MAST ANALYSIS OF A RAVENSCAR PRECEDENCE-CONSTRAINED APPLICATION WITH FPS AND EDF SCHEDULING

TECHNICAL REPORT

Giovanni Jiayi Hu
Department of Mathematics
University of Padua, Italy I-35121
Email: giovannijiayi.hu@studenti.unipd.it

Alessio Gobbo
Department of Mathematics
University of Padua, Italy I-35121
Email: alessio.gobbo@studenti.unipd.it

March 9, 2020

ABSTRACT

This paper describes a hard real-time application built with the Ravenscar profile of the Ada programming language and running on a real-time kernel of reduced size and complexity. The application is comprised of several tasks whose activation events present dependency relationships, and this characteristic allows interesting considerations during the different analysis we provide. We consider the same tasks under both Fixed-Priority Scheduling (FPS) and Earliest Deadline First (EDF) and evaluate different metrics like response times, jitters and blocking times. Throughout the sections, we also test the ability of the MAST analysis tools to describe a formal model of dependent tasks and to check their deadline satisfaction with less pessimism as possible without compromising correctness. Lastly, we offer some insight into the behaviour of both FPS and EDF systems under permanent and transient overload, a showcase of how classic real-time considerations may be revaluated to consider dependent tasks.

1 Introduction

Embedded systems have to satisfy strict timing requirements and especially in the case of such hard real-time applications, predictability of the timing behaviour is an extremely important aspect. So it is the choice of a suitable design and development method, in conjunction with supporting tools that enable the real-time performance of a system to be analysed and simulated. The result can lead to a high level of confidence that the final system meets its real-time constraints.

As a matter of fact, the use of Ada has proven to be of great value within high integrity and real-time applications, thanks to language subsets of deterministic constructs which ensure full analysability of the code. In embedded systems, the programmer is tied to become more concerned with the implementation and efficient manipulation of the abstract program entities representing the underlying hardware. Ada solves this issue by i.e. providing the programmer facilities for interrupt handling, access to shared variables and definition of task attributes.

Notably, the Ravenscar profile [1] is a subset of the tasking model, restricted to meet the real-time community requirements for determinism, schedulability analysis and memory-boundedness, as well as being suitable for mapping to a small and efficient run-time system that supports task synchronization and communication.

Along with the Ravenscar profile, we have used a model for representing the temporal and logical elements of real-time applications, called MAST [4]. This model allows a very rich description of the system, including the effects of an event or message-based synchronization, multiprocessor and distributed architectures as well as shared resource synchronization.

The bare-board used throughout our analysis is the STM32F429I-Discovery, a low-cost and easy-to-use development kit to start with an STM32F4 microcontroller and equipped with an ARM Cortex-M4 core. The board has been used

along with the `ravenscar-full-stm32f429disco` runtime environment. The runtime implementation is provided by GNAT, a free-software compiler for the Ada language, and it is based upon the Open Ravenscar Real-Time Kernel [3], which provides full conformance with the Ravenscar profile.

We have preferred the usage of a bare-board instead of the GNAT ARM emulator as we have noticed significant variance with the execution times measured on the latter. The board also includes a ST-LINK/V2 embedded debug tool, which can halt the processor, insert/remove breakpoints and execute instructions line by line [20].

We shall start the paper assuming Fixed Priority Scheduling (FPS), since its behaviour is more predictable and easier to reason about, and then introduce the Earliest Deadline First (EDF) scheduling by means of their differences.

The rest of the paper is organized as follows. The remaining Section 1 will provide an introduction to the fixed-priority scheduling and analysis, the application under consideration and some general notions of the Ada tasking model. A more formal description of our system is then presented in Section 2, followed by considerations about measuring the execution times and detecting deadline misses in Section 3 and 4, respectively.
Section 5 offers an in-depth description of the MAST model and its available analysis tools. Later, Sections 6 provides the insight of our FPS analysis with both stand-alone tasks and precedence relations, followed by a comparative EDF analysis in Section 7. Lastly, Section 8 includes some considerations of both systems under overload, whereas conclusions are contained in Section 8.

In the next sections, whenever we will use the term [RM], we will refer to a section of the Ada Language Reference Manual[1].

1.1 Fixed-priority scheduling and analysis

In the fixed priority system under analysis, each task is assigned a static priority, and the schedule is generated based on the current priority value. According to the Rate Monotonic analysis [11], the fixed priorities are ordered based on the rates, so the task with the smallest period receives the highest priority. The rate (of job releases) of a task is the inverse of its period.

Another well-known fixed-priority algorithm is the Deadline Monotonic algorithm [11]. This algorithm assigns priorities to tasks according to their relative deadlines: the shorter the relative deadline, the higher the priority.

Clearly, when the relative deadline of every task is proportional to its period, the two algorithms are identical. When the relative deadlines are arbitrary, the Deadline Monotonic algorithm performs better in the sense that it can sometimes produce a feasible schedule when the Rate Monotonic algorithm fails. In contrast, the RM algorithm always fails when the DM algorithm fails.

The schedulability analysis uses as inputs the given tasks of periods T_i and execution times C_i and checks one task τ_i at a time to determine whether the response times of all its jobs are equal to or less than its relative deadline D_i.

FPS analysis does not count on any relationship among the release times to hold, and identifies the worst-case combination of release times of any job of task τ_i, and all the jobs in the other tasks that have higher priorities. This combination is the worst because the response time of a job released under this condition is the largest possible for all combinations of release times.

This worst-case time instant is called the critical instant and corresponds to when the job is released at the same time with a job in every higher-priority task, e.g. all of the latter tasks are in phase. This is the case where the response time of the task is the largest and the analysis checks if it is still equal to or less than its relative deadline D_i.

To determine whether a task can meet all its deadlines, an analysis called time-demand analysis computes the total demand for processor time by a job released at a critical instant of the task and by all the higher-priority tasks as a function of time from the critical instant. It then checks whether this demand can be met before the deadline of the job.

To carry out the time-demand analysis on the taskset, we consider one task at a time, starting from the task τ_1 with the highest priority in order of decreasing priority. Assuming t_0 as the release time of the job from task τ_i at the critical instant, at time $t_0 + t$, for $t \geq 0$, the total (processor) time demand $w_i(t)$ of this job and all the higher-priority jobs released in $[t_0, t]$ is given by the following formula for $0 < t \leq T_i$

$$w_i(t) = C_i + \sum_{k=1}^{i-1} \lceil \frac{t}{T_k} \rceil C_k$$

[1] `http://www.ada-auth.org/standards/rm12_w_tc1/html/RM-TOC.html`

If $w_i(t) > t$ for all $0 < t \leq D_i$, this job cannot complete by its deadline. The task τ_i, and hence the given system of tasks, cannot be feasibly scheduled by the fixed-priority algorithm.

Time-demand analysis can be usually depicted plotting the time-demand function as in Figure 1.

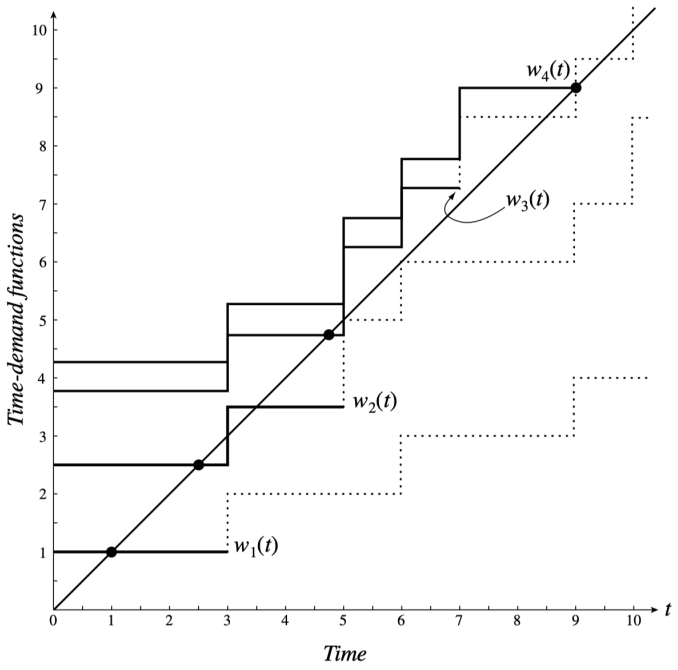

Figure 1: Time-demand analysis example of four tasks (T, C_i): (3,1), (5, 1.5), (7, 1.25), and (9, 0.5) [11]

1.2 The application

The example application presented in this paper is extracted from "Guide for the use of the Ada Ravenscar Profile in high integrity systems" [1]. It includes a periodic process that handles orders for a variable amount of workload. Whenever the request level exceeds a certain threshold, the periodic process farms the excess loadout to a supporting sporadic process. While such orders are executed, the system may receive interrupt requests from an external manual push-button. Each interrupt treatment records an entry in an activation log.

When specific conditions hold, the periodic process releases a further sporadic process to perform a check on the interrupt activation entries recorded in the intervening period. The policy of work delegation adopted by the system allows the periodic process to ensure the constant discharge of a guaranteed level of workload.

The correct implementation of this policy also requires assigning the periodic process a higher priority than those assigned to the sporadic processes, so that guaranteed work can be performed in preference to subsidiary activities.

The application is comprised of the tasks and attributes in Table 1. Static priorities are given based on the Deadline Monotonic scheduling [11], which is the most optimal between the fixed priority algorithms [13].

Task name	Task type	Period / Minimum inter-arrival time (ms)	Deadline (ms)	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

Table 1: Attributes of the tasks in the application [1]

Ada protected objects [RM 9.4] are used to ensure mutually exclusive access to shared resources, whereas protected entries are used only for task synchronization purposes where data exchange is involved.

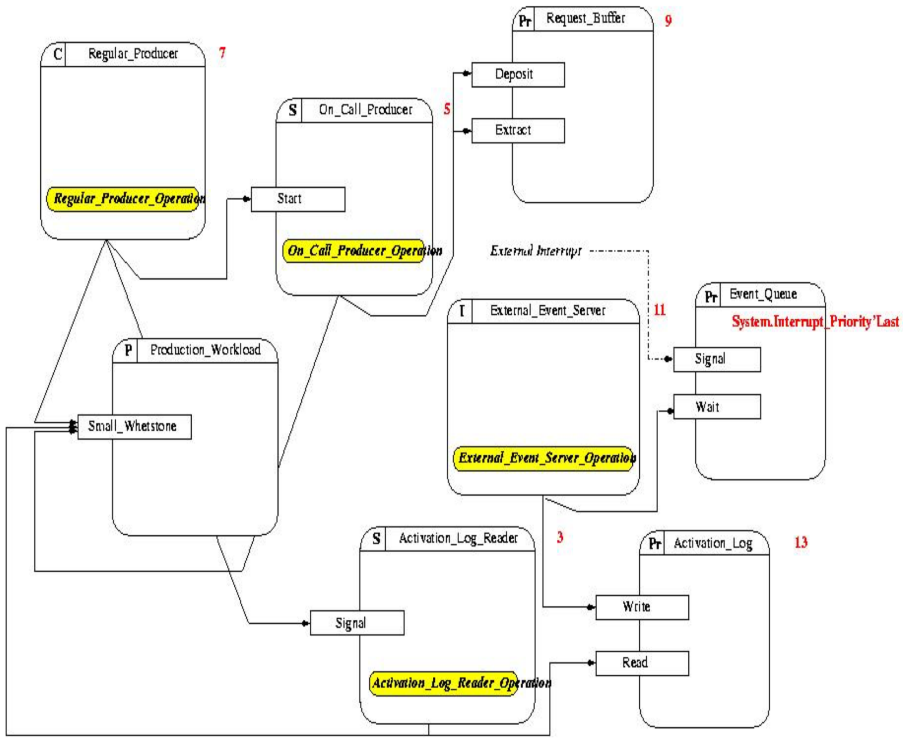

Figure 2: Architecture of the example application [1].

In a real-time application, each protected object has a priority ceiling which represents the maximum priority of any task that calls the object. The Ada Real-Time Systems Annex supports the definition of `Locking_Policy` [RM D.3] and implements the resource locking protocol called Immediate Priority Ceiling Protocol (IPCP) [5], which is similar to the Priority Ceiling Protocol (PCP).

PCP is an improvement of the Priority Inheritance Protocols (PIP) which allow a task to execute with an enhanced priority if it is blocking (or could block) a higher-priority task. In addition to PIP, PCP prevents deadlock and reduces blocking to its minimum value: every job is blocked at most once for the duration of a critical section, no matter how many jobs conflict with it [14].

The IPCP is similar to PCP in its use of the ceiling priority, but it has a different set of rules on how a task behaves under the ceiling locking protocol.

1. A task may lock a protected object if it is not yet locked.
2. When it enters a critical section, it immediately inherits the priority ceiling of the protected object and recovers its entry priority when it exits the section.

This protocol effectively prevents any task from starting to execute until all the shared resources it needs are free. This means that no separate mutual exclusion mechanism, such as semaphores, is needed to lock shared resources. It is also cheap to implement at run time and incurs in fewer context switches. By raising priorities as soon as a resource is locked, whether a higher priority task is trying to access it or not, the protocol avoids the need to make complex scheduling decisions while tasks are already executing.

Protected object names	User tasks	Ceiling priority
Request_Buffer	Regular_Producer (`Deposit`), On_Call_Producer (`Extract`)	9
Event_Queue	External interrupt (`Signal`), External_Event_Server (`Wait`)	System.Interrupt_Priority'First
Activation_Log	External_Event_Server (`Write`), Activation_Log_Reader (`Read`)	13

Table 2: Attributes of the protected objects in the application [1]

1.3 Ada tasks for real-time systems

In the Ada Ravenscar, a periodic task has an infinite loop within which there is a self-suspension statement that ensures that the task regularly executes [32]:

```
with Ada.Real_Time; use Ada.Real_Time;
...
task Periodic_Task;
   task body Periodic_Task is
      Period : Time_Span := Milliseconds(1000);
      -- define the period of the task, 1000 ms in this example
      Next : Time;
   begin
      Next := Clock;
      -- start time
      loop
         -- undertake the work of the task
         Next := Next + Period;
         delay until Next;
      end loop;
end Periodic_Task;
```

However, we should bear in mind that `Period` is the minimum length of time between the release times of instances of the task. The subsequent jobs will be released periodically only if the loop always completes within `Period` time units. If the response time of an instance of the thread exceeds the value, the next instance is released only as soon as the current instance completes. Therefore there will be both a deadline miss of the current job and a delay in activation of the subsequent instance.

A sporadic task requires a protected object instead to control its release:

```
task Sporadic_Task;
protected Sporadic_Controller is
   entry Wait_Next_Invocation;
   procedure Release_Sporadic;
private
   Barrier : Boolean := False;
end Sporadic_Controller;

task body Sporadic_Task is
begin
   loop
      Sporadic_Controller.Wait_Next_Invocation;
      -- undertake the work of the task
   end loop;
end Sporadic_Task;

protected body Sporadic_Controller is
   entry Wait_Next_Invocation when Barrier is
   begin
      Barrier := False;
   end;

   procedure Release_Sporadic is
   begin
      Barrier := True;
   end;
end Sporadic_Controller;
```

The task body for an event-triggered task that conforms to the Ravenscar Profile typically has, as its last statement, an outermost infinite loop whose first statement is either a call to a protected entry or a call to a Suspension Object [1]. The Suspension Object is used when no other effect is required in the signalling operation; for example, no data is to be transferred from signaller to waiter. In contrast, the protected entry is used for more elaborate event signalling, when additional operations must accompany the resumption of the event-triggered task.

1.4 Software interrupts

As mentioned above, the system may receive interrupt requests (IR) from an external manual push-button. Such IR is handled by the NVIC Interrupt Controller, which is a part of the Cortex-M processor that handles the exceptions and the interrupt configurations, prioritization and masking [2].

To automate the arrival of interrupts throughout our following analysis, we periodically trigger an IR via software by using a Software Trigger Interrupt Register (STIR). We first define a system package `ST.EXTI` where we gain access to the STM32F4 Interrupts and Events registers. Then, setting a pending bit to the appropriate interrupt line in the STIR triggers the IR for the external push-button.

Lastly, the whole process is executed within a new periodic task Force_Interrupt, whose loop body simulates the worst-case sporadic behaviour of the interrupt. The worst-case happens when the subsequent interrupts arrive at the minimum interarrival time, used as period of the task.

```
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Text_IO;
with ST; use ST;
with ST.EXTI; use ST.EXTI;

package body Force_Interrupt is
   Period : constant Ada.Real_Time.Time_Span :=
      Ada.Real_Time.Milliseconds (5000);
   Button_Line : constant Interrupt_Line := 0;
   --  The User Button is connected to EXTI0 (aka Line 0)

   task body Force_Interrupt is
      --  for periodic suspension
      Next_Time : Ada.Real_Time.Time;
   begin
      loop
         Next_Time := Next_Time + Period;

         EXTI.Software_Trigger_Interrupt_Register.Line :=
            (Button_Line => True, others => False);
         Ada.Text_IO.Put_Line ("Interrupt␣generated");

         delay until Next_Time; --  delay statement at end of loop
      end loop;
   end Force_Interrupt;
end Force_Interrupt;
```

The overhead introduced by the newly defined task is negligible and thus will not be considered during the different analysis of the paper.

2 System model and notation

The described application is a set of tasks executing in the same processor, grouped into entities called transactions [37]. Each transaction Γ_i is activated by a periodic sequence of external events with period T_i , and contains a set of tasks. Each task is released when a relative time offset elapses after the arrival of the external event. Each activation of a task releases the execution of one instance of that task, called a *job*.

Figure 3 shows an example of such system: the horizontal axis represents time; down-pointing arrows represent the arrival of the external events associated to each transaction, while up-pointing arrows represent the activation times of each task; and shaded boxes represent task execution [36]. Each task has its own unique priority, and in this example, the task set is scheduled using a preemptive FPS.

Each task will be identified with two subscripts: the first one identifies the transaction to which it belongs, and the second one the position that the task occupies within the tasks of its transaction when they are ordered by increasing offsets. In this way, τ_{ij} will be the j-th task of transaction Γ_i, with an offset of Φ_{ij} and a worst-case execution time of C_{ij}. In addition, we will allow each task to have jitter, that is to have its activation time delayed by an arbitrary amount of time between 0 and the maximum jitter for that task, which we will call J_{ij}. This means that the activation time of task τ_{ij} may occur at any time between $t_0 + \Phi_{ij}$ and $t_0 + \Phi_{ij} + J_{ij}$, where t_0 is the instant at which the external event arrived.

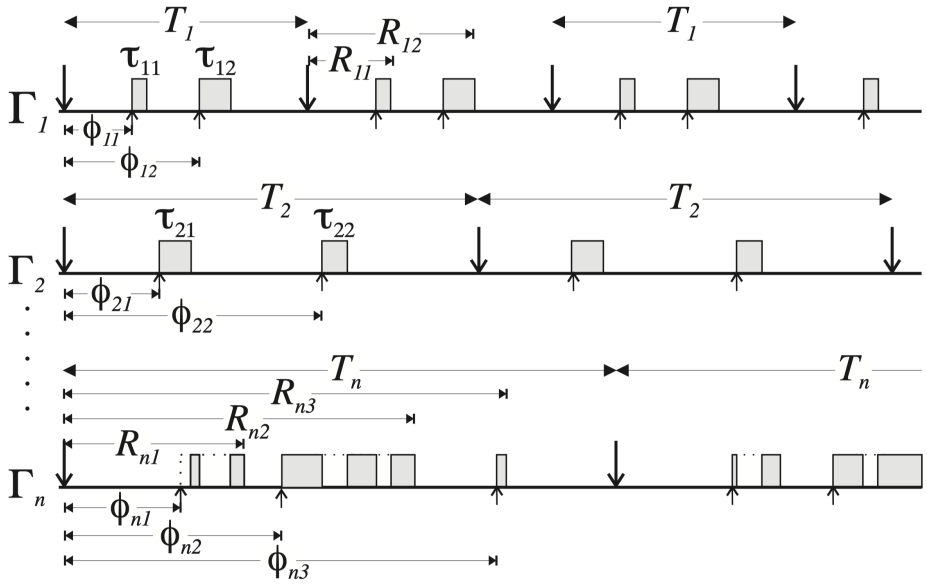

Figure 3: Timeline of a system composed of transactions with offsets [36]

The reason for this is that tasks must execute in order, i.e. On_Call_Producer can start executing only after the preceding task in the transaction, Regular_Producer, has completed. The precedence constraints are modelled by assigning each task an initial offset and a maximum jitter [37]. The initial offset Φ_{ij} of a periodic task is the instant of the first activation of the task. However, a task belonging to a transaction may start only after it has been activated, and the preceding task in the transaction has completed execution. Hence maximum jitter is the maximum time interval it can occur from the task activation until the completion time of the preceding task in the transaction.

In addition to maximum jitter, tasks offsets are allowed to vary dynamically, from one activation to the next, within a minimum and a maximum value: $\Phi_{ij} \in [\Phi_{ij\ min}, \Phi_{ij\ max}]$. Dynamic offsets are useful in systems in which tasks suspend themselves, like in the case of protected object entries. The task On_Call_Producer τ_{i2} calls the protected entry `Extract` and suspends itself until the task Regular_Producer τ_{i1} replenishes the `Request_Buffer`. The activation time of On_Call_Producer depends on the completion time of the Regular_Producer, and thus the offset for task τ_{i2} is variable in the interval $\Phi_{i2} \in [R_{i1\ min}, R_{i1\ max}]$, where $R_{i1\ min}$ and $R_{i1\ max}$ are respectively the best-case and worst-case response times of task Regular_Producer.

For each task τ_{ij} we define its response time as the difference between its completion time and the instant at which the associated external event arrived. The worst-case response time will be called R_{ij}. Each task has also an associated global deadline, D_{ij}, which is again relative to the arrival of the external event.

If tasks synchronize using shared resources in a mutually exclusive way, they will be using the aforementioned Immediate Priority Ceiling Protocol. The effects of lower priority tasks on a task under analysis τ_{ab} are bounded by an amount called the blocking term B_{ab}, calculated as the maximum of all the critical sections of lower priority tasks that have a priority ceiling higher than or equal to the priority of τ_{ab}.

2.1 Offset-based analysis

Rate monotonic analysis (RMA) [11] allows an approximate calculation of the worst-case response time of tasks in single-processor real time systems, including the effects of task synchronization, the presence of aperiodic tasks, the effects of deadlines before, at or after the periods of the tasks, tasks with varying priorities, overhead analysis, etc. However, classic RMA [29] cannot provide exact solutions in systems in which tasks with precedence relations. Classic techniques for these systems are based on the assumption that all tasks are independent, and thus they lead to pessimistic results [36].

For building the worst-case scenario for a task τ_{ab} under analysis, the analysis must consider the critical instant that leads to the worst-case busy period. A task τ_{ab} busy period is an interval of time during which the CPU is busy processing task τ_{ab} or higher priority tasks. For tasks with offsets, it must take into account that the critical instant may

not include the simultaneous arrival of all higher priority jobs, as it was the case when all tasks were independent. The existence of offsets makes it impossible for some sets of tasks to become active simultaneously.

Works on such problem have been the base of offset-based analysis, first proposed by Tindell and Clark [37] and later improved by Palencia and Gonzàlez [36] who called it Worst-Case Analysis of Dynamic Offsets. In such analysis, the best and worst-case response times of each task are used to set the offset and the jitter of the successive task in the same transaction.

3 Execution times

To use the described model, upper bounds on the execution times are needed. Unfortunately, precise Worst-Case Execution Time (WCET) is hard to find due to pipelines, caches and other performance-enhancing techniques used on contemporary computer architectures [38]. These effects are reduced in the case of a more predictable bare-board environment, which can nevertheless suffer a small amount of indeterminism. Therefore pessimistic scheduling is needed in order to provide an offline guarantee that all hard deadlines will be met, but leads to poor processor utilization.

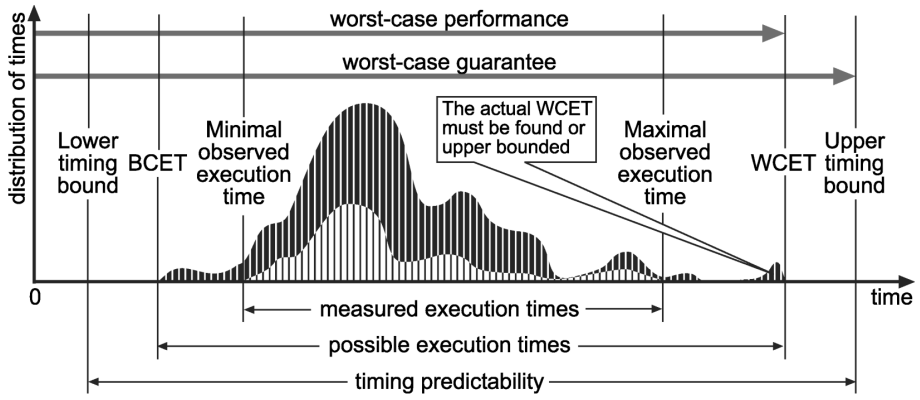

Figure 4: The lower curve represents a subset of measured executions. The darker curve, an envelope of the former, represents the times of all executions. [38].

Figure 4 shows the set of all execution times as the upper curve. Its minimum and maximum are the best- and worst-case execution times, respectively, abbreviated BCET and WCET. In most cases, the space is too large to exhaustively explore all possible executions and thereby determine the exact worst- and best-case execution times.

The conventional method to estimate execution time bounds is to measure the end-to-end execution time of the task for a subset of the possible executions. This determines the minimal observed and maximal observed execution times. These will, in general, overestimate the BCET and underestimate the WCET.

Nevertheless, we have adopted the same approach, aware of the mentioned perils. In most cases, we had deterministic execution times with always the same exact number of CPU cycles or with a difference less than 1μs, except for the Whetstone operations, which showed more significant variation. However, we always had shallow standard errors minor than 1%. The example application is quite simple, comprised of few tasks with predictable executions and the only interrupts are the periodic ticker and the external push button.

Execution times are measured using two custom packages: `System_Overhead` and `Task_Metrics`. The former is able to provide the exact number of elapsed CPU ticks, which is then converted as seconds by dividing it with the clock frequency. It's used to measure runtime overhead, whereas the latter provides task execution time if self-suspension can happen, which would make usage of the clock ticks unsuitable.

```
-- system-overhead.ads
with System.BB.Time; use System.BB.Time;
with System.Semihosting;

package System_Overhead is
   pragma Preelaborate;

   procedure Start_Tracking;
```

```
   --  Avoid counting sub-program execution time
   procedure Start_Sub_Program;
   procedure End_Sub_Program;

   procedure End_Tracking (Item : String := "");

   procedure Log_Time;
   --  Just log the current clock time
end System_Overhead;

with System.BB.Time; use System.BB.Time;
with System.Semihosting;



-- system-overhead.adb
package body System_Overhead is
   Initial_Value : Time := 0;
   Start_Sub_Value : Time := 0;
   End_Sub_Value : Time := 0;

   procedure Start_Tracking is
   begin
      Initial_Value := Clock;
      Start_Sub_Value := 0;
      End_Sub_Value := 0;
   end Start_Tracking;

   procedure Start_Sub_Program is
   begin
      Start_Sub_Value := Clock;
   end Start_Sub_Program;

   procedure End_Sub_Program is
   begin
      End_Sub_Value := Clock;
   end End_Sub_Program;

   procedure End_Tracking (Item : String := "") is
      Now : constant Time := Clock;
      Sub_Program : Time;
      Elapsed : Time;
   begin
      -- Sometime End_Tracking may be called before Start_Tracking
      if Initial_Value = 0 then
         return;
      end if;

      Sub_Program := End_Sub_Value - Start_Sub_Value;
      Elapsed := Now - Initial_Value - Sub_Program;

      Put_Line (Item & Time'Image (Elapsed));
   end End_Tracking;

   procedure Log_Time is
   begin
      Put_Line (Time'Image (Clock));
   end Log_Time;

   procedure Put_Line (Item : String) is
   begin
      System.Semihosting.Put (Item & ASCII.CR & ASCII.LF);
   end Put_Line;
end System_Overhead;
```

The `System_Overhead` package uses the board-specific `System.BB.Time` package, which provides the `Clock` function to read the real-time monotonic clock. It's the same primitive used under-the-hood by `Ada.Real_Time` [RM D.8] to provide physical time as observed in the external environment.

The package `Task_Metrics` has the same interface as `System_Overhead`, but it replaces `System.BB.Time` with `Ada.Execution_Time` [RM D.14] to measure the elapsed execution time of a task. The `ravenscar-full-stm32f429disco` runtime supports the Ada 2012 implementation to separately account for the execution time of interrupt handlers [35].

The functionality of the real-time clock (RTC) and execution time clocks (ETCs) are quite similar: both clocks support high accuracy measurement of the monotonic passing of time since an epoch, and both support calling a protected handler when a given timeout time is reached. The main difference is that the RTC is always active, while an ETC is active only when its corresponding task or interrupt is executed.

3.1 Semi-hosting

It is worth mentioning the usage of semi-hosting [21], which allows print messages to be transferred from the board to the host computer using the debug connection. Using semi-hosting for printing is usually much slower than UART because the semi-hosting mechanism needs to halt the processor, but on the other hand, the system tick timer `Sys_Tick` counter is stopped during the transmission, thus avoiding affecting the schedule of the tasks. The example application has no timing requirements relative to external interrupts, with the exception of the manual push-button.

Both `System_Overhead` and `Task_Metrics` use semi-hosting to send execution time data to the host computer. Besides, it is leveraged also in the `ravenscar-full-stm32f429disco` runtime implementation of the `Ada.Text_IO` package, whose method `Put_Line` is called by the tasks.

The runtime defines a semi-hosting buffer size of 128 characters before flushing a string, therefore we have padded all the print messages with white space to reach the fixed size of 50 characters. By doing so, we have fixed execution time due to buffer insertion, simplifying MAST modelling of the `Put_Line` operation.

4 Deadline miss detection

In later analysis, we will want to achieve the maximum schedulable utilization by analyzing a MAST model with low utilization and then increasing tasks utilization until the system no longer meets its deadlines. However, for design attributes to turn into system properties, we must enforce them at runtime. In particular, we have to check that the jobs of the tasks always complete before their respective deadline, to ensure consistency between the MAST analysis and the execution [34].

Fortunately, Ada 2005 introduced a lower-level facility that maps a handler to a specific time without the need to use a separate task. The handler is associated with a timing event. When the event time is due and detected by the runtime, the handler code is executed.

The most effective way for an implementation to support timing events is to execute the handlers directly from the interrupt handler of the clock [33], and this is indeed what happens in `ravenscar-full-stm32f429disco`.

```
-- deadline_miss.ads
with System;
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Real_Time.Timing_Events; use Ada.Real_Time.Timing_Events;

package Deadline_Miss is
   type Deadline_Handler is limited private;

   procedure Set_Deadline_Handler (
      H: in out Deadline_Handler;
      Name : String;
      At_Time : in Time);
   procedure Cancel_Deadline_Handler (H: in out Deadline_Handler);

private
   protected type Deadline_Handler
     with Priority =>
       System.Interrupt_Priority'Last
   is
```

```
      procedure Notify_Deadline_Miss (Event : in out Timing_Event);

      procedure Set_Deadline_Handler (Name : String; At_Time : in Time);
      procedure Cancel_Deadline_Handler;

      private
         Tag : String(1..3) := "N/A";
         Event : Timing_Event;
   end Deadline_Handler;
end Deadline_Miss;

-- deadline_miss.adb
with Ada.Real_Time; use Ada.Real_Time;
with Ada.Text_IO; use Ada.Text_IO;

package body Deadline_Miss is
   protected body Deadline_Handler is
      procedure Notify_Deadline_Miss (Event : in out Timing_Event) is
      begin
         --raise Program_Error with "Detected deadline miss";
         Ada.Text_IO.Put_Line ("Deadline␣Miss␣Detected␣-␣" & Tag );
      end Notify_Deadline_Miss;

      procedure Set_Deadline_Handler (Name : String; At_Time : in Time) is
      begin
         Tag := Name;

         Set_Handler (Event, At_Time, Notify_Deadline_Miss'Access);
      end Set_Deadline_Handler;

      procedure Cancel_Deadline_Handler is
         Cancelled : Boolean;
         pragma Unreferenced (Cancelled);
      begin
         Cancel_Handler (Event, Cancelled);
      end Cancel_Deadline_Handler;
   end Deadline_Handler;

   procedure Set_Deadline_Handler (
      H: in out Deadline_Handler;
      Name : String;
      At_Time : in Time) is
   begin
      H.Set_Deadline_Handler (Name, At_Time);
   end Set_Deadline_Handler;

   procedure Cancel_Deadline_Handler (H : in out Deadline_Handler) is
   begin
      H.Cancel_Deadline_Handler;
   end Cancel_Deadline_Handler;
end Deadline_Miss;
```

The measured execution times include overrun detection overhead for Regular Producer, On Call Producer and Activation Log Reader.

5 MAST

MAST [4] is a Modeling and Analysis Suite for Real-Time Applications, and its main goal is to provide an open-source set of tools that enables engineers developing real-time applications to check the timing behaviour of their application, including schedulability analysis with hard timing requirements.

It is designed to handle both fixed priority and dynamic priority scheduled systems, although offset-based analysis for Earliest Deadline First scheduling is still missing as of the time of writing. However, within fixed priorities, different

scheduling strategies are allowed, including preemptive and non-preemptive scheduling, interrupt service routines, sporadic server scheduling, and periodic polling servers.

The MAST model is designed to handle both single-processor as well as multiprocessor or distributed systems. In both cases, the emphasis is placed on describing event-driven systems in which each task may conditionally generate multiple events at its completion. A task may be activated by a conditional combination of one or more events. The external events arriving at the system can be of different kinds: periodic, unbounded aperiodic, sporadic, bursty, or singular (arriving only once).

The system model facilitates the independent description of overhead parameters such as processor overheads (including the overheads of the timing services). This frees us from the need to include all these overheads in the actual application model, thus simplifying it and eliminating much redundancy.

MAST provides also a graphical editor to generate the system using the MAST ASCII description. However, it's still immature to be reliable, and the presence of several graphical bugs causes an annoying experience. A graphical display of results is also available.

5.1 The MAST Model

We now proceed to describe the MAST model of the example application. In this phase, it will represent an FPS set of independent tasks, further sections will provide the needed changes to match a chain of dependant tasks or to support EDF scheduling. For a full reference to the MAST syntax, visit "Description of the MAST Model" [6].

5.1.1 Processing Resources

Processing Resources represent resources that are capable of executing abstract activities, including conventional CPU processors. Among its attributes, we have the range of priorities valid for normal operations on that processing resource and the speed factor. We have left the default value as speed factor, meaning that execution times will be expressed as seconds.

Normally when dealing with hard real-time analysis, we would also define only the Worst-Case Execution Time (WCET) of the operations but, since we have dynamic offsets depending on them, we include both best and worst execution times because we don't know for sure that always having the WCET corresponds to worst system performance. We may, for instance, have anomalies as in the case of multiple processors [17].

```
Processing_Resource (
   Type                    => Regular_Processor,
   Name                    => cpu,
   Max_Interrupt_Priority  => 255,
   Min_Interrupt_Priority  => 241,
   Worst_ISR_Switch        => 2.578E-06,
   System_Timer            =>
      ( Type           => Ticker,
        Worst_Overhead => 3.844E-06,
        Period         => 0.001000),
   Speed_Factor            => 1.00);
```

The board is built with only one CPU, whereas the interrupt ranges are taken from the `System` package in the `ravenscar-full-stm32f429disco` runtime. Task priorities span from 1 to 240, while interrupt priorities go from 241 to 255. Thus it's possible to have at max 240 distinct task priorities if more priorities are needed, one can use the technique described in [15].

The Interrupt Service Routine (ISR) overhead is measured as the time taken to run the `Interrupt_Handler` in `System.BB.Board_Support` package, without counting the execution time of the application interrupt handler. In Ada, the code in the handler itself executes at the hardware interrupt level. In contrast, the major part of the processing of the response to the interrupt is moved into an event response task, which executes at a software priority level with interrupts fully enabled.

The first procedure executes for a very short time-typically executing only the instructions that are strictly necessary to service the interrupt and reset the associated piece of hardware. The second one is implemented as a task that is activated from the interrupt handler and its priority is assigned as defined in Table 1.

Both parts are not accounted into the ISR overhead. However, the overhead takes into account the management of the aforementioned Execution Time Clocks (ETC) [35].

The system timer used by the board is Tick Scheduling [16], which represents a system that has a periodic clock interrupt that arrives at the system. When this interrupt arrives, all timed events whose expiration time has already passed, are activated.

Tick scheduling introduces two additional factors that must be accounted for in schedulability analysis. First, the fact that a job is ready may not be noticed and acted upon by the scheduler until the next clock interrupt. This introduces additional jitter that may delay the completion of the job.

Second, a self-suspended task is held in a queue which we will call the delay queue. When the scheduler executes, it scans the delay queue and moves the jobs that have been released since the last clock interrupt to the ready job queue and places them there in order of their priorities. Once in the ready queue, the jobs execute in priority order without intervention by the scheduler. The time the scheduler takes to scan and move the jobs introduces additional scheduling overhead. Similar overhead must be accounted for any timing events that need to be triggered.

The scheduling overhead is accounted for in the analysis using the technique described in [30]. MAST can model the scheduler as a periodic task τ_0 whose period is p_0. This task has the highest priority among all tasks in the system. Its execution time C_0 is the amount of time the scheduler takes to service the clock interrupt. This time is spent even when there is no job in the pending job queue.

In the `ravenscar-full-stm32f429disco` runtime, the period p_0 of the tick is 1ms, defined in the `System.BB.Board_Support` package and the worst overhead is measured as the time taken to execute `Timer_Interrupt_Handler`, the trap handler defined in the same package for the `Sys_Tick` trap.

5.1.2 Schedulers

Schedulers represent the runtime procedures that implement the appropriate scheduling strategies to manage the amount of CPU processing capacity. They can have a hierarchical structure to model hierarchical scheduling [18], but the example application has only one primary scheduler with fixed priority policy.

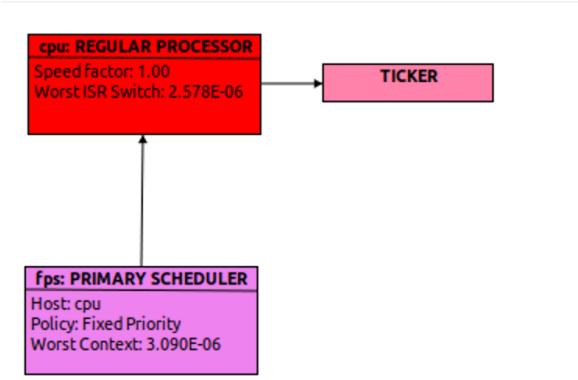

Figure 5: Fixed Priority Scheduler which manages the CPU

```
Scheduler (
   Type             => Primary_Scheduler,
   Name             => fps,
   Host             => cpu,
   Policy           =>
      ( Type                  => Fixed_Priority,
        Worst_Context_Switch  => 3.090E-06,
        Max_Priority          => 240,
        Min_Priority          => 1));
```

The context switch overhead is measured as time to set the context switch interrupt `Pend_SV` as pending and the execution time of `Pend_SV_Handler` in the `System.BB.CPU_Primitives.Context_Switch_Trigger` package, which saves the registers of active context and restores the ones of the new context. One some platforms, like in the case of the STM32F429I-Discovery board equipped with an ARM Cortex-M4 core, the context switch requires the triggering of a trap [22]. Then context switching is usually carried out in the `Pend_SV` trap handler.

5.1.3 Scheduling Servers

Scheduling Servers represent schedulable entities in a processing resource, in particular, if the resource is a processor, the scheduling server is a task or thread of control. As a matter of fact, each of them has a priority and a type, which for our application may be `Fixed_Regular_Policy` or `Interrupt_FP_Policy`. The former represents a regular preemptive fixed priority, whereas the latter models an interrupt service routine. In reality, we have not used a `Interrupt_FP_Policy` as the interrupt overhead is negligible.

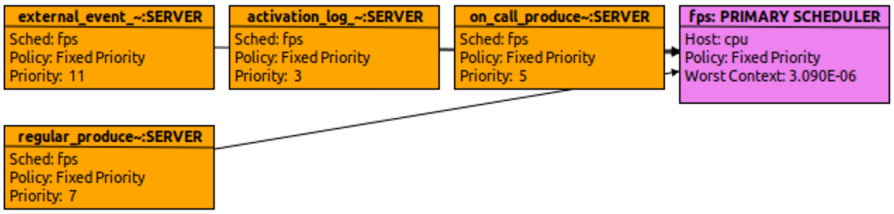

Figure 6: Scheduling Servers representing the application tasks

```
Scheduling_Server (
   Type                        => Regular,
   Name                        => regular_producer,
   Server_Sched_Parameters     =>
      ( Type          => Fixed_Priority_Policy,
        The_Priority => 7,
        Preassigned  => YES),
   Scheduler                   => fps);

Scheduling_Server (
   Type                        => Regular,
   Name                        => on_call_producer,
   Server_Sched_Parameters     =>
      ( Type          => Fixed_Priority_Policy,
        The_Priority => 5,
        Preassigned  => YES),
   Scheduler                   => fps);

Scheduling_Server (
   Type                        => Regular,
   Name                        => activation_log_reader,
   Server_Sched_Parameters     =>
      ( Type          => Fixed_Priority_Policy,
        The_Priority => 3,
        Preassigned  => YES),
   Scheduler                   => fps);

Scheduling_Server (
   Type                        => Regular,
   Name                        => external_event_server,
   Server_Sched_Parameters     =>
      ( Type          => Fixed_Priority_Policy,
        The_Priority => 11,
        Preassigned  => YES),
   Scheduler                   => fps);
```

5.1.4 Shared Resources

Shared Resources represent resources that are shared among different tasks, and that must be used in a mutually exclusive way. Therefore, protected objects are modelled as Shared Resourses that use the Immediate Priority Ceiling Protocol described above.

```
Shared_Resource (
```

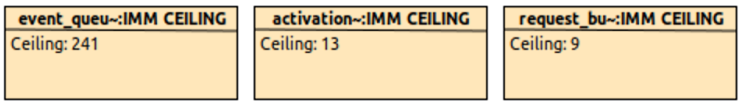

Figure 7: Shared Resources of the application

```
   Type        => Immediate_Ceiling_Resource,
   Name        => request_buffer,
   Ceiling     => 9,
   Preassigned => YES);

Shared_Resource (
   Type        => Immediate_Ceiling_Resource,
   Name        => activation_log,
   Ceiling     => 13,
   Preassigned => YES);

Shared_Resource (
   Type        => Immediate_Ceiling_Resource,
   Name        => event_queue,
   Ceiling     => 241,
   Preassigned => YES);
```

5.1.5 Operations

MAST Operations represent a piece of code to be executed by the processor. We have used the following classes of operations:

- **Simple**: it represents a simple piece of code or a message. It may have the list of shared resources to lock before executing the operation, and the list of shared resources that must be unlocked after executing the operation. Simple Operations have been used to model methods of protected objects. The execution time is measured from the first line of the method to the last one, thus it doesn't include the runtime overhead associated with invoking protected methods.
- **Composite**: it represents an operation composed of an ordered sequence of other operations, simple or composite. The execution time attribute of this class cannot be set because it is the sum of the execution times of the comprised operations.
- **Enclosing**: it represents an operation that contains other operations as part of its execution, but in this case the total execution time must be set explicitly; it is not the sum of execution times of the comprised operations, because other pieces of code may be executed in addition. For each protected method, there is an Enclosing Operation which takes into account the overhead associated with calling protected methods. Sometimes it corresponds to a method defined by the application, other times it's defined in the model specifically to include the runtime overhead. By doing so, we can define the caller procedures as simple Composite operations.

Examples of protected methods as Simple Operations:

```
Operation (
   Type                       => Simple,
   Name                       => rb_deposit,
   Worst_Case_Execution_Time  => 2.000E-06,
   Shared_Resources_To_Lock   =>
      ( request_buffer),
   Shared_Resources_To_Unlock =>
      ( request_buffer));

Operation (
   Type                       => Simple,
   Name                       => rb_extract,
   Worst_Case_Execution_Time  => 2.000E-06,
   Shared_Resources_To_Lock   =>
```

```
      ( request_buffer),
   Shared_Resources_To_Unlock =>
      ( request_buffer));
```

Examples of Enclosing Operations including protected methods overhead:

```
Operation (
   Type                      => Enclosing,
   Name                      => ocp_start,
   Worst_Case_Execution_Time=> 6.000E-06,
   Composite_Operation_List =>
      ( rb_deposit));

Operation (
   Type                      => Enclosing,
   Name                      => rb_extract_enclosing,
   Worst_Case_Execution_Time=> 7.000E-06,
   Composite_Operation_List =>
      ( rb_extract));
```

A complete example of the MAST representation of a job of the task Regular_Producer:

```
Operation (
   Type                       => Simple,
   Name                       => rp_small_whetstone,
   Worst_Case_Execution_Time  => 0.019363);

Operation (
   Type                       => Simple,
   Name                       => due_activation,
   Worst_Case_Execution_Time  => 1.000E-06);

Operation (
   Type                      => Enclosing,
   Name                      => ocp_start,
   Worst_Case_Execution_Time=> 6.000E-06,
   Composite_Operation_List =>
      ( rb_deposit));

Operation (
   Type                       => Simple,
   Name                       => check_due,
   Worst_Case_Execution_Time  => 1.000E-06);

Operation (
   Type                       => Simple,
   Name                       => alr_signal,
   Worst_Case_Execution_Time  => 5.000E-06);

Operation (
   Type                       => Simple,
   Name                       => put_line,
   Worst_Case_Execution_Time  => 1.400E-05);

Operation (
   Type                      => Composite,
   Name                      => rp_operation,
   Composite_Operation_List =>
      ( rp_small_whetstone,
        due_activation,
        ocp_start,
        check_due,
        alr_signal,
        put_line));
```

```
Operation (
   Type                       => Composite,
   Name                       => regular_producer,
   Composite_Operation_List =>
      ( overrun_detection,
        rp_operation,
        delay_until));
```

The Small_Whetstone algorithm allows controlling the computational workload of Regular_Producer, On_Call_Producer and Activation_Log_Reader. By changing the workload parameters of `Small_Whetstone` in the application, we will be able to test different utilisation of the system with likewise ease in updating the MAST model.

The Whetstone execution time is proportional to the workload parameter and exhibits deterministic behaviour. If we wanted to try what would happen by increasing the load of factor 10, we would just multiply the WCET in the model by 11, without the need to measure all the Enclosing operations again, since all the methods which use Whetstone are defined as Composite. However, we have been careful to avoid forgetting to include any overhead in an Enclosing method, and we have made sure they are not impacted by any change of the Whetstone workload.

5.1.6 Transactions

A Transaction represents a transaction of our model (see Section 2) as a graph of event handlers and events, and form interrelated activities executed in the system. A Transaction is defined with three different components: a list of External Events, a list of Internal Events (with their timing requirements if any), and a list of Event Handlers.

Events may be internal or external and represent channels of event streams, through which individual event instances may be generated.

Internal Events are generated by an Event Handler. Internal Events have timing requirements, a Global Deadline relative to the arrival of a Referenced External Event. The MAST language also allows using Local Deadlines, relative to the arrival of the event that activated that Event Handler. All of our deadlines are Hard Deadlines, e.g. they must be met in all cases, including the worst case.

External events model the interactions of the system with external components or devices through interrupts, signals, etc., or with hardware timing devices. They have a double role in the model: on the one hand, they establish the rates or arrival patterns of activities in the system. On the other hand, they provide references for defining global timing requirements. MAST supports different arrival patterns, of which we used the following: *Periodic* represents a stream of events that are generated periodically, such as from the Tick Scheduling; *Sporadic* as a stream of aperiodic events that have a minimum interarrival time.

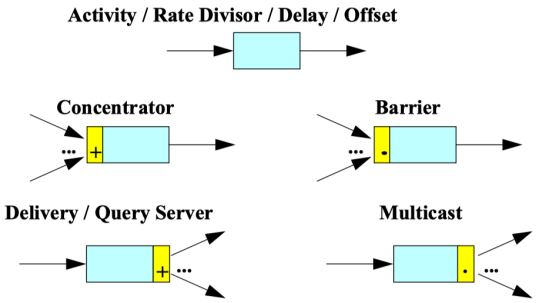

Figure 8: Event Handlers

Event Handlers in figure 8 represent actions that are activated by the arrival of one event, and that in turn generate one or more events at their output. There are two fundamental classes of Event Handlers. The Activities represent the execution of an operation by a Scheduling Server (a task), in a processing resource (the CPU). The other kinds of Event Handlers are just a mechanism for handling events, with no runtime effects. In the model, we have used the following classes:

- *Activity*: an instance of an operation to be executed by a Scheduling Server;

- *System Timed Activity*: an activity that is activated by the system timer, and thus is subject to the aforementioned jitter associated with it;
- *Multicast*: it is an event handler that generates one event in every one of its outputs each time an input event arrives;
- *Rate Divisor*: it is an event handler that generates one output event when a number of input events equal to the Rate Factor have arrived;
- *Offset*: an event handler that generates its output event after a time interval has elapsed from the arrival of some (previous) external event. If the time interval has already passed when the input event arrives, the output event is generated immediately.

We now proceed to model the three transactions which model the respective independent tasks. We will start with an initial analysis of the system as stand-alone tasks, then compare its maximum utilisation with the model using dynamic offsets to represent dependant tasks.

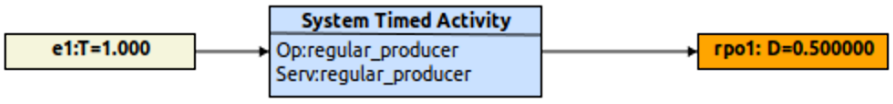

Figure 9: Regular_Producer transaction

```
Transaction (
   Type             => regular,
   Name             => rp_transaction,
   External_Events =>
      ( ( Type       => Periodic,
          Name       => e1,
          Period     => 1.000,
          Max_Jitter => 0.000,
          Phase      => 0.000)),
   Internal_Events =>
      ( ( Type => Regular,
          Name => rpo1,
          Timing_Requirements =>
            ( Type             => Hard_Global_Deadline,
              Deadline         => 0.500000,
              Referenced_Event => e1))),
   Event_Handlers  =>
      ( (Type               => System_Timed_Activity,
         Input_Event        => e1,
         Output_Event       => rpo1,
         Activity_Operation => regular_producer,
         Activity_Server    => regular_producer)));
```

The main event stream is modelled as a transaction activated by the periodic system timer, with a period of 1s. The event is handled by the `regular_producer` operation, representing a job of the same name. The Event Handler is of type `System_Timed_Activity` to take into account the jitter caused by the tick scheduling.

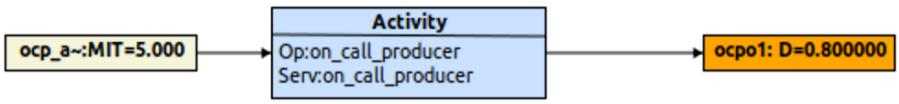

Figure 10: On_Call_Producer transaction

```
Transaction (
   Type             => regular,
   Name             => ocp_transaction,
```

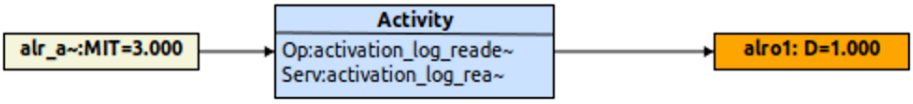

Figure 11: Activation_Log_Reader transaction

```
   External_Events =>
      ( ( Type             => Sporadic,
          Name             => ocp_activation,
          Min_Interarrival => 3.000)),
   Internal_Events =>
      ( ( Type => Regular,
          Name => ocpo1,
          Timing_Requirements =>
            ( Type             => Hard_Global_Deadline,
              Deadline         => 0.800000,
              Referenced_Event => ocp_activation))),
   Event_Handlers  =>
      ( (Type               => Activity,
         Input_Event        => ocp_activation,
         Output_Event       => ocpo1,
         Activity_Operation => on_call_producer,
         Activity_Server    => on_call_producer)));
```

The sporadic On_Call_Producer event stream is modelled as activated by a bounded aperiodic event, with a minimum interarrival time of 3s. Similar modelling has been done for the Activation_Log_Reader sporadic task.

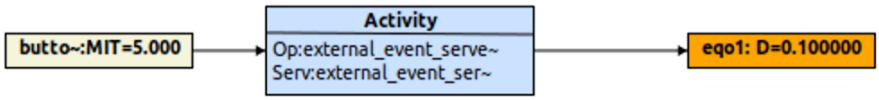

Figure 12: External push-button transaction

```
Transaction (
   Type            => regular,
   Name            => interrupt_transaction,
   External_Events =>
      ( ( Type             => Sporadic,
          Name             => button_click,
          Avg_Interarrival => 0.000,
          Distribution     => UNIFORM,
          Min_Interarrival => 5.000)),
   Internal_Events =>
      ( ( Type => Regular,
          Name => eqo1,
          Timing_Requirements =>
            ( Type             => Hard_Global_Deadline,
              Deadline         => 0.100000,
              Referenced_Event => button_click))),
   Event_Handlers  =>
      ( (Type               => Activity,
         Input_Event        => button_click,
         Output_Event       => eqo1,
         Activity_Operation => external_event_server,
         Activity_Server    => external_event_server)));
```

The push-button interrupt event stream is modelled as triggered by a sporadic event of 5s as minimum interarrival time and it's handled by the `external_event_server` job at software priority. The interrupt handler at hardware interrupt priority has not been modelled since it's execution time is negligible.

5.2 MAST analysis

As of the time of writing, MAST is at version 1.5.1 and supports the analysis tools [7] in Figure 13. The techniques relevant for this paper are:

Table 1. Fixed-priority schedulability analysis tools

Technique	Single-Processor	Multi-Processor	Simple Transact.	Linear Transact.	Multipath Transact.
Classic Rate Monotonic	☑		☑		
Varying Priorities	☑		☑	☑	
Holistic	☑	☑	☑	☑	☑
Offset Based	☑	☑	☑	☑	

Table 2. EDF schedulability analysis tools

Technique	Single-Processor	Multi-Processor	Simple Transact.	Linear Transact.	Multipath Transact.
Single Processor	☑		☑		
EDF_Within_Priorities	☑		☑		
Holistic_Local	☑	☑	☑	☑	☑
Holistic_Global	☑	☑	☑	☑	☑
Offset Based	☑	☑	☑	☑	

Figure 13: MAST analysis tools [7].

- *Classic RM Analysis*: it implements the classic exact response time analysis for single-processor fixed-priority systems and corresponds to the Technique "Calculating response time with arbitrary deadlines and blocking" in [26]. Although it's called Rate Monotonic, it bases on the final work by Tindell regarding Deadline Monotonic analysis to include jitters [23];
- *Holistic Analysis*: this analysis extends the response time analysis to multiprocessor and distributed systems. It is not an accurate analysis because it makes the assumption that tasks of the same transaction are independent. It was first developed for fixed priority systems by Tindell and Clark [24]. It has no use for our purposes, but it is worth mentioning to the reader because it can support both FPS and EDF monoprocessor and has fewer restrictions compared to *Classic RM Analysis*, as explained below. In terms of our example application, both techniques provide equivalent results;
- *EDF Monoprocessor / Single Processor*: it implements the exact response time analysis for single-processor EDF systems first developed by Spuri [39];
- *Offset Based Approximate Analysis*: this is a response time analysis for multiprocessor and distributed systems that improves the pessimism of the holistic analysis by taking into account that tasks of the same transaction are not independent, through the use of offsets. Offset based analysis for fixed priorities was first introduced by Tindell [37] and then extended to distributed systems by Palencia and González [36];
- *Offset Based Approximate with Precedence Relations Analysis*: this is an enhancement of the offset based approximate analysis for fixed priority systems in which the priorities of the tasks of a given transaction are used together with the precedence relations among those tasks to provide a tighter estimation of the response times;
- *Offset Based Slanted Analysis*: this is another enhancement of the offset based approximate analysis for fixed priority systems in which the maximum interference function is defined with a tighter approximation. This

method provides better results that the Offset-Based Approximate Analysis, but it is uncertain if it gets better results than the method with precedence relations.

In addition, the analysis tools are subject to different restrictions [8]. The most significant ones are:

- *No_Hard_Local_Deadlines*: Hard Local Deadlines cannot be used as Timing Requirements;
- *Referenced_Events_Are_External_Only*: no internal events can be referenced by Global Deadlines;
- *Simple_Transactions_Only*: checks that every transaction has only a continuous sequence of activities executed by the same server. This restriction is required by the Rate Monotonic analysis and the EDF Monoprocessor analysis;
- *Linear_Plus_Transactions_Only*: less restrictive than *Simple_Transactions_Only*, checks that every transaction only has one external event and is not multipath, e.g. it has no Multicasts. This restriction is required by the Holistic Analysis and the different Offset based analysis tools;
- *Restricted_Multipath_Transactions_Only*: checks that every transaction has a single input event, has no branch elements (Delivery or Query Servers), and has no Rate Divisors. It also checks that the transaction follows the set of allowed constructs mentioned in [8]. This restriction is required by the Holistic analysis.

As final note, as of the time of writing, offset-based analysis with EDF tasks fallbacks to holistic analysis [7], which in turn does not support shared resources in EDF yet.

6 FPS analysis

We start the analysis with fixed priority scheduling (FPS) and a MAST model which represents the tasks as stand-alone. Later, we will try to more strictly model the formal transactions comprised of dependent tasks.

To check that the system meets the deadlines, it suffices to run it for at least the first hyperperiod amount of time. Assuming sporadic tasks as periodic with a period equal to the minimum interarrival time, which is the worst case, the hyperperiod is $LCM(1, 3, 5) = 15s$. The hyperperiod of a set of tasks is least common multiple of all periods.

6.1 Independent tasks

We start with the Rate Monotonic analysis of the initial system.

```
Optimum Resource Ceilings:
request_buffer =>  7
activation_log =>  11
event_queue =>  241
```

A first analysis suggests that smaller values can be used as ceilings for the protected objects Request_Buffer and Activation_Log. This possible improvement is expected since the two values are the highest priorities of the tasks Regular_Producer and External_Event_Server, respectively. We leave the ceilings intact nevertheless since the ceiling is required to be an upper bound of the priorities between the tasks that request the resource, not the least upper bound. Having some spare priorities between the task priorities and the ceilings might prove to be useful if we need to separate a task into two distinct tasks with proper offset to better model the application [37].

Task	WCET			
regular_producer	0.019333			
on_call_producer	0.007126			
activation_log_reader	0.003582			
Transaction	R_{max}	**Slack**	**Worst blocking time**	**Jitter**
rp_transaction	0.020434	2470.0%	2.000E-06	0.001101
ocp_transaction	0.026592	10776.6%	1.000E-06	0.019472
alr_transaction	0.030195	26600.8%	0.00	0.026613
interrupt_transaction	2.102E-05	>=100000.0%	1.000E-06	1.102E-05
System slack	2401.2%			
Total utilisation	2.68%			

Table 3: Rate Monotonic analysis results for FPS

Table 3 first shows the WCET of the tasks as defined in the MAST model and which are controlled by the Whetstone workloads. Then the results of the analysis are displayed, containing the worst-case response time R_{max}, the slack, the blocking time and the jitter for each transaction. The MAST analysis tool also provides best-case response times R_{min}, but Rate Monotonic is a pessimistic analysis which assumes worst-case scenario at the critical instant, therefore only worst-case response time matters.

All transactions suffer jitter due to the system ticker interrupt running at the highest interrupt priority and the context switch overhead.

- *regular_producer*: suffers additional jitter due to the system clock with granularity 1ms and the possible execution of the interrupt handler. Its blocking time is caused by the On_Call_Producer and the Activation_Log_Reader which have lower priorities but can access resources with higher ceiling priority than Regular_Producer;
- *ocp_transaction*: suffers additional jitter due to interference by Regular_Producer. Its blocking time is caused by the Activation_Log_Reader;
- *alr_transaction*: suffers additional jitter due to interference by On_Call_Producer and Regular_Producer. It has no blocking time since it's the task with the lowest priority;
- *interrupt_transaction*: it's the software level handler of the interrupt. It suffers no additional jitter other than the aforementioned overheads. Its blocking time is caused by the Activation_Log_Reader;

We shall now increase the Whetstone workload of factor 24 in the first three transactions since 2477.0% is the smallest slack of the three of them. The factor corresponds to how much the execution time of all event responses can be increased while preserving system schedulability [28]. We leave the interrupt_transaction intact because it doesn't contain any Whetstone operation. The new results are as shown in table 4.

Task	WCET			
regular_producer	0.482597			
on_call_producer	0.177482			
activation_log_reader	0.088702			
Transaction	**R_{max}**	**Slack**	**Worst blocking time**	**Jitter**
rp_transaction	0.485486	2.73%	2.000E-06	0.002889
ocp_transaction	0.662657	76.95%	1.000E-06	0.485175
alr_transaction	0.751706	277.34%	0.00	0.663004
interrupt_transaction	2.102E-05	>=100000.0%	1.000E-06	1.102E-05
System slack	3.21%			
Total utilisation	57.52%			

Table 4: Rate Monotonic analysis results for FPS increased of factor 24

The blocking times have not changed because protected operations are the same as before, but the total utilisation has increased up to 57.52%. The 2.73% slack value of Regular_Producer is already very low, so we can leave it as it is. We proceed instead to increase the workload of On_Call_Producer and Activation_Log_Reader from factor 24 to $43 = 24 * 1.77$, using the slack value 77%.

Task	WCET			
regular_producer	0.482597			
on_call_producer	0.312341			
activation_log_reader	0.156086			
Transaction	**R_{max}**	**Slack**	**Worst blocking time**	**Jitter**
rp_transaction	0.485486	0.390625%	2.000E-06	0.002889
ocp_transaction	0.798039	0.390625%	1.000E-06	0.485698
alr_transaction	0.954730	28.13%	0.00	0.798644
interrupt_transaction	2.102E-05	15421.1%	1.000E-06	1.102E-05

System slack	0.390187%
Total utilisation	64.26%

Table 5: Rate Monotonic analysis results for FPS increased of factor 43

The system has reached utilisation 64.26%. We now increase the workload of Activation_Log_Reader from factor 43 to $55 = 43 * 1.28$, using the slack value 28%.

Task	WCET			
regular_producer	0.482597			
on_call_producer	0.312341			
activation_log_reader	0.198645			
Transaction	**R_{max}**	**Slack**	**Worst blocking time**	**Jitter**
rp_transaction	0.485492	0.0%	2.000E-06	0.002889
ocp_transaction	0.798039	0.390625%	1.000E-06	0.485698
alr_transaction	0.997454	0.390625%	0.00	0.798809
interrupt_transaction	2.102E-05	15421.1%	1.000E-06	1.102E-05
System slack	0.390187%			
Total utilisation	65.68%			

Table 6: Rate Monotonic analysis results for FPS increased of factor 55

The maximum utilisation reached is about 65.68%. The only transaction with significant slack left is `interrupt_transaction`, but tests show that an increase of the WCET of factor 154 in the operation *external_event_server* would improve the utilisation only up to 65.71%, hence we can ignore it.

6.2 Adding offsets

So far, tasks have been assumed to be scheduled independently, there are no relationships between the release of any pair of tasks. Consequently, the worst-case task release pattern has been assumed in the critical instants [12]; the resulting analysis is, therefore, sufficient for any task release pattern. It may, however, be advantageous to specify timing constraints on release patterns. We try to include time offsets into the computational model and, by taking account of time offsets, we try to reduce the pessimism when bounding the timing behaviour of the system.

By assuming all tasks are independent, the current analysis is subject to two pessimistic points:

1. Critical instant: for tasks with offsets, we must take into account that the critical instant may not include the simultaneous activation of all higher priority tasks, as it was the case when all tasks were independent. The existence of offsets makes it impossible for some sets of tasks to become active simultaneously [36];
2. Blocking time: offsets can be used to avoid the need for a dynamic concurrency control protocol for access to shared resources. Two tasks in the same transaction may not need to use locks to guard access to a shared resource if certain constraints on response times and offsets hold [37].

The above pessimism can be avoided by modeling the precedence constraint: within a pair of tasks, one of them must complete execution before the other can be permitted to commence. If it can be shown that two tasks execute in exclusion, then any resources shared exclusively between these tasks need not be guarded by locks, the tasks are guaranteed never to access the shared resource concurrently. Besides, the two tasks cannot be active concurrently, which means that neither task can be permitted to preempt the other, causing interference in the critical instant.

6.2.1 Critical instant

Offsets can be used to express precedence constraints: the existence of offsets makes it impossible for some sets of tasks to become active simultaneously. This is achieved by "spreading out" the computation of tasks so that all the tasks are not released together.

In our system, the task On_Call_Producer (OCP) is actually not truly sporadic given that it's activated by the Regular_Producer (RP) every 3 jobs. The two tasks are not thus independent and both the RP job which activates OCP

and the latter should belong to the same transaction. The equivalent argument holds true for the RP job that awakens Activation_Log_Reader (ALR). The remaining instances of the RP tasks should belong to another transaction again.

Formally, with a precedence constraint, for two tasks τ_C and τ_D that are members of the same transaction, τ_C must complete before task τ_D is run. We have in theory two possible priority situations: task τ_C is of higher priority, or task τ_D is of higher priority. Fortunately, in our system, $\tau_C = RP, \tau_D = OCP$ and RP is the one of higher priority, which means that task OCP (of lower priority) will simply not execute if task RP has been released before task OCP and has remaining computation. It can be seen that the condition for the precedence constraint to be met is [37]:

$\Phi_C + J_C \leq \Phi_D \Leftrightarrow \Phi_{RP} + J_{RP} \leq \Phi_{OCP}$

Otherwise, if task τ_D were of higher priority, we could not use the priority mechanism to ensure precedence and would rely on offsets. Therefore, for the precedence relation to hold, the latest finish time of τ_C must be before the earliest release time of τ_D, i.e.:

$O_C + r_c < O_D$, where r_c is the response time of C

MAST Offset-based analysis tools are able to derive such conditions from the following representations of the newly described transactions.

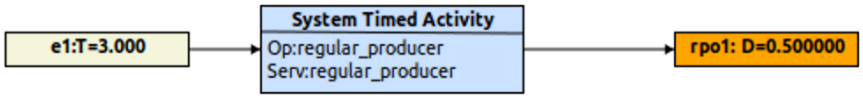

Figure 14: Transaction A

The transaction A has a period of 3, representing the stand-alone RP instance.

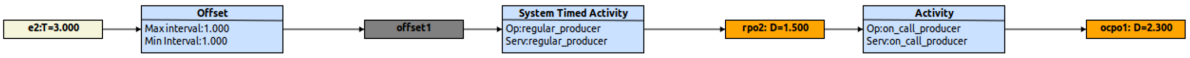

Figure 15: Transaction B

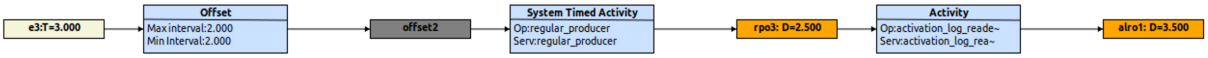

Figure 16: Transaction C

The transactions B and C also have a period of 3 and include the RP job which activates the OCP and ALR jobs respectively. Both transactions have an initial offset from the external periodic event to differentiate each RP instance from the others.

The unfolding of the three transactions reproduces the timeline in Figure 17, assuming the Whetstone values reached by the previous analysis. It is clear from the timeline that there is a possibility for the OCP and ALR jobs to further expand their executions without provoking any deadline miss. Nevertheless, the pessimistic Rate Monotonic analysis returns zero slack because of the possible interference between stand-alone tasks.

As can be seen below, the Offset Based Slanted analysis provides an improvement of the best response times. Offset-based analysis tools are able to consider the offset values, so that chained tasks are not released together.

Task	WCET
regular_producer	0.482597
on_call_producer	0.312341
activation_log_reader	0.198645

Transaction	R_{min}	R_{max}	Slack	Worst blocking time	Jitter

transaction_a	0.482597	1.454	-66.02%	2.000E-06	0.971820
transaction_b RP	1.483	2.454	-66.02%	2.000E-06	0.971820
transaction_b OCP	1.795	3.737	-66.02%	1.000E-06	1.942
transaction_c RP	2.483	3.454	-66.02%	2.000E-06	0.971820
transaction_c ALR	2.681	4.936	-66.02%	0.00	2.255
interrupt_transaction	1.000E-05	2.102E-05	-100.0%	1.000E-06	1.102E-05
System slack	-65.55%				
Total utilisation	65.68%				

Table 7: Offset Based Slanted analysis results

Both the R_{min} and R_{max} response times produced by the analysis include the initial offsets of the transaction, but the former RP R_{min} values are also used as offset Φ_{ij} of the dependant tasks OCP and ALR for their own response times. The R_{max} values are instead the sum of the corresponding R_{min} and jitter.

Between R_{min} and R_{max} response times, we consider only the best-case response time R_{min}. The timeline in Figure 17 clearly shows that the three transactions have the same source of activation, a RP job, and there is no interference between tasks. Thus there cannot be any significant jitter and only the R_{min} value is valuable for our considerations since it already takes into account the offsets and examines the actual case with zero interference. Besides, by not considering the worst-case response times, we ignore the slack values as well. If the best-case response time is within the deadline, then it's sufficient for our considerations.

Task	WCET
regular_producer	0.482597
on_call_producer	0.312341
activation_log_reader	0.198645

Transaction	R_{min}	R_{max}	Slack	Worst blocking time	Jitter
transaction_a	0.482597	1.454	-66.02%	2.000E-06	0.971820
transaction_b RP	1.483	2.454	-66.02%	2.000E-06	0.971820
transaction_b OCP	1.795	**2.768**	-66.02%	1.000E-06	**0.973028**
transaction_c RP	2.483	3.454	-66.02%	2.000E-06	0.971820
transaction_c ALR	2.681	**3.967**	-66.02%	0.00	**1.286**
interrupt_transaction	1.000E-05	2.102E-05	-100.0%	1.000E-06	1.102E-05
System slack	-65.55%				
Total utilisation	65.68%				

Table 8: Offset Based Approximate with Precedence Relations analysis results

Compared to Offset Based Slanted, Offset Based Approximate with Precedence Relations analysis is able to provide even tighter worst-case response times by using the precedence relation and therefore considering a dynamic offset $\Phi_{i2} \in [R_{ISR\ min}, R_{ISR\ max}]$. I.e. the OCP task is never activated before the RP completion, therefore its jitter can be approximately reduced to $J_{B,OCP} = J_{B,RP}$.

Future analysis in this section will provide only the results from the Offset Based Approximate with Precedence Relations technique, referred only as Offset-based analysis, as it has proved to be the best of the two tools even if are interested only in best-case response times.

6.2.2 Blocking time

We can further improve the analysis by reducing unnecessary blocking time: the precedence constraint between the RP and OCP means the former cannot suffer blocking time from the latter. There is also no need to use any lock to guard access to the Request_Buffer resource given that concurrent access is not possible.

By removing the resource lock on Request_Buffer, we obtain the results in table 9 for the three transactions.

The IPCP ensures that each task can be blocked at most once, at its beginning, by a single lower-priority task [5]. Then the removal of the lock on Request_Buffer reduces the maximum blocking time suffered the tasks to a value equal to the execution time of the lowest priority Activation_Log_Reader within the shared resource Activation_Log.

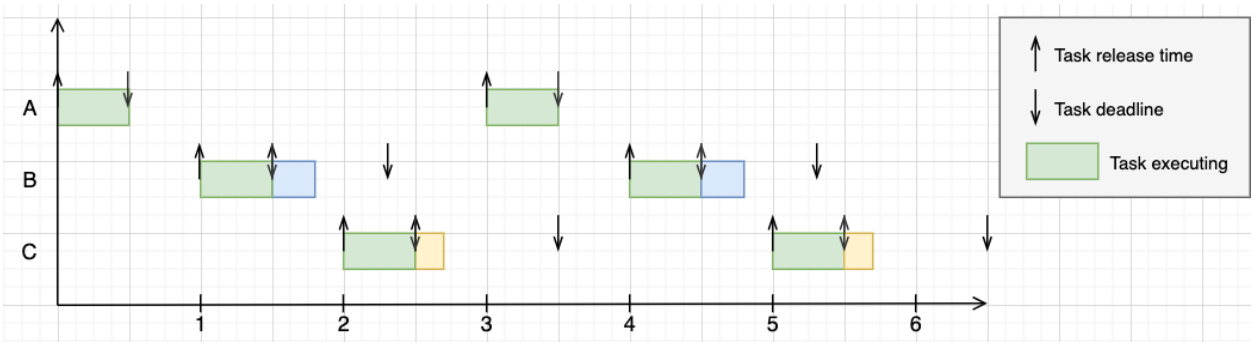

Figure 17: Unfolded timeline

Task	WCET		
regular_producer	0.482597		
on_call_producer	0.312341		
activation_log_reader	0.198645		
Transaction	R_{min}	**Worst blocking time**	**Jitter**
transaction_a	0.482597	**1.000E-06**	0.971820
transaction_b RP	1.483	**1.000E-06**	0.971820
transaction_b OCP	1.795	1.000E-06	**0.973028**
transaction_c RP	2.483	**1.000E-06**	0.971820
transaction_c ALR	2.681	0.00	**1.286**
interrupt_transaction	1.000E-05	1.000E-06	1.102E-05
Total utilisation	65.68%		

Table 9: Offset-based analysis without `request_buffer`

Further in-depth analysis is presented later in Section 7.4, which gathers the observations for both FPS and EDF blocking times actually suffered by our application.

6.2.3 Maximum utilisation

Leveraging the newly defined offset-based model, we can try to achieve maximum system utilisation. By having a look at the unfolded timeline in Figure 17, it is evident that RP has already reached its maximum workload. However OCP can still increase up to approximately 0.5s and likewise ALR. Bearing in mind the possible jitter caused by the system ticker ($0.5/0.001 * 3.844E-06 = 0.001922$), both tasks can raise their execution time way to $0.5 - 0.001922 = 0.498078$.

Actually, compared to the Rate Monotonic Analysis, RP can be increased by a tiny amount to get the WCET up to 0.5s as well. The final offset-based FPS analysis with maximum utilisation is displayed in Table 10.

All the best-case response times are within the deadlines and the concluding maximum utilisation, with proven runtime feasibility, of the FPS system is approximately 83.28%. If we flatten the final transactions onto a single timeline, we obtain Figure 18. Within the hyperperiod of 3 seconds, or equivalently 6 blocks of 0.5seconds, there is only a single block of 0.5s of task idleness. The theoretical utilisation is then $5/6 = 0.8\bar{3}\%$, very close to the value provided by the MAST analysis.

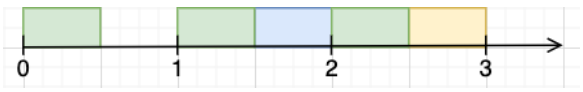

Figure 18: Flattened timeline

Analytically the system utilisation is approximately $\sum_{i \in \{RP,OCP,ALR\}} \frac{C_i}{T_i} = \frac{0.5}{1} + \frac{0.5}{3} + \frac{0.5}{3} = \frac{5}{6}$. Runtime overhead, such as system ticker or context switch, and also blocking times have been left out the formula given that they are negligible in the approximation.

Task	WCET	
regular_producer	0.496961	
on_call_producer	0.497936	
activation_log_reader	0.497844	
Transaction	**R_{min}**	**Worst blocking time**
transaction_a	0.497003	1.000E-06
transaction_b RP	1.497	1.000E-06
transaction_b OCP	1.995	1.000E-06
transaction_c RP	2.497	1.000E-06
transaction_c ALR	2.995	0.00
interrupt_transaction	1.000E-05	1.000E-06
Total utilisation	83.28%	

Table 10: Offset-based analysis with max utilisation

7 EDF Analysis

The Earliest Deadline First (EDF) [19] scheduling is dynamic-priority algorithm that assigns priorities to individual jobs of a task based on its absolute deadline. In particular, the earlier the deadline, the higher the priority. The absolute deadline of a job is computed as the sum of its release event plus its relative deadline. Such relative deadline is a static attribute, missing in FPS algorithm and corresponding to the maximum response time allowed to each task instance.

The runtime we have adopted is an Ada Ravenscar runtime variant implementing an EDF scheduling coupled with the Deadline Floor Protocol (DFP) [42] as resource locking policy.

DFP is the EDF counterpart of the Immediate Priority Ceiling Protocol (IPCP), outlined in Section 1.2. Indeed, rather than assigning a ceiling priority to each shared resource r_i, a deadline floor value D^i is computed as the minimum relative deadline of any task accessing such resource.

Besides, instead of raising the priority of a task to the resource's ceiling, when a task τ_i released at time s accesses resource r_i at time t (so $s < t$) its relative deadline is immediately reduced to D^i. As a result, its active absolute deadline d_i is also (potentially) reduced to $d_i \leftarrow min\{t + D^i, s + D_i\}$. Finally, when the task frees the resource, its deadline immediately returns to its original value.

Clearly, the IPCP for fixed-priority systems and DFP for dynamic-priority systems are structurally equivalent. FPS uses a ceiling value as dispatching urgency, whereas a floor value is expected as earlier deadline under EDF.

7.1 A MAST model for EDF

Unfortunately, as mentioned in Section 5.2, MAST requires even stricter restrictions for EDF analysis tools than FPS. The Offset-based analysis is not yet available for EDF at the time of writing and rollbacks to the Holistic analysis, which in turn does not yet support shared resources. Therefore, we had to discard both Offset-based and Holistic analysis tools.

The last standing option was the EDF Monoprocessor tool, which implements the exact response time analysis for single-processor in EDF systems. Unfortunately, the *Simple_Transaction_Only* restriction (§5.2) forbids definining a transaction comprising a sequence of activities executed by different Execution Servers. Hence, dependency constraints between the task releases could not be expressed.

In addition, the EDF Monoprocessor analysis is able to support only Stack Resource Policy (SRP) [42] as resource access protocol. Under SRP, each job is assigned a preemption level $\pi(\tau_i)$ inversely proportional to the task relative deadline D_i, e.g. $\pi(\tau_i) < \pi(\tau_j) \Leftrightarrow D_i > D_j$.

In turn, each resource is assigned a ceiling preemption level $\Pi(r_i)$ defined as the maximum preemption level of any job that may access it. After defining the system ceiling $\hat{\pi}$ as the highest ceiling of all the resources which are held by some job at any time t, the following definition of the SRP locking policy is provided. A job j_i released at time t can start execution only if:

- the absolute deadline of this job ($t + D_i$) is the earliest deadline of the active requests in the task set;
- its preemption level is higher than the system ceiling $\pi(r_i) > \hat{\pi}$.

Nevertheless, the disparity in resource access control protocols between runtime implementation and MAST can be ignored because of the worst-case bound equivalence between SRP and DFP [42]. Indeed they lead to even the same worst blocking time analysis.

As a result, within the MAST model, we must provide the preemption level for each task according to the SRP. An available assignment is displayed in Table 11.

Task name	Deadline (ms)	Preemption Level
External_Event_Server	100	40
Regular_Producer	500	30
On_Call_Producer	800	20
Activation_Log_Reader	1000	10

Table 11: Preemption Levels for the task set

According to the newly defined preemption levels, Table 12 shows a correct definition of the resource ceilings.

Resource name	Ceiling Preemption Level
Request_Buffer	30
Activation_Log	40
Event_Queue	50

Table 12: Ceiling Preemption Levels for the resource set

7.2 EDF Monoprocessor analysis

The EDF Monoprocessor tool is based on the formal analysis developed by Spuri [39], which considers the busy period to study the feasibility of the schedule. As depicted in Figure 19, the worst-case response time (WCRT) of a task τ_i is found in a busy period $[t_1, t_2]$ in which all other tasks are released synchronously at $t = 0$ and then at their maximum rate. Such busy period is characterized by the job j_i released at time $t = a$, $a \geq 0$, preceded by other jobs of any task which do not let the CPU idle, possible by other instances of task τ_i itself. t_1 is the first instant preceding the release of j_i without CPU idleness, whereas t_2 is the completion time of the job i under consideration.

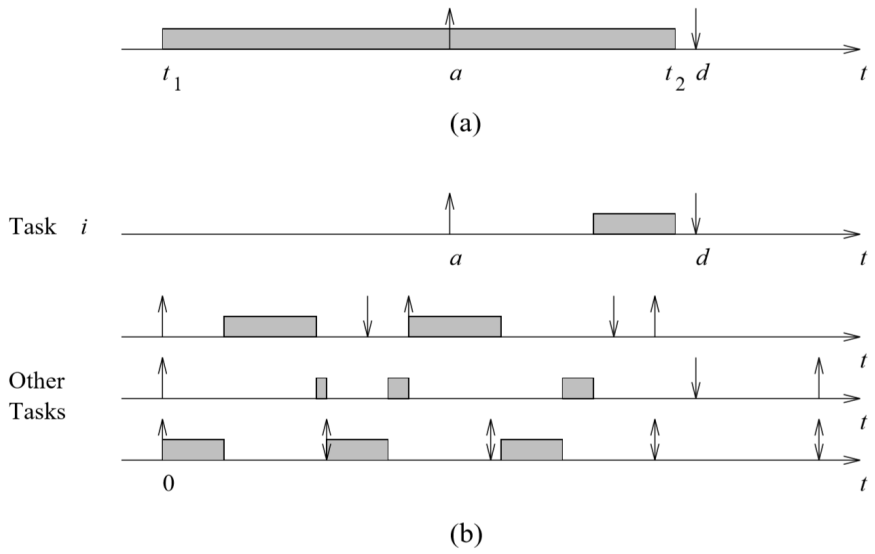

Figure 19: Busy period (a) leading to the job j_i WCRT (b) [39]

With this being said, the transactions of this model are precisely those in the FPS model with the independent task set as outlined in §6.1. Thus we still have four transactions, each one composed of a single activity.

As workload values, we have set the best values reached by the Offset-based analysis and fed the model to the EDF Monoprocessor analysis to compare the two tools. The results for the latter are pasted in Table 13.

Task	WCET			
regular_producer	0.496961			
on_call_producer	0.497936			
activation_log_reader	0.497844			
Transaction	**R_{max}**	**Slack**	**Worst blocking time**	**Jitter**
rp_transaction	0.992883	-99.22%	2.000E-06	0.495880
ocp_transaction	1.293	-99.22%	2.000E-06	0.794917
alr_transaction	1.493	-100.00%	2.000E-06	0.995006
interrupt_transaction	0.592882	-100.00%	1.000E-06	0.592872
System slack	-32.98%			
Total utilisation	82.90%			

Table 13: EDF monoprocessor analysis results

As expected, this is an unfair comparison because a busy period in which all tasks but one are released synchronously leads to a great pessimism in the worst-case response time. Figure 20 presents the worst arrival pattern considered by MAST to cause the WCRT of Regular_Producer. However, because of the dependencies between release events, RP will never compete for the CPU with ALR because it is the completion of the former which provokes the release event of latter and without overloading the two tasks are never active simultaneous.

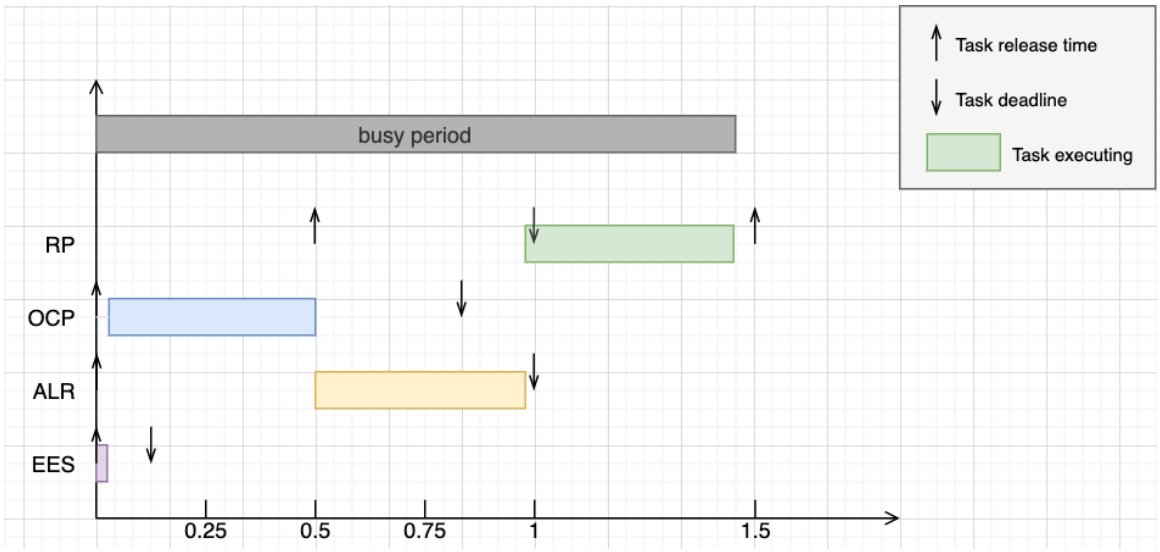

Figure 20: Busy period leading to RP's WCRT

A more fair comparison would pick Rate Monotonic analysis as the FPS counterpart as response time analysis tool for single-processor since both assume tasks to be standalone. Indeed, providing the best workloads achieved previously via Rate Monotonic, EDF Monoprocessor prints the results in Table 14. The slack values are close to the previous FPS results and the reason can be found again in Figure 20. Assuming this pessimistic case, no further significant improvement can be made to any task workload without missing a deadline.

We believe instead that the slight slack improvement is due to the absence of the system ticker in EDF analysis, that is it doesn't take into account the possible release jitters and interferences suffered by each task.

To sum up, MAST can ensure a relatively poor utilisation under EDF scheduling because we aren't allowed to maintain consistency between model and application. We may question what the real performance granted by such dynamic-scheduling is..

Task	WCET
regular_producer	0.482555
on_call_producer	0.312311
activation_log_reader	0.198612

Transaction	R_{max}	Slack	Worst blocking time	Jitter
rp_transaction	0.494969	0.781250%	2.000E-06	0.012372
ocp_transaction	0.794969	1.56%	2.000E-06	0.482628
alr_transaction	0.993620	3.13%	2.000E-06	0.794975
interrupt_transaction	0.094968	41928.5%	1.000E-06	0.094958

System slack	0.783430%
Total utilisation	65.29%

Table 14: EDF Monoprocessor analysis results

7.3 Runtime behaviour

Despite the pessimistic MAST analysis results, EDF is an optimal scheduler and can handle a total theoretic utilization up to 1. In addition, any feasible preemptive FPS schedule can be transformed into an EDF schedule without affecting its feasibility [39]. Hence, we expect the EDF runtime to perform at least as well as the FPS counterpart under the maximum workload reached with Offset-based analysis §6.2.3.

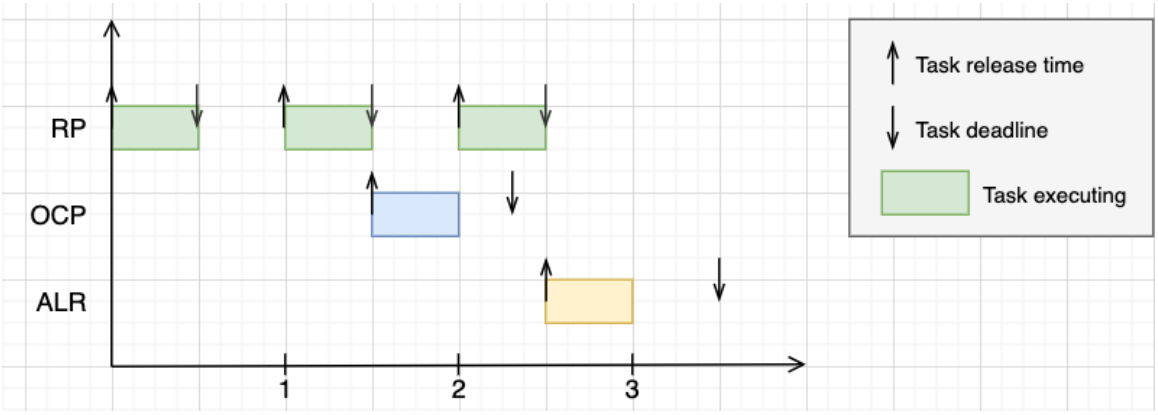

Figure 21: Application execution under EDF scheduling with maximum utilisation

Figure 21 exhibits the schedule produced by EDF up to the hyperperiod $H = 3$. The total utilisation of 83.28% is given by the Whetstone values in Table 15.

Task	WCET
regular_producer	0.496961
on_call_producer	0.497936
activation_log_reader	0.497844

Table 15: Whetstone values leading to maximum FPS and EDF utilisation

Despite the EDF optimality, by observing the aforementioned timeline, there is, unfortunately, no room for utilisation increase. RP already runs at zero laxity, whereas OCP and ALR cannot exceed an execution time of 0.5 seconds without causing a fatal release jitter to RP itself.

This is further proved by noting that both EDF and FPS algorithms lead to the same schedule of our application, under the condition that no task misses its deadline. FPS assigns priorities inversely proportional to the relative deadline D_i, whereas EDF calculates them inversely proportional to the absolute deadline d_i. In general, the relation $D_i < D_j \Leftrightarrow d_i < d_j$ does not hold, but in our specific case we can demonstrate it's valid.

Given a pair of task τ_i and τ_j, for the task τ_i with relative deadline $D_i > D_j$ to have absolute deadline $d_i < d_j$ it must be released at time $t_i < t_j$, i.e. for OCP to have a lower absolute deadline than RP. But under the condition that no task overruns, this situation is clearly not possible. Both OCP and ALR are always activated by the RP, and they complete before the next release of RP. This proves that, within our application with precedence relations, $D_i < D_j \Leftrightarrow d_i < d_j$ and therefore both algorithms assign the same priority.

7.4 Blocking Time

Further consideration can be made about the blocking time by reasoning with the timeline in Figure 21. The blocking time suffered by each job is equal to zero, except for one case, for that no higher-priority job will be suspended waiting for a lower-priority job to complete its use of a (non-preemptable) resource.

Since DFP is structurally equivalent to IPCP, there are two kinds of possible blocking [5].

1. Direct blocking, a situation in which a higher priority task is blocked by a lower-priority task which accesses a resource shared between the two of them. In our application, this may happen only between ALR and External_Event_Server (EES), which have independent release events. Direct blocking cannot happen between RP and OCP, again because they are never simultaneously active;
2. Push-through blocking happens when a medium priority task can be blocked by a lower priority task, which inherits the priority of a high priority task. In our analysis, the only plausible case would be RP being push-blocked by ALR, which receives the priority of EES. However, since ALR is activated when RP terminates and vice versa, this situation cannot happen between the two tasks.

Because of the comparability between PCP and DFP and the schedule correspondence between FPS and EDF with our taskset, the aforementioned observations about the blocking time can also be made for the fixed-priority application.

8 Overloading

A job is said to overrun when it executes for more than its guaranteed execution time. We say that a system is overloaded when it is not schedulable on the basis of the maximum execution times of its tasks, and hence it is likely that some jobs will miss their deadlines [40].

Any algorithm for scheduling jobs with potential for overrun should meet two criteria if it is to perform well. First, it should guarantee that jobs which do not overrun meet their deadlines and, second, the algorithm tries to maximize the number of deadlines met.

In this section, we compare the behaviour of our application under FPS and EDF during permanent overload situations, which occur in literature when the system utilisation $U > 1$. In our case, our limit is not the theoretical full CPU utilisation 1, we have seen we should consider 0.83 as limits for both FPS and EDF.

8.1 FPS overloading

When tasks have fixed priorities, overruns of jobs in a task can never affect higher-priority tasks, and it is possible to predict which tasks will miss their deadlines during an overload. Likewise, another equivalent point of view is that a permanent overload may cause a complete blocking of the lower priority tasks.

We have observed this behaviour in our tests by increasing the RP workload of a small amount $\epsilon = 0.02s$, reaching a WCET of approximately 0.52s.

```
Interrupt generated
Deadline Miss Detected -  RP
End of cyclic activation.
Deadline Miss Detected -  RP
```

```
End of cyclic activation.
Deadline Miss Detected - OCP
Deadline Miss Detected -  RP
End of cyclic activation.
End of sporadic activation.
Deadline Miss Detected -  RP
End of cyclic activation.
Deadline Miss Detected - ALR
End of parameterless sporadic activation.          1
```

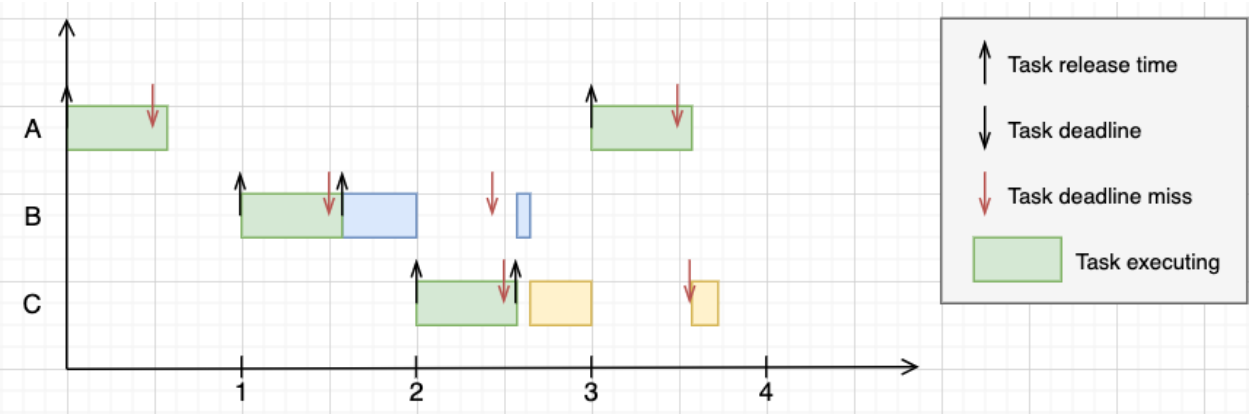

Figure 22: Timeline with overruning RP

As proved by the runtime log and shown in the timeline in Figure 22, an overruning RP affects all the lower-priority tasks and causes their deadline miss as well.

```
Interrupt generated
End of cyclic activation.
End of cyclic activation.
Deadline Miss Detected - OCP
End of cyclic activation.
Elapsed time: 0.530376517
End of sporadic activation.
End of cyclic activation.
Deadline Miss Detected - ALR
End of parameterless sporadic activation.          1
```

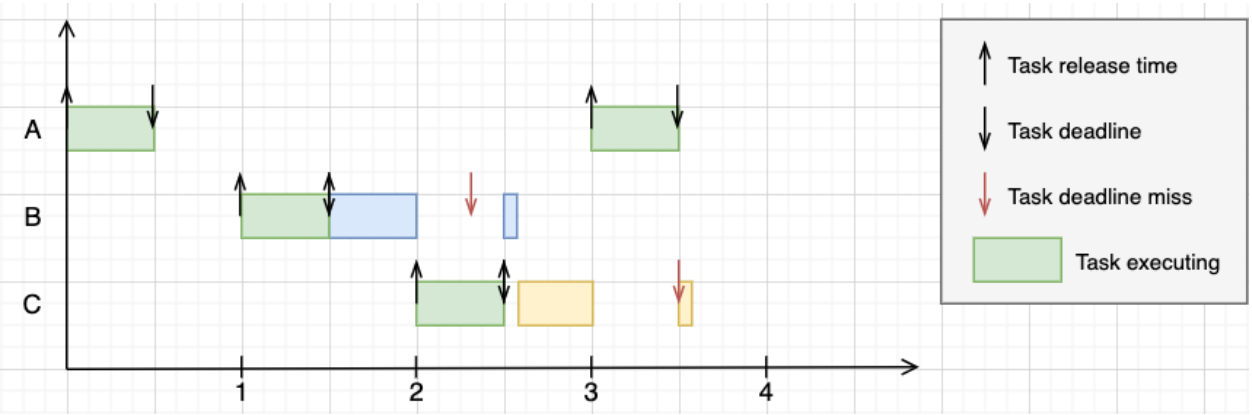

Figure 23: Timeline with overruning OCP

Runtime log and Figure 23 show that an overload of OCP never impacts the higher-priority task RP which can meet all its deadlines. ALR will instead miss its deadline once again.

8.2 EDF overloading

In literature, the EDF exhibits an unstable behaviour during an overload: a late EDF job which has already missed its deadline has a higher priority than a job whose deadline is still in the future. Consequently, if the execution of a late job is allowed to continue, it may cause the other jobs to be late.

```
Interrupt generated
Deadline Miss Detected -  RP
End of cyclic activation.
Deadline Miss Detected -  RP
End of cyclic activation.
End of sporadic activation.
Deadline Miss Detected -  RP
End of cyclic activation.
Deadline Miss Detected -  RP
End of cyclic activation.
Deadline Miss Detected - ALR
End of parameterless sporadic activation.          1
```

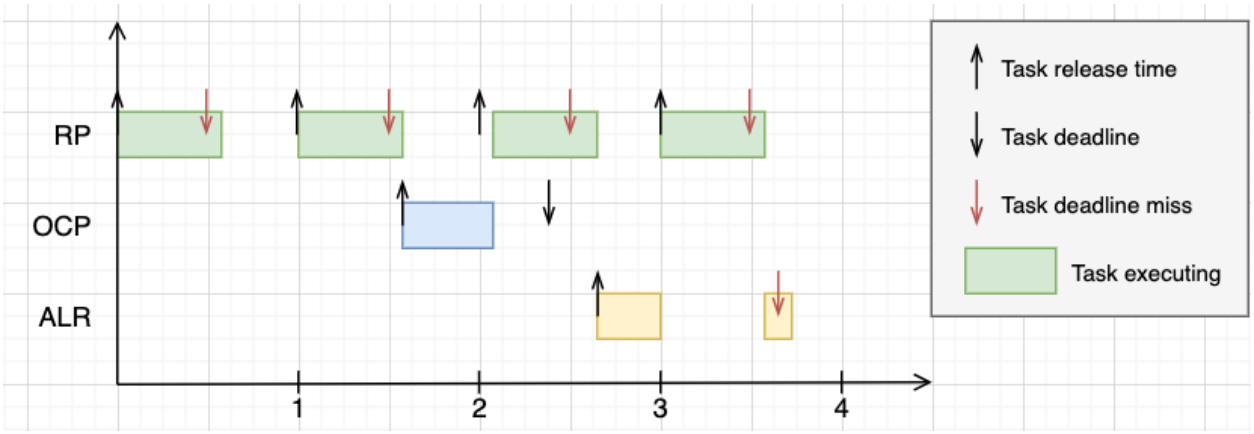

Figure 24: Timeline with overruning RP in EDF

A RP overload of $\epsilon = 0.02s$ causes the deadline miss of the ALR, as shown in the runtime log and Figure 24, but OCP seems to meet its deadline nevertheless. This behaviour is apparently in contrast with the FPS overload with the same overruning task. However, if we consider a more significant overrun of $\epsilon = 0.2$ and expand the timeline beyond the hyperperiod, an interesting pattern emerges.

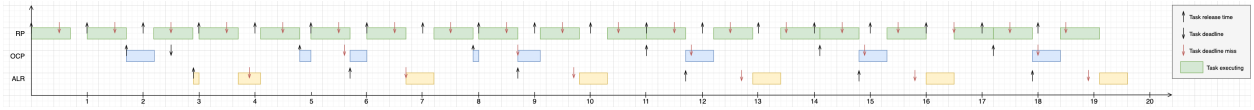

Figure 25: Extended timeline with overruning RP in EDF

All OCP jobs, except for the first instance, fail to meet the deadline and a regular pattern comprising the three tasks emerges as soon as the timeline goes beyond time instant 10. As time elapses from the first activation, OCP and ALR suffer an increasing interference from the overruning RP until time 10, beyond which the jitter is regular. This peculiar EDF behaviour, where there is an initial interval of irregularity followed by regular executions and events, is not exhibited by the analogous FPS algorithm with RP overruning of the same amount. In fixed-priority scheduling, all jobs miss their deadline, and the regularity emerges from the first instant of execution.

When the first task that misses its deadline causes all subsequent tasks to miss their deadlines, the effect is called *the domino effect* [45]. EDF is prone to the domino effect, and it rapidly degrades its performance during overload intervals. This is due to the fact that EDF gives the highest priority to those processes that are close to missing their deadlines. Even worse, we note that a late job which has already missed its deadline has a higher priority than a job whose deadline is still in the future [19].

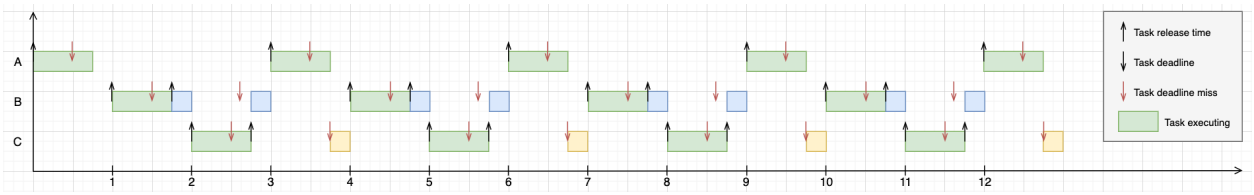

Figure 26: Extended timeline with overruning RP in FPS

The application under consideration doesn't seem to provoke domino effect. Although a permanent overload of RP causes all subsequent tasks to miss their deadline, it's trivial to show with a timeline that a transient overload of it causes a finite amount of subsequent deadline misses. Eventually, the system is able to recover as long as the system utilisation is below 1. The same property holds for an overruning OCP, as made evident by Figure 27. We believe this is again because of the precedence relationships. Even if RP misses a deadline, the later OCP and ALR are not activated independently. They wait for RP completion and are less subject to domino effect.

Nevertheless, a difference between FPS and EDF is that, under the former, a transient overrun in a task cannot cause tasks with higher priority to miss their deadlines, whereas under EDF any other task could miss its deadline. That is, the latter does not provide any type of guarantee on which tasks will meet their timing constraints.

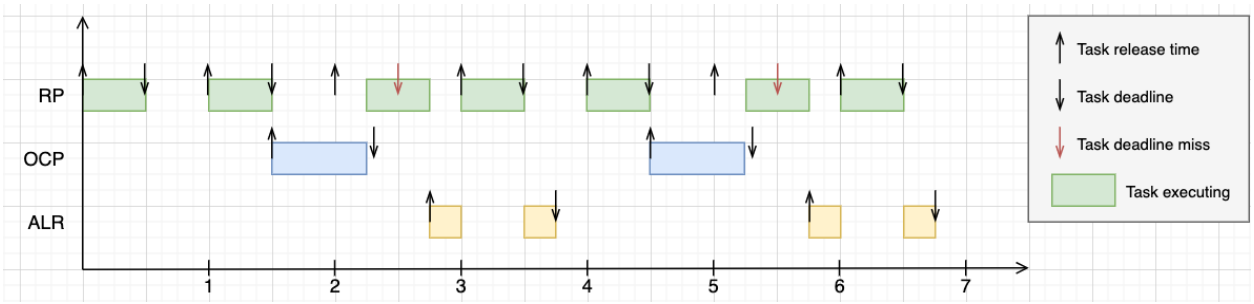

Figure 27: Timeline with overruning OCP in EDF

Lastly, an additional interesting property of EDF during permanent overloads is that it automatically performs a period rescaling, and tasks start behaving as they were executing at a lower rate [43]. The following theorem has been proven by Celvin et al. (2002) [44]:

Assume a set of n periodic tasks, where each task is described by a fixed period T_i, a fixed execution time C_i, a relative deadline D_i, and a release offset Φ_i. If U > 1 and tasks are scheduled by EDF, then the average period $\tilde{T}_i$ of each task τ_i is given by $\tilde{T}_i = T_i U$.

With our previous overruning case, the rescaling factor would be $U = (0.7/1 + 0.5/3 + 0.5/3) \approx 1.0\bar{3}$. According the to theorem, the tasks are executing with average periods $\tilde{T}_{RP} = T_{RP} * U = 1.0\bar{3} * 1 = 1.0\bar{3}$ and $\tilde{T}_{OCP} = \tilde{T}_{ALR} = 1.0\bar{3} * 3 = 3.1$. Indeed, it can be verified with the help of Figure 28 that in the repeated interval of 3.1 seconds, RP executes 3 times ($3.1/1.0\bar{3} = 3$), whereas OCP and ALR execute once respectively ($3.1/3.1 = 1$).

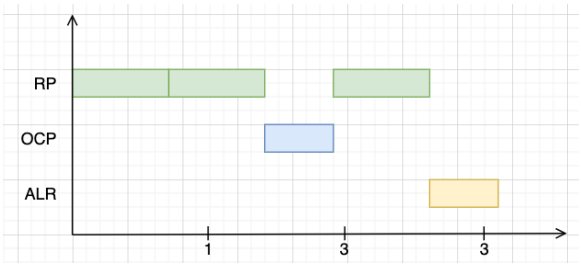

Figure 28: Timeline of the repeated interval in EDF overloading

9 Conclusions

We have started off the analysis by seeing the uses of offsets as a mechanism to improve the consistency between a formal model and the runtime execution of a real-time application subject to precedence relations. In turn, this has helped to reduce the pessimism in the fixed-priority analysis and offsets have proved to be extremely useful in increasing the schedulability of a given task set.

Throughout this paper, we have used MAST as model for describing real-time applications and representing not only the characteristics of the architecture of the application but also the hard real-time requirements that are imposed. However, we believe there is still some scheduling theory needed to eliminate some of the restrictions. The combination of all the restrictions has posed a severe blocking issue in our ability to describe a consistent model which could lead to a sound analysis of maximum utilisation. From our experience, the most critical missing pieces are the enhancement of the FPS offset-based analysis between transactions, to make it less pessimist, and the support for EDF offset-based analysis of linear transactions.

Besides, in this paper, we compared the behaviour of the two most famous policies: the FPS and the EDF algorithm. EDF allows a theoretic full processor utilization, which implies a more efficient exploitation of computational resources, but the statement doesn't hold in general with applications composed of dependant tasks.

At the same time, predictability during overload conditions only apply for the highest priority task, and it is not valid in general for the other tasks. Such a property of FPS can be of little use if we do not know a priori which other task is going to overrun [43]. Under permanent overload conditions, both the behaviours of FPS and EDF are predictable except for an initial interval, but deciding which one is better is highly application conditional. Nevertheless, what we have presented in this paper is just a shallow insight. Further work is undoubtedly needed to explore better the implications of relation dependencies in FPS and EDF overload.

In this paper, we have also introduced the Deadline-Floor Protocol for controlling access to shared resources within the EDF scheduling framework. This protocol has already been proved to be equivalent to the Stack Resource Protocol, the defacto protocol to use with EDF, by A. Burns (2005) [42]. We have instead shown that the combination EDF+DFP can exhibit the same schedule and blocking times of FPS+IPCP in a system where job releases are not stand-alone but instead form a chain.

Therefore, as shown by the aforementioned cases, we believe that the implications of precedence constraints are worth to be the subject of further research.

References

[1] A Burns, B Dobbing, T Vardanega. "Guide for the use of the Ada Ravenscar Profile in high integrity systems". In *University of York Technical Report YCS-2003-348*. January 2003.

[2] J. Yiu. "Exceptions and Interrupts". In *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*. October 2013.

[3] Juan A. de la Puente, José F. Ruiz, Juan Zamorano. "Open Ravenscar Real-Time Kernel. Design Definition File, Software Design Document". 2001.

[4] M. Gonzalez Harbour, J.J. GutiCrrez Garcia, J.C. Palencia GutiCrrez, and J.M. Drake Moyano. "MAST Modeling and Analysis Suite for Real Time Applications". In *Proceedings 13th Euromicro Conference on Real-Time Systems*. 2001.

[5] Albert M. K. Cheng, James Ras. "The Implementation of the Priority Ceiling Protocol in Ada-2005". In *ACM SIGAda Ada Letters*. Volume XXVII Issue 1, pp 24-39. 2007.

[6] J. M. Drake, M. G. Harbour, J. J. Gutiérrez, P. L. Martínez, J. L. Medina, J. C. Palencia "Description of the MAST Model". `https://mast.unican.es/mast_description.pdf`

[7] J. M. Drake, M. G. Harbour, J. J. Gutiérrez, P. L. Martínez, J. L. Medina, J. C. Palencia "MAST Analysis Techniques". `https://mast.unican.es/mast_analysis_techniques.pdf`

[8] J. M. Drake, M. G. Harbour, J. J. Gutiérrez, P. L. Martínez, J. L. Medina, J. C. Palencia "MAST Restrictions". `https://mast.unican.es/mast_restrictions.pdf`

[9] Juan Zamorano, Alejandro Alonso, José Antonio Pulido, Juan Antonio de la Puente. "Implementing Execution-Time Clocks for the Ada Ravenscar Profile". In *Reliable Software Technologies - Ada-Europe 2004*. pp 132-143. Ada-Europe 2004.

[10] Juan Zamorano, Jose F. Ruiz, Juan Antonio de la Puente. "Implementing Ada.Real Time.Clock and Absolute Delays in Real-Time Kernels". In *Reliable SoftwareTechnologies — Ada-Europe 2001*. pp 317-327. Ada-Europe 2004.

[11] Jane W. S. W. Liu. "Rate-Monotonic and Deadline-Monotonic Algorithms". In *Real-Time Systems*. pp 118-119. 2001.

[12] Jane W. S. W. Liu. "Critical Instants". In *Real-Time Systems*. pp 131-134. 2001.

[13] Jane W. S. W. Liu. "Optimality of the RM and DM algorithms". In *Real-Time Systems*. pp 118-119. 2001.

[14] Jane W. S. W. Liu. "Basic Priority Ceiling Protocol - Duration of Blocking". In *Real-Time Systems*. pp 295-296. 2001.

[15] Jane W. S. W. Liu. "Limited-Priority Levels". In *Real-Time Systems*. pp 166-168. 2001.

[16] Jane W. S. W. Liu. "Tick Scheduling". In *Real-Time Systems*. pp 168-171. 2001.

[17] Jane W. S. W. Liu. "Anomalous Behavior of Priority-Driven Systems". In *Real-Time Systems*. pp 72-73. 2001.

[18] Jane W. S. W. Liu. "Schedulability Test of Hierarchically Scheduled Periodic Tasks". In *Real-Time Systems*. pp 177-179. 2001.

[19] Jane W. S. W. Liu. "Fixed-Priority versus Dynamic-Priority Algorithms". In *Real-Time Systems*. pp 117-124. 2001.

[20] Joseph Yiu. "Introduction to the Debug and Trace Features". In *The Definitive Guide to ARM CORTEX-M3 and CORTEX-M4 Processors*. Chapter 4 pp 443-485. 2014.

[21] Joseph Yiu. "Semi-hosting". In *The Definitive Guide to ARM CORTEX-M3 and CORTEX-M4 Processors*. Chapter 18.3 pp 591-595. 2014.

[22] Joseph Yiu. "PendSV exception". In *The Definitive Guide to ARM CORTEX-M3 and CORTEX-M4 Processors*. Chapter 18.3 pp 591-595. 2014.

[23] K. Tindell. "An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks". In *Journal of Real-Time Systems*. Vol. 6, No. 2, March 1994.

[24] KenTindell, JohnClark. "Holistic schedulability analysis for distributed hard real-time systems". In *Microprocessing and Microprogramming*. Volume 40, Issues 2–3, pp 117-134. April 1994.

[25] C. L. Liu, James W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". In *Journal of the ACM*. Volume 20 Issue 1, pp 46-61. Jan. 1973.

[26] Klein, M., Ralya, Th., Pollak, B., Obenza, R., Harbour, M.G. . "A Practitioner's Handbook for Real-Time Analysis". 1993.

[27] Klein, M., Ralya, Th., Pollak, B., Obenza, R., Harbour, M.G. . "Using Utilization Bounds for Each Event when Deadlines Are Within the Period". In *A Practitioner's Handbook for Real-Time Analysis* . chapter 4.1.2. 1993.

[28] Klein, M., Ralya, Th., Pollak, B., Obenza, R., Harbour, M.G. . "Calculating Growth by Increasing Resource Usage of All Events". In *A Practitioner's Handbook for Real-Time Analysis* . chapter 4.3.8. 1993.

[29] Klein, M., Ralya, Th., Pollak, B., Obenza, R., Harbour, M.G. . "Designing Tasks that Must Synchronize to Share Common Data". In *A Practitioner's Handbook for Real-Time Analysis* . chapter 5.2. 1993.

[30] Klein, M., Ralya, Th., Pollak, B., Obenza, R., Harbour, M.G. . "Effects of Operating System and Runtime Services on Timing Analysis". In *A Practitioner's Handbook for Real-Time Analysis* . chapter 7. 1993.

[31] Klein, M., Ralya, Th., Pollak, B., Obenza, R., Harbour, M.G. . "Service the Event at a Specified Software Priority". In *A Practitioner's Handbook for Real-Time Analysis* . chapter 5.3.5.2. 1993.

[32] Alan Burns, Andy Wellings. "Scheduling real-time systems - Fixed Priority Dispatching". In *Concurrent and real-time programming in Ada*. chapter 13.1. 2007.

[33] Alan Burns, Andy Wellings. "Timing events". In *Concurrent and real-time programming in Ada*. chapter 15.2. 2007.

[34] Enrico Mezzetti, Marco Panunzio, Tullio Vardanega. "Preservation of Timing Properties with the Ada Ravenscar Profile". In *Reliable Software Technologiey – Ada-Europe 2010*. pp 153-166. 2010.

[35] Kristoffer Nyborg Gregertsen, Amund Skavhaug. "Implementation and Usage of the new Ada 2012 Execution Time Control Features". In *Ada User Journal*. 2011.

[36] J.C. Palencia ; M. Gonzalez Harbour. "Schedulability analysis for tasks with static and dynamic offsets". In *Proceedings 19th IEEE Real-Time Systems Symposium*. 1998.

[37] K. Tindell. "Adding Time - Offsets to Schedulability Analysis". Technical Report YCS 221, Dept. of Computer Science, University of York, England, January 1994.

[38] R. Wilhelm et al.. "The worst-case execution-time problem—overview of methods and survey of tools". In *Trans. on Embedded Computing Sys*. vol. 7, no. 3, pp. 153, 2008.

[39] M. Spuri. "Analysis of Deadline Scheduled Real-Time Systems". In *[Research Report] RR-2772, INRIA*. 1996.

[40] M. K. Gardner, J. W.S. Liu. "Performance of Algorithms for Scheduling Real-Time Systems with Overrun and Overload". In *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*. 9-11 June 1999.

[41] P. Carletto, T. Vardanega. "Ravenscar-EDF: Comparative Benchmarking of an EDF Variant of a Ravenscar Runtime". In *Ada-Europe 2017: Reliable Software Technologies – Ada-Europe 2017*. pp 18-33. 2017.

[42] A. Burns. "A Deadline-Floor Inheritance Protocol for EDF Scheduled Embedded Real-Time Systems with Resource Sharing" Technical Report YCS-2012-476, Department of Computer Science, University of York, UK.

[43] G.C. Buttazzo. "Rate Monotonic vs. EDF: Judgment Day". In *Real-Time Systems, 2005 - Springer*. 2005.

[44] A. Cervin, J. Eker, B. Bernhardsson, K.E. Arzén. "Feedback-Feedforward Scheduling of Control Tasks". In *Real-Time Systems, 2002 - Springer*. 2002.

[45] G. Buttazzo, M. Spuri, F. Sensini . "Value vs. Deadline Scheduling in Overload Conditions". In *Real-Time Systems, 2002 - Springer*. 2002.