Towards Unit Testing Real-Time Schedulers in LITMUS$^{\mathrm{RT}}$

Malcolm S. Mollison, Björn B. Brandenburg, and James H. Anderson
Department of Computer Science, The University of North Carolina at Chapel Hill

Abstract

The problem of unit testing multiprocessor real-time schedulers in operating systems such as LITMUS$^{\mathrm{RT}}$ is discussed. A tool intended to aid debugging by identifying deviations from an intended scheduling policy and performance regressions is proposed. This paper gives a specification for the tool and also discusses ongoing work on a prototype implementation.

1 Introduction

The advent of multicore computing has led to renewed interest in real-time scheduling algorithms for multiprocessor systems. In research on this topic, the greatest emphasis has been on purely algorithmic issues. To impact the development of real systems, such work must be complemented by prototype development, so that scheduler-related overheads can be measured and the practicality of proposed scheduling algorithms assessed. The **LI**nux **T**estbed for **MU**ltiprocessor **S**cheduling in **R**eal-**T**ime Systems (LITMUS$^{\mathrm{RT}}$) project [2, 4, 8] was launched to enable such prototype-oriented research. LITMUS$^{\mathrm{RT}}$ provides a testbed for multiprocessor real-time schedulers by extending the Linux kernel to support the implementation of such schedulers as plugins.

Unfortunately, developers of real-time schedulers, including LITMUS$^{\mathrm{RT}}$ scheduler plugins, face two pressing challenges. First, it is very difficult to ascertain if a scheduler is actually making correct decisions, and thus is implemented correctly. For example, mistaken scheduling decisions may not result in deadline overruns for a particular benchmark task set, and thus may go unnoticed. Second, it is easy for scheduler developers to introduce extra, unwanted scheduling overhead. Such overhead will increase the degree to which the scheduler deviates from a desired scheduling policy.

These challenges are exacerbated by serious difficulties associated with multiprocessor operating system development. For example, concurrent programming issues such as synchronization, race conditions, etc., must be properly taken into account. Furthermore, collecting relevant data to aid debugging can be an obstacle in itself due to programming limitations commonly encountered in kernel environments.

Currently, developers must exert much effort after each subsequent update to scheduler code to manually discover and correct faults, such as those described above. Despite this extra effort, it is still likely that latent defects will be introduced into the scheduler code over time due to limitations in scope and coverage of purely manual testing approaches. An automated tool that facilitates extensive testing after each code update would both eliminate much of the need for manual testing and increase developers' confidence that scheduler code has been implemented correctly.

A sensible strategy for such a tool to utilize is *unit testing* [6], which is a widely-followed software engineering practice [9] wherein individual units of a software system are programmatically tested for appropriate behavior, ideally after each code revision. Besides detecting undesired behavior in LITMUS$^{\mathrm{RT}}$ scheduler plugins, a unit-testing tool should produce detailed output that can help developers identify and fix detected problems, including incorrect scheduling decisions, deadline overruns, and regressions in scheduling overhead.

In this paper, we present a specification for such a tool, and discuss both the current state of a prototype implementation and desired extensions.

Prior work. The Linux Test Project [7] has implemented a testing mechanism for the Linux scheduler, including the two POSIX-mandated real-time scheduling policies included in stock Linux, `SCHED_FIFO` and `SCHED_RR`. Like the proposed tool, the Linux Test Project performs scheduling overhead regression testing and conducts a series of checks that can alert developers to various problems. However, the Linux Test Project lacks a lightweight event tracing mechanism that would allow it to analyze individual scheduler decisions and provide very precise scheduling overhead analysis. (Our tool makes use of Feather-Trace [1, 5], a toolkit used by LITMUS$^{\mathrm{RT}}$ to record events.) Furthermore, from a real-time scheduling perspective, the Linux Test Project's scope is limited to only static-priority scheduling (`SCHED_FIFO` and `SCHED_RR`).

LITMUS$^{\mathrm{RT}}$ currently supports a much wider range of real-time scheduling algorithms, including several global algorithms that have been the subject of much recent research. The development of LITMUS$^{\mathrm{RT}}$ has thus far relied on purely manual techniques for debugging and testing of adherence to desired scheduling policies. Due to the scarcity of developer time, such testing is only conducted infrequently and involves running only a few simple, hand-crafted task sets. With the increasing complexity and maturity of LITMUS$^{\mathrm{RT}}$, a more thorough testing effort is clearly warranted. Therefore, we desire a high-quality unit-testing tool for use with LITMUS$^{\mathrm{RT}}$. To the best of our knowledge, the topic of testing of multiprocessor real-time scheduler *implementations* has not been considered in prior work.

The rest of this paper is organized as follows. In Section 2, we provide background on real-time scheduling on multiprocessor platforms, including LITMUS$^{\mathrm{RT}}$ and associated tools. In Section 3, we provide a specification of the proposed tool. In Section 4, we describe progress to date on a prototype of the tool and planned future extensions. Finally, in Section 5, we conclude.

2 Background

The LITMUS$^{\mathrm{RT}}$ operating system [2, 4, 8] executes task sets under a real-time scheduler selected by the user. Feather-Trace [1, 5] records scheduling decisions made by LITMUS$^{\mathrm{RT}}$ in machine-readable, binary trace files. Our proposed tool, in turn, reads these files and performs analysis upon them under a unit-testing paradigm. This section provides necessary background on these topics.

2.1 Real-Time Task Model

A *real-time task set* consists of a number of tasks that are subject to real-time constraints. Tasks are invoked, or *released*, repeatedly during their lifetimes; each such invocation is known as a *job*. Under the *sporadic* task model, each release of a task must be separated by a minimum interval of time known as the task's *period*. Each task also has an associated *worst-case execution time* (WCET), which is an estimate of the execution time of any job of the task in the worst case. The ratio of each task's WCET and period defines its *utilization*. In this paper, we limit attention to sporadic tasks with *implicit deadlines*, *i.e*, each job is expected to complete before the end of its period; otherwise, it has exceeded its deadline and is considered *tardy*. A job is *eligible* if it has been released, is unfinished, and the previous job of the same task has completed execution.

2.2 Real-Time Scheduling Policies

A *scheduling policy* is an algorithm used by a *scheduler* at runtime to determine how to allocate available cores[1] to jobs.

Under the *global earliest-deadline-first* (G-EDF) policy, contending jobs are prioritized in order of non-decreasing deadlines. Hence, on an m-processor system, at any time, if there are more than m eligible jobs, then the m eligible jobs with the earliest deadlines should be executing, with ties broken consistently according to some (perhaps implementation-specific) policy. In order to meet this condition, the scheduler can preempt jobs or migrate them to different cores.

Our prototype unit-testing tool only supports the analysis of G-EDF, though support for other policies is planned. These include *partitioned earliest-deadline-first* (P-EDF) scheduling, under which each task is assigned to a particular core and only contends for execution time with other tasks assigned to the same core; *clustered earliest-deadline-first* (C-EDF) scheduling, under which tasks are assigned to user-defined clusters of cores; and the *PD*^{2} scheduling algorithm [10], under which tasks make progress at a rate proportional to their utilizations.

2.3 LITMUS$^{\mathrm{RT}}$

The LITMUS$^{\mathrm{RT}}$ operating system provides a platform for executing real-time task sets according to a real-time scheduling policy. LITMUS$^{\mathrm{RT}}$ is implemented as a patch to the Linux kernel,[2] modifying its scheduling facilities to allow for the use of more specialized real-time scheduling policies than those that are included in stock Linux. LITMUS$^{\mathrm{RT}}$ scheduler plugins have been written for the G-EDF, P-EDF, C-EDF, and PD^2 scheduling policies (among others).

2.4 Feather-Trace

Feather-Trace is a light-weight event tracing toolkit. It allows scheduler code in the kernel to buffer binary data and make it available for asynchronous export to userspace, from whence it can be written to disk. Feather-Trace is designed to incur very little overhead, so that time-critical code can record data without being disrupted.

In LITMUS$^{\mathrm{RT}}$, this mechanism can be used to record scheduler events (for example, the release or completion of a job) in trace files. Each event record is stored in a

[1]In this paper, we denote any logical CPU available for scheduling by the OS as a "core."

[2]The latest release patches kernel version 2.6.24.

special binary format and includes a precise timestamp. These records can later be retrieved and examined by analysis tools, such as our unit-testing tool.

3 Specification

The goal of our proposed tool is to help real-time scheduler developers produce correct scheduler code for the LITMUS$^{\mathrm{RT}}$ system. Towards this end, the tool will conduct a series of tests to determine if desired criteria are met by a scheduler. An error will be reported when a criterion is not met, and details will be provided to help developers resolve the problem.

In typical unit testing, modules of source code are executed in a test framework. Each module is executed with pre-specified input and checked for expected output. If unexpected output is produced, then the test framework reports an error. This approach works well for source code that can be factored into small, modular components that accept well-defined inputs and give predictable outputs.

Scheduler code is not amenable to such piecemeal testing. This is due to three fundamental reasons. First, the absence of errors when testing individual components for *logical* correctness does not imply that the system as a whole exhibits *temporally* correct behavior (for example, scheduling errors due to race conditions cannot be found by piecewise testing). Second, scheduling code tends to rely heavily on side effects and external entities such as hardware timers; such dependencies are hard to emulate correctly in a traditional unit-testing environment. Third, scheduling code is invoked from many different code paths within the kernel since the scheduler constitutes the very core of the OS; it is infeasible to recreate appropriate system and task state for all possible invocations. In essence, traditional unit-testing techniques rely heavily on the tested system being well-structured and modular, whereas scheduling code tends to (implicitly) interact with large parts of the whole system. Furthermore, testing on a per-module basis would not allow for accurate sampling of scheduling overhead.

To work around these concerns, the proposed tool executes unit tests offline against records of scheduler events as they occurred on *real hardware* in the *actual system*. To automate the testing process, the generation and execution of task systems can also be automated.

This approach permits testing on any task system for a supported scheduling policy, alleviating the need to provide rigid, module-specific test input. It also allows for accurate measurement of scheduling overhead.

As with typical unit testing, errors will be reported when desired criteria are not met. Information to help developers correct bugs will be included with each error. If all tests complete without errors, developers can conclude that the scheduler behaved correctly for the criteria being tested.

3.1 Architecture

As depicted in Figure 1, the proposed tool consists of four components: the Driver, the Parser, the Validator, and the Report Generator.

Driver. The Driver automates the testing process, facilitating unit testing after each incremental update to the scheduler code. Each testing session begins with the Driver reading a developer-supplied configuration file that specifies task systems to execute and a suite of unit tests. Any unit-test tuning variables (for example, acceptable tardiness bounds) are also specified in the configuration file. The Driver executes the specified task system under LITMUS$^{\mathrm{RT}}$ with Feather-Trace recording enabled. When the task system completes execution, the Driver invokes the Parser, Validator, and Report Generator in sequence. For more thorough testing, the Driver can repeatedly generate and test random task systems (generated according to distributions specified in the configuration).

Parser. Feather-Trace exports scheduler event records to a buffer in a compact format to avoid incurring unnecessary overhead. The contents of this buffer are later written to a set of trace files. The Parser reads the trace files, extracting event records and storing them in an easy-to-use format in memory. The in-memory records are represented as a "stream" data structure, which the Validator and Report Generator iterate over in the course of performing their work.

Feather-Trace records the following types of events.

- Release: A job is released.
- Switch To: A job begins executing.
- Switch Away: A job stops executing.
- Completion: A job completes.
- Block: A job blocks. Even in task systems without synchronization requirements, blocking can be caused by the Linux I/O subsystem.
- Resume: A job resumes, after being blocked.

Each scheduling event record identifies the job and task it is associated with and includes a timestamp indicating the time of the event. Records for events associated with a particular core include the logical CPU number for that core.

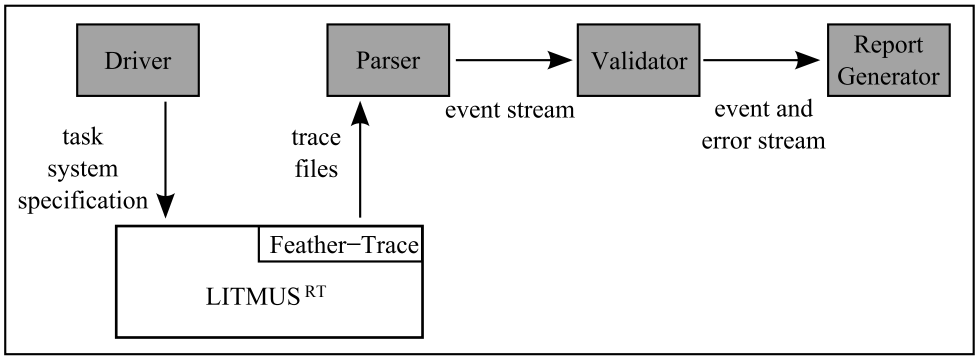

Figure 1: Data flow between components of the proposed tool (shaded) and LITMUS$^{\mathrm{RT}}$.

Additional Feather-Trace records specifying the WCET and period for each task are recorded at the beginning of task system execution.

Validator. The Validator executes each unit test, using as input the scheduler event stream. Each unit test produces an error record for each error it finds. Each error record includes the time of the scheduler event that triggered the error. This allows the Report Generator to display errors in the context of surrounding scheduling events. The Validator outputs a stream of all error records produced by the unit tests.

Report Generator. The Report Generator receives as input the event and error streams. It generates meaningful output to help developers correct bugs and detect problematic scheduling latency. A plugin system allows for output in various formats. For example, plugins to support HTML output or graphical output could be developed.

3.2 Unit Tests

The current needs of LITMUS$^{\mathrm{RT}}$ have motivated the creation of algorithms for a number of unit tests.

Completion Test. The Completion Test checks whether all released jobs actually complete. That is, for each Release record in the scheduler event stream, there should be a Completion record corresponding to the same job. An error is produced for any released jobs that do not complete.

Sporadic Task Model Test. The Sporadic Task Model Test checks whether all jobs of the same task are separated by at least the period of the task.

The algorithm for this unit test is straightforward: for each Release record in the event stream, if there is a previous Release record of the same task, then a check is made to ensure that the separation time between them is at least the corresponding task's period.

Some premature releases are expected in the case that periodic releases are desired, since scheduling overhead will delay releases by a variable, nonzero amount of time. To accommodate this, a "tolerance" tuning variable can be specified in the Driver configuration file. Releases that are premature by an amount of time less than the tolerance value will not cause an error to be generated.

Deadline Test. The Deadline Test checks to see if jobs meet their deadlines. Once again, the algorithm for the test is straightforward: for each Release record, there must be a Completion record for the same job, and the difference in their timestamps must be at most the period for the corresponding task.

A tolerance variable, similar to the one used for the Sporadic Task Model Test, allows developers to specify an acceptable tardiness bound.

G-EDF Decision Test. No scheduler can achieve true G-EDF scheduling in practice, due to scheduling overhead. For example, when a new job becomes eligible that has an earlier deadline than one of the jobs currently in execution, the new job should begin executing immediately. However, the overhead of halting the execution of the job to be preempted and starting execution of the new job takes a nonzero amount of time. Thus, testing for true adherence to the G-EDF policy would be an exercise in futility.

However, it still holds that whenever a scheduler does decide to switch execution to a particular job, that job should be one of the m earliest-deadline eligible jobs. The G-EDF Decision Test checks against this constraint. Successful validation of an event stream over both the Completion Test and G-EDF Decision Test assures developers that the scheduler allocated all eligible jobs in earliest-deadline-first order.

The G-EDF Decision Test algorithm models the state of the task system over the course of its execution. The model consists of a list of executing jobs (up to m) and a list of released jobs that are eligible for execution. The algorithm progresses by iterating over the event stream, updating the model for each record. For example, a Switch To record indicates that a job began execution on a particular core; upon encountering such a record, the algorithm would add that job to the list of executing jobs.

The algorithm generates an error each time it determines that a job that did not have sufficiently high priority was scheduled for execution. A job has sufficiently high priority for execution if it is one of the m earliest-deadline eligible jobs.

G-EDF Latency Test. The G-EDF Latency Test helps developers locate areas of high overhead in scheduler code and quantitatively evaluate their efforts to optimize code to reduce overhead. It also may help developers discover bugs that do not violate the decision-making constraint checked by the G-EDF Decision Test, as these bugs often contribute to overhead.

Scheduler overhead causes departure from the G-EDF policy when a job becomes one of the m earliest-deadline eligible jobs but has not yet begun execution. Therefore, to measure G-EDF scheduling latency, we examine each such occurrence, and measure the specific latency components contributing to overhead. Note that this condition occurs prior to each switch in execution to a job, and thus once for each job release.

Which latency components are present depends on the context of the system when switching to a new job becomes necessary. There are three such contexts, explained in the following paragraphs.

In Context 1, as depicted in Figure 2, a job is released into a system with an idle core and is eligible (i.e., the previous job of the same task has completed). The only latency component—and, thus, the total scheduling overhead that occurs in this context—is the difference between the time of the release of the job and the time when it is switched to for execution on a core.

In Context 2, as depicted in Figure 3, a job is released into a system with no idle cores, but it is not one of the m highest-priority eligible jobs. When it becomes one of the m highest-priority eligible jobs due to the completion of another job, that job should be switched away from execution and the new job should commence execution. Scheduling overhead is the sum of two latency components. The first is the difference between the time of completion of the previously-executing job and the time when it is switched away from execution. The second is the difference between the time when the previously-executing job is switched away from execution, and the time when the new job is switched to for execution.

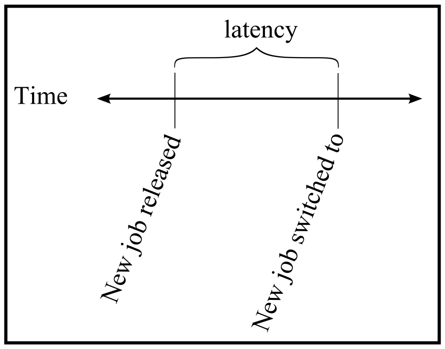

Figure 2: Context 1: The new job is scheduled immediately on an idle core.

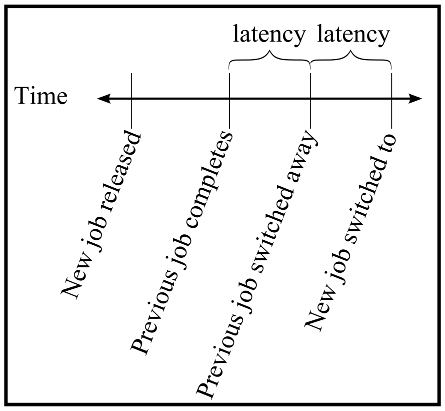

Figure 3: Context 2: There is no idle core available, and the new job has insufficient priority to preempt one of the currently-scheduled jobs, i.e., it must await the completion of a previous job.

In Context 3, as depicted in Figure 4, a job is released into a system with no idle cores, but it is one of the m highest-priority eligible jobs. Another job should be switched away from a core and the new job should commence execution. Scheduling overhead is the sum of two latency components. The first is the difference between the time when the new job is released and the time when the preempted, previously-executing job is switched away from execution. The second is the difference between the time when the previously-executing job is switched away from execution, and the time when new job is switched to execution.

The goal of the G-EDF Latency Test is to measure each instance of overhead, classified as one of the latency components discussed previously. When latency greater than a desired threshold is discovered, it can be reported to the developer. Furthermore, statistical analysis can be performed. For example, a developer could

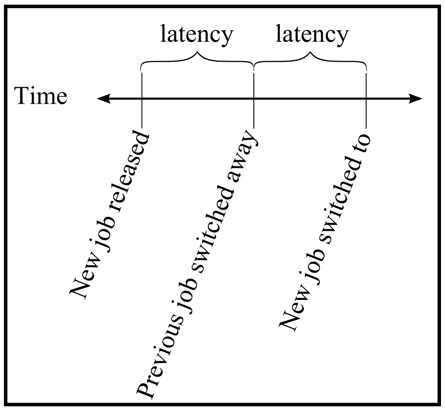

Figure 4: Context 3: There is no idle core available, but the new job has sufficient priority to preempt a running job.

use the latency measures to calculate the average amount of latency for a particular latency component over the course of execution of a task system. Historical data of this kind, accumulated from past testing, can be used to guard against regressions in scheduler performance.

Note that overhead potentially begins being accumulated when a job is released, and ends when the job is switched to. Thus, the G-EDF Latency Test algorithm iterates over the event stream, extracting latency information each time it encounters a release record.

The algorithm proceeds as follows. For each Release record, the corresponding Switch To record is found. This record is used to determine the CPU on which the events leading up to the beginning of execution of the job will occur. The events on that CPU from the time of the release to the time of the switch to execution match one of the three contexts. Once the context is identified, extracting the specific latency measures is straightforward.

Fortunately, each context represents a unique set of scheduler events, making it possible to identify the context for a particular release from the scheduler event stream.

- Context 1 is indicated by a Release record followed by a Switch To record.
- Context 2 is indicated by a Release record followed by a Completion record.
- Context 3 is indicated by a Release record followed by a Switch Away record.

The Driver configuration file allows developers to specify a large number of criteria for the G-EDF Latency Test. In the simplest case, acceptable thresholds for latency components can be provided. An error is generated each time these values are exceeded.

4 Prototype Description and Future Extensions

In this section, we provide a summary of ongoing work on a prototype of the tool, and discuss desired future extensions of the tool.

4.1 Prototype

Most of the functionality described in the specification has been successfully implemented in our prototype, including each of the unit tests discussed in this paper. The code has been made available online [3]. The prototype is implemented in the Python programming language, which was chosen for its ease of use and will be retained for the official release version of the tool.

The prototype does not yet support some of the more advanced proposed features, as discussed below, though support for them is planned for future releases.

The Driver does not automatically create and test randomized task systems; instead, a specific task system must be provided by the user. This feature would be useful for testing schedulers over a wide variety of task systems, increasing the probability of finding bugs that occur only in rare or specialized cases.

The Report Generator does not yet provide support for creating a graphical depiction of scheduling events, which would aid developers in understanding bugs.

The G-EDF Latency Test does not perform any automatic statistical validation or regression checking. For example, automatic calculation of average latency for each latency component over the course of execution of a task system is desired. These values could automatically be compared to ideal values provided by the user.

4.2 Future Extensions

Significant expansion of the tool beyond the specification described in this paper is desired. Most importantly, support for testing additional scheduling policies beyond G-EDF—in particular, C-EDF, P-EDF, and PD^2—is planned.

Supporting C-EDF and P-EDF will be straightforward. C-EDF unit-test algorithms will not be notably more complex than the G-EDF algorithms provided in this paper. As P-EDF is a special case of C-EDF, a separate unit tests for P-EDF will not be needed.

However, PD^2 will present a significant challenge, and will require the development of a large number of additional and complex unit tests. The need to debug PD^2 schedulers was the most pressing motivation for the development of the tool, but G-EDF was chosen as a more easily achievable initial target. Lessons learned in developing the overall framework for the unit tester and the G-EDF unit tests will likely prove invaluable in the extension of the tool to allow for testing of PD^2 schedulers.

In addition, the current specification for the tool does not account for jobs that block. A tool for testing real-time schedulers running under LITMUS$^{\mathrm{RT}}$ needs to account for blocking, since it can be caused by some synchronization protocols as well as the Linux I/O subsystem. Fortunately, extending the specification to account for blocking will be straightforward. Most of the unit tests remain the same, though a slight change to the model used in the G-EDF Decision Test is necessary. The most significant change will be the need to analyze new latency components in the G-EDF Latency Test.

5 Conclusion

We have established the need for an automated tool for testing multiprocessor real-time schedulers in LITMUS$^{\mathrm{RT}}$. The tool should uncover incorrect scheduling decisions. It should also allow for rigorous analysis of scheduling overhead. We have presented a specification of such a tool, and have discussed ongoing work on a prototype. Finally, we have described significant desired extensions of the tool beyond the features provided for in our current specification.

References

[1] B. Brandenburg and J. Anderson. Feather-Trace: A Light-Weight Event Tracing Toolkit. In *Proceedings of the Third International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 20-27, 2007.

[2] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A Status Report. In *Proceedings of the 9th Real-Time Linux Workshop*, pages 107-126, 2006.

[3] LITMUS$^{\mathrm{RT}}$ Scheduler Testing.
Homepage. http://cs.unc.edu/˜mollison/unit-trace/.

[4] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111-123, 2006.

[5] Feather-Trace.
Homepage. http://www.cs.unc.edu/˜bbb/feathertrace/.

[6] IEEE Standard for Software Unit Testing. 1986.

[7] The Linux Test Project.
Homepage. http://ltp.sourceforge.net/.

[8] The LITMUS$^{\mathrm{RT}}$ Project
Homepage. http://www.cs.unc.edu/˜anderson/litmus-rt/.

[9] P. Runeson. A Survey of Unit Testing Practices. In *IEEE Software*, 23(4):22-29, 2006.

[10] A. Srinivasan and J. Anderson. Optimal rate-based scheduling on multiprocessors. *Journal of Computer and System Sciences*, 72(6):1094-1117, September 2006.