MAST analysis of a Ravenscar 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

December 27, 2019

Abstract

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

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 behavior is an extremely important aspect.

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, 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, to ensure full analysability of the code. In the next sections we will use the term [RM] to refer to a section of the Ada Reference Manual[1].

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 [3]. This model allows a very rich description of the system, including the effects of event or message-based synchronization, multiprocessor and distributed architectures as well as shared resource synchronization.

The board-specific values shown in the paper are relative to the bare-board STM32F429I-Discovery and we have used the GNAT `ravenscar-full-stm32f429disco` runtime environment for supporting the Ravenscar restricted tasking model. The runtime implementation is based upon the Open Ravenscar Real-Time Kernel [2], whose design document can help the reader to understand the GNAT source code.

We have instead preferred the usage of a bare-board instead of the GNAT emulator as we have noticed significant standard deviation with the execution times measured on the latter. The board also includes an ST-LINK/V2 embedded

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

debug tool, which can stop the processor (halting), insert/remove breakpoints and execute instructions line by line (single stepping) [18].

1.1 The application

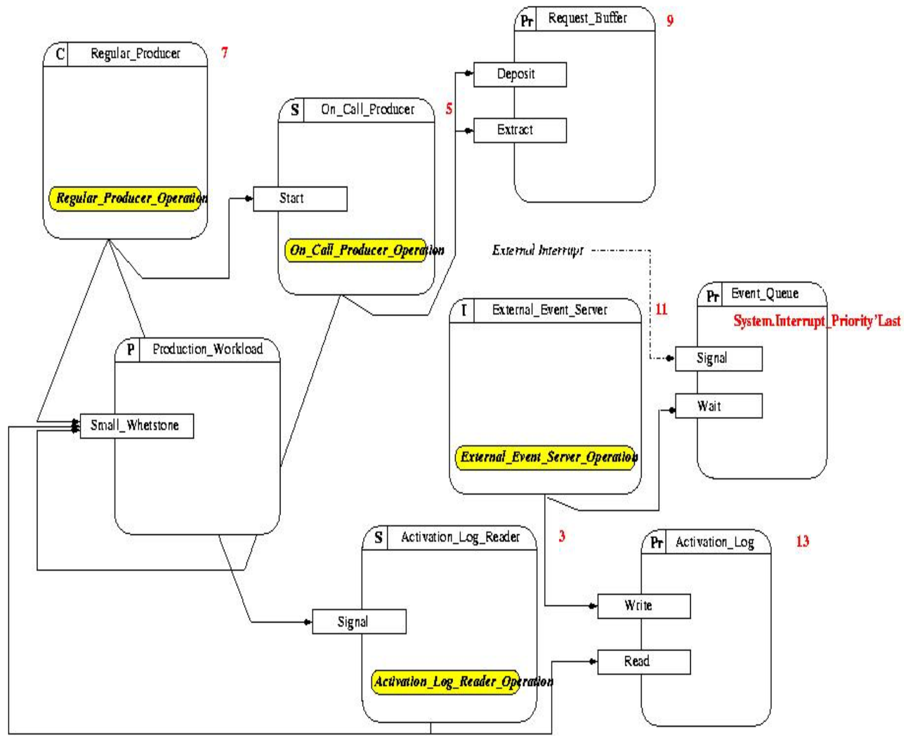

Figure 1: Architecture of the example application [1].

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 load out 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 by the following tasks and attributes. Static priorities are given based on the deadline monotonic scheduling [10], which the most optimal between the fixed priority algorithms [12].

Task name	Task type	Period / Minimum inter-arrival time (ms)	Deadline (ms)	Priority
Regular_Producer	Cyclic	1000	500	7
On_Call_Producer	Sporadic	5000	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.

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) [4], 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 [13].

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 set behave 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 less 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]

2 Ada tasking model

$\beta\mu\varsigma h\iota\ \delta\epsilon l\ \zeta\mu lo.$

1. Ada task model 2. Delay until pro & cons and implementation [9]

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. 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.

3 System model and notation

The described application is a set of tasks executing in the same processor, grouped into entities called transactions [33]. 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 2 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 [32]. Each task has its own unique priority and in this example the task set is scheduled using a preemptive fixed priority scheduling.

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

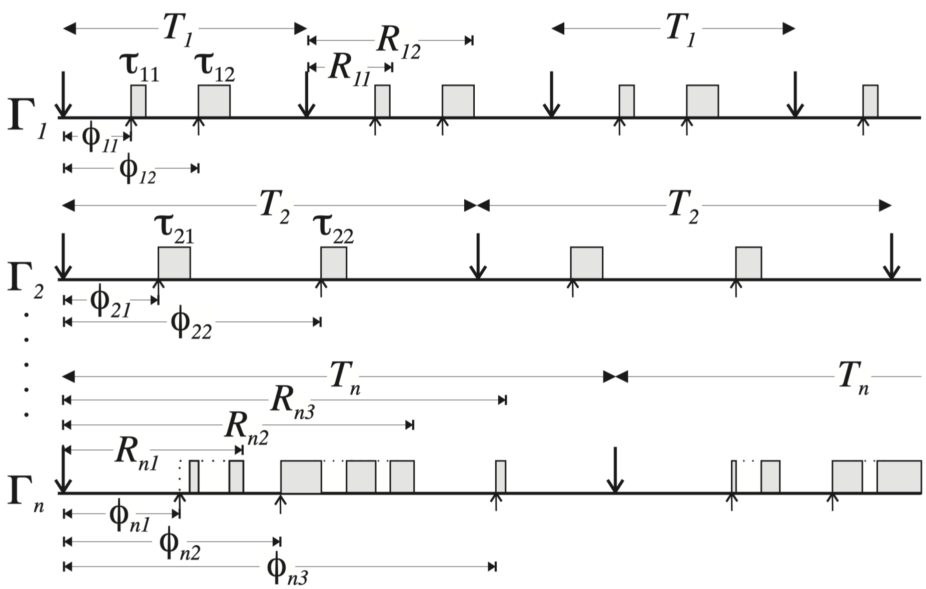

Figure 2: Timeline of a system composed of transactions with offsets [32]

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.

The reason for this is that tasks must execute in order, e.g. On_Call_Producer can start executing only after the preceding task Regular_Producer in the transaction has completed. The precedence constraints are modeled by assigning each task an initial offset and a maximum jitter [33]. 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 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}.

3.1 Offset-based analysis

Rate monotonic analysis (RMA) [10] allows an exact 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, precedence constraints and tasks with varying priorities, overhead analysis, etc. However, classic RMA [26] cannot provide exact solutions in systems in which tasks suspend themselves. Classic techniques for these systems are based on the assumption that all tasks are independent, and thus they lead to pessimistic results [32].

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 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 simultaneously become active.

Works on such problem has been the base of offset-based analysis, first proposed by Tindell and Clark [33] and later improved by Palencia and Gonzàlez [32] who called it Worst-Case analysis of Dynamic Offsets (WCDO). In such analysis, the worst-case response time of each task is used to set the offset and the jitter of the successive task in the same transaction. Then, the computation of worst-case response times is iterated until a stable solution is found. If response times are bounded, the offset-based method is guaranteed to converge to a solution.

The MAST analysis tool implements both different offset-based techniques and the more pessimistic holistic approach [21], which assumes tasks are scheduled independently. Holistic analysis is included in the toolset for completeness and comparison.

4 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 [34]. Therefore pessimistic scheduling is needed in order to provide an offline guarantee that all hard deadlines will be met, which leads to poor processor utilization.

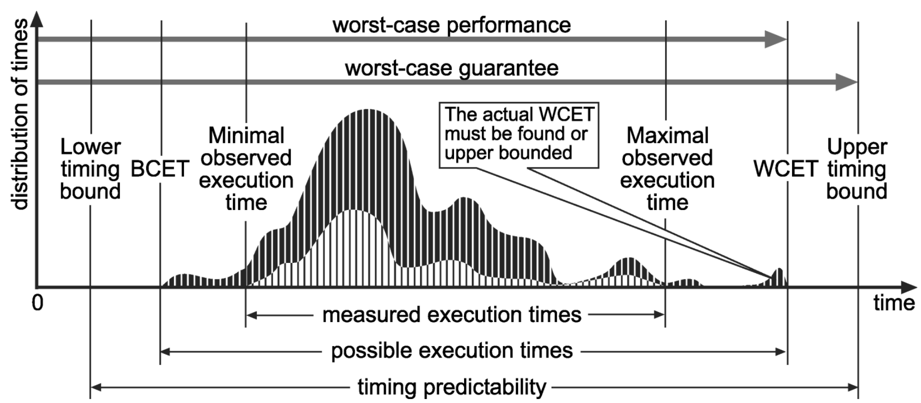

Figure 3: The lower curve represents a subset of measured executions. The darker curve, an envelope of the former, represents the times of all executions. [34].

Figure 3 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 common 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 very low 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 board push button.

Execution times are measured using two custom packages: `System_Overhead` and `Task_Overhead`. 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 and task operations which do not self-suspend. The latter measures 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_Overhead` 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 [31].

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.

4.1 Semi-hosting

It is worth mentioning the usage of semi-hosting [19], which allow 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_Overhead` 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 modeling of the `Put_Line` operation.

5 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 [30].

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's time is due, and detected by the runtime (see Chapter 6.1.1), 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 [29], 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 Task_Name is (RP, OCP, ALR);
   type Deadline_Events_Array is array (Task_Name) of Timing_Event;

   protected Handler
     with Priority =>
       System.Interrupt_Priority'Last
```

```
   is
      procedure Notify_Deadline_Miss (Event : in out Timing_Event);
   end Handler;

   procedure Set_Deadline_Handler (Name : Task_Name; In_Time : in Time);
   procedure Cancel_Deadline_Handler (Name : Task_Name);
end Deadline_Miss;


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

package body Deadline_Miss is
   Deadline_Events : Deadline_Events_Array;

   protected body Handler is
      procedure Notify_Deadline_Miss (Event : in out Timing_Event) is
      begin
         raise Program_Error with "Detected␣deadline␣miss";
      end Notify_Deadline_Miss;
   end Handler;

   procedure Set_Deadline_Handler (Name : Task_Name; In_Time : in Time) is
   begin
      Set_Handler (Deadline_Events (Name),
         In_Time, Handler.Notify_Deadline_Miss'Access);
   end Set_Deadline_Handler;

   procedure Cancel_Deadline_Handler (Name : Task_Name) is
      Cancelled : Boolean;
      pragma Unreferenced (Cancelled);
   begin
      Cancel_Handler (Deadline_Events (Name), Cancelled);
   end Cancel_Deadline_Handler;
end Deadline_Miss;
```

Ispiration from [8]. The measured execution times include overrun detection overhead for Regular Producer, On Call Producer and Activation Log Reader.

6 MAST

MAST [3] 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 behavior of their application, including schedulability analysis for checking 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, 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 a lot of redundancy.

MAST provides also a graphical editor to generate the system using the MAST ASCII description, but 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.

6.1 The MAST Model

We now proceed to describe the MAST model of the example application. In this phase, it will represent a 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" [5].

6.1.1 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 [16].

```
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 [14].

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, whereas 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 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) [31].

The system timer used by the board is Tick Scheduling [15], 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 in the analysis using the technique described in [27]. We 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.

6.1.2 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 [17], but the example application has only one primary scheduler with fixed priority policy.

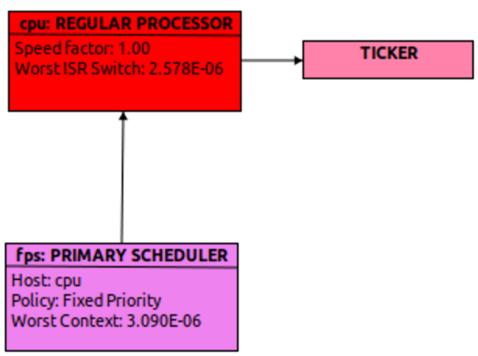

Figure 4: Fixed Priority Scheduler which manages the CPU

```
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 [20]. Then context switching is usually carried out in the `Pend_SV` trap handler.

6.1.3 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.

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

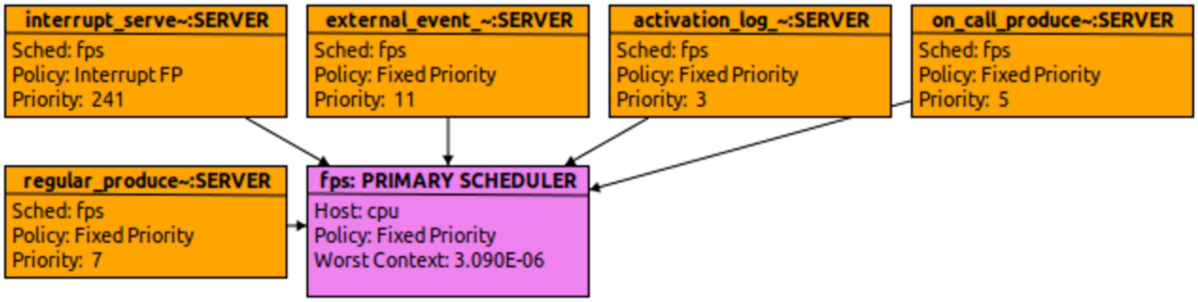

Figure 5: Scheduling Servers representing the application tasks

```
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);

Scheduling_Server (
   Type                         => Regular,
   Name                         => interrupt_server,
   Server_Sched_Parameters      =>
      ( Type           => Interrupt_FP_Policy,
        The_Priority => 241),
   Scheduler                    => fps);
```

The Scheduling Server `interrupt_server` allows to model the runtime which runs the Interrupt Service Routine using the technique described in [28]. Each aperiodic response will be rappresented as two MAST operations. The first operation is the interrupt handler and executes as `interrupt_server` at interrupt priority 241. The second part is implemented as an operation of the task External_Event_Server at software priority 11.

6.1.4 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 modeled as Shared Resourses that use the Immediate Priority Ceiling Protocol described above.

```
Shared_Resource (
```

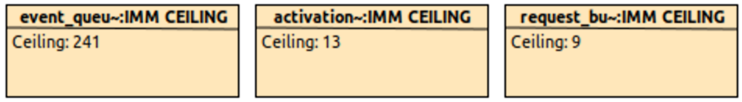

Figure 6: Shared Resources of the application

```
   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);
```

6.1.5 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 corresponds to a method defined by the application, other times it's defined in the model specifically. By doing so, we can define the caller procedures as simpler Composite operations, which have the execution time with the runtime overhead included.

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));
```

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 to control 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 happens by increasing the load of factor 10, we would just multiply the WCET in the model by 11, without the need to measure again all the Enclosing operations, since all the methods which use Whetstone are defined as Composite. However we have been careful to avoid forgetting any overhead in a Enclosing method and make sure they are not impacted by any change of the Whetstone workload.

6.1.6 Transactions

A Transaction represents a transaction of our model (see Section 3) as a graph of event handlers and events, that represents 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, most of the time a Global Deadlines relative to the arrival of a Referenced External Event. Local Deadlines are instead 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.

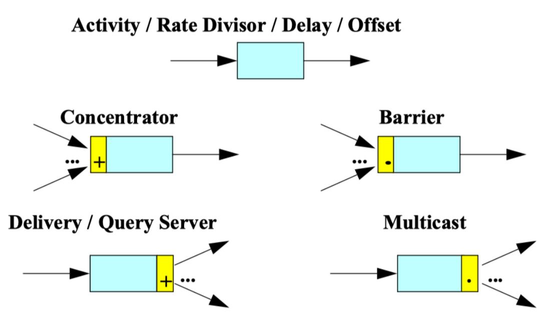

Figure 7: Event Handlers

Event Handlers in figure 7 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 of some (previous) 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.

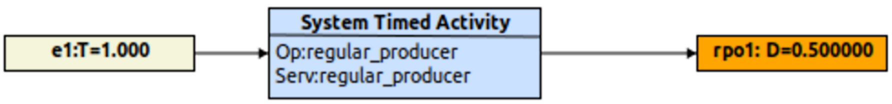

Figure 8: Regular_Producer transaction

```
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 modeled as a transaction activated by the periodic system timer, with 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.

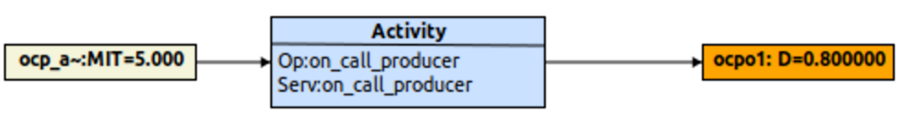

Figure 9: On_Call_Producer transaction

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

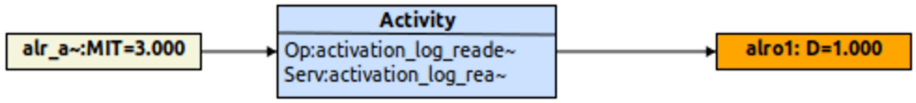

Figure 10: Activation_Log_Reader transaction

```
   External_Events =>
      ( ( Type              => Sporadic,
          Name              => ocp_activation,
          Avg_Interarrival  => 5.000,
          Distribution      => UNIFORM,
          Min_Interarrival  => 5.000)),
   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 modeled as activated by a bounded aperiodic event, with minimum interarrival time of 5s and uniform distribution. Actually we know that the interarrival time is precisely 5s, thus the same value as average interarrival time. Similar modelling has been done for the Activation_Log_Reader sporadic task.

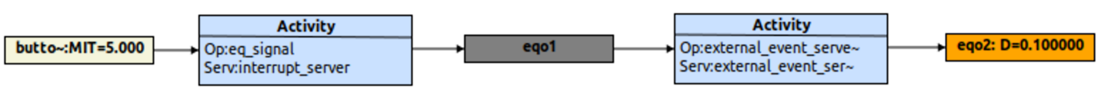

Figure 11: External push-button transaction

```
Transaction (
   Type            => regular,
   Name            => event_queue_interrupt,
   External_Events =>
      ( ( Type              => Sporadic,
          Name              => button_click,
          Avg_Interarrival  => 0.000,
          Distribution      => UNIFORM,
          Min_Interarrival  => 5.000)),
   Internal_Events =>
      ( ( Type => Regular,
          Name => eqo1),
        ( Type => Regular,
          Name => eqo2,
          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 => eq_signal,
```

```
 Activity_Server    => interrupt_server),
(Type               => Activity,
 Input_Event        => eqo1,
 Output_Event       => eqo2,
 Activity_Operation => external_event_server,
 Activity_Server    => external_event_server)));
```

The push-button interrupt event stream is modeled as triggered by a sporadic event of 5s as minimum interarrival time and it's first handled by the `interrupt_server` which runs the interrupt handler at hardware interrupt priority and then by the `external_event_server` job at software priority.

7 MAST analysis

As of the time of writing, MAST is at version 1.5.1 and supports the analysis tools [6] in figure 12. 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 12: MAST analysis tools [6].

- *Classic RM Analysis* (not used): it implements the classic exact response time analysis for single-processor fixed-priority systems and corresponds to Technique 5, "Calculating response time with arbitrary deadlines and blocking" in [23];
- *Holistic Analysis* (used): this analysis extends the response time analysis to multiprocessor and distributed systems. It is not an exact 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 [21];
- *Offset Based Approximate Analysis* (not used): 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 [33] and then extended to distributed systems by Palencia and González [32];
- *Offset Based Approximate with Precedence Relations Analysis* (used): 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* (used): 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 [7]. 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;
- *Linear_Plus_Transactions_Only*: checks that every transaction only has one external event and is not multipath, i.e. it has no Multicasts. This restriction is required by the 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 [7]. This restriction is required by the Holistic analysis.

The transaction which defines the button interrupt event stream is considered as a linear transaction, since it only has one external event, but unfortunately it is not a Simple transaction since Activities are not executed by the same server. An Interrupt Scheduling Server first handles the Interrupt Service Routine and then a regular Scheduling Server executes the application handler defined by External_Event_Server.

As a consequence, it's not possible to use the classic Rate Monotonic algorithm [10] with our model and we are restricted to the traditional Holistic and the Offset-based tools. We believe the reason is that the Rate Monotonic assumes independent tasks, but the interrupt transaction is composed by an ISR executed by the Interrupt_FP_Server, followed by the interrupt handler in External Event Server, which is activated by the former and thus dependent on it for its activation.

As final note, as of the time of writing, offset-based analysis fallbacks to holistic analysis with EDF processing resources [6].

8 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 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.

8.1 Independent tasks

We start with an Holistic analysis of the initial system.

```
Optimum Resource Ceilings and Levels:
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 a required to be an upper bound of the priorities between the tasks the 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 transaction into two distinct tasks with proper offset to better model the application [33].

Task	WCET
regular_producer	0.019333
on_call_producer	0.007126

activation_log_reader	0.003582				
Transaction	R_{min}	R_{max}	Slack	Worst blocking time	Jitter
rp_transaction	0.019333	0.020440	2470.0%	2.000E-06	0.001107
ocp_transaction	0.007126	0.026598	10776.6%	1.000E-06	0.019472
alr_transaction	0.003582	0.030201	26600.8%	0.00	0.026619
event_queue_interrupt ISR	6.156E-06	1.100E-05	>=100000.0%	1.000E-06	4.844E-06
event_queue_interrupt EES	1.616E-05	3.818E-05	>=100000.0%	1.000E-06	2.202E-05
System slack	2401.2%				
Total utilisation	2.58%				

Table 3: Holistic 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 best-case response time R_{min}, the worst-case response time R_{max}, the slack, the blocking time and the jitter for each transaction. 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 ISR. 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 and the ISR. Its blocking time is caused the Activation_Log_Reader.;
- *alr_transaction*: suffers additional jitter due to interference by On_Call_Producer, Regular_Producer and the ISR. It has no blocking time since it's the task with lowest priority;
- *event_queue_interrupt ISR*: it's the first part of the interrupt transaction. It suffers additional jitter due to the runtime ISR overhead. Its blocking time is caused by the External_Event_Server;
- *event_queue_interrupt EES*: it's the second part of the interrupt transaction. It suffers additional jitter due to the ISR, considered as independent from the former in the Holistic analysis. Its blocking time is caused by the External_Event_Server;

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 correponds to how much the execution time of all event responses can be increased while preserving system schedulability [25]. We leave the Event_Queue 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_{min}	R_{max}	Slack	Worst blocking time	Jitter
rp_transaction	0.482597	0.485492	2.73%	2.000E-06	0.002895
ocp_transaction	0.177482	0.662663	76.95%	1.000E-06	0.485181
alr_transaction	0.088702	0.751713	277.34%	0.00	0.663011
event_queue_interrupt ISR	6.156E-06	1.100E-05	>=100000.0%	1.000E-06	4.844E-06
event_queue_interrupt EES	1.616E-05	3.818E-05	>=100000.0%	1.000E-06	2.202E-05
System slack	3.21%				
Total utilisation	55.15%				

Table 4: Holistic analysis results for FPS increased of factor 24

The blocking times have not changed because protected operations are same as before, but the total utilisation have increased up to 55.15%. 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_{min}	R_{max}	Slack	Worst blocking time	Jitter
rp_transaction	0.482597	0.485492	0.390625%	2.000E-06	0.002895
ocp_transaction	0.312341	0.798045	0.390625%	1.000E-06	0.485704
alr_transaction	0.156086	0.954736	28.13%	0.00	0.798650
event_queue_interrupt ISR	6.156E-06	1.100E-05	17737.1%	1.000E-06	4.844E-06
event_queue_interrupt EES	1.616E-05	3.818E-05	17737.1%	1.000E-06	2.202E-05
System slack	0.390187%				
Total utilisation	60.10%				

Table 5: Holistic analysis results for FPS increased of factor 43

The system has reached utilisation 60.10%. We now increase 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_{min}	R_{max}	Slack	Worst blocking time	Jitter
rp_transaction	0.482597	0.485492	0.0%	2.000E-06	0.002895
ocp_transaction	0.312341	0.798045	0.390625%	1.000E-06	0.485704
alr_transaction	0.198645	0.997460	0.390625%	0.00	0.798650
event_queue_interrupt ISR	6.156E-06	1.100E-05	13963.3%	1.000E-06	4.844E-06
event_queue_interrupt EES	1.616E-05	3.818E-05	13963.3%	1.000E-06	2.202E-05
System slack	0.390187%				
Total utilisation	61.51%				

Table 6: Holistic analysis results for FPS increased of factor 55

The maximum utilisation reached is about 61.51%. The only transaction with significant slack left is `event_queue`, but tests show that an increase of the WCET of factor 140 in the operation *external_event_server* would improve the utilisation only up to 61.54%, hence we can ignore it.

8.2 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 [11]; 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. 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 [33];

2. 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 simultaneously become active [32].

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.

Implicitly, we already have modeled a precedence constraint in the `event_queue_interrupt` transaction. By being a linear transaction, it's composed of the two distinct tasks Interrupt Service Routine (ISR) and External_Event_Server (EES). The former execution always preceeds the latter and the relation is represented by having the EES Activity wait for the arrival of the ISR Output event, signalling its completion. The ISR and the EES cannot be active simultaneously.

In fact, using the Offset Based Slanted and Offset Based Approximate with Precedence Relations analysis provides an improvement of the jitter, and thus R_{max}, for the transaction.

Transaction	R_{min}	R_{max}	Slack	Worst blocking time	Jitter
event_queue_interrupt ISR	6.156E-06	1.100E-05	13963.3%	1.000E-06	4.844E-06
event_queue_interrupt EES	1.616E-05	3.818E-05	13963.3%	1.000E-06	2.202E-05

Table 7: Holistic analysis results for `event_queue_interrupt`

Transaction	R_{min}	R_{max}	Slack	Worst blocking time	Jitter
event_queue_interrupt ISR	6.156E-06	1.100E-05	13963.3%	1.000E-06	4.844E-06
event_queue_interrupt EES	1.616E-05	**3.202E-05**	13963.3%	1.000E-06	**1.587E-05**

Table 8: Offset Based Slanted analysis results for `event_queue_interrupt`

Transaction	R_{min}	R_{max}	Slack	Worst blocking time	Jitter
event_queue_interrupt ISR	6.156E-06	1.100E-05	13963.3%	1.000E-06	4.844E-06
event_queue_interrupt EES	1.616E-05	**2.718E-05**	13963.3%	1.000E-06	**1.102E-05**

Table 9: Offset Based Approximate with Precedence Relations analysis results for `event_queue_interrupt`

The Offset Based Slanted tool reduces interference suffered by the EES handler since the ISR cannot be executing at the same time. Thus the jitter and the R_{max} of the former have been reduced by an amount equal to the best-case response time R_{min} of the ISR. We believe this value is used as offset Φ of the task EES, whereas the difference $R_{ISR\ max} - R_{ISR\ min}$ is counted as possible interference.

Offset Based Approximate with Precedence Relations is instead able to provide even a tighter approximation by using the precedence relation and therefore considering a dynamic offset $\Phi_{i2} \in [R_{ISR\ min}, R_{ISR\ max}]$. The EES task is never activated before the ISR completion.

Future analysis will provide only the results from the Offset Based Approximate with Precedence Relations technique, referred as only offset-based analysis, as it has proved to be the best of the two tools.

8.2.1 Blocking time

We can further improve the analysis by reducing unnecessary blocking time: the precedence constraint between the ISR and EES means the former cannot suffer blocking time from the latter. There is also no need to use any lock to guard the access to the Event_Queue resource. The Protected Object can be considered as just a synchronization mechanism with no shared data, like a Suspension Object.

By removing the shared resource Event_Queue, we obtain the results in table 10 for the interrupt transaction.

EES results are unaltered, but ISR values have improved as expected. The latter is no more subject to any blocking time from the EES, and actually it suffers no blocking at all from other tasks because it runs at interrupt priority. The EES, whereas, continues to have blocking caused by the Activation_Log_Reader.

Transaction	R_{min}	R_{max}	Slack	Worst blocking time	Jitter
event_queue_interrupt ISR	6.156E-06	**1.000E-05**	13963.3%	**0.000**	**3.844E-06**
event_queue_interrupt EES	1.616E-05	2.718E-05	13963.3%	1.000E-06	1.102E-05

Table 10: Offset Based Approximate with Precedence Relations analysis results for `event_queue_interrupt`

The rest of the analysis has not been affected by the removal of the resourse. The ISR runs at interrupt priority, whereas the EES still has the highest task priority and consequently they don't cause any blocking.

8.2.2 Critical instant

The existence of offsets makes it impossible for some sets of tasks to simultaneously become active. 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 5 `regular_producer` transaction invocations. The two tasks are not thus independent and should belong to the same transaction. Equivalent argument holds true for Activation_Log_Reader (ALR).

Formally, for the two tasks RP and OCP that are members of the same transaction, task RP must complete before task OCP is run. We have two priority situations: task RP is of higher priority, or task OCP is of higher priority. Fortunately, in our system RP is of higher priority, which means that task OCP (of lower priority) will 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 [33]:

$\Phi_{RP} \ + \ J_{RP} \ \leq \ \Phi_{OCP}$

Figure 13 shows what we believe to be the best representation in MAST of the precedence constraints.

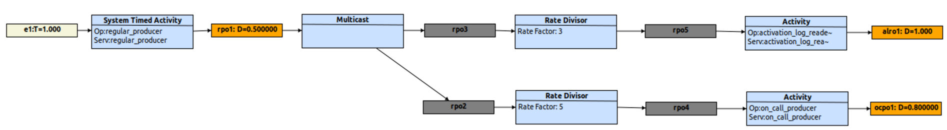

Figure 13: Transaction with all the precedence constraints

The transaction is initiated by an External Event with period 1s and served by the RP Activity. Its completion generates two Output Events, modeled with a Multicast, which are able to release the ALR and/or the OCP. The first Output Event activates the ALR every 3 transaction invocations, as shown by the Rate Divisor with factor 3. The second Output Event has a similar effect on the OCP Activity every 5 transaction invocations. Both ALR and OCP Activity have their respective Hard Local Deadlines, which start as soon as the Input Events arrive.

Unfortunately, at the time of writing, MAST doesn't support this kind of transaction. First of all no transaction can use a Hard Local Deadline due to the No_Hard_Local_Deadlines restriction. Besides, also the usage of Multicast along with Rate Divisors is forbidden by the Restricted_Multipath_Transactions_Only restriction. We were not able to think a transaction which can more strictly model the precedence constraints of our system and yet respect this restrictions. Ours attempts of using workarounds based on offsets had analysis results at least equivalent to modeling as independent tasks.

Nevertheless, we believe that the maximum utilisation reached by assuming stand-alone tasks is very close to the pratictical limit, as shown by the timeline in figure 14.

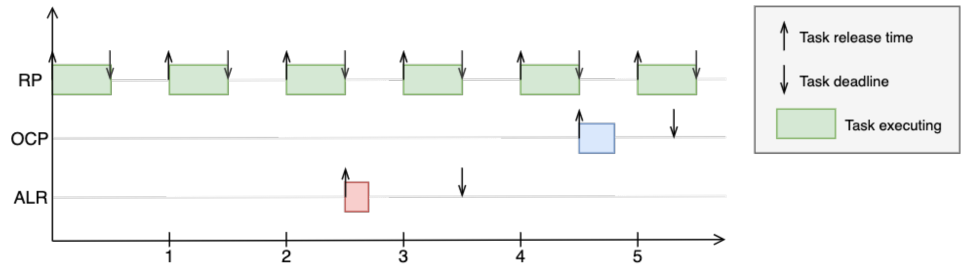

Figure 14: Timeline with the three tasks

Having fixed priority scheduling, the behaviour of the tasks is easily predictable. RP can reach a maximum execution of 0.5s equal to its deadline. Then there is an idle interval of 0.5s before its next release event and, within this interval, the tasks OCP and ALR must complete their executions. The critical instant happens when OCP and ALR are released at the same time, after a completion of the RP, as shown in figure 15 at time 14. OCP, ALR and the next RP must complete in less or equal than 1s, otherwise at least one of them will not meet its deadline.

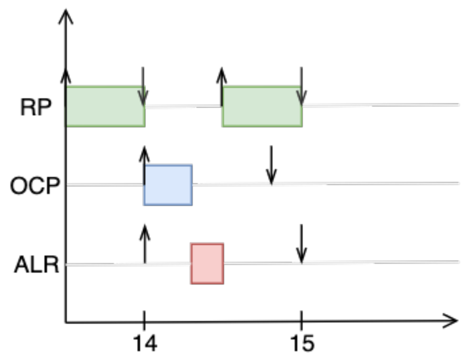

Figure 15: Timeline with the three tasks

The WCET used in the application for ALR, OCP and ALR are respectively 0.482597, 0.312341s and 0.198645s. Their sum is 0.993583, hence the CPU utilisation is very close to 100% during the busy period, despite having total utilisation at 61.51%.

9 Conclusions

Non funziona mai un cazzo.

Facile sbagliare con le analisi ed avere bias.

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

[3] 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.

[4] 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.

[5] 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`

[6] 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`

[7] 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`

[8] 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.

[9] 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.

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

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

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

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

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

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

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

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

[18] 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.

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

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

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

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

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

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

[34] 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.