UNIVERSITÄT AUGSBURG
Universitätsbibliothek

OPEN ACCESS
OPUS AUGSBURG

TiM2s, a lightweight, OS-agnostic Time-Multiplexed Multilevel Scheduler

Julian Linne, Severin Brunner, Sebastian Altmeyer

Angaben zur Veröffentlichung / Publication details:

Linne, Julian, Severin Brunner, and Sebastian Altmeyer. 2026. “TiM2s, a lightweight, OS-agnostic Time-Multiplexed Multilevel Scheduler.” In *2026 IEEE 31st International Conference on Emerging Technologies and Factory Automation (ETFA), 8-11 September 2026, Vasteras, Sweden*, edited by Lucia Lo Bello, Moris Behnam, and Tullio Facchinetti, 1–8. Piscataway, NJ: IEEE. https://doi.org/10.1109/etfa68377.2026.11724831.

Nutzungsbedingungen / Terms of use:

licgercopyright

Dieses Dokument wird unter folgenden Bedingungen zur Verfügung gestellt: / This document is made available under these conditions:
Deutsches Urheberrecht
Weitere Informationen finden Sie unter: / For more information see:
https://www.uni-augsburg.de/de/organisation/bibliothek/publizieren-zitieren-archivieren/publiz/



TiM2S, a Lightweight, OS-agnostic Time-Multiplexed Multilevel Scheduler

Julian Linne
University of Augsburg, Germany
linne@es-augsburg.de

Severin Brunner
University of Augsburg, Germany
brunner@es-augsburg.de

Sebastian Altmeyer
University of Augsburg, Germany
altmeyer@es-augsburg.de

***Abstract*—Temporal isolation between applications is a powerful feature: standards such as ARINC 653 mandate it as a safety requirement in tightly regulated domains such as avionics. There, it also enables collaborative system development across independent teams. However, open-source real-time operating systems rarely provide it, despite serving an increasingly broad range of mixed-criticality and multi-application use cases. Without such isolation, applications sharing a system can interfere with each other's timing. This forces schedulability analyses to either account for cross-application interference at the cost of substantial pessimism, or risk unreliable timing guarantees. To address this issue, we introduce Time-Multiplexed Multilevel Scheduler (TiM2S), a non-intrusive framework that enforces temporal isolation between co-located applications without requiring modifications to the underlying operating system. The framework is OS-agnostic by design and can be ported to any Real Time Operating System (RTOS) using fixed-priority preemptive scheduling, the de facto standard among open-source RTOSes, along with only minor additional requirements. We demonstrate this portability through instantiations for both FreeRTOS and RIOT OS. We validate our framework using a custom task set as well as a case study and demonstrate that the introduced relative runtime overhead is limited to below $0.6\ \%$ in both cases.**

***Index Terms*—Real Time Systems, FreeRTOS, RIOT OS, ARINC653, Hierarchical Scheduling**

I. Introduction

Over the last decades, real-time scheduling research has produced a myriad of different scheduling techniques and analyses [1]. Even if we restrict ourselves to unicore scheduling [2], we can choose among scheduling policies that favor predictability over performance [3], or vice versa [4], or that restrict preemption to variable degrees [5], [6]. Some of these scheduling policies have found their way into industry standards, such as ARINC [7] or OSEK/VDX [8], and commercial real-time operating systems, such as vxWorks [9] or EMCOS [10].

In stark contrast to this wide array of available scheduling policies, open-source RTOSes, such as FreeRTOS, Zephyr, or RIOT OS, only provide native support for a very limited choice of scheduling policies. Typically, only fixed-priority preemptive scheduling (FPPS) policy is supported, with Zephyr being the exception, offering additional scheduling algorithms. Limited support for other scheduling policies, such as FIFO scheduling or dynamic task priorities, is feasible, but requires more intricate task implementations [11] and knowledge about the operating system from the system developer. Not only does this increase the susceptibility to implementation errors, but it also hampers the integration of third-party software or collaborative system development.

To minimize these issues for embedded avionics systems, ARINC 653 groups the tasks of applications into separate partitions. Partitions are then scheduled using cyclic time-division multiplexing, whereas tasks within a partition are typically (but not exclusively) scheduled using FPPS. A further advantage of this hierarchical scheduling is an increase in safety and security due to temporal isolation.

In this paper, we present TiM2S, a lightweight scheduling framework that enables hierarchical scheduling similar to ARINC 653 in real-time operating systems that natively support only FPPS. Our framework ensures temporal isolation and thus enables the seamless integration of independently developed applications on the same platform. Spatial partitioning is not part of TiM2S and remains a concern of the underlying platform. TiM2S therefore provides the temporal, but not the spatial, isolation required by ARINC 653, and does not by itself meet the isolation requirements of the standard. It does not require any changes to the typical task implementation or changes to the RTOS kernel. To demonstrate the capabilities of our framework, we evaluate our implementation with a case study [12] executed on two popular open-source operating systems, FreeRTOS [13] as a tick-based and RIOT OS [14] as a tickless system. The results show that we can realize hierarchical scheduling similar to ARINC 653 with limited overhead.

II. Background

We assume a set of m independent applications $\{A_1, \ldots, A_m\}$. Each application A_k consists of a set of n_k sporadic tasks. Each task τ_i^k has an execution time C_i^k, a period or minimal interarrival time T_i^k, and a deadline D_i^k. Tasks within an application are scheduled using FPPS, optionally with release offsets O_i^k and release jitter J_i^k. Tasks are allowed to use shared mutexes within an application. Sharing mutexes across application boundaries in any case contradicts the isolation paradigm of hierarchical scheduling. Currently, TiM2S does not support resource access protocols such as the priority inheritance or the priority ceiling protocol, since these adapt task priorities independently of TiM2S. Therefore, demoting a task while it holds a lock is safe: the lock is simply held across the inactive window, and as mutexes are not shared across applications, no task of another application can be

blocked by it. The application designer assigns each task a *nominal* priority $\overline{P}$ and assumes that tasks are scheduled with FPPS according to these nominal priorities. The notation and symbols used throughout this paper are displayed in Table I.

A. ARINC 653-like Timeline Scheduling

In 1996, Aeronautical Radio, Inc. (ARINC) specified ARINC 653 as an RTOS API for aviation, also known as APplication EXecutive (APEX). To ensure that different applications run independently of one another on a system, ARINC 653 encapsulates them in partitions. The goal is not only to limit errors but also to facilitate verification, validation, and certification.

Partitions are separated from one another both in terms of memory and time. While spatial separation in memory is usually achieved using MMUs or MPUs, temporal separation must be ensured directly through the operating system's scheduling. Partitions use table-based scheduling in ARINC 653 [7]. Each table entry is a partition window, often referred to as a *minor timeframe*, with an application to be executed for a specific time. The sequence of minor timeframes repeats cyclically and such a cycle is often referred to as a *major timeframe*. Each partition must be scheduled for at least one partition window in a major timeframe. The partition scheduling table can be used to implement round robin or other, more complex scheduling policies and is provided by the system integrator. Deterministic behavior of table-based scheduling is the key aspect: This ensures the temporal separation of the partitions. Nevertheless, interrupts or communication acknowledgments can preempt a partition, even if they are unrelated to the currently executing partition, which can impair the temporal behavior of the application and thus the desired determinism [7, p. 13 ff.].

Process scheduling in ARINC 653 is preemptive and priority-based, and a process's priority can be changed even during runtime [7, p. 20].

B. Scheduling Algorithms provided by open-source RTOSes

While priority-based scheduling with fixed priorities is widely used in real-time operating systems, neither established real-time operating systems such as FreeRTOS, Zephyr, or ThreadX, nor relatively newer real-time operating systems such as RIOT OS, natively offer timeline scheduling. We note that support of FPPS by the operating system does not mean that priorities are immutable at runtime [2, Chapter 7]. Instead, the priority level must be set explicitly using system calls such as `vTaskPrioritySet()` in FreeRTOS. Using these calls, it is possible to support other scheduling policies, such as EDF [4].

We assume an operating system that supports FPPS with a priority range from $P_{min} := 0$ (lowest priority) to P_{max} (highest priority). We assume that each application requires at most $P_{max}/2 - 1$ levels of nominal priorities. Since these are typically in the range of e.g. 32 for FreeRTOS[1] or 16 for RIOT OS[2], we believe that this restriction does not pose any practical issues[3]. Except for the limited priority range, the application designer can use the scheduler interface provided by the operating system as-is without further restriction.

m	Number of independent applications
n_k	Number of tasks within application k
C_i^k	Execution time of task τ_i within application A_k
D_i^k	Deadline of task τ_i within application A_k
T_i^k	Period of task τ_i within application A_k
ϕ_i^k	Offset of task τ_i within application A_k
J_i^k	Jitter of task τ_i within application A_k
$\overline{P}_i^k$	Nominal priority of task τ_i within application A_k
s	Number of time slots of the scheduling table
$M(j)$	Mapping of time slot j to the application executed at slot j
l_j	Length of time slot j
P_i^k	Active priority of task τ_i within application A_k
P_{min}	Minimal priority supported by the operating system
P_{max}	Maximal priority supported by the operating system
τ_{idle}	Idle task provided by the operating system

TABLE I: Notations and symbols used throughout the paper.

Whenever the operating system has no active task to execute, it idles either by setting the processor to a sleep state or by executing a dedicated idle task that can always execute. We assume an RTOS that supports the latter solution of executing an idle task. If not, we can define a dedicated idle task ourselves. This task typically has the lowest priority and is treated just like a regular task.

III. TiM2S, a Time-Multiplexed Multilevel Scheduler

Our framework aims to mimic timeline scheduling with minimal overhead using priority-based preemptive scheduling as available in most open-source real-time operating systems. To be explicit, TiM2S imposes the following requirements on the RTOS:

1) Priorities are fixed, but mutable at runtime
2) There is an idle task τ_{idle} or an idle task can be defined that executes whenever there is no other active task available
3) The priority of the idle-task τ_{idle} can be changed at runtime

In addition, each application must only require at most half of the available priority range.

With TiM2S, we aim for the least-intrusive approach to activate and deactivate tasks based on the currently active application/partition. Instead of using task suspension, for instance using `vTaskSuspend()` in FreeRTOS, or similar often RTOS-specific calls that may have unforeseen side-effects and change the internal task state (in FreeRTOS[4]:

[1] https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/02-Customization#configuse_port_optimised_task_selection

[2] https://api.riot-os.org/group__core__sched.html

[3] If necessary, the nominal priorities can be saved in memory instead of separating active and inactive partitions in the available priority range, leaving more priority levels available to the application designers at the cost of one table lookup per promotion.

[4] https://freertos.org/Documentation/02-Kernel/02-Kernel-features/01-Tasks-and-co-routines/02-Task-states

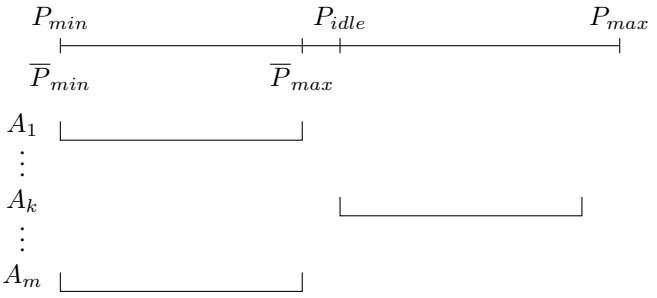


Fig. 1: Priority distribution for a system with m applications.

Blocked, Ready, Suspended, Running), we only adapt task priorities: We use the idle-task τ_{idle} as a barrier between active and inactive tasks and assign it a priority in the middle of the available priorities. Moving a task's priority below this barrier will prevent its execution without having to adapt its internal state, effectively freezing the current execution state of a task. Lifting its priority above the barrier reactivates the task.

This concept can be extended to implement hierarchical timeline scheduling for applications and FPPS for tasks inside partitions, similar to the partition scheduling required for ARINC 653. The corresponding pseudocode can be found in Algorithm 1. For an explanation of the notation and symbols used, refer to Table I. The partitions $A_1, \ldots, A_m$ are executed according to the time slots $(M(1), l_1), \ldots, (M(s), l_s)$. The available priorities P_{min} to P_{max} are split in half and priorities P_{min} to $\lfloor (P_{max} - P_{min})/2 \rfloor - 1 = P_{idle} - 1$ can be used to internally schedule the tasks of each partition. At every partition boundary, the priority of every τ_i of the previous application A_c, if any, is reset to its nominal priority $\overline{P}_i^c$ (demote), and the priority of every task of the next application A_k is increased to $P_i^k \coloneqq \overline{P}_i^k + P_{idle}$ (promote). This preserves their relative priority order within the application across a partition switch. Statically configured ceilings are not shifted along, so ceiling-based protocols would require the ceilings to be updated at every partition boundary, and we leave their analysis to future work. The wait at the end of each iteration is a delay relative to the current time, so the execution time of the demotion and promotion loops is added to the length of the corresponding time slot rather than absorbed by it; subsection IV-A quantifies this contribution.

The scheduler task receives an assignment of the application tasks to their respective partitions and the sequence of time slots, i.e., their length and the application executed for that duration. To maintain the priority distribution within each partition, the possible priority values are split: The scheduler task has priority P_{max} and the idle task gets priority $P_{idle} \coloneqq \lfloor P_{max}/2 \rfloor$. For prioritizing the application tasks within a partition, the priorities P_{min} through $P_{idle} - 1$ are available. This is shown in Fig. 1. A_k is currently active with the priorities of all tasks between P_{idle} and $P_{max} - 1$, whereas the priorities of the tasks of all other applications lie between $\overline{P}_{min} = P_{min}$ and $\overline{P}_{max} = P_{idle} - 1$.

We note that priority P_{min} can only be used at the application designer's discretion as this leaves the task in competition with the idle task of the operating system. Depending on the implemented scheduling policy, this can starve either task of any execution time and lead to unintended behavior.

Algorithm 1: The Time-Multiplexed Multilevel Scheduler.

Input: partition task sets $\Gamma_{A_1}, \ldots, \Gamma_{A_m}$, time slots $(M(1), l_1), \ldots, (M(s), l_s)$, idle task τ_{idle}, priority P_{max}
Precondition
$\forall\, 1 \leq k \leq m, \forall\, 1 \leq i \leq n_k : \overline{P}_i^k < \lfloor P_{max}/2 \rfloor$

```
P_idle ← ⌊P_max/2⌋;
set_prio(τ_idle, P_idle);
c ← ⊥;
while true do
    for j ← 1 to s do
        if c ≠ ⊥ then
            for τ_i ∈ A_c do
                set_prio(τ_i, P̄_i^c);
            end
        end
        c ← M(j);
        for τ_i ∈ A_c do
            set_prio(τ_i, P̄_i^c + P_idle);
        end
        sleep(l_j);
    end
end
```

Fig. 2 illustrates the execution of two partitions $A_1 = \{\tau_1^1, \tau_2^1\}$ and $A_2 = \{\tau_1^2, \tau_2^2\}$ with $\overline{P}_1^1 = \overline{P}_1^2 > \overline{P}_2^1 = \overline{P}_2^2$, using the scheduler described in Algorithm 1. Both partitions are assigned a time slot of equal length T. The execution of all tasks involved is displayed in Fig. 2a and the priorities of τ_{sched}, τ_{idle}, and the tasks of A_1 are shown in Fig. 2b. Partition boundaries t_0, t_2, t_5 are shown as vertical red lines. The execution duration of τ_{sched} is exaggerated for clarity. At t_0, τ_{sched} increases the priority of the idle task from P_{min} to P_{idle}. Until t_1, τ_1^1 and τ_2^1 retain their nominal priorities of $\overline{P}_1^1$ and $\overline{P}_2^1$, respectively. At t_1, τ_{sched} promotes both by adding P_{idle} to their priorities. In reality, this does not happen in an instant, but since the priority of τ_{sched} is higher than that of any other task, they can only start their execution after τ_{sched} is finished. τ_1^1 and τ_2^1 are executed between t_1 and t_2. At t_2, τ_2^1 is interrupted by the partition boundary and τ_{sched} demotes both tasks of A_1 by resetting their priorities to their nominal values, and promotes τ_1^2 and τ_2^2. The minor timeframe for A_2 runs between t_3 and t_5. Since τ_2^2 finishes its execution at t_4, τ_{idle} is executed for the remaining time in this time slot, even though τ_2^1 is waiting to be executed. τ_2^1 resumes its execution in the next partition window, since the second job of τ_1^1 has not yet arrived. The promotion and demotion repeat at every partition boundary.

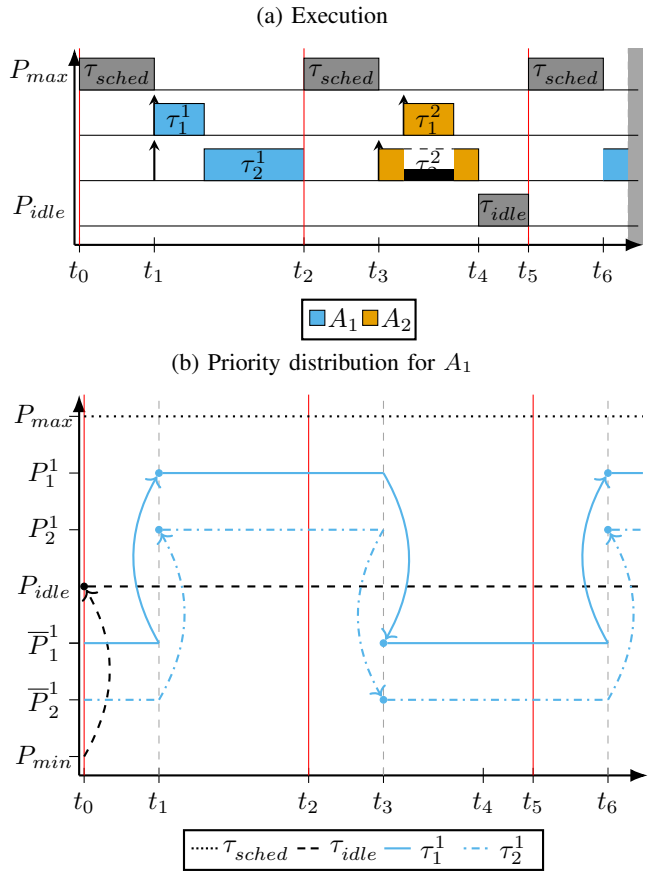


Fig. 2: Example of ARINC 653-like scheduling using priority-based scheduling.

A. Implementation Details for FreeRTOS and RIOT OS

To ensure independence from the underlying operating system, we introduce an abstraction layer between the components needed for the framework and their actual implementation in the operating system. For the OS-specific realization of TiM2S, we only need to instantiate wrapper functions to the actual system calls adapting the tasks' priorities and configure the OS accordingly, i.e., setting the number of priority levels and the tick rate, if any. No changes to the OS implementations are required.

We implemented TiM2S for FreeRTOS and RIOT OS. Both are popular, open-source real-time operating systems and fulfill the requirements laid out in the previous section. While FreeRTOS features a tick-based scheduler that runs in a fixed period, the RIOT OS scheduler is inherently tickless, meaning it only runs when an event occurs that might require the rescheduling of tasks. This could be an event a task is waiting for, the creation of a new task, etc. In this section, we discuss relevant details of implementing TiM2S for these operating systems, creating a guideline for porting TiM2S to other real-time operating systems.

The first step of implementing the wrappers for the task and mutex data structures as well as functions such as modifying the priority of a task, sleeping for a specific time, etc., is straightforward, since the framework data structures are designed not to leak any OS details to the applications. Since in RIOT OS, a value of 0 corresponds to the highest possible priority, whereas 0 in FreeRTOS represents the lowest priority, we introduced a priority abstraction that must be translated for the target operating system. To implement TiM2S, the priority of the idle task must be adjusted. While FreeRTOS natively provides a function to obtain the handle of the idle task, the idle thread in RIOT OS must first be identified by its name.

FreeRTOS distinguishes between binary semaphores[5] and mutexes[6]. The relevant difference here is that binary semaphores do not entail any kind of priority inheritance mechanism, while mutexes use the priority inheritance protocol. Hence, TiM2S restricts the use of FreeRTOS mutexes, while binary semaphores are supported. RIOT OS provides optional mitigation against priority inversion, which is disabled by default[7] and also restricted by TiM2S.

IV. Framework Evaluation

We used an ESP32 Pico Kit 1 V1.0 for the evaluation, where execution was limited to the first core only. Both RIOT OS and FreeRTOS support this board.

RIOT OS offers native tracing that can be buffered and output all at once when needed. Because the output requires significant computing resources, this type of tracing is preferred: After a desired number of major timeframes have been executed, the collected log is reported. Since FreeRTOS natively does not provide a log buffer, we adopted the RIOT OS tracing for the FreeRTOS implementation of the framework. Each trace contains the current time since system start in μs in addition to an integer used as a binary mask for logging purposes. To obtain comparable results, the log entries are generated in the operating-system-independent part of the framework before and after the execution chunk is run when measuring execution times, and at the start and end of each iteration of the outer for-loop in Algorithm 1 when measuring the minor timeframe durations.

For the framework overhead, we logged the difference in CPU cycles before and after a combination of partition demotion and promotion. After flashing the application with the current task set to the microcontroller, the system is reset, and the measurements are transferred via USB through the board's serial interface after the test has run.

The mutexes listed in Table II and Table IV provide mutual exclusion without priority inheritance: under FreeRTOS they are realized as binary semaphores, and under RIOT OS the optional mitigation against priority inversion remains disabled.

A. Evaluation of core functionalities

To demonstrate the capabilities of TiM2S in a comprehensible manner, we analyze a custom task set to display

[5]https://freertos.org/Documentation/02-Kernel/02-Kernel-features/02-Queues-mutexes-and-semaphores/02-Binary-semaphores

[6]https://freertos.org/Documentation/02-Kernel/02-Kernel-features/02-Queues-mutexes-and-semaphores/04-Mutexes

[7]https://api.riot-os.org/group__core__sync__mutex.html

TABLE II: Custom task set Γ_{cf} to evaluate core functionalities.

A	τ	$T\,[ms]$	$O\,[ms]$	Prio	$C\,[ms]$	Mutex
1	τ_1	10	1	2	2	$m_1[1\ ms]$
	τ_2	20	0	1	5	$m_1[2\ ms]$
2	τ_1	10	0	2	2	–

the temporal separation as well as the correctness of the preemption logic and mutexes. The task set is displayed in Table II. The partitions are executed in the specified order for 5 ms each. Higher priority values indicate higher priority, and tasks are numbered in order of decreasing priority, i.e., $i < j$ implies $prio\,(\tau_i) > prio\,(\tau_j)$. The time given in brackets next to the mutex is the duration of the corresponding critical section.

For τ_1^1, the critical section is at the end, while for τ_2^1, it is at the beginning of the respective execution time.

The task set consists of two partitions A_1 with two and A_2 with a single task. The major timeframe has a total duration of 20 ms and consists of 4 partition windows of 5 ms, alternating between A_1 and A_2. The theoretical execution is displayed in Fig. 3a. The first partition window of A_1 starts at $t = 0\ ms$. Due to the offset of τ_1^1, τ_2^1 should execute first for 1 ms before being interrupted by τ_1^1, since the first chunk of τ_1^1 does not require the mutex m_1. Then τ_2^1 is allowed to continue while τ_1^1 waits for the lock on m_1 to be released. After the first job of τ_1^1 is done, τ_2^1 continues its execution at $t = 4\ ms$ and is interrupted by the partition boundary at $t = 5\ ms$. Jumping forward to the second partition window of A_1 at $t = 10\ ms$, τ_2^1 has another 1 ms of execution time prior to the arrival of the second job of τ_1^1 at $t = 11\ ms$. This job of τ_1^1 finishes its execution completely before τ_2^1 completes its first job at $t = 14\ ms$. For the last 1 ms of this partition window, τ_{idle} is executed. Regarding A_2, τ_1^2 starts and finishes its execution at the start of the according partition window. Even though the processor is idle between $t = 7\ ms$ and $t = 10\ ms$ and τ_2^1 is ready to be executed, it cannot run during the partition window of A_2, leaving the idle task to be executed.

Examining the execution of Γ_{cf} under FreeRTOS displayed in Fig. 3b, there is a small gap of 0.15 ms before the execution of τ_2^1. We assume this is due to the initialization of τ_1^1 at the start of the partition and thus choose to ignore it. In the second minor timeframe of A_1, τ_2^1 finishes later than expected. This can be explained by the aforementioned overhead and small fluctuations in the durations of the minor timeframes. A_2 is executed as expected. The RIOT OS implementation also behaves as expected for A_2. While the small 0.18 ms gap before executing τ_2^1 can be explained in the same way for RIOT OS, the minor timeframe durations vary significantly. This results in the interruption of τ_2^1 at $t = 14.77\ ms$ before it can finish its first job.

To analyze the distribution of minor timeframes, we ran 250 major timeframes for a total of 1000 minor timeframes and measured the duration of each partition window. The distribution of deviation in minor timeframe duration for

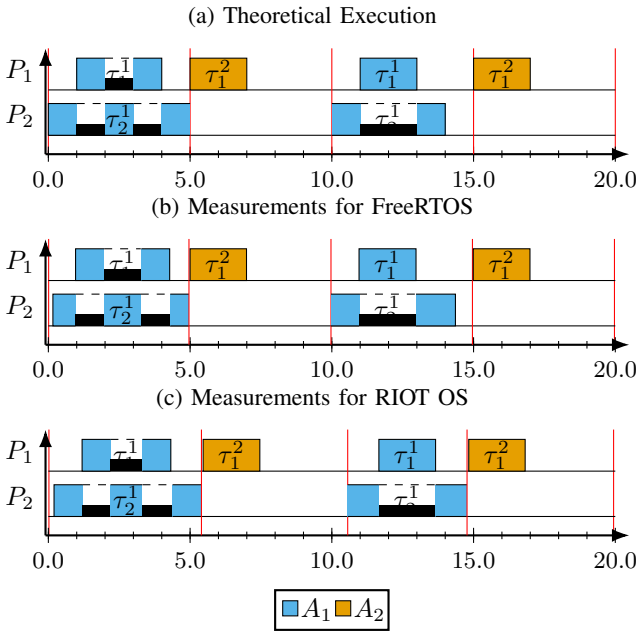
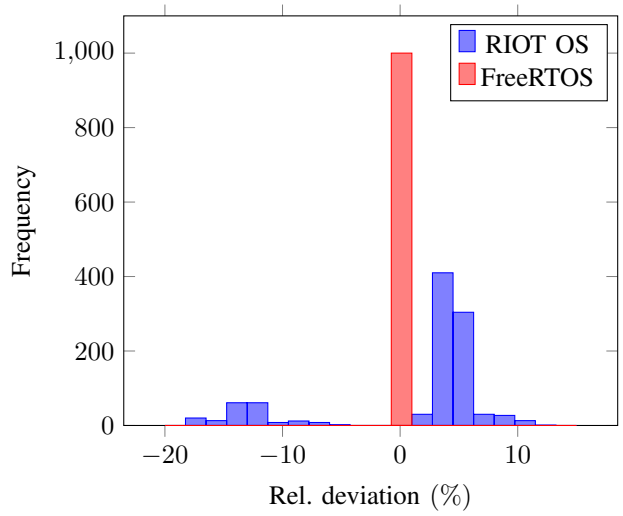


Fig. 3: The first two major timeframes of Γ_{cf}.

FreeRTOS and RIOT OS is displayed in Fig. 4a with summary statistics in Table III. For RIOT OS, no significant difference was observed by excluding the first minor timeframe. To analyze whether the observed deviations increase with longer timeframes, we conducted additional measurements with all time-related parameters of Γ_{cf} scaled by a factor of 10, leading to an expected minor timeframe duration of 50 ms. The distribution of deviation in minor timeframe duration for the scaled task set is displayed in Fig. 4b with summary statistics in Table III. Since the first minor timeframe was a significant outlier for FreeRTOS for both the unscaled and the scaled task set, we attributed this to the system's startup period and excluded it from both evaluations. Comparing both figures, it is clear that the fluctuations do not scale with minor timeframe duration for either operating system implementation. Even though the expected minor timeframe duration is increased by a factor of 10, the deviations remain in the same order of magnitude, resulting in a decrease in relative deviation by an order of magnitude. This is particularly noticeable with RIOT OS. The stark difference in distributions between FreeRTOS and RIOT OS, particularly with the original task set, can be explained by the nature of both operating systems' schedulers: FreeRTOS features a tick-based scheduler with a tick size set to 1 ms. This combines very well with the desired minor timeframe durations of 5 ms or 50 ms. In contrast, RIOT OS uses a tickless scheduler, which makes timer deviations on this timescale visible. Since the absolute deviation does not increase when scaling the task set, and since the priority updates performed at a partition boundary can only prolong a time slot but never shorten it, we attribute the fluctuation in minor timeframe duration to the timer facility of the operating system rather than to our framework.

(a) Original task set



(b) Scaled task set

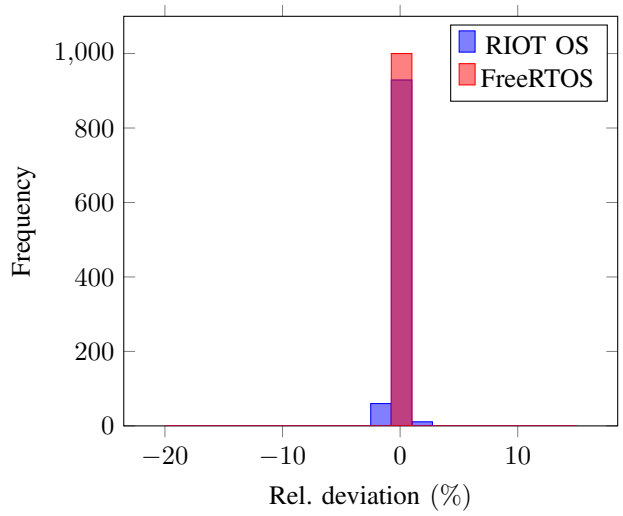


Fig. 4: Distribution of relative deviations of minor timeframe durations running the original and scaled version of Γ_{cf} for FreeRTOS and RIOT OS with $n = 999$. The expected minor timeframe duration is $5\ ms$ and $50\ ms$ for (a) and (b), respectively.

TABLE III: Distribution of minor timeframe durations running Γ_{cf} for FreeRTOS and RIOT OS with $n = 999$. The task set was executed for 1000 minor timeframes.

Minor time-frame duration	OS	Deviation $[ms]$	$\bar{x}[\%]$	$\sigma[\mu s]$
$5\ ms$	RIOT OS	$[-0.881, 0.629]$	101.21	348.48
	FreeRTOS	$[-0.006, 0.001]$	99.98	1.59
$50\ ms$	RIOT OS	$[-0.885, 1.097]$	100.24	252.35
	FreeRTOS	$[-0.006, 0.001]$	100.00	1.59

B. Evaluation based on a case study

After validating the framework's core functionalities in the previous section, we now examine how they interact using a complex real-world case study. For this purpose, we use the "Case Study with Interapplication Communications" from Carnevali et al. [12]. However, we make two adjustments. First, we ignore communication between partitions because the necessary behavior is not relevant to our framework. Second, the execution times of the original task set are non-deterministic. To achieve comparable measurements for both operating systems, we use the mean of each execution time interval as a deterministic value. Repeating the experiments with randomized execution times led to the same results. The resulting task set is shown in Table IV. The partitions are executed in the specified order for $5\ ms$ each. Sporadic tasks have a minimum interarrival time as their period, signified by $\geq$. Higher priority values indicate higher priority, and tasks are numbered in order of decreasing priority, i.e. $i < j$ implies $prio(\tau_i) > prio(\tau_j)$. The time given in brackets next to the mutex is the duration of the corresponding critical section.

TABLE IV: Case study based on [12].

A	τ	$T\,[ms]$	$O\,[ms]$	$J\,[ms]$	Prio	$C\,[ms]$	Mutex
1	τ_1	25	2	0	5	1.2	
	τ_2	50	3	0	4	0.2	
	τ_3	50	3	0	3	3.5	
	τ_4	50	0	0	2	0.2	$m_1[0.2\ ms]$
	τ_5	≥ 120	0	0	1	1	$m_1[0.2\ ms]$
2	τ_1	5	0	0.5	4	2.5	
	τ_2	50	2	0	3	0.9	
	τ_3	100	0	0	2	0.2	$m_2[0.2\ ms]$
	τ_4	≥ 100	10	0	1	1.4	$m_2[0.3\ ms]$
3	τ_1	25	0	0.5	4	0.7	
	τ_2	50	0	0	3	0.9	
	τ_3	50	0	0	2	1.3	
	τ_4	≥ 100	11	0	1	1.1	
4	τ_1	25	3	0.2	5	1	
	τ_2	50	5	0	4	1.6	
	τ_3	50	25	0	3	0.2	
	τ_4	100	11	0	2	0.9	
	τ_5	200	13	0	1	4.8	
5	τ_1	50	0	0.3	4	0.9	
	τ_2	50	2	0	3	1.6	
	τ_3	200	0	0	2	0.8	$m_5[0.3\ ms]$
	τ_4	≥ 200	14	0	1	2	$m_5[0.2\ ms]$

There are some notable features in the task properties. Due to the given offsets, tasks τ_4^2 and τ_3^4 are not ready to start in the first minor timeframe of their respective partitions. Other offsets, such as those of τ_2^2 or τ_1^4, are irrelevant because their respective partitions are not active before the offset expires. Since no task in the first partition has $J > 0$, and J does not exceed the length of the first minor timeframe for any affected task, J does not actually play a role for any task in our observations. Since neither J nor a second job of a sporadic task is relevant in the first major timeframe, the theoretical behavior is deterministic and displayed in Fig. 5a.

When comparing the execution for FreeRTOS in Fig. 5b and for RIOT OS in Fig. 5c, the gap before the execution of the first task is again visible for both operating systems at $0.18\ ms$ and $0.3\ ms$, respectively. In contrast to the theoretical execution, τ_4^2 is executed in the first major timeframe for RIOT OS. We assume this is related to the shifted minor timeframe durations. Looking at A_5 for RIOT OS, the end of the minor timeframe is delayed so much that τ_4^5 can finish executing in this minor

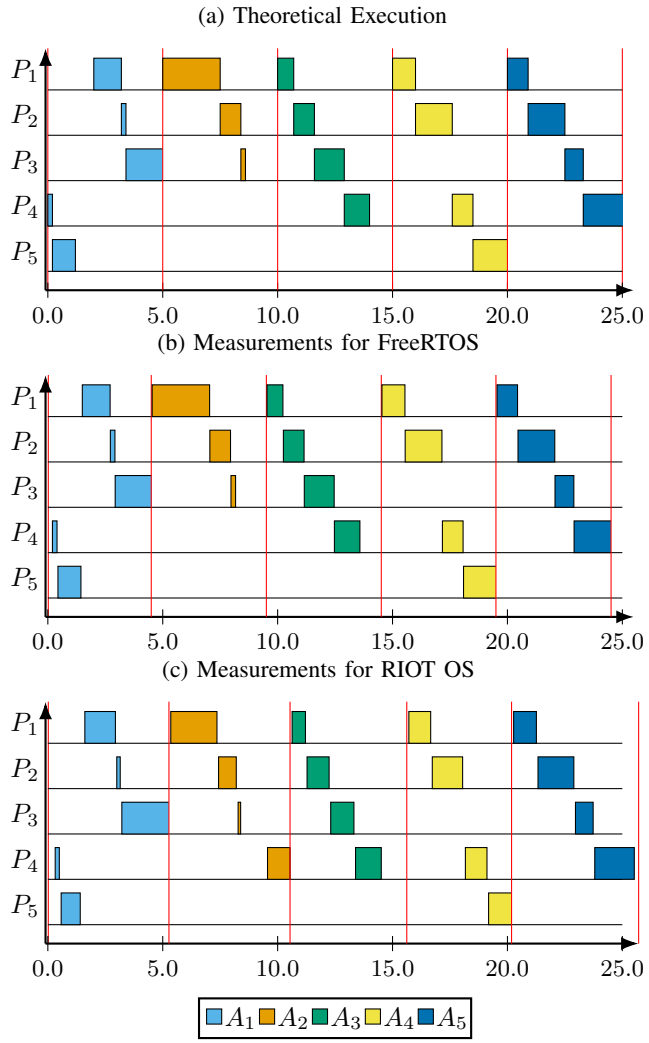


Fig. 5: First major timeframes of the case study from [12].

timeframe. These deviations can both be explained by the fluctuation of minor timeframe duration. The distribution of minor timeframe durations is very similar to the one discussed in subsection IV-A and will not be discussed any further.

For the case study, we also analyzed the overhead of TiM2S and measured the CPU cycle count before and after demoting and promoting the previous and next task sets. A fair comparison to the execution of the case study without our framework is hardly possible, as table-based execution of partitions cannot be cleanly mapped to the default FPPS of FreeRTOS or RIOT OS. Therefore, the distribution of relative overhead is displayed in Fig. 6 with summary statistics in Table V. Even though both operating systems use the same implementation of our framework scheduler, the underlying data structures used for the ready queues seem to significantly impact the overhead when changing task priorities. However, this topic will not be discussed further.

We conducted additional overhead measurements using custom task sets consisting of between 2 and 90 tasks split evenly between two partitions with alternating minor timeframes. At

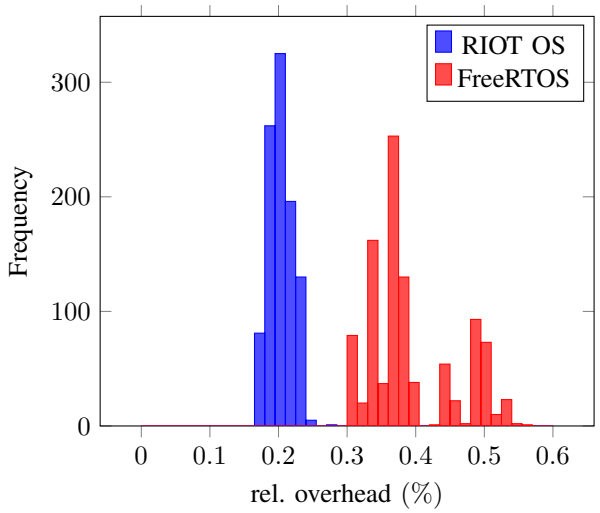


Fig. 6: Distribution of TiM2S overhead for FreeRTOS and RIOT OS executing the case study [12]. Values are relative to the expected minor timeframe duration of 5 ms.

TABLE V: Distribution of relative overhead for TiM2S executing the case study [12] on FreeRTOS and RIOT OS. The values are given relative to the minor timeframe duration of 5 ms.

OS	Range [%]	$\bar{x}$[%]	σ [pp]
RIOT OS	$[0.17, 0.28]$	0.20	0.016
FreeRTOS	$[0.31, 0.56]$	0.39	0.06

every partition boundary, half of the tasks are demoted and half are promoted. We chose 90 as the upper limit for this analysis due to the limited memory available even when using the smallest feasible task stack size. For each task set, we measured 500 runs for a total of 999 partition boundaries where all n task priorities must be adapted. As expected, the framework's CPU clock cycle overhead increases linearly with the number of tasks. For FreeRTOS, the average clock cycle count increases from 1065 for $n = 2$ to 84876 for $n = 90$, and for RIOT OS, it increases from 286 for $n = 2$ to 17132 for $n = 90$. At 240 MHz, 84876 cycles correspond to roughly 7 % of a 5 ms partition window. The implementation overhead stays below 0.6 % of the minor timeframe in our case study. However, systems combining large partitions with short partition windows or lower clock frequencies must budget this cost explicitly.

V. Related Work

As mentioned in the introduction, the set of available scheduling algorithms in most open-source real-time operating systems is limited. Most open-source RTOSes only provide native support for fixed-priority preemptive scheduling. FIFO is often implicitly implemented as a tie-breaker in case of equal priorities. Consequently, various research exists that aims to integrate different real-time scheduling algorithms within open-source RTOSes. Inam et al. [15] have presented support for hierarchical scheduling with deferred and idle

servers in FreeRTOS and evaluated their approach on the AVR 32-bit board EVK1100. Buttazzo and Gai [16] have integrated EDF scheduling within ERIKA OS, and Taji et al. [11] have integrated it within FreeRTOS.

Several attempts have been made to implement the required functionality of ARINC 653, the inspiration for TiM2S, on top of existing operating systems or standards. Santos et al. [17] implement the corresponding Application Programming Interface (API) on top of POSIX, and Han and Jin [18], [19] realize the hierarchical scheduling of ARINC 653 in Linux through virtualization. An implementation of hierarchical scheduling according to ARINC 653 within VxWorks can be found in [9] and [20]. Jo et al. [21] present an implementation of the temporal partitioning of ARINC 653 within Linux and RTEMS, along with a case study of a civil quadcopter. Easwaran et al. [22] and Han et al. [23] offer analyses of ARINC 653 partition scheduling as well as corresponding heuristic optimization approaches.

In stark contrast to prior work, our framework TiM2S is largely non-intrusive and does not require any error-prone changes to the operating system's internals. TiM2S is thus robust against updates to the underlying RTOS, assuming backward compatibility, and can easily be instantiated to other RTOSes. Unless actively supported, prior efforts to integrate a scheduling paradigm within existing open-source RTOSes are likely to be deprecated within a few releases.

VI. Conclusion

In this paper, we have introduced TiM2S, a framework that enables partitioned, hierarchical scheduling similar to ARINC 653 scheduling using priority-based, preemptive scheduling available in most open-source RTOSes. In contrast to prior work, our framework is largely non-intrusive and does not require any changes to the RTOS itself. We instantiated TiM2S for FreeRTOS and RIOT OS and evaluated it using a custom task set and a case study. The execution of both the custom task set as well as the case study behaved as expected: temporal isolation was provided while preserving mutex-based task interaction within partitions. Our evaluation showed that TiM2S is particularly suited for tick-based operating systems such as FreeRTOS and is able to provide the desired hierarchical scheduling behavior with only minor deviations. With a tickless RTOS such as RIOT OS, the maximum deviations were significant for a $5\ ms$ window, but could be attributed to the timer resolution of the operating system rather than to the framework. Relative overheads for both implementations are below $0.6\ \%$ and increase linearly with the number of tasks as expected.

References

[1] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, Oct. 2011. [Online]. Available: https://doi.org/10.1145/1978802.1978814

[2] G. C. Buttazzo, *Hard Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications, 4th Edition*. Springer, 2024. [Online]. Available: https://doi.org/10.1007/978-3-031-45410-3

[3] A. Minaeva and Z. Hanzálek, "Survey on Periodic Scheduling for Time-triggered Hard Real-time Systems," *ACM Comput. Surv.*, vol. 54, no. 1, Mar. 2021. [Online]. Available: https://doi.org/10.1145/3431232

[4] G. C. Buttazzo, "Rate monotonic vs. EDF: judgment day," *Real-Time Syst.*, vol. 29, no. 1, p. 5–26, Jan. 2005. [Online]. Available: https://doi.org/10.1023/B:TIME.0000048932.30002.d9

[5] S. Baruah, "The Limited-Preemption Uniprocessor Scheduling of Sporadic Task Systems," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, ser. ECRTS '05. USA: IEEE Computer Society, 2005, p. 137–144. [Online]. Available: https://doi.org/10.1109/ECRTS.2005.32

[6] R. J. Bril, M. van den Heuvel, U. Keskin, and J. Lukkien, "Generalized fixed-priority scheduling with limited preemptions," in *24th ECRTS*, July 2012. [Online]. Available: http://www.es.mdu.se/publications/3347-

[7] AEEC, "ARINC Specification 653P1-2, Version Supplement 2," Aeronautical Radio, Inc., Standard, 3 2006.

[8] ISO, "Road vehicles - Open interface for embedded automotive applications - Part 3: OSEK/VDX Operating System (OS)," International Organization for Standardization, Standard ISO 17356-3:2005, 2005.

[9] W. Ruan and Z. Zhai, "Kernel-Level Design to Support Partitioning and Hierarchical Real-Time Scheduling of ARINC 653 for VxWorks," in *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, 2014, pp. 388–393.

[10] eSOL Co., Ltd., "eMCOS AUTOSAR, an AUTOSAR-Compliant Scalable RTOS, Receives ISO 26262 ASIL-D Certification," 2018, product announcement / technical release.

[11] H. Taji, J. Miranda, M. Pełoń-Quirós, S. Balasi, and D. Atienza, "Dynamic Scheduling for Event-Driven Embedded Industrial Applications," in *2023 IFIP/IEEE 31st International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE, 2023, pp. 1–6.

[12] L. Carnevali, A. Pinzuti, and E. Vicario, "Compositional Verification for Hierarchical Scheduling of Real-Time Systems," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 638–657, 2013.

[13] R. Barry, *FreeRTOS reference manual: API functions and configuration options*. Real Time Engineers Limited, 2009.

[14] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, 2018.

[15] R. Inam, J. Mäki-Turja, M. Sjödin, and M. Behnam, "Hard real-time support for hierarchical scheduling in FreeRTOS," in *23rd Euromicro Conference on Real-Time Systems*, 2011, pp. 51–60.

[16] G. C. Buttazzo and P. Gai, "Efficient EDF Implementation for Small Embedded Systems," in *Proceedings of the 2nd Workshop on Operating System Platforms for Embedded Real-Time Applications (OSPERT 2006)*, Dresden, Germany, July 2006, pp. 1–6.

[17] S. Santos, J. Rufino, T. Schoofs, C. Tatibana, and J. Windsor, "A portable ARINC 653 standard interface," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, 2008, pp. 1.E.2–1–1.E.2–7.

[18] S. Han and H.-W. Jin, "Full virtualization based ARINC 653 partitioning," in *2011 IEEE/AIAA 30th Digital Avionics Systems Conference*, 2011, pp. 7E1–1–7E1–11.

[19] ——, "Kernel-level ARINC 653 partitioning for Linux," in *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, ser. SAC '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 1632–1637. [Online]. Available: https://doi.org/10.1145/2245276.2232037

[20] M. Behnam, T. Nolte, I. Shin, M. Åsberg, and R. J. Bril, "Towards Hierarchical Scheduling on top of VxWorks," in *Proceedings of the 4th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT08)*, July 2008, pp. 63–72. [Online]. Available: http://www.es.mdu.se/publications/1261-

[21] H.-C. Jo, J.-K. Park, H.-W. Jin, H.-S. Yoon, and S. H. Lee, "Portable and Configurable Implementation of ARINC-653 Temporal Partitioning for Small Civilian UAVs," *IEEE Access*, vol. 7, pp. 142 478–142 487, 2019.

[22] A. Easwaran, I. Lee, O. Sokolsky, and S. Vestal, "A compositional framework for avionics (ARINC-653) systems," *University of Pennsylvania, Tech. Rep. MS–CIS–09–04*, 2009. [Online]. Available: https://repository.upenn.edu/handle/20.500.14332/7863

[23] P. Han, Z. Zhai, B. Nielsen, and U. Nyman, "Model-based optimization of ARINC-653 partition scheduling," *International Journal on Software Tools for Technology Transfer*, vol. 23, no. 5, pp. 721–740, Feb. 2021.