PROCEEDINGS OF

OSPERT 2024

The 18th Annual Workshop on
Operating Systems Platforms for
Embedded Real-Time Applications

July 9th, 2024 in Lille, France

in conjunction with

ECRTS

The 36th Euromicro Conference on Real-Time Systems
July 9–12, 2024, Lille, France

Editors:
Alexander Zuepke
Kuan-Hsun Chen

Contents

Message from the Chairs

Welcome to OSPERT'24, the 18th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications. This year, OSPERT will provide a combined program with the RT-AutoSec and RT-Cloud workshops. We invite you to join us in participating in a workshop of lively discussions, exchanging ideas about systems issues related to real-time and embedded systems.

The workshop will open with a keynote by Dr.-Ing. Martin Ring. He will discuss the challenges the interdependence of safety and security pose in a product's lifecycle, their impact on Bosch and its products, potential solutions, and open research questions After the technical presentations, we will have a second keynote, shared between RTAutoSec and OSPERT, from Dr.-Ing. Zain Hammadeh, discussing safety and security in software development at the German Aerospace Center (DLR). In the afternoon, we will have technical sessions from the RT-Cloud workshop and a keynote given by Dr. George Violettas from SYSGO GmbH, Germany, discussing safety-critical cloud applications. At the end, we conclude with an overarching panel.

OSPERT'24 received two submissions from which all were selected by the program commitee to be presented at the workshop. Each paper received four individual reviews. Our special thanks go to the program committee, a team of eight experts for volunteering their time and effort to provide useful feedback to the authors, and of course to all the authors for their contributions and hard work.

OSPERT'24 would not have been possible without the support of many people. The first thanks are due to Rodolfo Pellizzoni, Julien Forget, and the whole ECRTS organizing team for entrusting us with organizing OSPERT, and for their continued support of the workshop. We would also like to thank the chairs of prior editions of the workshop who shaped OSPERT and let it grow into the successful event that it is today.

Last, but not least, we thank you, the audience, for your participation. Through your stimulating questions and lively interest you help to define and improve OSPERT. We hope you will enjoy this day.

The Workshop Chairs,

Alexander Zuepke
Technische Universität München
Germany

Kuan-Hsun Chen
Universiteit Twente
The Netherlands

Program Committee

Keynote Talk

Safety and Security on a Journey to Outer Space: Navigating the Complex Relationship

Zain Hammadeh
Research Scientist, German Aerospace Center (DLR)

As humanity ventures further into the cosmos, the complexities of space exploration extend beyond engineering marvels and scientific discoveries. Central to these challenges is the development of robust, real-time yet secure software applications essential for the success and safety of space missions. This keynote addresses the unique obstacles faced by software developers in this high-stakes environment, particularly focusing on the effects of space radiation and other extraterrestrial hazards on software reliability, security and performance.

Zain Hammadeh is a research scientist at the German Aerospace Center (DLR) since 2019. He earned his Ph.D. (Dr.-Ing.) in 2019 from TU Braunschweig, Germany, under the supervision of Prof. Rolf Ernst. His research group, Real-Time Software and Security (RTS), focuses on developing safe and secure flight software for space systems.

A Preliminary Assessment of the real-time capabilities of Real-Time Linux on Raspberry Pi 5

Wannes Dewit
SOFT Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
wannes.guido.v.dewit@vub.be

Antonio Paolillo
SOFT Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
antonio.paolillo@vub.be

Joël Goossens
Département d'Informatique
Université libre de Bruxelles
Brussels, Belgium
joel.goossens@ulb.be

***Abstract*—This preliminary study evaluates the practical real-time capabilities of Real-Time Linux with the `PREEMPT_RT` patch on the Raspberry Pi 5, using Cyclictest and various stressors to simulate extreme operational conditions. By comparing the performance and predictability of Linux kernels with and without the `PREEMPT_RT` patch, we establish quantifiable metrics that demonstrate significant improvements in determinism and reduced latency: notably, we observed on Real-Time Linux a $294\times$ shorter maximum latency than on regular Linux. Our findings contribute to a deeper understanding of Real-Time Linux's potential in industrial applications. Our work aims, in the longer term, to establish a measurement-based assessment methodology of the real-time performance and capabilities of real-time operating systems.**

***Index Terms*—Real-Time Linux, PREEMPT_RT, scheduling latency, benchmarking, determinism**

I. Introduction

The explosion of the number of connected devices, automation, and the increasing need for real-time operations in various sectors have prompted the search for operating systems that can meet strict timing requirements. Along proprietary solutions, Linux, the open-source operating system kernel, has seen adaptations like Real-Time Linux to address these real-time demands [1]. Such adaptations are crucial for sectors such as industrial control [2], automotive [3], and even in video games [4], where milliseconds can make a difference.

Recently, the `PREEMPT_RT` patch series, aimed at making the kernel fully preemptive and real-time but currently not yet fully merged into mainline, has gained traction in the Linux community [5], [6].

Research question. How does the implementation of Real-Time Linux with the `PREEMPT_RT` patch affect the scheduling latency and predictability on the Raspberry Pi 5 compared to previous models and other platforms? The Raspberry Pi 5 introduces significant hardware improvements over its predecessors, potentially offering better performance and more deterministic behavior under real-time conditions. Evaluating Real-Time Linux on this updated platform can provide insights into how these hardware advancements contribute to real-time capabilities and whether they justify (or not) the use of the Raspberry Pi 5 in real-time applications. Our goal is to scrutinize and understand the real-time capabilities of Real-Time Linux and evaluate whether it qualifies as a Real-Time Operating System (RTOS). By doing so, we aim to establish a methodology for RTOS assessment enabling us to compare, for example, the behaviour of Linux when applying (or not) the `PREEMPT_RT` patch. This requires the definition of metrics adapted to real-time workloads (as "fast" does not mean "predictable") and associated benchmarks simulating real-time applications.

Overall project. We plan to use existing benchmarking tools and suites, like `Cyclictest` [7], `benchkit` [8] `RTEval` [9], and `rtbench` [10], to evaluate the impact of the `PREEMPT_RT` patch on both performance and real-time metrics. The eventual goals of this project are (1) to produce a comprehensive analysis detailing the real-time capabilities of Real-Time Linux on the chosen platforms, and (2) to provide a benchmarking-based methodology that can be reused for different hardware platforms, other versions of the kernel or even for different RTOS.

This paper. In this experience report, we present preliminary evaluation results of latencies measured with `Cyclictest` when running on Linux, with and without `PREEMPT_RT`, when the system is under heavy stressing conditions, using the `stress-ng` and `iperf3` stressing tools. Using that setting, applying `PREEMPT_RT` results in reducing the maximum observed latency by a $294\times$ factor. While being currently restricted to a single platform and a single benchmark, these results enable us to show key differences between Linux and its real-time equivalent.

II. Background

A. Linux and real-time

The main goal of the `PREEMPT_RT` patch is to make Linux real-time compliant by making the kernel fully preemptible. In the long term, the patch aims to be upstreamed, merging these real-time capabilities as build options available within mainline Linux. As such, a lot of work from the real-time Linux community has already been merged [11] and has even improved Linux performance in non-real-time scenarios [12]. However, notice that it was never the ambition of the real-time Linux community to make the Linux kernel completely hard real-time. This would lead to the loss of a lot of features expected from a modern general-purpose operating system and nowadays considered standard for Linux users.

For example, in the overloaded case – when the task set utilization is greater than the number of CPUs, i.e. there is more work demand than the available processing capacity of the multi-core platform – users expect higher latencies and generally some performance loss. In a real-time environment, such scenario must be avoided to meet the tasks' deadlines. Upstreaming the real-time patch would encourage more users to try it by simply enabling the build option, therefore making the switch as smooth as possible [11], [13].

B. Assessing real-time capabilities

Formally verifying Linux real-time capability is a challenging endeavor [14], [15], and we contend that these initiatives must be complemented with practical testing, i.e., benchmarking, as pursued in prior work [16].

In the first iteration of this project, we propose to use `Cyclictest` to measure the scheduling latency of tasks while the system is under heavy load. `Cyclictest`, developed by Thomas Gleixner, is the de facto benchmarking tool for real-time Linux. It was used in many prior work [17]–[23]. `Cyclictest` measures the so called *scheduling latency* of a real-time system — i.e., the difference between a thread's intended wake-up time and the time at which it actually wakes up. The tool gets periodically invoked in order to calculate the max, min and average scheduling latency [7]. Since the main feature of the `PREEMPT_RT` patch is to make the Linux kernel more deterministic, important metrics to benchmark system performance are its response-time latency and jitter [16], [21], and `Cyclictest` often gets used to provide these metrics. `Cyclictest` *"provides an easy-to-interpret metric that reflects various sources of unpredictability as a single, opaque measure."* [23], making it a very useful tool to quickly compare, for example, different kernel versions. Since it is the most widely accepted benchmarking tool for real-time Linux, it is also very easy to compare new benchmarking results with results of prior work.

`Cyclictest` often gets used together with certain stressing tools [17], [18], [20]. The decision for which parts of the system need to be stressed is usually application-specific, but in general, studies pick stressers which put a load on the CPU (e.g., *stress-ng, phoronix-test-suite's openssl*), I/Os (e.g., *stress-ng, fio, build-linux-kernel*), and networking (e.g., *iperf*) [17], [19], [20].

Since we selected the recently-released Raspberry Pi 5 to conduct our experiments, notice that existing performance-related studies exist [24]. To the best of our knowledge, although many prior work used prior Raspberry Pi models to conduct Real-Time Linux studies [22], there are no other reported studies assessing real-time metrics on this device.

III. Proposed Methodology

A. Selected hardware and OS

We ran the below stressors and benchmark on a Raspberry Pi 5 (Model B Rev 1.0) running Debian 12. The platform has a 2.4 GHz quad-core 64-bit Arm Cortex-A76 CPU and a VideoCore VII GPU.

We ran our experiments in the following scenarios: (1) with a stock kernel, version 6.6.21, and (2) with a `PREEMPT_RT` patched kernel, with the same version and the compatible patch. We analyzed and compared the real-time performance of both kernels in order to determine the benefits of `PREEMPT_RT` in terms of real-time capabilities. Notice that for the stock kernel, we used the vanilla configuration provided by the instructions on Raspberry Pi website [25]. Further configuration tuning could lead to better results – we will explore this as future work. To avoid extra interrupt noise due to graphic processing, we disabled the desktop environment and ran the experiments on the Raspberry Pi through an `ssh` terminal.

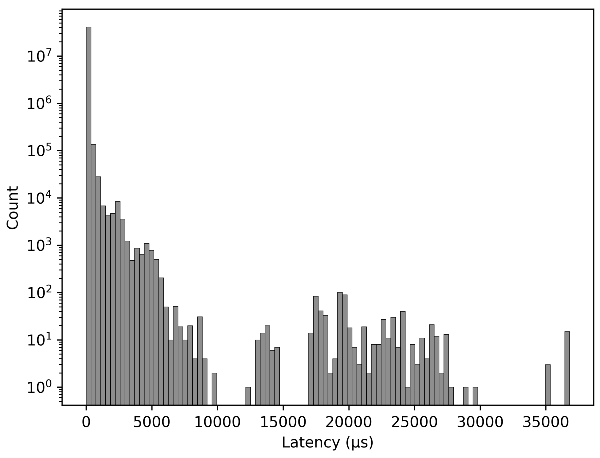
B. Selected software benchmark and stressors

Stressors are used in benchmarking practices in order to generate a computing load to push specific parts of the system to their limit. Since we aim to assess the real-time capabilities of Linux, we use different stressors concurrently with `Cyclictest`. We expect that the combination of these stressors will generate a system wide load to approach the worst-case scenario – this to determine whether the RTOS still performs deterministically under pressured circumstances.

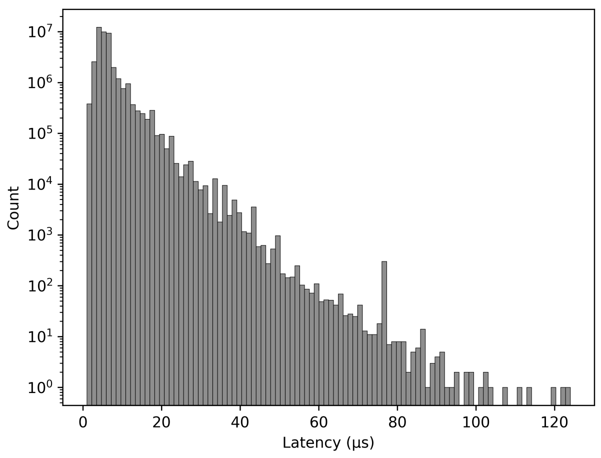
To this end, we used `stress-ng` for generating CPU and I/O load. `stress-ng` is a library of stressors, ranging from tests stressing the CPU, virtual memory, file system or memory/CPU cache [26], allowing us to generate a diverse set of resource-intensive tasks. We also used `iperf3` [27] for generating networking load, which comes with its own set of interrupts and service routines and thus possible sources of latency.

Cyclictest was configured according to best practices in the field [17], [20]–[22], using: `sudo cyclictest -vmn -i100 -p99 -t --duration=1h`. Firstly, memory allocations were disabled (`-m`), `clock_nanosleep` is used instead of POSIX interval timers (`-n`) and the output was set to verbose in order to correctly gather the needed results (`-v`). The real-time tasks that are measured by `Cyclictest` are created every 100 microseconds (`-i100`) with a priority of 99 (`-p99`). The amount of tasks that is created every interval is equal to the amount of processors of the system (`-t`). The tests were run for a duration of 1 hour (`--duration=1h`). As suggested by Adam [22], [28], processor affinity was intentionally not set with `--smp` since this would mitigate task migration. This is an important source of possible latency due to potential acquisition of locks, so it should be included in our tests. Notice that this flag was set in another study by Oliveira [19].

stress-ng ran through docker using the following command: `sudo docker run --rm colinianking/stress-ng --all 1 -t1h 1`. We decided to run all 320+ stressers in parallel (`--all 1`) for a duration of 1 hour (`-t1h`) [17], [20], [29]. Using all stressors in the library in order to generate system wide CPU and I/O load was an idea lent from Delgado, et al [20], with the only difference being that they execute all stressors sequentially, whereas we run all of them in



(a) Latencies for the stock kernel v6.6.21, without the `PREEMPT_RT` patch.



(b) Latencies for the real-time kernel v6.6.21, with the corresponding `PREEMPT_RT` patch applied.

Fig. 1: Histograms of `Cyclictest` measured scheduling latencies for both non-real-time/real-time versions of the kernel. The real-time kernel shows better observed latencies and better predictability.

parallel to trigger a highly demanding workload to maximize the stressing. We expect the system to be able to handle a multitude of different stressors at the same time, and sustain its determinism even under heavy load. We acknowledge that running all stressors in parallel on a Raspberry Pi 5, an embedded system, introduces an unrealistic degree of parallelism. However, our goal was to push the system to its absolute limits to understand the worst-case scenario performance and to stress test the capabilities of the `PREEMPT_RT` patch under extreme conditions. Future work will involve more realistic stress scenarios that reflect typical workloads encountered in embedded systems.

iperf3 was run from a remote computer using: `iperf3 -c <IP> -w 64K -P 100 -t 3800`. In order to generate networking load for our tests, the remote computer sends out 64KB packets (`-w 64K`) from 100 different virtual clients (`-P 100`) at the same time during more than an hour (`-t 3800`). Notice `iperf3` is a client/server application. We configured the Raspberry Pi 5 to act as a server (by running `iperf3` with the `-s` flag on the Raspberry Pi 5) to receive the networking load from the remote computer acting as the client (by running `iperf3` with the `-c` flag on the remote computer) [17], [20].

C. Reproducibility

We documented our system settings and reproducible methodology on a publicly available repository[1]. To reproduce our results, the provided kernel configuration must be used. To ease the process, we streamlined the process of patching and building Linux with/without `PREEMPT_RT` in a Dockerfile and provided a `README` with the commands to run the benchmarks on the target system – here the Raspberry Pi 5. In the future, we aim to create a reusable process for running stressors and benchmarks on real-time Linux (or other RTOSes) across different hardware platforms. This will enable us to build a large database of experiments that assess the real-time performance and capabilities of various Platform/OS combinations.

[1]https://github.com/apaolillo/rtlinux

TABLE I: Observed scheduling latencies with `Cyclictest`

	Average	Max	Std.
Stock kernel	14.69 µs	36802.00 µs	122.08 µs
RT kernel	5.91 µs	124.00 µs	3.25 µs

IV. Results

`Cyclictest` measures the scheduling latency of four real-time tasks on the system. These tasks are periodically woken up every 100 microseconds for a period of one hour. During our tests, where `Cyclictest` and the stressors were running concurrently, we observed the system reached 100% CPU load and about 70% RAM load. Figures 1a and 1b show the measured scheduling latencies during our experiment. The benefits of `PREEMPT_RT` in terms of predictability are clear. The results for the stock kernel are very scattered, with a maximum observed latency of 36802 microseconds. In contrast, the negative slope on the graph for the real-time kernel indicates greater stability, with a maximum observed scheduling latency of 125 microseconds – achieving a $\times 294$ improvement of the maximum observed latency. These observations, together with the general lack of outliers on Figure 1b indicates that the `PREEMPT_RT` patch succeeds in its goal of making the kernel more deterministic.

The same results are aggregated in Table I for convenience. Notice that the average scheduling latency also improved in the patched system, from 14.69 to 5.91 microseconds, with a less impressive improvement of $\times 2.49$. This points to the fact that

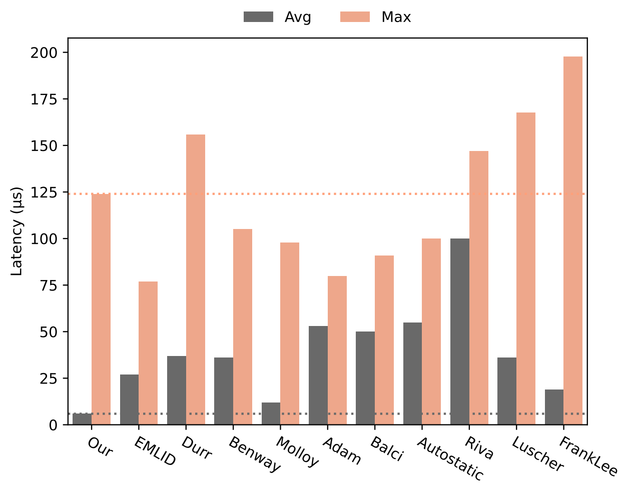


Fig. 2: Average and maximum observed scheduling latencies with `Cyclictest` in our study and various prior studies. Our own measured average and maximum scheduling latencies are drawn as horizontal lines across the graph to ease the comparison.

the goal of the `PREEMPT_RT` patch is not necessarily to make the kernel faster but mainly to make it more deterministic for real-time tasks. The reduction of the standard deviation further supports this claim and indicates how much the determinism of the kernel was improved by the patch.

In Figure 2, we compare our results obtained with the patched kernel to a series of benchmarks gathered by Adam, et al. [22], supplemented with some more recent studies [30], [31]. These benchmarks were gathered from other studies which benchmarked other Raspberry Pi models with `Cyclictest` [22], [30]–[38]. Notice that the objective of this comparison is primarily to corroborate the validity of our findings, noting that precise quantitative comparisons are not expected. Instead, an affirmation of the order of magnitude is sought, as the referenced benchmarks employed no stressors or utilized different ones, yielding varying testing conditions for the experiments. It is clear that the average scheduling latency that we obtained is substantially better than the other results. This was to be expected however, since these benchmarks were run on previous iterations of the Raspberry Pi, namely the Raspberry Pi 1, 2, 3 and 4. The Raspberry Pi 5 that we used for our benchmarks has a CPU with a clock frequency of 2.4 GHz, which is double the clock frequency of the CPU on for example the Raspberry Pi 3. The leap in average scheduling latency could thus be ascribed to the improvement in hardware, but also to improvements in the Linux kernel or the `PREEMPT_RT` patch (since other studies used different versions of those). Results of the maximum observed scheduling latency is quite comparable to the other results, which is a bit disappointing. We ascribe these subpar results to the fact that we decided to run the benchmarks on an untuned kernel for this first iteration of our project.

V. Future work

The next step of our project will be to further experiment with configuring and tuning the kernel in order to get better real-time performance results. We are confident that the Raspberry Pi 5 can still be pushed further in order to obtain better results. For example, kernel configuration options that could be explored are: disabling RT-throttling, disabling CPU frequency scaling, and raising software interrupt priority [29]. We are also interested in experimenting with the best practices in configuring the Linux kernel for real-time as described in the Red Hat manual [39].

As the goal of the research project is to establish a methodology for assessing the real-time capabilities of RTOSes, a natural next step for comparison would also be to assess the capabilities of other, non-Linux RTOSes, such as Zephyr, FreeRTOS, LITMUS^{RT}, or proprietary products. We also plan to explore other benchmarking tools such as `rt-bench` [10] or `RTEval` [9]. RTEval also uses `Cyclictest` as a measuring tool but has another approach to stressing the system. In future work, besides scheduling latency, other metrics will be considered to complete our assessment methodology, such as end-to-end response latency and RTOS jitter.

We will also consider running our experiments on other hardware platforms such as other embedded systems (e.g., Orange Pi, Raspberry Pi 4, Rock Pi 4) or many-core processors (e.g., AMD EPYC, Huawei Kunpeng, Ampere Altra).

Finally, we will streamline the process of getting real-time KPIs through the benchkit [8], aiming for a fully open-source and reproducible benchmarking pipeline.

References

[1] Canonical, "Real-time Ubuntu is now generally available," February 2023, [Online] Accessed: 2024-05-06. [Online]. Available: https://canonical.com/blog/real-time-ubuntu-is-now-generally-available

[2] C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. M. Vilches, "Towards a distributed and real-time framework for robots: Evaluation of ROS 2.0 communications for real-time robotic applications," *CoRR*, vol. abs/1809.02595, 2018. [Online]. Available: http://arxiv.org/abs/1809.02595

[3] I. Wind River Systems, "Wind River Acquires Hard Real-Time Linux Technology from FSMLabs," February 2007, [Online] Accessed: 2024-05-06. [Online]. Available: https://www.windriver.com/news/press/news-4261

[4] M. Larabel, "SteamOS Compositor Details, Kernel Patches, Screenshots," December 2013, [Online] Accessed: 2024-05-06. [Online]. Available: https://www.phoronix.com/news/MTU0MzY

[5] ——, "PREEMPT_RT Might Be Ready To Finally Land In Linux 5.20," July 2022, [Online] Accessed: 2024-05-06. [Online]. Available: https://www.phoronix.com/news/520-Maybe-Real-Time-PREEMPT_RT

[6] ——, "Real-time "rt" patches updated against linux 6.6 git," September 2023, [Online] Accessed: 2024-05-06. [Online]. Available: https://www.phoronix.com/news/Linux-RT-Patches-Linux-6.6

[7] T. L. Foundation, "Linux Foundation Realtime Wiki - HowTo - Cyclictest," [Accessed 2024-05-06]. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start

[8] open s4c, "benchkit: Benchmarking Toolkit for Performance Analysis," 2024, [Accessed 2024-05-07]. [Online]. Available: https://github.com/open-s4c/benchkit

[9] "Rteval," [Online] Accessed: 2024-05-06. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/rteval

[10] M. Nicolella, S. Roozkhosh, D. Hoornaert, A. Bastoni, and R. Mancuso, “Rt-bench: an extensible benchmark framework for the analysis and management of real-time applications,” in *Proceedings of the 30th International Conference on Real-Time Networks and Systems*, ser. RTNS ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 184–195. [Online]. Available: https://doi.org/10.1145/3534879.3534888

[11] J. Perlow. (2021) In the trenches with thomas gleixner: Real-time linux kernel patch set. [Accessed 2024-05-06]. [Online]. Available: https://www.linux.com/news/in-the-trenches-with-thomas-gleixner-real-time-linux-kernel-patch-set/

[12] T. Gleixner. (2022) A guided tour through the preempt rt castle. [Accessed 2024-05-09]. [Online]. Available: https://www.youtube.com/watch?v=o58ff38eD64&t=440s

[13] I. Stoica and H. Abdel-Waheb, “Earliest eligible virtual deadline first: A flexible and accurate mechanism for proportional share resource allocation,” Old Dominion University, USA, Tech. Rep., 1996. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=805acf7726282721504c8f00575d91ebfd750564

[14] S. Rostedt, “Kernel Recipes 2016 - Who needs a Real-Time Operating System (Not You!),” https://youtu.be/4UY7hQjEW34?si=EL8w75sHzS9WjqwK, 2016, [Accessed 2024-05-06].

[15] Bielmeier, Benno and Mauerer, Wolfgang, “Formal Verification of Embedded Linux Systems Using Trace-Based Models,” https://www.youtube.com/watch?v=w42ab8-CH1o, 2022, [Accessed 2024-05-09].

[16] F. Reghenzani, G. Massari, and W. Fornaciari, “The Real-Time Linux Kernel: A Survey on PREEMPT_RT,” *ACM Comput. Surv.*, vol. 52, no. 1, 02 2019. [Online]. Available: https://doi.org/10.1145/3297714

[17] Y. H. Jo and B. W. Choi, “Performance Evaluation of Real-time Linux for an Industrial Real-time Platform,” *International Journal of Advanced Smart Convergence*, vol. 11, no. 1, pp. 28–35, 2022. [Online]. Available: https://doi.org/10.7236/IJASC.2022.11.1.28

[18] K. Koolwal, “Investigating latency effects of the Linux real-time Preemption Patches (PREEMPT RT) on AMD ’ s GEODE LX Platform Kushal Koolwal,” in *Proceedings of the 11th OSADL Real-Time Linux Workshop*, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:11519524

[19] D. B. de Oliveira, D. Casini, R. S. de Oliveira, and T. Cucinotta, “Demystifying the Real-Time Linux Scheduling Latency,” in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Völp, Ed., vol. 165. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, pp. 9:1–9:23. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2020.9

[20] R. Delgado and B. W. Choi, “New Insights Into the Real-Time Performance of a Multicore Processor,” *IEEE Access*, vol. 8, pp. 186 199–186 211, 2020.

[21] N. Litayem and S. Ben Saoud, “Impact of the Linux real-time enhancements on the system performances for multi-core Intel architectures,” *International Journal of Computer Applications*, vol. 17, no. 3, pp. 17–23, Mar. 2011.

[22] G. K. Adam, N. Petrellis, and L. T. Doulos, “Performance assessment of linux kernels with preempt_rt on arm-based embedded devices,” *Electronics*, vol. 10, no. 11, 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/11/1331

[23] F. Cerqueira and B. B. Brandenburg, “A comparison of scheduling latency in linux, preempt-rt, and litmus rt,” in *Proceedings of the 9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT 2013)*, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:14096981

[24] M. Larabel, “Raspberry Pi 5 Benchmarks: Significantly Better Performance, Improved I/O,” September 2023, Accessed: 2024-05-08. [Online]. Available: https://www.phoronix.com/review/raspberry-pi-5-benchmarks/4

[25] Raspberry Pi Foundation. Building the Linux Kernel. [Accessed 2024-05-09]. [Online]. Available: https://www.raspberrypi.com/documentation/computers/linux_kernel.html#building

[26] C. I. King, “stress-ng (stress next generation),” [Accessed 2024-05-06]. [Online]. Available: https://github.com/ColinIanKing/stress-ng

[27] ESnet, “iPerf - The ultimate speed test tool for TCP, UDP and SCTP,” [Accessed 2024-05-06]. [Online]. Available: https://iperf.fr/

[28] G. K. Adam, “Real-time performance and response latency measurements of linux kernels on single-board computers,” *Computers*, vol. 10, no. 5, 2021. [Online]. Available: https://www.mdpi.com/2073-431X/10/5/64

[29] P. Karachatzis, J. Ruh, and S. S. Craciunas, “An evaluation of time-triggered scheduling in the linux kernel,” in *Proceedings of the 31st International Conference on Real-Time Networks and Systems*, ser. RTNS ’23. New York, NY, USA: Association for Computing Machinery, 2023, p. 119–131. [Online]. Available: https://doi.org/10.1145/3575757.3593660

[30] M. Lüscher, “Real-Time Linux on the Raspberry Pi,” 2018, [Accessed 2024-05-09]. [Online]. Available: https://www.get-edi.io/Real-Time-Linux-on-the-Raspberry-Pi/

[31] F. Lee, “How to optimize real-time performance,” 2023, [Accessed 2024-05-09]. [Online]. Available: https://forums.raspberrypi.com/viewtopic.php?t=363635

[32] D. Molloy, *Exploring Raspberry Pi Interfacing to the Real World with Embedded Linux*. Indianapolis, IN, USA: John Wiley & Sons, Inc., 2016.

[33] EMLID, “EMLID Raspberry Pi Real-Time Kernel,” [Accessed 03-05-2024]. [Online]. Available: https://emlid.com/raspberry-pi-real-time-kernel/

[34] Durr, Frank, “Raspberry Pi Going Realtime with RT Preempt,” [Accessed 03-05-2024]. [Online]. Available: http://www.frank-durr.de/?p=203

[35] Benway, Joel, “Real-Time on Raspberry Pi,” [Accessed 03-05-2024]. [Online]. Available: https://www.distek.com/blog/part-2-real-time-on-raspberry-pi/

[36] Metehan Balci, “Latency of Raspberry Pi3 on Standard and Real-Time Linux 4.9 Kernel,” [Accessed 03-05-2024]. [Online]. Available: https://metebalci.com/blog/latency-of-raspberry-pi-3-on-standard-and-real-time-linux-4.9-kernel/

[37] AUTOSTATIC, “RPi3 and the Real Time Kernel,” [Accessed 03-05-2024]. [Online]. Available: https://autostatic.com/2017/06/27/rpi-3-and-the-real-time-kernel/

[38] Lema Riva, “Raspberry Pi: Preempt-RT vs. Standard Kernel 4.14.y,” [Accessed 03-05-2024]. [Online]. Available: https://lemariva.com/blog/2018/02/raspberry-pi-rt-preempt-vs-standard-kernel-4-14-y

[39] Red Hat, “Real-Time Kernel Tuning in RHEL 8,” [Accessed 03-05-2024]. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_for_real_time/8/html/optimizing_rhel_8_for_real_time_for_low_latency_operation/real-time-kernel-tuning-in-rhel-8_optimizing-rhel8-for-real-time-for-low-latency-operation#doc-wrapper

Towards Enabling Synchronous Releases for Periodic Tasks in RTEMS

Tristan Seidl*, Mario Günzel*, Jian-Jia Chen*, Kuan-Hsun Chen†
*Department of Computer Science, TU Dortmund University, Germany
†Department of Computer Science, University of Twente, the Netherlands
E-mail: {tristan.seidl, mario.guenzel, jian-jia.chen}@tu-dortmund.de, k.h.chen@utwente.nl

***Abstract*—In addition to functional correctness, real-time systems offer temporal correctness which is important for safety-critical domains like avionics. Inherently, these systems deal with reoccurring functionality that can be modeled as periodic tasks. These task-sets are managed by specialized real-time operating systems (RTOS) because they feature predictability and advanced scheduling mechanisms. Within research, RTOSs are commonly used to evaluate analytical results.**

When periodic task-sets are analyzed in the literature, their fixed release pattern can be exploited analytically. Furthermore, if all tasks have constrained deadlines and release their first job synchronously, the schedulability can be determined by only analyzing the first job of each task. The well-established RTOS *Real-Time Executive for Multiprocessor Systems (RTEMS)* provides the specification of periodic tasks without task phases. Usually, without specification of phases, periodic tasks are considered synchronous. However, in this work, we demonstrate that RTEMS invokes task phases dependent on the task execution behavior. Hence, in RTEMS tasks are not released synchronously. Furthermore, since periodic tasks do not even have a fixed release pattern, many analytical results for periodic tasks from the literature are not applicable. Our objective in this work is to shift RTEMS' release strategy towards a fixed synchronous release of periodic task-sets with implicit deadlines. We propose two treatments that are implemented on kernel- and user-level respectively.

***Index Terms*—Real-Time Operating Systems, RTEMS, Periodic Task-Sets, Synchronization**

I. Introduction

Safety-critical systems often have to comply to specific timing constraints, i.e., the response of jobs has to be delivered in-time. Such systems do not only require functional correctness, but also temporal correctness has to be guaranteed, so-called real-time systems. Prime examples are navigation and path planning systems in the automotive or avionic domain, where violation of temporal correctness may lead to catastrophic failures. Recently, a survey-based industry study has shown that for critical domains like avionics, timing constraints are one of the most important development factors [2].

In the real-world practice, the operating system has to coordinate such real-time behaviors. Such real-time operating systems (RTOS) feature timing predictability and advanced scheduling mechanisms. An important factor for real-time operating systems is the ability to analyze and provide guarantees on the system behavior. One popular RTOS is called Real-Time Executive for Multiprocessor Systems (RTEMS) [9], which is an open source RTOS that is used in space flight, medical, networking and many more embedded devices. Due to its outreach, it is also a common choice in the research community for evaluation and case studies [3], [8], [10].

Inherently, real-time applications handle reoccurring functionality that can be modeled as periodic tasks. RTEMS provides the ability to implement periodic task-sets by utilizing its Rate-Monotonic Manager. A periodic task releases jobs recurrently, and two subsequent job releases are exactly one task period apart. Periodic tasks are known for their fixed release pattern that repeats every hyperperiod, i.e. the least common multiple of task periods, which can be exploited analytically [4]–[6]. Moreover, when all tasks release simultaneously at time 0, i.e. it is a synchronous periodic task-set, and it is a task-set with constrained or implicit deadlines, i.e. the tasks' relative deadlines are less or equal to their periods, its schedulability can be determined by solely analyzing all first jobs. [7].

Our Contributions: In this work, we show that RTEMS invokes phases on periodic task-sets dependent on the execution behavior of the tasks. Therefore, many analytical results, such as [4]–[6], are not applicable to RTEMS. Even for the classical result in [7], the schedulability analysis is only sufficient but not exact anymore. More specifically, our contributions of this work are summarized as follows:

- In Section III, we demonstrate that in the current version of RTEMS, tasks are released with phases dependent on the execution behavior.
- In Section III, we briefly give insight on the content of our previous work [6] where we first discovered RTEMS current release strategy.
- In Section IV, we analyze RTEMS' support of periodic tasks to identify the cause of phase invocation.
- In Section V, we propose two treatments that shift the release strategy towards synchronous releases. While the first solution achieves the ultimate goal of synchronous and predictable release patterns, it requires a patch of the RTEMS kernel and therefore might not be compliant with future versions of RTEMS. The second solution is a user-level solution. This solution achieves synchronous release pattern up to a neglectable overhead, independent of the task execution time, and allows high portability.

II. Background

In this section, we introduce the system model and the notation considered in this work. Furthermore, we explain in detail what RTEMS does to housekeep the management of periodic tasks to ensure their temporal correctness.

A. Task Model

We consider that an application designed to be executed on a real-time operating system (RTOS) is organized in tasks. Each of the tasks τ_i, with $i \in \mathbb{N}_0$, encapsulates a single functionality, and a priority P_i is assigned to each of them. Considering preemptive scheduling schemes, the priority assignment enables higher priority tasks to preempt lower priority tasks during the application's execution. In this work, we assume that $P_i > P_{i+1}$, i.e. a lower task index indicates a higher priority.

A task τ_i that repeatedly executes its functionality regulated by a fixed period T_i is classified as a periodic task. Each periodic execution of τ_i is defined to be its j-th job $J_{i,j}$, with $j \in \mathbb{N}_0$. The execution time it actually takes for such a job to be executed is denoted as $c_{i,j}$, which is assumed to be upper bounded by a task's worst-case execution time (WCET) C_i. The point in time when $J_{i,j}$ becomes ready for execution to the RTOS is defined by its release time $r_{i,j}$, where

$$T_i = r_{i,j+1} - r_{i,j}. \tag{1}$$

$J_{i,j}$ is required to finish its execution until its absolute deadline $d_{i,j}$, where $D_{i,j} = d_i - r_{i,j}$ is that job's relative deadline. In this work, we only consider tasks with implicit deadlines, i.e., $D_{i,j} = T_i$ for all $J_{i,j}$. The consequences of a job missing its deadline is dependent on the type of the real-time system, i.e., hard, soft or firm real-time system, and can range from system failure to normal operation. The release time of a task's first job $r_{i,0}$ is determined by that task's phase ϕ_i. If there is no phase specified for a task, it is assumed to be 0, releasing $J_{i,0}$ at the start of the application's execution, i.e. $r_{i,0} = 0$. In case ϕ_i of all τ_i in a task-set Γ is 0 or unspecified, their initial jobs are all released simultaneously. Such a task-set is called a synchronous periodic task-set.

Overall, we specify $\tau_i \in \Gamma$ with a tuple $\tau_i = (P_i, \phi_i, C_i, T_i)$, as shown in Fig. 1. For all schedules that we present in the remaining work, we assume that $c_{i,j} = C_i$.

B. Management of Periodic Tasks in RTEMS

In the following, we give an overview on RTEMS' management of real-time applications and its take on periodic tasks. RTEMS allows structuring the application by defining Γ. Each $\tau_i \in \Gamma$ is represented by a task object that RTEMS manages [1]. RTEMS provides an initialization task through that these task objects can be created ahead of time to setup the application and prepare its actual execution.

Throughout the application's lifetime, a task's functionality is only executed once. A periodic task is therefore required to execute its functionality in a loop where each loop iteration corresponds to the execution of one of its jobs $J_{i,j}$. To ensure a periodic task's temporal correctness, i.e.

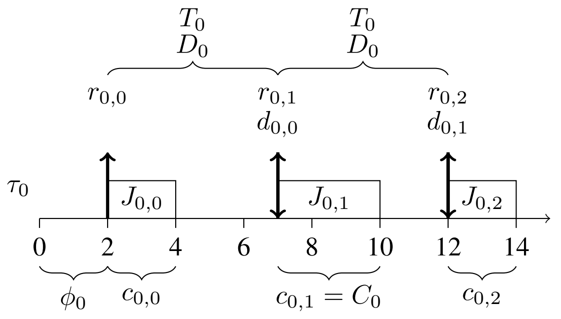


Fig. 1. The notations of the task model visualized by a possible schedule of $\Gamma = \{(1, 2, 3, 5)\}$.

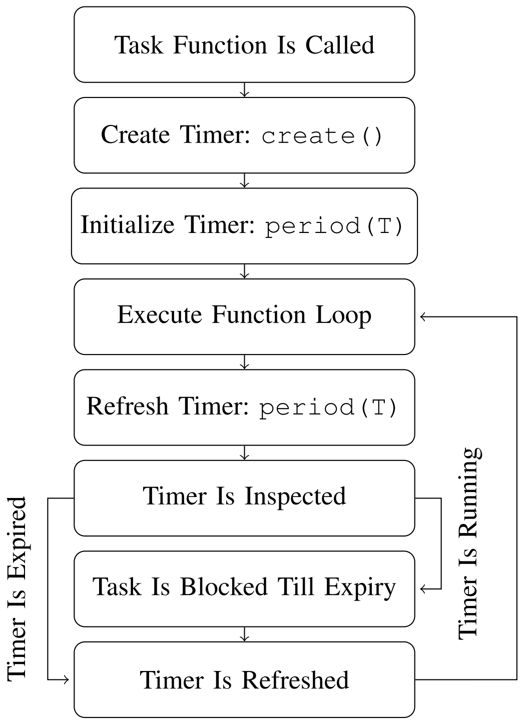


Fig. 2. The execution of a periodic RTEMS task.

maintain its periodicity specified by T_i, the looping needs to be spaced properly. For this, RTEMS provides a Rate-Monotonic-Manager that offers the required tools. It comes with a *rate monotonic period* that can be used to manage a task's periodicity. Abstracting from all data structures and implementation details, in essence, the *rate monotonic period* behaves as a timer. To avoid confusing a task's period T_i with the Rate-Monotonic-Manager's *rate monotonic period*, we just refer to the latter as a task's timer throughout this work.

Fig. 2 shows the control flow of a periodic task's execution. For preparations, the task's timer needs to be created by calling `rtems_rate_monotonic_create()`, abbr. `create()`. This is done at the beginning of a task's function, ahead of its actual looped functionality. Once created, this timer needs to be initialized and refreshed regularly by calling `rtems_rate_monotonic_period()`, abbr. `period()`.

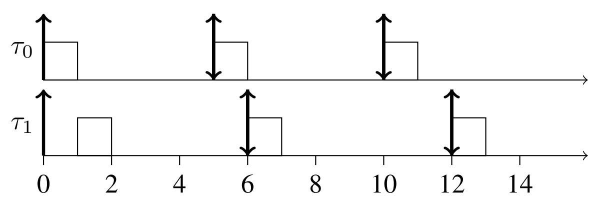


Fig. 3. The expected schedule of $\Gamma = \{(2, 0, 1, 5), (1, 0, 1, 6)\}$.

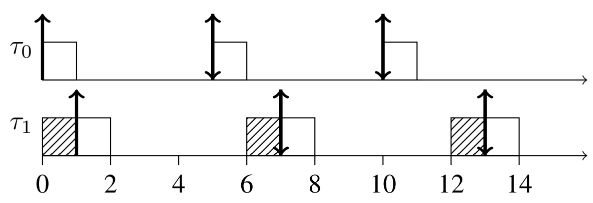


Fig. 4. The observed schedule of $\Gamma = \{(2, 0, 1, 5), (1, 0, 1, 6)\}$ implemented as an RTEMS application. The hatched area indicates the offset of $r_{i,j}$ from its intended value when the task-set would have been released synchronously.

T_i is passed as an argument to ensure subsequent loop iterations are delayed accordingly. The initialization takes place directly after calling `create()` to ensure that the timer is initialized before the function loop is entered. Since neither `create()` nor `period()` allow to specify D_i, the Rate-Monotonic-Manager only supports tasks with an implicit deadline. Whenever a job has been fully executed, i.e. a loop iteration has passed, the timer is inspected as part of its refreshment. If the timer is still running, the task will be blocked until the timer expires. As a blocked task cannot be assigned to the CPU for execution, this effectively prevents the execution of the next loop iteration and thus the execution of the subsequent job. On the contrary, if the timer is already expired, no block will be issued and the task moves to its next loop iteration, i.e. instantly releasing its upcoming job. The timer will be restarted as soon as the new job is released. If all periods passed to `period()` throughout the task's execution are identical, it will be executed in a periodic manner.

III. Problem description

RTEMS' Rate-Monotonic-Manager enables the implementation of periodic task-sets with implicit deadlines in RTEMS. In the classical literature on real-time systems, periodic tasks entail a fixed release pattern. Hence, if no task phases are specified, usually synchronous release (i.e., all phases are equal to 0) is assumed. However, in RTEMS, the release pattern of periodic tasks is not fixed. Specifically, tasks are released with phases that depend on their execution behavior.

To identify RTEMS' release strategy, we execute the following task-set Γ:

$$\Gamma = \{\tau_0, \tau_1\} = \{(2, 0, 1, 5), (1, 0, 1, 6)\} \tag{2}$$

It is a synchronous periodic task-set with implicit deadlines. Its expected schedule when released as intended, i.e., synchronously, is shown in Fig. 3. However, when Γ is executed as an RTEMS application, the schedule that is shown in Fig. 4 can be observed instead. The observed schedule differs from the expected schedule by an offset δ, marked by the hatched areas in Fig. 4. For this example, the offset δ is the size of the execution time of the first job of τ_0. Hence, the offset (or phase) of task τ_1 is not fixed but depends on the execution behavior of τ_0.

We first discovered this issue when implementing the methodology of our previous work *Scheduling Periodic Segmented Self-Suspending Tasks without Timing Anomalies* (Lin et al.) [6]. There, we focus on the problem of timing anomalies. These are counter-intuitive phenomena where a feasible segmented task-set schedule can become infeasible when one of the job's segments finishes early or a job suspends itself for less than its maximum suspension time. Common analyses account for these phenomena by over-approximating a task's τ_i worst case response time R_i (WCRT), i.e. the worst case duration from a job's release $r_{i,j}$ to its finishing time. This can result in a schedulable task-set to be considered the opposite. We implement two treatments in our previous work, namely *segment release time enforcement* and *segment priority modification*, to eliminate such timing anomalies. When we applied the latter to a segment-level fixed priority scheduling algorithm that we had added to RTEMS and executed a synchronous periodic task-set, the resulting schedule did not match the task-sets expected schedule. Since the observed behavior did not match the scope of our previous work, we focus on it in detail in this work.

Our objective in this work is twofold. First, we analyze the cause for the unpredictable task phases in RTEMS by looking into RTEMS' periodicity management. Second, we present two treatments, that allow to shift RTEMS release strategy for such a task-sets towards fixed synchronous releases. They take place at the level of RTEMS' kernel and the user's application, respectively.

IV. Analysis

In this Section, we analyze RTEMS' support of periodic tasks in detail to identify the cause for unpredictable task phases. When implemented in RTEMS, a periodic task's functionality needs to be looped in order to be executed repeatedly. This allows the task to release its functionality periodically as jobs. RTEMS' Rate-Monotonic-Manager enables to ensure their periodic releases by providing a timer that can be used to space the looping correctly. Fig. 2 visualizes that since the task's function loop is only ever executed after initializing or refreshing the task's timer, a job's release is equal to one of these two timer updates. Thus, $r_{i,j}$ equals the point in time when `period()` is called. Note that this equality is not entirely correct due to implementation details. $r_{i,j}$ is actually equal to the inspection or expiry of the task's timer, depending on the timer's state when it is inspected. However, we simplify by using the presented equality as a sufficient abstraction from the implementation details in the scope of this work.

We reason from the above, that the earliest point in time a task's first job $J_{i,0}$ can be released is when it is possible to initialize the timer of τ_i. As the period management in

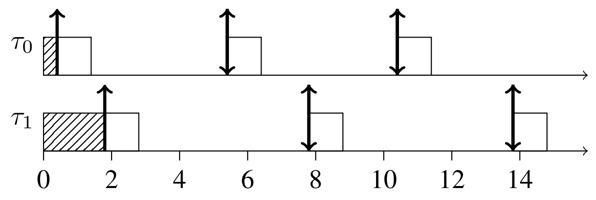


Fig. 5. The schedule of $\Gamma = \{(2, 0, 1, 5), (1, 0, 1, 6)\}$ implemented as an RTEMS application. Φ_i is indicated by the hatched areas. $\theta_i = 0.4$.

RTEMS is fully done from within the task itself, the earliest that can happen is when τ_i is executed by RTEMS for the first time. Thus, it is implied that $r_{i,0} \neq 0$, which effectively precludes the implementation of synchronous periodic task-sets in RTEMS. This implication does indeed hold true for τ_0, even though it is the first task to be executed, because it requires some time to create the timer prior to initializing it.

Let θ_i be the duration from calling a task's function to the initialization of the task's timer. This duration covers the timer's creation as well as other things like e.g., the validation of the arguments that are passed to the task's function. Accounting for θ_i, any task-set Γ given to RTEMS will have an unintended phase Φ_i added to each of its tasks that is lower bounded by

$$\Phi_0 = \theta_0 \tag{3}$$

$$\forall i \neq 0 \quad \Phi_i \geq \theta_i + \sum_{\tau_k \in \gamma_i} (c_{k,0} + \Phi_k), \tag{4}$$

where $\gamma_i = \{\tau_k \mid P_k > P_i\}$. Since $c_{k,0}$, unlike C_k, is not a fixed duration and does contribute to Φ_i, the latter cannot be predicted a priori and varies for multiple executions of the same task-set configuration. The resulting release pattern of Γ is therefore unpredictable.

Fig. 5 shows the resulting schedule of Γ from Section III when it is executed as an RTEMS application and, exemplary, $\theta_i = 0.4$. It accounts for the adjusted $r_{i,j}$, with the hatched areas indicating Φ_i. Note that the release pattern in Fig. 4 differs from the release pattern in Fig. 5 because θ_i is chosen to be 0.4 for demonstration purpose. In practice, it usually only takes up to a few system ticks.

Strictly speaking, $r_{i,j}$ is offset even more from the intended releases than it is shown in Fig. 5 due to overhead that is introduced by RTEMS itself. It consists of influences like handling the scheduling, context switches and interrupts. It is unavoidable in order to ensure the system's integrity. However, since it is a legitimate part of any RTOS and not just RTEMS, examining it in detail exceeds the scope of this work.

Avoiding the introduction of Φ_i by initializing a task's timer prior to the execution of the task's function is not a feasible solution. It requires another task, e.g., RTEMS' initialization task, to call `create()`, which in turn leads to the respective task automatically becoming the owner of the period controlling timer. Due to the Rate-Monotonic-Manager's design, only the timer's owner may call related functions like `period()` [1]. However, the task whose periodicity is to be maintained by the timer likewise needs to be the owner, as it needs to refresh its timer regularly. Thus, initializing the timer ahead of the maintained task's execution is impossible.

Overall, RTEMS' design to support periodic tasks invokes phases Φ_i dependent on the execution behavior of the task-set, and thus precludes the implementation of synchronous periodic task-sets. To shift RTEMS' towards releasing a periodic task-set synchronously as intended, modifications to RTEMS itself are required. Alternatively, the application needs to be adapted accordingly without changing its semantics.

V. Treatments

In the following subsections, we propose two treatments on different implementation levels to shift RTEMS' release strategy towards releasing tasks synchronously and thus supporting the implementation of synchronous periodic task-sets with implicit deadlines. First, we present a modification to RTEMS' kernel that adapts the initialization of the tasks' timer such that they reference the application's start instead of the point in time when the initialization is triggered. Second, we show an approach on user-level that targets to release the given task-set's first jobs *almost* simultaneously using only tools provided by RTEMS' Rate-Monotonic-Manager.

A. Kernel-Level Solution

Since `period()` initializes a task's timer by setting it to the passed argument, the point in time when the function is called is referenced. Thus, we deduce in Section IV that the release of a task's first job equals to the initialization of the timer. As `period()` needs to be called from within the task, the introduction of Φ_i is inevitable.

To shift towards a synchronous release strategy we propose to modify RTEMS' kernel such that `period()` references the start of the application when it initializes any timer. For this, we record the respective system tick t_s at that RTEMS schedules any task of the given synchronous periodic task-set with implicit deadlines Γ for the first time. We consider t_s to be the start of the application. Then, whenever a task's timer is initialized, the period T_i that gets passed to `period()` is modified before setting the timer to it such that

$$\mathcal{T}_i = T_i - (t_p - t_s), \tag{5}$$

where t_p is the system tick at that `period()` is called. By applying this modification, the task's second job $J_{i,1}$ will be released at $t_s + T_i$, i.e., as if $J_{i,0}$ is released at the application's start. The new reference point leads to the expected schedule that is displayed in Fig. 3 when executing Γ as an RTEMS application, satisfying $r_{i,0} = t_s$ for $\forall \tau_i \in \Gamma$ and thus shifting RTEMS' strategy towards synchronous releases.

The proposed treatment is simple to implement. The only modifications required are the addition of a new variable to store t_s, setting it just before calling the function of the task that is scheduled first and modifying T_i according to Eq. 5. It introduces a constant temporal overhead $\mathcal{O}(1)$ to record the system tick at the application's start as well as a constant temporal overhead $\mathcal{O}(1)$ when reading t_s, reading t_p and calculating $\mathcal{T}_i$ whenever a timer is initialized, i.e. a temporal overhead of $\mathcal{O}(|\Gamma|)$ in total. Note that Eq. 5 does not account for this overhead. Additionally, this treatment causes a small constant spatial overhead $\mathcal{O}(1)$ for storing t_s.

This approach is generically applicable for any application. Once RTEMS is modified, the user does not need to adapt their application at all but can instead solely rely on the tools that the Rate-Monotonic-Manager provides them with. While this solution achieves the ultimate goal of this paper, that is synchronous releases and fixed release patterns, it requires a patch of the kernel of RTEMS.

B. User-Level Solution

In addition to the presented kernel modification, we propose an alternative treatment to shift RTEMS' release strategy towards supporting synchronous periodic task-sets. Relying only on tools provided by RTEMS' Rate-Monotonic-Manager, it is fully implemented on user-level which allows to maximize portability. In case a kernel modification is unfavourable, this treatment can be chosen instead.

For this treatment, we introduce the concept of setup jobs. A periodic task's setup job $J_{i,setup}$ does not execute its task's usual functionality but instead, it initializes the task's timer such that a targeted $r_{i,0}$ can be achieved. If all $J_{i,setup} \in \Gamma$ coordinate the timing correctly, the task-set will behave synchronously from $r_{i,0}$ on forward.

In more detail, our treatment adds a preparation stage that precedes the application's actual execution. This stage extends every $\tau_i \in \Gamma$ with a $J_{i,setup}$. Since a setup job's sole objective is to time $r_{i,0}$ such that all $J_{i,0}$ release simultaneously, it only needs to perform two operations. First, it is required to create the task's timer by calling `create()`. Second, it needs to initialize the created timer, referencing the current point in time, which `period()` does by default. Exploiting that the timer is set to the argument that `period()` is called with, we pass a value S_i for the timer's initialization during the preparation stage. It differs from the argument T_i, which is otherwise used for the timer's periodic refreshment.

S_i needs to be chosen in such a way that $J_{i,setup}$ can ensure the simultaneously release of all $J_{i,0}$. It is dependent on $|\Gamma|$ and $c_{i,setup}$, resulting in

$$S_i = \sum_{\tau_k \in \zeta_i} c_{k,setup}, \tag{6}$$

where $\zeta_i = \{\tau_k | P_k < P_i\}$.

However, Eq. 6 has two weaknesses. First, S_i needs to be known a priori but $c_{k,setup}$ cannot be predetermined. To overcome this issue, the setup job's WCET C_{setup} is used in place of $c_{k,setup}$. Since the functionality of $J_{i,setup}$ is identical for every τ_i, besides the value of S_i, so is C_{setup}. It can be calculated using static WCET analysis. Second, S_i does not account for the temporal context switch overhead in between setup jobs that we comment on in Section IV. Assuming

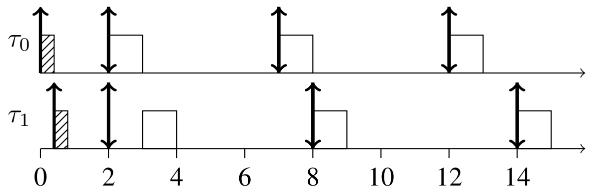


Fig. 6. A possible schedule of $\Gamma = \{(2,0,1,5),(1,0,1,6)\}$ with the approach presented in Section V-B applied. The hatched areas indicate $J_{i,setup}$.

that the context switch executes for c_{CS}, the release of two consecutive setup jobs is spaced by

$$r_{i+1,setup} - r_{i,setup} = c_{i,setup} + c_{CS}. \tag{7}$$

This effect cascades throughout all setup jobs such that

$$r_{m,0} - r_{i,0} = c_{CS} * (m - i). \tag{8}$$

Thus, S_i needs to be increased by $c_{CS} * (|\Gamma| - i - 1)$ to ensure that all $J_{i,0}$ release simultaneously. Since c_{CS} is unpredictable, the WCET C_{CS} of the context switch needs to be used instead. In total, Eq. 6 is updated such that

$$S_i = C_{CS} * (|\Gamma| - i - 1) + \sum_{\tau_k \in \zeta_i} C_{k,setup}. \tag{9}$$

Alternatively, S_i could also be determined by an educated guess. Note that choosing a value too small can lead to the preemption of $J_{i,setup}$ by some $J_{i,0}$, causing unintended behavior.

Applying this treatment to the task-set that we introduce in Section III, its execution results in a schedule similar to Fig. 6, depending on the chosen S_i.

While maximizing portability and avoiding kernel modifications, this treatment does introduce some temporal overhead. The computation time needed for the preparation stage scales with the number of tasks in Γ, i.e., $\mathcal{O}(|\Gamma|)$. Furthermore, while this approach does not achieve perfectly synchronized release of tasks, the proposed solution only invokes small task phases, only dependent on the execution behavior of setup jobs, which are neglectable in most cases.

Since we needed to implement a synchronous periodic task-set in our previous work that we briefly describe in Section III, we applied this treatment to shift RTEMS' release strategy towards synchronous task releases. After applying it, we were able to produce the expected schedule and to successfully validate the functionality of our *segment priority modification* treatment.

VI. Conclusion

Real-time operating systems (RTOS) are essential to manage safety-critical real-time systems. Such real-time systems inherently handle reoccurring functionality and can therefore be modeled as a set of periodic tasks which offers a fixed release pattern that can be exploited analytically.

In this work, we examine the release behavior of periodic tasks by the RTOS *Real-Time Executive for Multiprocessor*

Systems (RTEMS). Specifically, RTEMS invokes task phases dependent on the execution behavior of jobs. Consequently, this results in an unpredictable release pattern and analytical approaches, making classical results on periodic task-sets, such as [4]–[6], invalid under RTEMS.

To shift towards releasing the tasks of a given task-set synchronously, we propose two treatments on different implementation levels. The first option modifies RTEMS' kernel, altering the implementation of `period()` to reference the application's start when initializing a task's timer instead of the point in time when the initialization is triggered. This avoids the introduction of phases and achieves synchronous fixed release patterns. The second option offers an alternative, more portable solution on user-level. It introduces a preparation stage of setup jobs that precede the task-set's actual execution. These setup jobs exploit the timer's implementation to time the release of all first jobs of the given task-set such that they release simultaneously. This option achieves almost synchronous release patterns. Specifically, only small phases are introduced that dependent on artificial setup jobs and are neglectable in most cases.

While this paper focuses on obtaining fixed release patterns by synchronous releases, this is mainly due to the limitation of the RTEMS specification of periodic tasks. That is, if future versions of RTEMS enable the specification of task phases, our proposed treatments can be easily extended to account for these as well.

Acknowledgment

This work has been supported by European Research Council (ERC) Consolidator Award 2019, as part of PropRT (Number 865170), and by Deutsche Forschungsgemeinschaft (DFG), as part of "Suspension-Aware Designs and Analyses for Real-Time Embedded Systems" (Project no. 398602212) and the priority program "SPP 2377: Disruptive Memory Technologies" under project: "Reconfigurable Architectures and Real-Time Systems Co-Design for Non-Volatile Main Memory (ARTS-NVM)" (Project no. 502308721).

References

[1] RTEMS Classic API Guide. https://ftp.rtems.org/pub/rtems/people/joel/docs-eng/c-user/index.html.

[2] B. Akesson, M. Nasri, G. Nelissen, S. Altmeyer, and R. I. Davis. An empirical survey-based study into industry practice in real-time systems. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 3–11, 2020.

[3] G. Bloom, G. Parmer, B. Narahari, and R. Simha. Shared hardware data structures for hard real-time systems. In *ACM International Conference on Embedded Software (EMSOFT)*, page 133–142, 2012.

[4] M. Günzel, K. Chen, N. Ueter, G. von der Brüggen, M. Dürr, and J. Chen. Compositional timing analysis of asynchronized distributed cause-effect chains. *ACM Trans. Embed. Comput. Syst.*, 22(4):63:1–63:34, 2023.

[5] T. Kloda, A. Bertout, and Y. Sorel. Latency analysis for data chains of real-time periodic tasks. In *ETFA*, pages 360–367. IEEE, 2018.

[6] C.-C. Lin, M. Günzel, J. Shi, T. T. Seidl, K.-H. Chen, and J.-J. Chen. Scheduling periodic segmented self-suspending tasks without timing anomalies. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 161–173, 2023.

[7] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[8] J. Shi, C.-C. von Egidy, K.-H. Chen, and J.-J. Chen. Formal verification of resource synchronization protocol implementations: A case study in rtems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4157–4168, 2022.

[9] The RTEMS Project. Real-Time Executive for Multiprocessor Systems (RTEMS). http://www.rtems.org/, Last accessed on 2024-05-13.

[10] G. von der Brüggen, K.-H. Chen, W.-H. Huang, and J.-J. Chen. Systems with dynamic real-time guarantees in uncertain and faulty execution environments. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 303–314, 2016.

OSPERT, RT-AutoSec and RT-Cloud 2024 Program

	Tuesday, July 9 2024
8:00 – 8:45	Registration
9:00 – 10:00	OSPERT and RT-AutoSec Opening Remarks Session 1: RT-AutoSec Industry Keynote Safety & Security in Perfect Harmony through Life(cycles) *Martin Ring*
10:00 – 10:30	Coffee Break
10:30 – 11:35	Session 2: OSPERT and RT-AutoSec Technical Session [RT-AutoSec] Assured Micropatching of Race Conditions in Legacy Real-time Embedded Systems *R. Chatterjee, H. Simpa, B. Karel, R. Baratto, M. Gordon and J. Daily* [OSPERT] A Preliminary Assessment of the real-time capabilities of Real-Time Linux on Raspberry Pi 5 *W. Dewit, A. Paolillo, J. Goossens* [OSPERT] Towards Enabling Synchronous Releases for Periodic Tasks in RTEMS *T. Seidl, M. Guenzel, J.-J. Chen, and K.-H. Chen*
11:35 – 12:15	Session 3: OSPERT and RT-AutoSec Keynote Safety and Security on a Journey to Outer Space: Navigating the Complex Relationship *Zain Hammadeh*
12:15 – 13:30	Lunch
13:30 – 15:00	RT-Cloud Opening Remarks Session 3: RT-Cloud Technical Session [RT-Cloud] Dynamic Offloading of Control Algorithms to the Edge using 5G and WebAssembly *A. A. Bayati, K.-E. Årzén* [RT-Cloud] Safety-Critical Edge Robotics Architecture with Bounded End-to-End Latency *G. Gala, T. Unte, L. Maia, J. Kühbacher, I. Kadusale, M. I. Alkoudsi, G. Fohler, and S. Altmeyer* [RT-Cloud] Integrating Containers and Partitioning Hypervisors for Dependable Real-time Industrial Clouds *M. Barletta, F. Boccola, M. Cinque, L. D. Simone, R. D. Corte and D. Ottaviano* [RT-Cloud] Orchestration Done Upside Down: Self-aware Applications for Substation Automation *C. Göttel, D. Kozhaya, E. Fregnan, P. Sommer, S. Schönborn*
15:00 – 15:30	Coffee Break
15:30 – 16:20	Session 4: RT-Cloud Keynote Safety-critical cloud applications *George Violettas*
16:20 – 17:00	Session 5: Joint Panel OSPERT + RT-AutoSec + RT-Cloud Panel
17:00 – 18:00	ECRTS First-timer Reception
	Wednesday, July 10th – Friday, July 12th 2024
	ECRTS main conference.