Transforming an Ada Ravenscar Real-Time application into an equivalent system in Rust

NICOLA RAVAGNAN, Department of Mathematics, University of Padua, Italy

This report includes the result of the experiment of re-implementing the extended example application from *Guide for the use of the Ada Ravenscar Profile in high integrity systems*, an application that is designed to illustrate the expressive power of the Ravenscar Profile to facilitate offline scheduling analysis, with another programming language, like Rust the chosen language for this experiment, and the Real-time Interrupt-driven Concurrency (RTIC) framework, the hardware accelerated Rust RTOS. The report will show how tasks and resources have been adapted to the RTIC framework. Then the resulting system was put on analysis with Cheddar GPL, a real-time scheduling simulator, using a Deadline Monotonic scheduling algorithm, and then the results were compared with the previously obtained results from the technical report *MAST Analysis of a Ravenscar precedence-constrained application with FPS and EDF scheduling*, considering only the Fixed Priority Scheduling variant of the model. The models tested were a simple model with independent tasks and then a model with time offsets on the tasks to reproduce the precedence constraints in the application, the results found that the Rust application had lower response times than the Ada application, however, the results were influenced by many factors such as the scheduling analysis tool used and the microcontroller unit used to run the application. To obtain the performance indicators needed for the analysis, the implemented Rust application was run on a *STM32F303VC* microcontroller unit.

CCS Concepts: • **General and reference** → **Evaluation**; • **Computer systems organization** → **Embedded software**; **Real-time operating systems**.

Additional Key Words and Phrases: Embedded programming, Real-time systems, Rust, RTIC, Ravenscar, MAST, Scheduling Analysis

ACM Reference Format:
Nicola Ravagnan. 2024. Transforming an Ada Ravenscar Real-Time application into an equivalent system in Rust. 1, 1 (November 2024), 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Embedded systems are becoming more diffused as IoT and mobile devices are more common around the world. As more devices are available for development, we should be seeking more programming languages and paradigms to use for coding on embedded devices, developers should be willing to try different programming languages on new or previous systems and test the developed systems to compare and find the best framework for the required use case or to increase the range of programming languages supported.

This report considers the extended example application using the Ravenscar profile presented in [12], and I had to re-implement said application with another programming language well suited for embedded programming such as Rust, then I compared the newly implemented application with the previous one in terms of performance indicators, and empirical evaluation.

Author's Contact Information: Nicola Ravagnan, nicola.ravagnan.2@studenti.unipd.it, Department of Mathematics, University of Padua, Padua, Italy.

Manuscript submitted to ACM

For the programming of the application I decided to use Rust: Rust is a low-level programming language with no runtime or garbage collector that focuses on thread and memory safety, making it a good candidate for embedded programming, also with Rust, I decided to the *Real-time Interrupt-driven Concurrency*[5][21] framework (or RTIC), a concurrency framework for building real-time systems with Rust, inspired from the RTFM (Real-time for the masses) SRP based scheduler [13].

For the analysis I have decided to run the application on bare-metal using a *STM32F303VC* micro-controller unit (or MCU), a cheap device that comes with an Arm Cortex M4 core and 256 Kilo bytes of flash memory, it's worth saying that the application has also been tested on a QEMU virtual machine that emulated the *LM3S6965EVB*, a board that uses an ARM Cortex M3 core, however, I will not cover it this paper. The hosts used to flash and test the application on the MCU are Windows 11 and Fedora Linux 40.

In this paper, I analyzed the new Rust implementation in terms of performance indicators and empirical evaluation, then compared the results of the *Fixed Priority Scheduling* variant of the Ada extended application example found by previous students [15].

2 Related Work

The Ravenscar Profile is a subset of the Ada tasking features designed for safety-critical hard real-time computing. It was defined by a separate technical report in Ada 95; it is now part of the Ada 2012 Standard. The purpose of the Ravenscar profile is to restrict the use of many tasking facilities so that the effect of the program is predictable, programs coded with the Ravenscar Profile result be easier to analyze with offline scheduling and static analysis methods.

In the research, I considered the *extended example application* coded in Ada, in the *Guide for the use of the Ada Ravenscar Profile in high integrity systems*, written in Ada that uses the Ravenscar profile.

The application example uses all of the concurrency components permitted by the Ravenscar Profile. The structure of the example models, on a reduced and simplified scale, is the operation of real-world embedded real-time systems.

The original example application is composed by:

- A Regular producer task: a cyclic task that carries out work at a fixed periodic rate;
- An On Call Producer task: a sporadic task triggered by the Regular Producer invoked to take over its excess workload;
- A Request Buffer: A shared buffer where the On Call Producer waits on this queue for incoming requests from the Regular Producer; at first when On Call Producer starts, it will wait on an empty queue until the top of the queue is filled;
- An Event Queue that keeps track of requests incoming from an external interrupt, then the request from the external interrupt is carried out by the External Event Server task;
- An External Event Server interrupt-sporadic task that is triggered by the signal function of the Event Queue, the External Event Server then writes on the Activation Log;
- An Activation Log protected object that contains the latest log of the incoming external interrupt;
- An Activation Log Reader that reads the latest log entry of the Activation Log.

This application was then put in analysis by testing the application running on the `ravenscar-full-stm32f429disco` runtime and running on the *STM32F429I-Discovery* micro-controller device, the system has been tested with two scheduling rules, Fixed Priority Scheduling, and Earliest Deadline First, by building a MAST model and running a scheduling simulation [15].

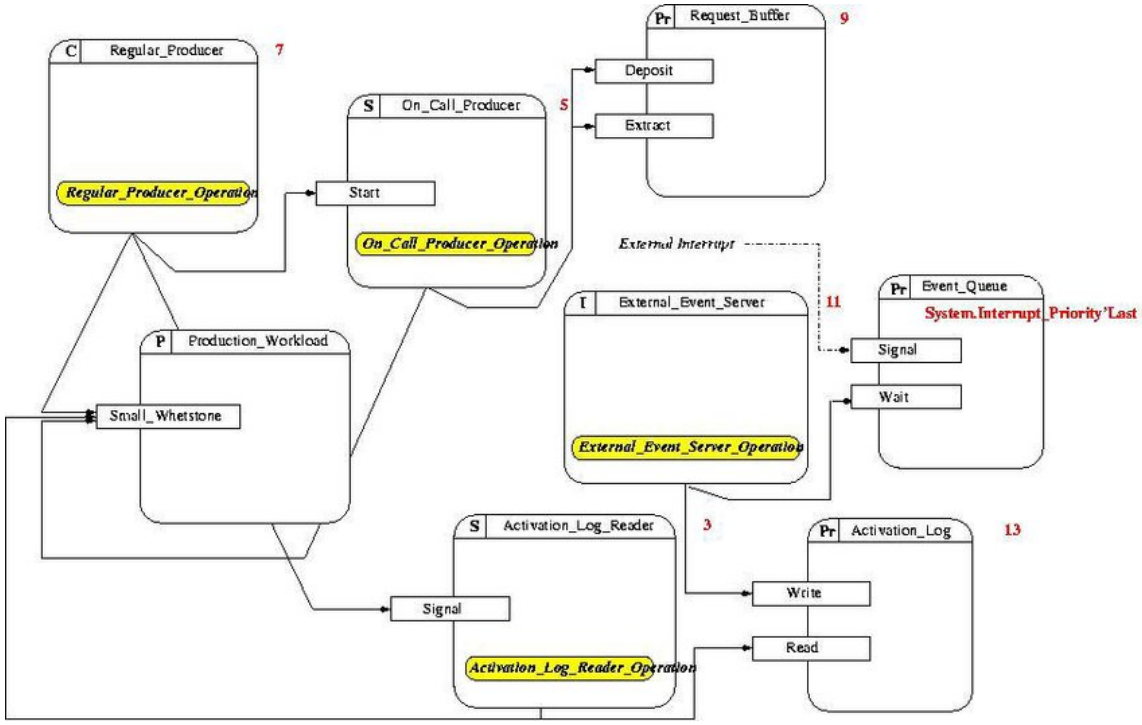

Fig. 1. Schematic architecture of the example Ravenscar application.

3 Motivation

The original Ravenscar Profile has been made to begin a subset of the tasking model, made to meet hard real-time system requirements for determinism constructs, schedulability analysis, and memory-boundness, as well as begin suitable for mapping to a small and efficient run-time system while supporting task synchronization and communication. This allowed programs made with the Ravenscar profile to be analyzed easily using static methods.

This report aims to see the functionality of the Ravenscar profile from another point of view using different but similar tools and methods, trying to mimic as closely as possible the implementation of the original application and the analysis approach done previously, the reason for this is to be able to compare two different programming languages and frameworks in terms of performance indicators and empirical evaluation.

In this report, I recreated the extended application example mentioned before using Rust and analyzed the resulting system with Cheddar GPL[20], a real-time scheduling simulator, using performance indicators such as worst-case execution time, worst-case response time, and processor utilization with a Deadline Monotonic scheduling protocol [9]. The results were compared with the Rate Monotonic [11] Fixed Priority Scheduling [16] results found on the original extended example application found previously.

4 Technical choices made

4.1 Response Time Analysis

In fixed priority scheduling tasks have a fixed priority value determined off-line, and the ready tasks are dispatched to execution in the order determined by their static priority. With priority-based scheduling, a high-priority task may be released during the execution of a lower-priority one, this can have two different effects if the system in question permits preemption.

The Response Time Analysis [8] is a good tool to define if a system is feasible or not, it's a necessary and sufficient test which means if it passes all tasks will meet their deadlines, otherwise, some task will miss their deadlines. to get the response time of a task, one (that may be a program) can use the following equation

$$R_i = C_i + \sum_{j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil C_j$$

Where R_i is the response time of the task i, C_i is the computation time for the task i and the sum term is the inference term caused by tasks that have higher priority than i.

To solve this equation, we solve the recurrence relationship:

$$W_i^{n+1} = C_i + \sum_{j \in hp(i)} \lceil \frac{w_i^n}{T_j} \rceil C_j$$

The set of values $W_i^0 ... W_i^n$ is monotonically non-decreasing. The solution of the equation is found when $W_i^n = W_i^{n+1}$ and if $R_i < T_i$, where T_i is the period of the task, then the task will finish before it's deadline, this is valid for Rate Monotonic scheduling. Response Time Analysis works well also for Deadline Monotonic scheduling, the stopping criterion becomes $W_i^{n+1} > D_i$.

4.2 The Rust programming language

Rust[4] is a low-level programming language that focuses on performance and memory safety, released in 2015 by Mozilla, with no runtime or garbage collector, it can power performance-critical services, and integrate with other languages such as C. Rust is also a good candidate for embedded systems when used with the `#[no_std]` attribute which makes the Rust application use platform-agnostic libraries and not the platform dependant standard library, this permits for any kind of bootstrapping (stage 0) code like bootloaders, firmware or kernels. Rust features include:

- Rust applications are free of mutable aliasing;
- Execution will always have defined behavior.

This is achieved by putting restrictions on the use of mutable references and raw pointers, so Rust provides two *modes*: safe and unsafe. Safe mode is the *normal* one, in which most Rust is written. In unsafe mode, the developer is responsible for the code's memory safety, which is used by developers for cases where the compiler is too restrictive. Usually, a developer writes most of the code in safe mode, and unsafe code is usually warped in safe functions with safety guarantees.

4.3 RTIC

RTIC [5] is a concurrency framework for building real-time systems using Rust, I used it for this project because it provides a common abstraction for tasks with priority levels and message passing between tasks with a Stack Resource Policy (SRP) based concurrency that extends the Priority Inheritance Protocols giving guarantees to single core scheduling such as:

- Preemptive deadlock and race-free scheduling;

- Resource efficiency;
- Predictable scheduling, with bounded priority inversion by a single (named) critical section;
- theoretical underpinning amenable to static analysis (e.g., for task response times and overall schedulability).

RTIC also manages shared resources using an immediate Priority Ceiling Protocol (IPCP). The Ceiling Priority Protocol [19] is applied when a task acquires a shared resource its priority is temporarily raised during its execution in the critical section of the shared resource if a higher-priority task requires access to that shared resource. IPCP is a variant of the original Ceiling Priority Protocol, in which the task's priority is immediately raised after acquiring the shared resource. This protocol complies with the SRP-based scheduling of RTIC. The Stack Resource Policy [10] is a refinement of the Priority Ceiling Protocols, which states that a job execution request to be blocked from starting execution (i.e. from receiving its initial stack allocation) until its ceiling is higher than the current system ceiling, as the maximum of the preemption level of the current job and the current ceilings of all the resources: as an example, consider two tasks A and B with static priorities $A = 4$ and $B = 2$ both accessing a shared resource R, the shared resource R will have ceiling priority of $\pi(R) = max(p(A) = 2, p(B) = 4) = 4$. The SRP model fully prevents deadlocks and data races while placing a tight bound on priority inversion.

4.4 Real-Time Transfer and debugging

Semihosting is a mechanism that lets embedded devices do I/O on the host and is mainly used to log messages to the host console. Semihosting requires only an active debug session. The downside is that it's prolonged: each write operation can take several milliseconds, depending on the hardware debugger (in my case, ST-Link) used[1].

A solution is to use a real-time transfer framework such as `defmt` [7], a highly efficient logging framework that targets resource-constrained devices, such as micro-controllers, `defmt` can be used for logging information through text, such as execution times, deadline misses and so on.

To work with the debugger, I used Probe-rs [6], which is a Rust-focused software designed to work with debuggers in embedded systems. Probe-rs is built with simplicity in mind and reduces the amount of configuration found in other debugging solutions supports various probes and targets and integrates directly with Rust tooling and with Visual Studio Code with its extension.

A hardware probe is a device used in the development and debugging of embedded systems to facilitate communication between a host computer and the target embedded device, the probe used for this assignment was ST-Link, a debugging and programming probe developed by STMicroelectronics primarily for their STM32 and STM8 micro-controller series.

4.5 Cheddar GPL

Cheddar [20] is a GNU GPL real-time scheduling simulator/schedulability tool. Cheddar allows the user to model software architectures of real-time systems and to check their schedulability or other performance criteria. Cheddar is developed and maintained by a team composed of the Lab-STICC laboratory UMR CNRS 6285, from the University of Brest and Ellidiss Technologies. Cheddar comprises two independent parts: an editor to model the real-time system to be analyzed, and a framework to perform analysis.

The editor allows you to describe systems composed of cores, processors, cache units, and network-on-chips which own software components (e.g. tasks, shared resources, buffers).

The framework called *Cheddar kernel*, can be called alone (e.g. by a shell script) or embedded in a toolset.

Cheddar offers features such as:

- Scheduling simulations;
- Extraction of information such as response time, number of preemptions, and context switches;
- Feasibility tests on tasks and buffers;
- Shared resources support (both scheduling simulation and worst-case blocking time analysis) with supporter d protocols such as PIP, PCP, and IPCP.

5 Design of the system

The following are some examples of tasks and code snippets from the Rust implementation of the extended application example, showing how the tasks and the resources have been implemented

5.1 Hardware tasks

```
#[task(binds = /*interrupt*/ , shared = [/*Shared resources*/], priority = /*Task's priority*/)]
fn interrupt_sporadic_task(cx: interrupt_sporadic_task::Context) {
    //Task's operation
}
```

Listing 1. Example of an interrupt sporadic task.

For example, the interrupt sporadic tasks of the Ravenscar profile can be reproduced in RTIC, as *hardware tasks*. The RTIC framework uses a hardware interrupt controller to schedule and start the execution of tasks (there are some exceptions, like the `init` task). Hardware tasks are bound to a specific interrupt, this means that they are triggered in answer to an external interrupt bound using the `binds` annotation, here at 2 is how the `external_event_server` task has been implemented.

```
#[task(binds = EXTI0 , shared = [activation_manager,activation_log], priority = 11)]
fn external_event_server(mut cx: external_event_server::Context) {
    cx.shared.activation_log.lock(|log|{
        log.write(Mono::now());
    });

    defmt::info!("External event server has written in the activation log.");
}
```

Listing 2. Implementation of the `external_event_server`.

5.2 Software tasks

Unlike hardware tasks, *software tasks* are not bound to a specific interrupt vector, but rather bound to a `dispatcher` interrupt vector running at the intended priority of the software task.

All software tasks at the same priority level share an interrupt handler acting as an async executor dispatching the software tasks. This list of dispatchers, `dispatchers = [FreeInterrupt1, FreeInterrupt2, ...]` is an argument to the `#[app]` attribute, where the user defines the set of free and usable interrupts.

Each interrupt vector acting as a dispatcher gets assigned to one priority level meaning that the list of dispatchers needs to cover all priority levels used by software tasks.

For example in my application, I have three software tasks, `regular_producer`, `on_call_producer` and `activation_log_reader`, with three different priorities, this means that the `dispatchers =` argument needs at least three entries for using three different priorities.

```
#[task(local = [/*Local resources*/], shared = [/*Shared resources*/], priority = /*Task priority*/)]
async fn periodic_task(cx: periodic_task::Context) {
    let mut next_time = Mono::now();
    loop {
        next_time = next_time + REGULAR_PRODUCER_PERIOD.millis();

        /*Task's operation*/
        Mono::delay_until(next_time).await;
        }
    }
}
```

Listing 3. Example of a periodic task.

5.2.1 Periodic task. In the application in Rust, the *periodic task* is a software task that executes a job inside an infinite loop after it completes the job and waits for its period to perform again. The example at 4 shows the `regular_producer` task from the application. The period is the inter-arrival time between two consecutive jobs, and, if the task hasn't finished its job before the next period (or the next deadline for Deadline Monotonic scheduling), it is considered a deadline miss. The `ProductionWorkload` object contains the *Whetstone* benchmark.

5.2.2 Sporadic task. *Sporadic tasks* are tasks that have a minimum inter-arrival time and are usually triggered by external events such as requests from other tasks, wake-ups from suspension objects, or external interrupts (already discussed before with hardware tasks), suspension objects or request buffers are used to wake up a sleeping task when it's ready to execute and, in this implementation, they are implemented through a producer-consumer channel. At 6, the implementation of the `on_call_producer` task.

The implementation of the `activation_log_reader` is shown at 7, this task uses a size 1 communication channel where it waits until the `regular_producer` sends a message to the channel and reads the `activation_log`'s latest entry.

```
#[task(local = [aux,p], shared = [&activation_manager], priority = 7)]
async fn regular_producer(mut cx: regular_producer::Context,mut writer: Sender<'static, bool, 1>) {
    let mut next_time = Mono::now();

    let mut work = production_workload::ProductionWorkload::new();

    Mono::delay_until(cx.shared.activation_manager.get_activation_time()).await;
    loop {
        next_time = next_time + REGULAR_PRODUCER_PERIOD.millis();
         work.small_whetstone(REGULAR_PRODUCER_WORKLOAD);
        if cx.local.aux.due_activation(2){
            if let Err(_) = cx.local.p.try_send(ON_CALL_PRODUCER_WORKLOAD) {
                defmt::error!("Failed sporadic activation.")
            }
        };
        if cx.local.aux.check_due(){
            writer.try_send(true).unwrap_or_default();
        };
        defmt::info!("End of cyclic execution.");
        Mono::delay_until(next_time).await;
        }
    }
}
```

Listing 4. Implementation of the `regular_producer`.

```
#[task(local = [c], shared = [/*Shared resources*/], priority = /*Task priority*/)]
async fn sporadic_task(cx: sporadic_task::Context) {

    loop {
        // c is a local suspension object
        if let Ok(_) = cx.local.c.recv().await{
            /*Sporadic task's operation*/
        }
    }
}
```

Listing 5. Example of a sporadic task.

5.3 The init task

The `init` task 8 is responsible for starting the system and initializing all the resources required for the execution:

- The `writer` and the `reader` are used as suspension objects to communicate between the `regular_producer` and the `activation_log_reader`;
- The request buffer is made as a producer-consumer channel, then the producer `p` is passed to the `regular_producer` and the consumer `c` is passed to the `on_call_producer` as local resources, this permits a non-blocking queue filling by the producer;

```rust
#[task(local = [c], shared = [&activation_manager], priority = 5)]
async fn on_call_producer(mut cx: on_call_producer::Context) {
    let actv_time = cx.shared.activation_manager.get_activation_time();

    let mut work = production_workload::ProductionWorkload::new();

    Mono::delay_until(actv_time).await;
    loop {
        if let Ok(w) = cx.local.c.recv().await{
            work.small_whetstone(w);
            defmt::info!("End of sporadic execution.");
        }
    }
}
```

Listing 6. Implementation of the on_call_producer.

```rust
#[task(shared = [&activation_manager,activation_log],priority = 3)]
async fn activation_log_reader(mut cx: activation_log_reader::Context,
                               mut reader : Receiver<'static, bool, 1>){
    let actv_time = cx.shared.activation_manager.get_activation_time();

    let mut work = production_workload::ProductionWorkload::new();

    Mono::delay_until(actv_time).await;

    reader.recv().await.unwrap();

    work.small_whetstone(ACTIVATION_LOG_READER_WORKLOAD);
    cx.shared.activation_log.lock(|log|{
        let (_count, _time) = log.read();
    });
    defmt::info!("End of parameterless sporadic activation.");
}
```

Listing 7. Implementation of the activation_log_reader.

- Other resources created are the `activation_log`, which is shared between the `regular_producer` and the `activation_log_reader`, and the `activation_manager`, an auxiliary object that offers a common epoch for all tasks in the system.

Along with the creation of the system resources, the `init` task starts the software tasks with the function `spawn()` and gives them the required resources as function arguments. A monotonic timer is also initialized using `Mono::start()` with the clock speed in hertz as the function argument, this timer is used to keep track of the execution times of the tasks. In short, the `init` task runs before any other tasks returning a set of resources and starting the running tasks.

Another thing to note is that, since RTIC calls a hardware task in answer to an external interrupt, there was no need to implement an event queue.

```rust
#[init]
fn init(cx: init::Context) -> (Shared, Local) {
    let (p, c) = make_channel!(u32, { REQUEST_BUFFER_CAPACITY as usize });
    let (writer,reader) = make_channel!(bool, 1);
    regular_producer::spawn(writer.clone()).unwrap();
    on_call_producer::spawn().unwrap();
    activation_log_reader::spawn(reader).unwrap();
    force_interrupt_handler::spawn().unwrap();

    Mono::start(36_000_000);
    (
        Shared {
            activation_manager : activation_manager::ActivationManager::new(),
            activation_log : activation_log::ActivationLog::new(),
        },
        // initial values for the `#[local]` resources
        Local {
            p,
            c,
            aux : auxiliary::Auxiliary::new()
        },
    )
}
```

Listing 8. `init` task and initialization of shared resources.

5.4 Software generated interrupts

```rust
#[task(priority = 0)]
async fn force_interrupt_handler(_: force_interrupt_handler::Context){
    let mut next_time = Mono::now();

    loop {
        next_time = next_time + INTERRUPT_PERIOD.millis();
        rtic::pend(interrupt::EXTI0);
        defmt::warn!("Interrupt submitted.");
        Mono::delay_until(next_time).await;
    }
}
```

Listing 9. The periodic task generating interrupts.

The system may receive interrupts from external devices such as button presses, I created an additional software periodic task 9 that sets an interrupt as pending using the function `pend()` offered by the RTIC framework, the performance overhead is small so this task will not be considered during the scheduling analysis. The period of the task has been set to the worst case where whenever the `external_event_server` is ready to execute, the system submits an interrupt that triggers the interrupt sporadic task.

6 Scheduling analysis

To analyze the system I needed the upper-bound Worst-Case Execution Times (WCET) estimates of the tasks which can be quite challenging to obtain due to pipelines, caches, and other performance-enhancing techniques used on contemporary computer architectures. One can obtain pessimistic estimates using static analysis programs like KLEE [2] but this leads to poor processor utilization, so to get the WCETs of the tasks I executed a hybrid software instrumentation approach as the tasks' operations are augmented with instrumental code with the end of measuring the execution time of code segments, a good point of this approach is that the WCETs obtained are not pessimistic estimates but are closer to the real execution time, even if response time is randomized, on the other hand, this approach suffers of the *probe effect* [14] as code instrumentation alters the timing of the system since the code used to measure time also may have a (small) execution time. Before explaining the scheduling results, here below are the respective attributes given to the tasks in the Rust application (from now all times are expressed in milliseconds):

Table 1. Real-time attributes of tasks.

Task name	Task type	Period/min inter-arrival time	Deadline	Priority
regular_producer	Cyclic	1000	500	7
on_call_producer	Sporadic	3000	800	5
activation_log_reader	Sporadic	3000	1000	3
external_event_server	Interrupt-sporadic	5000	100	11

Tasks priorities have been set according to Deadline Monotonic Priority Ordering (or DMPO), so if we consider two tasks i and j, D their relative deadlines and P their priorities, if $D_i < D_j$ then $P_i > P_j$. Here below are the attributes of the shared resources, the only shared resource that needs locking is the `activation_log`, since the `request_buffer` has been implemented as a non-blocking producer-consumer communication channel, this channel uses critical sections, however, they are extremely small.

Table 2. Real-time attributes of shared resources.

Resource	User tasks	Ceiling priority
request_buffer (p,c)	regular_producer(try_send), on_call_producer(recv)	-
activation_log	activation_log_reader(read), external_event_server(write)	11

Now I have measured the tasks' Worst-Case Response Time (WCRT) using Cheddar, considering the worst-case scenario where all tasks were periodic and ready to execute. The analysis has been done over the hyper period of the tasks which is the *least common multiple: LCM(1,3,5) = 15s* using a Deadline Monotonic scheduling protocol since in this case, it is optimal.

I noticed that the worst-case execution time for `external_event_server` is close to its worst-case response time since, in the simulation, it was the first task to start and had no interference from other tasks. Its blocking time is caused by `activation_log_reader` which acquires the `activation_log` shared resource, in that case the `external_event_server` waits since the shared resource is blocked by the lower (lowest) priority task.

All other tasks suffer from interference of higher priority tasks:

- `regular_producer` gets interference from `external_event_server`;

Table 3. Scheduling analysis results of the simulation

Task	WCET	WCRT	Blocking time
regular_producer	303	306	0
on_call_producer	111	417	0
activation_log_reader	51	468	0
external_event_server	1	3	2

- `on_call_producer` gets interference from `regular_producer` and `external_event_server`;
- `activation_log_reader` which is the lowest priority task suffers interference by all the other tasks when it is not accessing the `activation_log`.

For the usage of the `request_buffer` during the simulation, the message waiting time is influenced by the time `on_call_producer` waited for a message to come to the buffer, the maximum number of messages in the buffer is 5 and the average number of message is 3.2.

For Deadline Monotonic scheduling, in the preemptive case, processor utilization calculated in the simulation is 80,57%, the utilization test fails since the processor utilization surpasses the result of the utilization test which is $U(n = 4) = n(2^{1/n} - 1) = 75.68\% \leq 80,57\%$ [17] but the tasks do meet all their deadlines in the hyper period due to the priority order.

One thing to keep in mind is that this approach considers all tasks independent of each other, this leads to pessimistic WCRT values since it does not consider any kind of task transaction, for example, `on_call_producer` executes after `regular_producer` has executed two times, while `activation_log_reader` executes after, `regular_producer` has executed for three times, so to gain less pessimistic WCRT I decided to add static offsets [18] to both the tasks, time offsets refer to the delays or time gaps between the release of related tasks or between instances of periodic tasks. They help in structuring when different tasks begin their execution to avoid conflicts, reduce resource contention, or ensure timing constraints are met.

I added an offset of 2300 ms to `on_call_producer` and an offset of 2400 ms to `activation_log_reader` to model the precedence constraint between the `regular_producer` and `on_call_producer` tasks and between the `regular_producer` and `activation_log_reader` tasks:

Table 4. Scheduling analysis results with added offsets

Task	WCET	WCRT	Blocking time
regular_producer	303	2417	0
on_call_producer	111	117	0
activation_log_reader	51	2468	0
external_event_server	1	3	2

Tasks' WCRTs take account of the offset value so the WCRT of the tasks will be $J + O$ with J the actual response time of the task and O the time offset added. Processor utilization is reduced to 66,68%, at this point one can increase process utilization by increasing the workloads of the *Whetstone* benchmarks.

7 Overall results

I decided then to compare the results with the results found in the Ada implementation first considering the tasks begin independent from each other and considering the scenario where the system reaches maximum utilization:

Table 5. MAST analysis from the Ada implementation

Task	WCET	WCRT	Blocking time
regular_producer	482.59	485.49	0.002
on_call_producer	312.34	798.03	0.001
activation_log_reader	198.64	997.45	0
external_event_server	0.01	0.02	0.001

Some things to note are these results:

- Have been obtained using MAST[3];
- The WCET estimates are obtained on a different MCU unit;
- Have been analyzed using a Rate Monotonic FPS scheduling algorithm.

The latter changes the interpretation of processor utilization, which means that for each task $D = T$, this implies that processor utilization will be calculated based on the period rather than the deadline. The scheduling algorithm is a Fixed Priority Scheduling algorithm which means that the task ready with the highest priority will execute first.

In the previous results, the processor utilization is 65.68% which unlike my deadline monotonic scheduling analysis, passes the utilization test, however, if I picked the scheduling results that I got with Cheddar and calculated the processor utilization factor with the period I would get 35.72%, the reason for this is due to the lower computation times of the Rust implementation, changing the Whetstone workloads could alter these values.

Then I compared the values obtained from the analysis with added offsets with the MAST offset-based slanted analysis.

Table 6. MAST offset-based slanted analysis from the Ada implementation

Task	WCET	BCRT	WCRT	Blocking time
regular_producer	482.59	482.59	1454	0.002
on_call_producer	312.34	1795	2768	0.001
activation_log_reader	198.64	2681	3967	0
external_event_server	0.01	0.01	0.02	0.001

The response times produced by the analysis also included the initial offsets but the Best-Case Response Time (BCRT) values are also used as offset of the dependant tasks `on_call_producer` and `activation_log_reader` for their response times. The WCRT values are instead the sum of the corresponding BCRT and jitter.

I compared the BCRT since it already takes into account the offsets and examines the actual case with zero interference, with the WCRT found with Cheddar, notice that the former are lower for the `on_call_producer` and higher for the `activation_log_reader`, this is expected due to the response time values obtained.

8 Self critique

What I achieved in this report:

- All tasks and resources of the system have been implemented successfully in the Rust implementation;
- As a bonus this was a chance to learn a new programming language such as Rust and also a chance to learn embedded programming;
- The system has been tested on both a virtual machine and a bare metal micro-controller unit;
- The system has been analyzed using a deadline monotonic scheduling protocol with independent tasks;
- Then this model was improved by adding precedence constraints on the tasks that reflect closer to the real case scenario.

The reason for choosing Cheddar instead of MAST is only to have a tool that was quick and easy to use but also as competent as MAST, the only problem with Cheddar is that it is not as precise as MAST in terms of time, since it can only consider integer values for execution times, and the user can't analyze large periods.

For the added offsets analysis the WCRT values returned a less pessimistic estimate compared to the simple independent model, however, the offset values were just some estimates of the actual times the sporadic tasks were ready to execute.

Cheddar can analyze systems using the Earliest Deadline First protocol, but this was not requested in the assignment so it was not done.

To go further one could do scheduling with static analysis tools using also control flow graphs or with other Rust-specific tools that are around, which unfortunately I couldn't get to work since they were not compatible with RTIC 2.1.

9 Conclusion

As embedded and IoT systems become more common in our daily lives, we also need to decide which programming language is more adaptable based on the developer's current knowledge, how the developer is willing to learn a new programming language, or the frameworks and resources available and how much they are currently supported. I recreated all the components of the extended example application implemented in Ada with Rust and the RTIC framework, perfect candidates for embedded programming with Rust's safety and performance and RTIC abstractions for tasks and shared resources. Then I ran the re-implemented application on a bare metal micro-controller unit to obtain measurement units and then I operated with a similar scheduling analysis to the one done before, first considering the simplest model with independent tasks and then modeling precedence constraints using time offsets, this permitted me to put into the test the same system with two different programming languages and frameworks, what I found out is that the performance of the two applications is mostly similar, with the Rust implementation begin slightly faster, however, it is not clear what is the best programming language between the two, since there are many factors to take account such as the different MCUs used for the analysis and that RTIC still to this day is in active development. One can put on test any real-time application with multiple programming languages then compare the performance and select the more appropriate language for that use case, or re-implement an older real-time system with the newest cutting-edge technology and test how much the latest technology improves from the previous one. Re-implementing systems may also help developers take a first step into the field of embedded programming by choosing a programming language that they already know, hence having to learn only about the various device targets, the process of flashing a device, and other stuff.

References

[1] [n. d.]. *The Embedded Rust Book*. https://docs.rust-embedded.org/book/intro/index.html

[2] [n. d.]. *KLEE Symbolic Execution framework*. https://klee-se.org/docs/

[3] 2000. *MAST - Modeling and Analysis Suite for Real-Time Applications*. https://mast.unican.es/

[4] 2015. *Rust Programming Language*. https://www.rust-lang.org/

[5] 2017. *RTIC - The hardware accelerated Rust RTOS*. https://rtic.rs/2/book/en/

[6] 2020. *Probe-rs*. https://probe.rs/

[7] 2021. *defmt - deferred formatting for logging on embedded systems*. https://github.com/knurling-rs/defmt

[8] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. 1993. *Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling*. Technical Report. University of York. https://www.math.unipd.it/~tullio/RTS/2009/ABRTW-1993.pdf

[9] N.C. Audsley, A. Burns, M.F. Richardson, and A.J. Wellings. 1991. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. *IFAC Proceedings Volumes* 24, 2 (1991), 127–132. https://doi.org/10.1016/S1474-6670(17)51283-5 IFAC/IFIP Workshop on Real Time Programming, Atlanta, GA, USA, 15-17 May 1991.

[10] T.P. Baker. 1991. *A Stack-Based Resource Allocation Policy for Realtime Processes*. Technical Report. Florida State University. https://www.math.unipd.it/~tullio/RTS/2009/Baker-1991.pdf

[11] T Baker. 2008. Rate monotone scheduling.

[12] Alan Burns, Brian Dobbing, and Tullio Vardanega. 2017. *Guide for the use of the Ada Ravenscar Profile in high integrity systems*. Technical Report. University of York. https://www.math.unipd.it/~tullio/RTS/2019/YCS-2017.pdf

[13] Johan Eriksson, Fredrik Häggström, Simon Aittamaa, Andrey Kruglyak, and Per Lindgren. 2013. *Real-Time For the Masses, Step 1: Programming API and Static Priority SRP Kernel Primitives*. Technical Report. Luleå University of Technology.

[14] Jason Gait. 1986. A probe effect in concurrent programs. *Software: Practice and Experience* 16, 3 (1986), 225–233.

[15] Giovanni Jiayi Hu and Alessio Gobbo. 2020. *Mast Analysis of a Ravenscar precedence-constrained application with FPS and EDF scheduling*. Technical Report. University of Padua.

[16] John P. Lehoczky, Lui Sha, J. K. Strosnider, and Hide Tokuda. 1991. *Fixed Priority Scheduling Theory for Hard Real-Time Systems*. Springer US, Boston, MA, 1–30. https://doi.org/10.1007/978-1-4615-3956-8_1

[17] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (1973). https://doi.org/10.1145/321738.321743

[18] J.C. Palencia and M. Gonzalez Harbour. 1998. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*. 26–37. https://doi.org/10.1109/REAL.1998.739728

[19] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. September 1990. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE TRANSACTIONS ON COMPUTERS, VOL. 39,NO. 9* (September 1990). https://www.math.unipd.it/~tullio/RTS/2009/SRL-1990.pdf

[20] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. 2004. Cheddar: a flexible real time scheduling framework. In *Proceedings of the 2004 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems Using Ada and Related Technologies* (Atlanta, Georgia, USA) *(SIGAda '04)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/1032297.1032298

[21] Henrik Tjäder. 2021. RTIC - A Zero-Cost Abstraction for Memory Safe Concurrency. , 95 pages.

Received 21 October 2024