Thready: A fast scheduling simulator for real-time task systems

Robert Schmidt
Institute of Electrodynamics and Microelectronics
University of Bremen
rschmidt@uni-bremen.de

Alberto García-Ortiz
Institute of Electrodynamics and Microelectronics
University of Bremen
agarcia@uni-bremen.de

***Abstract*—Correct scheduling in hard real-time systems is of utmost importance to guarantee that deadlines are not missed. By mathematical proof, correctness can be demonstrated for the worst case. Such demonstrations usually do not consider errors during system run-time, and do not provide quality of service insights. Such insights can be derived from simulations, but typical simulators are too slow for long term simulations.**

We developed Thready, a fast simulator for sporadic task systems under errors to investigate the long term behavior of scheduled systems. Thready's three order of magnitude speedup in latency compared with the fastest state of the art simulator framework allows designers to investigate system performance in the average case, which facilitates understanding and better design decisions.

***Index Terms*—Scheduling algorithms, Real-time systems, Simulation**

I. Introduction

Real-time systems, like on-board flight computers, have timing requirements which need to be satisfied. To meet these requirements, scheduled jobs need to finish before their deadline. If job arrival- and execution times are not known beforehand, the schedule in which jobs are granted access to the processor has to be created during runtime, according to a scheduling algorithm.

Such an algorithm is correct, if it can always create a schedule where each job receives enough processor time to finish before its deadline. Correctness is usually demonstrated for the worst case, defined by upper bounds on execution times [1], but for system performance the average case is of interest, which can be evaluated by simulations. For example, in mixed criticality scheduling, the time spend in low criticality mode is a key quality of service (QoS) metric, which depends on the probability of having a job executing longer than on average. Usually the assumed probabilities of such an event are very low, which requires to simulate the system for a long time. For such an investigation, simulator performance is of utmost importance to gather results in reasonable wall clock time, but available simulators fall short in this regard.

Although many simulators have been written, only few are readily available to researchers [2]. As with most scientific software, their value as an scientific artifact has not been recognized, resulting in lack of software preservation and reproducibility problems [3]. Moreover, most available simulators are hard to use in heterogeneous environments due to low portability, are hard to automate due to their user interfaces, and sacrifice simulation speed over features or ease of implementation. Although such feature-rich simulation environments are great to explore new algorithms, or to investigate the relative short critical instant of a task system, they don't allow to run long term simulations.

We fill this gap with `Thready`, which is the first portable, easy to automate, integrate, reproduce and understand open source simulator to address long term simulations for sporadic task systems:

- `Thready` is easy to compile and distribute (Section IV-D), allowing to leverage the computational resources available in a heterogeneous laboratory computer environment (Section V)
- `Thready`'s user interface is easy to understand and instrument, which fosters integration into scientific data analysis pipelines and reproducibility (Section IV-A)
- `Thready`'s three orders of magnitude speedup in latency allows to investigate QoS and average case performance metrics in long term simulations (Section V-B)
- `Thready` allows to investigate systems by simulation which are infeasible with with current simulators, as demonstrated with a case study (Section VI)

II. Related work

Simulators are recognized as useful tools to teach, explore, and understand scheduling algorithms [4]. We differentiate three main approaches to simulators: Comprehensive simulation environments configured by the user, frameworks, or libraries, to construct new special purpose simulators from basic components, or single purpose programs simulating a specific aspect of a system or algorithm.

One of the early comprehensive simulators are the `STRESS` computer-aided software engineering tools, which use a domain-specific language to specify the simulated system [5]. Besides simulation of different algorithms, system configurations, and shared resources, `STRESS` allows to analyze and visualize simulation results. In comparison to other simulators, `STRESS` provides many features, as shown in Table I, but is not publicly maintained as open source.

Another extensive simulator is `Cheddar`, which allows to simulate different hard- and software configurations, scheduling algorithms, supports shared resources, and provides extensive analysis and visualization capabilities [2]. Compared with `STRESS`, `Cheddar` is actively maintained and available as open

source software. Written in Ada, `Cheddar` is well suited to simulate the critical instance of a task system to show the correctness of the scheduling algorithm, but inappropriate for long term simulations owing to its slow simulations speed.

A further comprehensive simulators is `Realtss`, which supports multiple scheduling algorithms, shared resources, tracing, and result evaluation [6]. The modular design, written in TCL, would allow to extend the simulator, but `Realtss` is not publicly available as open source. In this regard, it shares the same fate as `schesim`, which featured hierarchical, uni- and multiprocessor scheduling, tracing and analysis [7], and is not available anymore. `schesim` allows the user to specify tasks via a JavaScript Object Notation (JSON) interface, but does not model errors. Similar to `Realtss` in features, but without support for shared resources and evaluation, is `RTSim`. `RTSim` targets education and classroom usage [4], but does not consider errors in its model, which is a key feature of `Thready`.

Unlike comprehensive simulators, frameworks like `Fortissimo` or `rtlib` provide building blocks to develop tailored simulators for a specific system [8, 9]. Such tailored simulators can achieve better performance than comprehensive simulators, because they allow to reduce or disable unnecessary features. Their main drawback is the lack of interfaces, which need to be written by the user of such frameworks, else the resulting simulator is a single purpose program, where the problem definition is part of the simulator, without a possibility to modify it. Although single purpose programs have legitimate use cases [10], their creation and maintenance for many similar problems is a burden.

Contrary, `Thready` strives to combine the performance benefits of the framework approach with easy to instrument interfaces and support for diverse error models.

III. Thready model

`Thready` simulates preemptive earliest deadline first (EDF) scheduling of a sporadic task system on a uniprocessor. The processor is the only shared resource, and there is no overhead of scheduler operation and context switch. All tasks in the simulated sporadic task set are independent of each other, and generate a sequence of jobs. Jobs signal their completion to the scheduler, which is not aware of the completion time a priori. If jobs are preempted, they return to the scheduler's queue, as shown in Fig. 1. Time is discrete and time step resolution is defined by the user.

A. Sporadic task system and job generation

The simulator generates jobs according to the parameters of each task τ_i. Jobs are characterized by their arrival time α_{ij}, actual execution time γ_{ij}, and absolute deadline d_{ij}.

Task parameters are execution time distributions, minimum time between two job arrivals p_i, and relative deadline d_i. Execution time distributions are uniform random distributions describing the execution time of a task's job. Moreover, execution time distributions may change with a given error probability to model errors, as described in Section III-B. The minimum time between two job arrivals exhibits the worst case job arrival

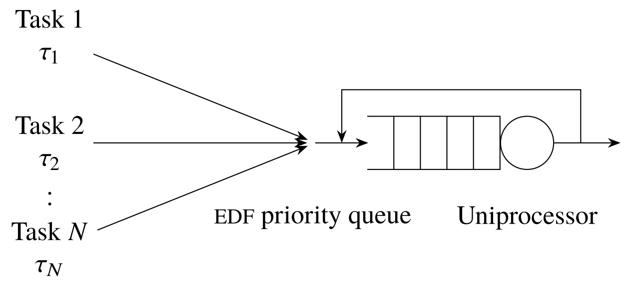

Figure 1. Simulation model of `Thready`. Each task τ_i generates jobs which are EDF scheduled on the uniprocessor. If a job with a higher priority (=earlier deadline) arrives, the current job on the uniprocessor is preempted and returns to the queue.

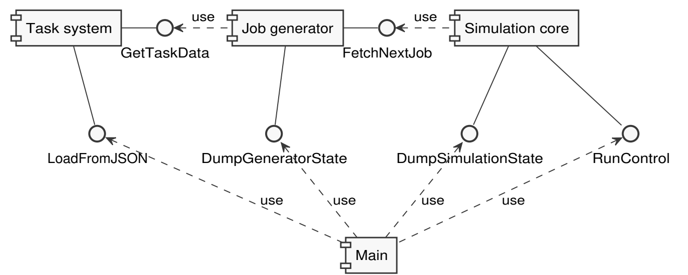

Figure 2. UML component diagram of `Thready`

sequence in terms of processor demand. Depending on the parameterization of the exponential inter-arrival time distribution for the task, the average time between two arrivals tends to be larger.

B. Error model

For any job of a task the execution time distribution may change with a given error probability. This allows to model different kinds of errors. For example, in a mixed criticality system one may specify multiple execution time distributions for high criticality tasks. Then the error probability describes the probability to overrun a job's execution time budget. Another example are transient errors that force a job to restart [12]. The user's application therefore defines the meaning of error, and the simulator is agnostic to the cause of the error.

IV. Interface and implementation

`Thready` consists of two major components as shown in Fig. 2: A job generator, and the simulation core which consumes jobs. For each task, the job generator raises jobs according to the task's parameterization, and provides the jobs sorted in order of arrival time to the simulation core. Generation of the next job takes place as soon as the job is handed to the simulation core. The simulation core fetches jobs from the generator, puts them into his job queue, and schedules the jobs according to EDF on the uniprocessor.

A. User interface

The task systems are described in JSON. `Thready` follows common UNIX conventions in specifying command line parameters

Table I
COMPARISON OF SIMULATOR FEATURES

	Scheduling algorithms		Essential features			Ancillary features		
Name	Uniprocessor	Multiprocessor	Shared resources	Error model	Performance	Tracing	Evaluation	Task generation
`RTSim` [4]	✓	✓				✓		
`rtlib` [9]	✓	✓	✓			✓		
`Cheddar` [2]	✓	✓	✓			✓	✓	✓
`YAO-SIM` [10, 11]	✓						✓	
`Realtss` [6]	✓	✓	✓			✓	✓	
`schesim` [7]	✓	✓				✓	✓	✓
`STRESS` [5]	✓	✓	✓			✓	✓	
`Fortissimo` [8]						✓		✓
`Thready`	✓			✓	✓			

for configuration, and keeps quiet during normal use, except mentioning the final simulation result to stdout. If an error was simulated, `Thready` outputs information about the first job that experienced an error. Once the simulation ends, `Thready` dumps the scheduler's job queue and current simulation time to a JSON file. This state dump can be investigated or used to continue the simulation. Moreover, the task systems are described in JSON as well, which allows to instrument `Thready` by any software capable of reading and writing JSON.

B. Job generator implementation

The job generator is responsible for creating a stream of jobs for each task according to the task's parameters. Every time the simulation core consumes a job of task τ_i, the job generator creates the next job for τ_i.

The on demand job creation keeps the memory footprint low, which increases performance by avoiding page misses and allows to run several simulations in parallel on a single machine. Contrary, generating and storing a full job trace slows down execution through page misses, and limits the possible simulation time to the available memory.

Jobs are sorted by arrival time, that is, the time the job enters the scheduler's queue, using a priority queue. This ensures correct order of job arrival from several tasks for the simulation core.

C. Simulation core implementation

The simulation core implements EDF scheduling by sorting arriving jobs by their deadline with a priority queue. The job with the earliest deadline is the job with the highest priority, and gets access to the uniprocessor. To speed up the simulation, time is progressed nonlinear by advancing from event to event. Events mark decision points where the EDF scheduler may switch the current running job. For example, if a new job arrives, the deadline of the currently running job is compared with the deadline of the arriving job. If the arriving job has an earlier deadline, the current job is preempted and returns to the scheduler's queue. Other examples are that a job finishes, or that a deadline is missed. Therefore, the simulation core can be described as an event loop, which traverses through all events until it returns. The return code of the event loop is evaluated to inform the user about the cause of the exit, and the current simulator state is dumped to a JSON file.

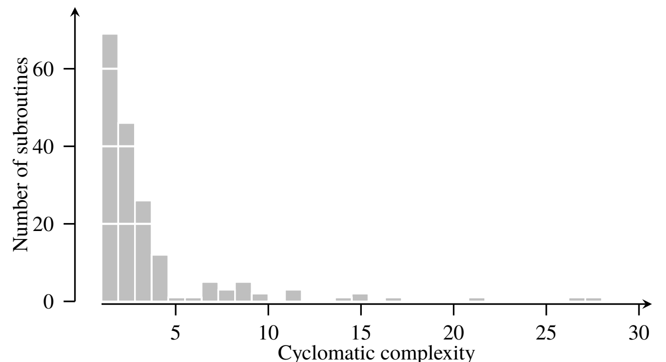

Figure 3. Histogram of subroutine cyclomatic complexity. Cyclomatic complexity corresponds to the number of independent paths through a piece of code, which increase with additional control flow statements, and should be as low as possible to ease unit testing and code maintenance. Values below 10 indicate easy to maintain code, showing that nearly every subroutine in `Thready`'s code base is of low complexity.

D. Distribution

The simulator is dedicated to the public domain, to foster access for the community in the spirit of early research software like SPICE, and to mitigate possible reproducibility issues. Motivated by performance `Thready` is written in modern C, and distributed in source form [13]. The small code base of 4599 lines of code and low complexity, as shown in Fig. 3, makes it easy to integrate `Thready` into scientific data analysis pipelines.

V. EVALUATION

`Thready` is written with simulation performance in mind to allow long term simulations in reasonable wall clock time. `Thready` allows to leverage the computation resources of a distributed, shared, and heterogeneous laboratory computer environment by portability and composability: `Thready` plays well together with other command line tools like GNU `Parallel` [14], which allows to run a plethora of simulations in parallel distributed on several physical machines.

`Thready` is trimmed down in size to fit in the level 1 (L1) cache of most low- and mid range desktop CPUs, which results in large performance gains and allows to run several simulation threads on a single machine without running into memory problems. This is especially beneficial on shared computers which do not

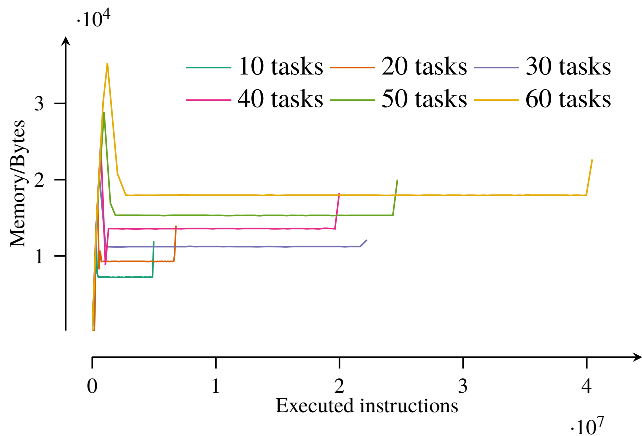

Figure 4. Heap memory of Thready during simulation of different task systems. Task systems of different size and fixed utilization of 0.7 are randomly created. Each simulation sees an increase in memory usage at the beginning and end of the simulation when JSON data is parsed or simulation results are serialized. Due to the job generation approach, the memory consumption stays constant during the actual simulation.

employ any user quotas on memory usage, or provide any other guarantee for available resources.

In the following subsections, we quantify Thready's advantages in simulation speed and memory usage, and compare our results with other simulators.

A. Memory usage

To quantify the memory consumption of Thready we profile its heap memory using Valgrind's massif [15]. We generate random task sets using the UUnifast algorithm, to ensure a uniform distributed utilization over all tasks [16]. To get results that are applicable to Thready's prime use case, some tasks are allowed to release jobs that experience an error, and therefore take longer to execute.

Memory consumption peaks at ≈ 34.5 KiB before the actual simulation by reading and parsing the JSON task system file. During simulation memory demand is constant at ≈ 2.6 KiB due to the job generation approach described in Section IV-B, which allows Thready to run long term simulations. Moreover, this is irrespective of the number of tasks in the task system, as can be seen by the qualitative similar curves in Fig. 4.

We selected rtlib to compare our results with other simulators, because rtlib can be considered as the fastest available simulator framework with minimal memory consumption available. Moreover, its flexibility allows to construct a simulator similar in features to Thready. We disabled tracing for the rtlib based EDF simulator because it severely penalizes simulation speed and memory consumption, and to make the comparison as fair as possible. The rtlib based EDF simulator and Thready are used to simulate the same task system, which is given in Table II, for a ten hour simulation in millisecond time step resolution. For this simulation, each task's relative deadline and period are equal. As an implicit deadline task system, it is EDF schedulable with c_1 as worst case execution time (WCET) [17]. Moreover, actual computation demand of jobs is uniformly distributed $\mathcal{U}(c_0, c_1)$, and there is no additional inter arrival time between two jobs of the same task.

Table II
IMPLICIT DEADLINE TASK SYSTEM FOR COMPARING SIMULATORS, WHICH IS EDF SCHEDULABLE WITH c_1 AS WCET.

Task i	Period	c_0	c_1
1	10	2	4
2	30	1	3
3	40	1	4
4	10	1	2

Table III
DETAILED PERFORMANCE RESULTS

		rtlib	Thready
	Runtime/s	9.96×10^3	1.68
	Context switches	2.49×10^3	1
	CPU migrations	1	0
	Page faults	168	70
L1 data cache	Loads	24.30×10^{12}	4.11×10^9
	Misses	8.76×10^9	36.44×10^3

The rtlib based EDF simulator allocates all required memory up front to the simulation, which results in a flat memory profile of ≈ 134.8 KiB reported by Valgrind's massif. We compare this value to Thready's constant memory profile of ≈ 1.7 KiB starting after JSON parsing finishes and the simulation core starts to operate. Compared with the rtlib based EDF simulator, Thready requires ≈ 98.7 % less heap memory, which enables to run many simulation threads in parallel on a single machine. Moreover, the small memory footprint facilitates the simulation process, as shown in Section V-B.

B. Performance

To quantify the simulation speed of Thready and the rtlib based EDF simulator we profile both with the LINUX tool perf. Both simulators run a long term simulation of the task system in Table II for 10 years simulated time in millisecond time step resolution. The system running the simulations is a low-end workstation, where the simulation threads are first in first out scheduled with highest priority to minimize the performance penalty of context switches and CPU migrations. Detailed results averaged from eight repetition runs are presented in Table III.

The rtlib based EDF simulator runs for (9959.96 ± 443.75) s to simulate the task system in Table II for 10 years. Thready runs for (1.6823 ± 0.0243) s to simulate the same, which is a speedup in latency of Thready with respect to the rtlib based EDF simulator of ≈ 5920. The three orders of magnitude improvement of Thready over rtlib in performance is facilitated by the small memory footprint of Thready, which avoids memory bottlenecks and leverages L1 data cache performance.

VI. CASE STUDY

The advantages of Thready are demonstrated best with a case study. We investigate a dual-criticality task system for ten years under transient errors. The system, shown in Table IV,

Table IV
CASE STUDY IMPLICIT DEADLINE TASK SYSTEM

Task i	Period	c_0	c_1	c_2	c_3	c_4	c_5
0	5	1	4	–	–	–	–
1	20	1	1	2	4	5	8
2	20	1	2	3	4	5	8

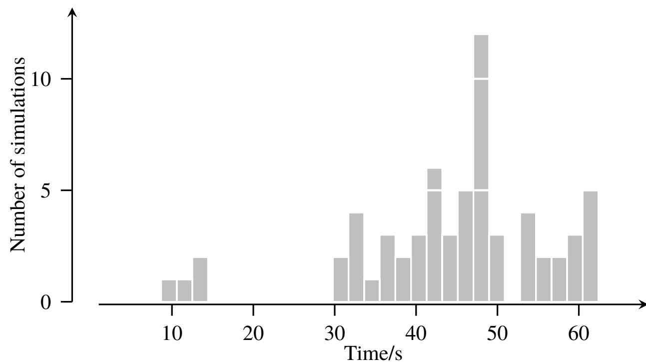

Figure 5. Histogram of simulation wall clock time. Measurements are gathered from 64 repetitions of a 10 year simulation of the task system in Table IV. Mean simulation wall clock time is 44.5 s.

consists of one low criticality task, and two high criticality tasks. Jobs from the high criticality tasks are not allowed to miss their deadline. The system is scheduled similar to earliest deadline first with virtual deadlines, where earlier, virtual deadlines are introduced to reserve time for a mode change which abandons all low criticality jobs.

Jobs from high criticality tasks may take longer to finish with a probability of $p = 0.05$, or may experience transient errors which require the job to restart with a probability of $p = 0.05$. In 90 % the job's execution time is uniformly drawn between c_0 and c_1. If a job may take longer, the job's execution time is uniformly drawn between c_2 and c_3. The transient error that requires the job to restart is modeled with a uniform execution time distribution $\mathcal{U}(c_4, c_5)$. `Thready`'s user interface allows to easily repeat the simulation experiment 64 times by running `Thready` with different random seeds using GNU `Parallel`. From all simulations, only two miss a virtual deadline, and no system violates its real deadline. The mean simulation wall clock time is 44.5 s, as indicated by the histogram in Fig. 5.

Sophisticated long-term simulations under errors are possible due to `Thready`'s flexible error model, easy to instrument interface, and simulation performance: The three order of magnitude speedup in latency allows to simulate in 44.5 s what else would take $\approx$ 12.4 h.

VII. Discussion

`Thready` combines the performance benefits from simulator framework approaches with easy to instrument interfaces and support for diverse error models. This frees the system designer from the burden of developing interfaces or single purpose programs, and fosters the development of reproducible scientific data analysis pipelines. Apparently `Thready` is a specialized and minimal tool in the tradition of UNIX: It solves one problem and solves it well [18]. Although other simulators provide more features like different scheduling algorithms, shared resources, or visualization, they lack in performance for long term simulations. Implementing such features for `Thready` requires to build further tools, which is easily enabled by `Thready`'s simple, short, and extensible code base.

VIII. Conclusion

`Thready` is the first portable and reproducible open source simulator to address long term simulations for sporadic task systems under errors. By `Thready`'s three order of magnitude speedup in latency compared with the fastest state of the art simulator framework, system designers can investigate average case performance and QoS metrics, which facilitates understanding and better design decisions. Moreover, `Thready` is easy to integrate with other programs due to its interface, which enables sophisticated simulations, fosters integration into scientific data analysis pipelines, and encourages reproducibility for scheduling simulation experiments.

References

[1] Reinhard Wilhelm et al. "The Worst-Case Execution-Time Problem: Overview of Methods and Survey of Tools". In: *ACM Trans. Embedded Comput. Syst.* 7.3 (2008), pp. 1–53.

[2] Frank Singhoff et al. "Cheddar: a Flexible Real Time Scheduling Framework". In: *ACM SIGAda Ada Lett.* Vol. 24. 4. 2004, pp. 1–8.

[3] Peter Ivie and Douglas Thain. "Reproducibility in Scientific Computing". In: *ACM Computing Surveys* 51.3 (2018), pp. 1–36.

[4] Aleardo Manacero Jr., Marcelo B. Miola, and Viviane A. Nabuco. "Teaching Real-Time with a scheduler simulator". In: *31st ASEE/IEEE Frontiers in Educ. Conf.* Vol. 2. IEEE. 2001, pp. 15–19.

[5] Neil C. Audsley et al. "STRESS: A Simulator for Hard Real-time Systems". In: *Software: Practice and Experience* 24.6 (1994), pp. 543–564.

[6] Arnoldo Diaz, Ruben Batista, and Oskardie Castro. "Realtss: a real-time scheduling simulator". In: *4th Int. Conf. Electr. and Electron. Eng.* IEEE. 2007, pp. 165–168.

[7] Yutaka Matsubara et al. "An Open-Source Flexible Scheduling Simulator for Real-time Applications". In: *IEEE 15th Int. Symp. Object/Compon./Service-Oriented Real-Time Distrib. Comput.* IEEE. 2012, pp. 16–22.

[8] Thorsten Kramp, Matthias Adrian, and Rainer Koster. "An Open Framework for Real-Time Scheduling Simulation". In: *Int. Parallel and Distrib. Process. Symp.* Springer. 2000, pp. 766–772.

[9] Giuseppe Lipari et al. *RTSIM: Real-Time system SIMulator*. http://rtsim.sssup.it/. 2009.

[10] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. *YAO-SIM: Yet Another Operating system SIMulator for real-time scheduling*. http://yaosim.sssup.it/. 2011.

[11] Gang Yao, Giorgio Buttazzo, and Marko Bertogna. "Feasibility analysis under fixed priority scheduling with limited preemptions". In: *Real-Time Systems* 47.3 (2011), pp. 198–223.

[12] Robert Schmidt et al. "Reliability Improvements for Multiprocessor Systems by Health-Aware Task Scheduling". In: *IEEE 24th Int. Symp. On-Line Testing and Robust Syst. Design*. 2018, pp. 247–250.

[13] Robert Schmidt. *thready - A lightweight and fast scheduling simulator*. http://doi.org/10.5281/zenodo.3601863. 2020.

[14] Ole Tange. "GNU Parallel: The Command-Line Power Tool". In: *;login:* 36.1 (2011), pp. 42–47.

[15] Nicholas Nethercote and Julian Seward. "Valgrind: a framework for heavyweight dynamic binary instrumentation". In: *ACM Sigplan notices*. Vol. 42. 6. 2007, pp. 89–100.

[16] Enrico Bini and Giorgio C. Buttazzo. "Measuring the Performance of Schedulability Tests". In: *Real-Time Systems* 30.1-2 (2005), pp. 129–154.

[17] C. L. Liu and James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In: *J. ACM* 20.1 (1973), pp. 46–61.

[18] Malcolm D. McIlroy, Elliot N. Pinson, and Berkley A. Tague. "UNIX Time-Sharing System: Foreword". In: *Bell Syst. Tech. J.* 57.6 (1978), pp. 1899–1904.