques for the Linux kernel were experimented and used. Indicatively, these methods can be grouped into seven categories [7].

2.1 Built-in debugging options

This kind of tests must not be done simultaneously with functional and performance tests. It consists in a series of debugging options (CONFIG_DEBUG_* in the Linux kernel) and fault insertion routines that allow the kernel to test itself.

2.2 Build tests

Build tests just compile the kernel code searching for warnings and errors from the compiler. This kind of tests is an extensive problem, for a number of different reasons:

- Different architectures to build for;
- Different configuration options that could be used;
- Different toolchains to build with;

2.3 Static verification tests

This kind of tests is designed to find bugs in the code without having to execute it. Static verification tests examine the code statically with tools like sparse, LClint [8] (later renamed as splint), and BLAST [9].

2.4 Functional and unit tests

Functional and unit tests are conceived to examine one specific system functionality. The code implementing a feature is tested in isolation, to ensure it meets some requirement for the implemented specific operation. Crashme [10] is an example of this kind of test.

2.5 Regression tests

Regression tests are designed to uncover new errors and bugs in existing functionalities after changes made on software, such as new features introduction or patches correcting old bugs. The goal for this kind of tests is to assure that a change did not introduce new errors.

Regression testing methods are different, but in general consist in rerunning previously ran tests and evaluating system behaviour, checking whether new errors appear or old errors re-emerge.

2.6 Performance tests

Performance tests measure the relative performance of a specific workload on a certain system. They produce data sets and comparisons between tests, allowing to identify performance changes or to confirm that no changes has happened. In this category we can include kernbench, a tool for CPU performance tests; iobench, a tool for disk performance tests; and netperf, a tool to test network performance.

2.7 Stress tests

Stress tests push the system to the limits of its capabilities, trying to identify anomalous behaviours. A test of this kind can be conceived as an highly parallel task, such as a completely parallelized matrix multiplication. A performance test running under heavy memory pressure (such as running with a small physical memory), or in a highly resource-competitive environment (competing with many other tasks to access the CPU, for example) can become a stress test.

3 Automated testsuites

Testing is expensive, both in terms of costs and time. Automation is a good way to reduce economic and human efforts on testing. A number of automated testing environments for the Linux kernel has been proposed, each with its own strengths and weaknesses.

In the following paragraphs the most important test suites for the Linux kernel are presented. Goals and basic concepts which guide their design are stated.

3.1 IBM autobench

IBM autobench is an open source test harness[1] conceived to supports build and system boot tests, along with support for profiling [7]. It is written in a combination of perl and shell scripts, and it is fairly comprehensive.

Autobench can set up a test execution environment, perform various tests on the system, and write logs of statistical data. Tests can be executed in parallel, but test control support is basic and the user have almost no control over the way tests are executed. Error handling includes the success or failure of the tests, but is a very complex activity and must be done explicitly in all cases. In addition the use of different languages limits testsuite's extensibility and maintainability.

IBM autobench project is inactive since 2004, when the last version was released.

3.2 Autotest

Autotest is an open source test harness capable of running as a standalone client. It is easy to plug it into an existing server harness [11], too. Autotest provides a large number of tests, including functional, stress, performance, regression and kernel build tests. It supports various profilers, too.

Autotest is written in python, which enables it to provide an object oriented and clean design. In this way testsuite is easy to extend and maintain. Including python syntax in job control file, for example, users can take more control on test execution. On the other hand, python is not widely used in the real-time community, and is not suited for real-time tests or applications development.

[1] A test harness is an automated test framework designed to perform a series of tests on a program unit in different operating conditions and load, monitor the behaviour of the system and compare the test results with a given range of good values.

Autotest has built-in error handling support. Tests produce machine parsable logs; their exit status are consistent and a descriptive message of them is provided. A parser is built into the server harness, with the task of summarizing test execution results from different testers, and formatting them in an easy consultation form.

Autotest includes very few tests to examine the Linux kernel from a real-time point of view; almost all of them are functional tests. Moreover, it does not include any compliance test on real-time standards.

3.3 Crackerjack

Crackerjack is a testsuite whose main goal is regression testing [12]. It provides:

- automatic assessment of kernel behaviours
- test results storage and analysis
- incompatibilities notification
- test result and expected test result management (register, modify, remove)

Crackerjack is initially developed to test Linux kernel system calls, but over time has been revised to easy future extension to other operating systems.

It is implemented using Ruby on Rails. This makes it easy to modify it, ensuring a low maintenance cost and simplifying development of new tests. However, as for python, Ruby is not suited for real-time tests or applications development.

Crackerjack integrates a branch tracer for the Linux kernel, called btrax. Btrax is a tool to analyse programs effectiveness. Crackerjack uses btrax to trace the branch executions of the target program, to analyse the trace log file, and to display data about coverage and execution path. btrax makes use of Intel processors' branch trace capabilities, recording how much code was tested.

Crackerjack does not support conformance, performance or stress tests, and does not include any functional test on real-time features.

3.4 rt-tests

rt-tests [13] is a popular testsuite developed to test the PREEMPT_RT patch to the Linux kernel. It is developed by Thomas Gleixner and Clark Williams, and it is used in the OSADL lab, across various hardware architectures, for a continuous testing. rt-tests includes ten different tests for real-time features.

cyclictest [14], for example, is a well known test that measures the latency of cyclic timer interrupts. Through command line options, the user can choose to pin the measurement thread to a particular core of a multi-core system, or to run one thread per core. Cyclictest works by creating one or more user space periodic thread, the period being specified by the user. The accuracy of the measurement is ensured by using different timing mechanism.

Another interesting test is *hackbench*. It is both a benchmark and a stress test for the Linux kernel scheduler. Hackbench creates a specified number of pairs of schedulable entities which communicate via socket. It measures how long it takes for each pair to send data back and forth.

rt-tests is a good and well established suite to test Linux kernel real-time features. However it is conceived primarily to test the PREEMPT_RT patch set, so it's quite difficult to extend it to other Linux based real-time systems. For example, it contains a couple of tests based on a driver for the Linux kernel; in systems such as Xenomai, the use of a driver as this causes a mode change in which a real-time task switch from the Xenomai to the Linux environment. Thus, the task experiences much longer latencies.

Test results are outputted in a statistical summary, rather than in a boolean "PASS" or "FAIL". Unfortunately rt-tests do not provide any mechanism for collecting the results and present them in a machine parsable form.

3.5 LTP - Linux Test Project

LTP (Linux Test Project) is a functional and regression testsuite [15]. It contains more than 3000 test cases to test much of the functionalities of the kernel, and the number of tests is increasingly growing. LTP is written almost entirely in C, except for some shell scripts.

In recent years LTP has been increasingly used by kernel developers and testers and today is almost a de-facto standard to test the Linux kernel [16] [17]. Linux distributors use LTP, too, and contributes enhancements, bug fixes and new tests back to the suite.

LTP excellence is testing Linux kernel basic functionality, generating sufficient stress from the test cases. LTP is able to test and stress filesystems, device drivers, memory management, scheduler, disk I/O, networking, system calls and IPC and provides a good number of scripts to generate heavy load on the system.

It also provides some additional testsuites such as pounder, kdump, open-hpi, open-posix, code coverage [18], and others.

LTP lacks support for profiling, build and boot tests. Even if it contains a complete set of tests, LTP is not a general heavy weight testing client.

LTP also lacks support for machine parsable logs. Test results can be formatted as HTML pages, but they are either "PASS" or "FAIL", and for tester is more complex to understand the reasons behind failures.

LTP has a particularly interesting real-time testsuite, that provides functional, performance and stress tests on the Linux kernel real-time features. To the best of our knowledge, LTP is one of the few testsuites that provides such a comprehensive and full featured set of tests for Linux real-time functionalities.

4 Lachesis

All analysed testsuites seem to suffer some key failings in relation to testing the Linux kernel real-time features.

Many of them seem to have little consideration for real-time features in the Linux kernel. A great part of them (with the notable exception of LTP and rt-tests) does not offer any real-time functional, performance or stress test.

Usually it is simple to design and develop a new test inside an existing testsuite. However it is very difficult to extend an entire testsuite in order to test some new real-time nanokernels. System calls analysed in tests, in fact, may differ syntactically from one kernel to another maintaining the same functionality. Moreover some nanokernels, such as RTAI and Xenomai, provides some real-time features through additional system calls, for which specific tests should be developed.

Another problem is the lack of machine parsable results. There is no standard way to consistently communicate results to the user; often we have not any detail on the reason that led a test to failure.

Lastly, every testsuites has grown rapidly and chaotically in response to the evolution of the Linux kernel. For this reason they are not easy to understand, maintain and extend.

The lack of a comprehensive testsuite to meet previously exposed needs led us to develop Lachesis.

The ambitious goal of Lachesis is to provide a common test environment for different Linux based real-time systems. Therefore it seems reasonable to start from an existing, accepted and widely used testsuite, to adopt its principles and apply them to a new testsuite, conceived with other goals and priorities.

We choose LTP as a starting point for Lachesis. There are many reasons behind this choice. First, LTP is one of the few testsuites able to provide a set of tests for Linux kernel real-time features, and a large number of testers use it. Second, LTP has a well established and clean architecture. It makes use of two main libraries, librttest which provides an API to create, signal, join and destroy real-time tasks, and libstats which provides an API for some basic statistical analysis and some functions to save data for subsequent analysis. Last, LTP provides a logging infrastructure. This is an important and desirable feature for Lachesis, too.

We believe it is of little significance to compare the results of tests to absolute values statically built inside the testsuite. In fact, varying the hardware to test, these values should vary as well. So, unlike LTP, Lachesis provides a boolean pass/no pass output only on functional tests; by contrast, in performance tests it outputs a statistical summary of the results.

4.1 Architecture

Lachesis is designed to analyse a variety of Linux based real-time systems; therefore it provides a straightforward method to build tests for different kernels. During the configuration Lachesis probes the system to determine which nanorkernels or real-time patches are present, and instructs the compiler to produce in output different executables for each system to be tested. A set of scripts is provided to execute tests sequentially; launching these scripts, tests are executed one after another and tests results are stored in logs for subsequent analysis.

librttest had to be rewritten to support both RTAI and Xenomai primitives. Lachesis maintains librttest API, extending it to provide advanced real-time features, typical of Linux-based nanokernels. Basic real-time features are provided encapsulating real-time specific function into the pre-existing API, thus concealing them from the user. For example, the `create_task()` primitive was modified to take into account the corresponding primitives for Xenomai and RTAI.

As a result, it is possible to write a single test for a specific real-time feature and use it to test all supported systems, thus increasing testsuite's portability and maintainability.

The logging infrastructure of Lachesis is based on LTP features. Libstat is used to provide a set of functions to store data and make statistical calculus on them; other functions are used to write files with reports on data. These files are written in an easy readable form (though they are not yet machine parsable) and stored in an unique subdirectory.

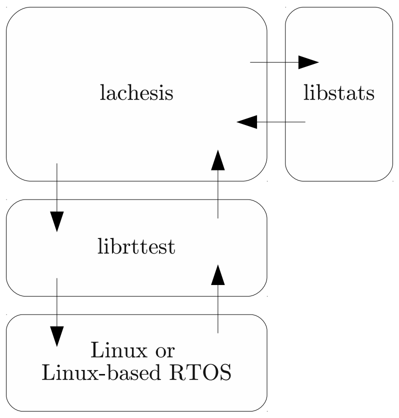

FIGURE 1: *Lachesis's architecture*

4.2 API

One of the principal goals of Lachesis is to provide a single API to develop tests for a number of different Linux based real-time systems. To do that, it provides an API that can be divided in four sections: tasks management, time management, buffers management and data management.

Tasks management API has been largely changed from the original LTP API, to provide support for testing on RTAI and Xenomai nanokernels. In particular, the `create_thread` primitive has been modified to call the corresponding Xenomai primitive, if Lachesis is requested to compile test for Xenomai nanokernel. RTAI's approach is quite different and consists in ensuring that certain section of code can be scheduled in accordance to some hard real-time algorithm, calling some specific functions immediately before and after them. We have addressed this particular mechanism defining four macros, to be used inside the tasks to be defined:

- `_RTAI_INIT` defines task descriptor, scheduler and system timer
- `_RTAI_START_TASK` starts hard real-time section
- `_RTAI_STOP_TASK` ends hard real-time section
- `_RTAI_END` deletes the task

In all the other primitives, where approaches do not differ so much from system to system, small changes have proved to be enough. Two primitives to create periodic and deadline tasks were added.

Time management API was extended with many primitives. A `RTIME` structure was defined to standardize time management between the different systems to be tested. Two functions have been defined to make additions and subtractions on it. Nanosleep and busy work primitives have been modified to call specific real-time sleep functions. A primitive was added to end the current period for the calling task.

Buffers management API remained almost unchanged from original API. Few changes were made to support RTAI's memory management primitives, changing `malloc()` and `free()` to `rt_malloc()` and `rt_free()` where appropriate.

Data management API remained also unchanged from original LTP API, since these functions only deal with data collected from tests previously carried out. We plan to work on this API in the near future, to support XML parsing and to format results adequately.

In addition to the API, we needed to introduce some macros to replace some standard functions with their real-time counterpart, provided by a particular nanokernel. For example, we have a macro to replace `printf()` with `rt_printf()` instances, if Lachesis have to build tests for Xenomai nanokernel.

4.3 Tests included in Lachesis

Below we briefly present all the tests currently included in Lachesis. Some of them were ported from different testsuites, others were developed as a part of this work. In the naming scheme we have tried to adopt the following principle: the first word is the parameter measured by the test, the second indicates the way in which measurement is made. We will indicate when the test is considered to be passed and the category it belongs to.

Tests are supposed to be done in an unloaded system. When a load is necessary, the test itself generates it. It's worth to say that Lachesis don't take into account power management features. So it's up to the user to ensure that tests are running under the same power management conditions. This could be done disabling the power management in the configuration of the Linux kernel.

1) blocktime_mutex is a performance test that measures the time a task waits to lock a mutex. Test creates a task with higher priority, one with lower priority, and some tasks at medium priority; highest priority task tries to lock a mutex shared with all the other tasks. Test is repeated 100 times.

2) func_deadline1 is a functional test we developed, conceived to be used only on SCHED_DEADLINE patched kernels. It creates a task set with $U = 1$, using the UUniFast [20] algorithm not to bias test's results. Task set is scheduled, and test is passed if no deadline is missed.

3) func_deadline2 is a functional test we developed, conceived to be used only on SCHED_DEADLINE patched kernels. It creates a task set with $U > 1$, using the UUniFast algorithm not to bias test's results. Task set is scheduled, and test is passed if at least one deadline is missed.

4) func_gettime verifies `clock_gettime()` behaviour. It creates a certain number of tasks, some of them setted to sleep, some other ready to be scheduled. Test is passed if the total execution time of sleeping tasks is close to zero. This is a functional test.

5) func_mutex creates a number of tasks to walk through an array of mutexes. Each task holds a maximum number of locks at a time. When the last task is finished, it tries to destroy all mutexes. Test is passed if all mutexes can be destroyed, none of them being held by a terminated task. This is a functional test.

6) func_periodic1 creates three groups of periodic tasks, each group with different priorities. Each task makes some computation then sleeps till its next period, for 6000 times. Test is passed if no period is missed. This is a functional test.

7) func_periodic2 is a functional test we developed, and is conceived to be used in kernels that support primitives for periodic scheduling. It creates a task set with $U = 1$, using the UUniFast algorithm not to bias test's results. Task set is scheduled, and test is passed if no period is missed.

8) func_periodic3 is a functional test we developed, and is conceived to be used in kernels that support primitives for periodic scheduling. It creates a task set with $U > 1$, using the UUniFast algorithm not to bias test's results. Task set is scheduled, and test is passed if at least one period is missed.

9) func_pi checks whether priority inheritance support is present in the running kernel. This is a functional test.

10) func_preempt verifies that the system is able to ensure a correct preemption between tasks. It creates 26 tasks at different priority, each of them trying to acquire a mutex. Test is passed if all task are appropriately preempted in 1 loop. This is a functional test.

11) func_prio verifies priority ordered wakeup from waiting. It creates a number of tasks with increasing priorities, and a master task; each of them waits on the same mutex. When the master task releases its mutex, any other task can run. Test is passed if tasks wakeup happened in the correct priority order. This is a functional test.

12) func_sched verifies scheduler behaviour using a football analogy. Two kinds of tasks are created: defence tasks and offence tasks. Offence tasks are at lowest priority and tries to increment the value of a shared variable (the ball). Defence tasks have an higher priority and they should block offence tasks, in such a way that they never execute. In this way ball position should never change. The highest priority task (the referee) end the game after 50 seconds. Test is passed if at the end of the test the shared variable is zero. This is a functional test.

13) jitter_sched measures the maximum execution jitter obtained scheduling two different tasks. The execution jitter of a task is the largest difference between the execution times of any of its jobs [19]. The first task measures the time it takes to do a fixed amount of work; it is periodically interrupted by an higher priority task, that simply wakes up and goes back to sleep. Test is repeated 1000 times. This is a performance test.

14) latency_gtod is a performance test. It measures the time elapsed between two consecutive calls of the `gettimeofday()` primitive. Test is repeated a million of times, at bulks of ten thousand per time.

15) latency_hrtimer one timer task and many busy tasks have to be scheduled. Busy tasks run at lower priority than timer task; they perform a busy wait, then yield the cpu. Timer task measures the time it takes to return from a nanosleep call. Test is repeated 10000 times, and is passed if the highest priority task latency is not increased by low priority tasks. This is a performance test.

16) latency_kill two tasks with different priority are to be scheduled. Lower priority task sends a kill signal to the higher priority task, that terminates. The test measures the latency between higher priority task start and termination. Test is repeated 10000 times, and we expect a latency under the tens of microseconds. This is a performance test.

17) latency_rdtsc is a performance test. It measures the average latency between two read of the TSC register, using the `rdtscll()` primitive. Test is repeated a million of times.

18) latency_sched is conceived to measure the latency involved in periodic scheduling in systems that do not support primitives for periodic scheduling. A task is executed, then goes to sleep for a certain amount of time; at the beginning of the new period the task is rescheduled. We measure the difference between expected start time and effective start time for the task. We expect this difference is under the tens of μs and, additionally, we expect the task does not miss any period. This is a performance test.

19) latency_signal schedules two tasks with the same priority. One task sends a signal, the other receives it. The test measures the time elapsed between sending and receiving the signal. Test is repeated a million of times. We expect a latency under the tens of μs. This is a performance test.

20) stress_pi stresses the Priority Inheritance protocol. It creates 3 real-time tasks and 2 non real-time tasks, locking and unlocking a mutex 5000 times per period. We expect real-time tasks to make more progress on the CPU than non real-time tasks. This is a stress test.

5 Conclusions and future work

In this paper we have presented Lachesis, a unified and automated testsuite for Linux based real-time systems. Lachesis tries to meet the need for a software tool to test Linux and Linux-based systems real-time features, having the following qualities:

- supports tests on Linux, RTAI, Xenomai, PREEMPT_RT and SCHED_DEADLINE real-time features, through a standard test API
- provides a series of functional, performance and stress tests to ensure the functionality of the examined kernels
- provides a series of tests for periodic and deadline tasks
- is easy to use: each feature to be tested is associated to a script, which runs tests and logs the results for every testable system.
- it includes a set of bash scripts that helps to execute tests in the correct order and in the correct conditions.

Several real-time tests were ported to Lachesis from other testsuites, in a simple and straightforward way. In many cases there were no needs to change the code except to add some macro calls at the beginning and at the end of the test's code.

Our extension to librttest API has made possible to develop some new tests for threads with fixed periods or deadlines. These tests are useful to value jitter and latency in periodic task scheduling. Similar tests can be developed in very short times.

Unfortunately, Lachesis is far from complete. First, its test coverage is very low. Second, tests included in Lachesis are somewhat general to be really useful in development. So Lachesis needs to expand its test coverage with more specific tests, and librttest needs to take into account more low level primitive, to make possible to develop more significant tests.

For this reasons, we believe it's very important to integrate the testsuite rt-tests in Lachesis. As underlined previously, rt-tests is very specific in respect to Lachesis, and so it's quite difficult to figure out how to extend these tests to other real-time nanokernels. We expect that a strong extension to librttest API is necessary to reach this goal.

Up to now Lachesis is tested and used only on x86 architecture. Given that we use only high-level kernel primitives, we are quite confident that the testsuite is easily portable on other architectures, with little or no effort. Recently we developed a porting of Xenomai 2.5.5.2 and RTAI 3.8 to a Marvell ARM9 board[2], and we plan to use Lachesis to test the functionalities and the performances of these portings.

However, just the variety of Linux based real-time systems that Lachesis is able to test proves that it is portable and easy to use. We plan to exploit this qualities porting Lachesis to other systems, such IRMOS [2], SCHED_SPORADIC [21], XtratuM [22] and PartiKle [23], and to other architectures.

Beyond this, we plan to develop some kernel-space tests for real-time nanokernels and to build a system to parse and XML format results. Test results quality can be improved, also, detailing possible reasons behind a test failure.

Lachesis is actually under active development, and can be downloaded from bitbucket.org[3].

[2]ARM Marvell 88F6281, equipped with a Marvell Feroceon processor, clocked at 1.2 GHz, ARMv5TE instruction set.
[3]https://bitbucket.org/whispererindarkness/lachesis

Acknowledgments

We would like to thank Andrea Baldini and Francesco Lucconi for their contributions in the development of the testsuite, and Massimo Lupi for his ideas and advices.

References

[1] S. Rostedt and D. Hart, *Internals of the RT Patch*, in Proceedings of the Linux Symposium, 2007, pp. 161-172.

[2] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari, *Hierarchical multiprocessor CPU reservations for the linux kernel.* OSPERT 2009, p. 15.

[3] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino, *An EDF scheduling class for the Linux kernel*, in Proceedings of the 11th Real-Time Linux Workshop, 2009, pp. 1-8.

[4] P. Mantegazza, E. Dozio, and S. Papacharalambous, *RTAI: Real time application interface*, Linux Journal, no. 72es, pp. 10-es, 2000.

[5] P. Gerum, *Xenomai - Implementing a RTOS emulation framework on GNU/Linux*, 2004. [Online]. Available: http://www.xenomai.org/documentation/

[6] K. Yaghmour, *Adaptive domain environment for operating systems*, 2001. [Online]. Available: http://www.opersys.com/adeos/

[7] M. Bligh and A. Whitcroft, *Fully Automated Testing of the Linux Kernel*, in Proceedings of the Linux Symposium, vol. 1, 2006, pp. 113-125.

[8] D. Evans, J. Guttag, J. Horning, and Y. Tan, *LCLint: A tool for using specications to check code*, ACM SIGSOFT Software Engineering Notes, vol. 19, no. 5, pp. 87-96, 1994.

[9] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar, *The software model checker Blast*, International Journal on Software Tools for Technology Transfer, vol. 9, no. 5-6, pp. 505-525, Sep. 2007.

[10] G. Carette, *CRASHME: Random Input Testing*, 1996. [Online]. Available: http://crashme.codeplex.com/

[11] J. Admanski and S. Howard, *Autotest-Testing the Untestable*, in Proceedings of the Linux Symposium, 2009.

[12] H. Yoshioka, *Regression Test Framework and Kernel Execution Coverage*, in Proceedings of the Linux Symposium, 2007, pp. 285-296.

[13] http://git.kernel.org/?p=linux/kernel/git/clrkwllms/rt-tests.git;a=summary

[14] https://rt.wiki.kernel.org/index.php/Cyclictest

[15] P. Larson, *Testing Linux with the Linux Test Project*, in Ottawa Linux Symposium, 2001, p. 265.

[16] S. Modak and B. Singh, *Building a Robust Linux kernel piggybacking The Linux Test Project*, in Proceedings of the Linux Symposium, 2008.

[17] S. Modak, B. Singh, and M. Yamato, *Putting LTP to test - Validating both the Linux kernel and Test-cases*, Proceedings of Linux Symposium, 2009.

[18] P. Larson, N. Hinds, R. Ravindran, and H. Franke, *Improving the Linux Test Project with kernel code coverage analysis*, in Proceedings of the Linux Symposium, 2003, pp. 1-12.

[19] G. C. Buttazzo, *Hard real-time computing systems - predictable scheduling algorithms and applications*, 2nd edition, Springer, 2005.

[20] E. Bini and G. C. Buttazzo, *Measuring the Performance of Schedulability Tests*, Real-Time Systems, vol. 30, no. 1-2, pp. 129-154, May 2005.

[21] D. Faggioli, A. Mancina, F. Checconi, and G. Lipari, *Design and implementation of a posix compliant sporadic server for the Linux kernel*, in Proceedings of the 10th Real-Time Linux workshop, 2008, pp. 65-80. [Online].

[22] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, *Xtratum: a hypervisor for safety critical embedded systems*, in Proceedings of the 11th Real-Time Linux Workshop. Dresden. Germany, 2009.

[23] S. Peiro, M. Masmano, I. Ripoll, and A. Crespo, *PaRTiKle OS, a replacement for the core of RTLinux-GPL*, in Proceedings of the 9th Real-Time Linux Workshop, Linz, Austria, 2007, p. 6.