A MAST analysis of a Ravenscar application example

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

November 30, 2019

Abstract

Buchi del culo. 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

There is increasing recognition that the software components of critical real-time applications must be provably predictable. This is particularly so for a hard real-time system, in which the failure of a component of the system to meet its timing deadline can result in an unacceptable failure of the whole system. 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.

The use of Ada has proven to be of great value within high integrity and real-time applications, albeit via language subsets of deterministic constructs, to ensure full analysability of the code. The research work in schedulability analysis has been mapped onto a number of new Ada constructs and rules that have been incorporated into the Real-Time Annex of the Ada language standard [RM D]. This has opened the way for these tasking constructs to be used in high integrity subsets whilst retaining the core elements of predictability and reliability.

The Ravenscar Profile 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, and which could be certifiable to the highest integrity levels.

The example presented in this paper is extracted from "Guide for the use of the Ada Ravenscar Profile in high integrity systems" [1] and it is designed to illustrate the expressive power of the Ravenscar Profile and the associated coding paradigms, which aim to facilitate off-line scheduling analysis.

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

The example system 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 source. 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.

MAST, a Modeling and Analysis Suite for Real-Time Applications, is a model for representing the temporal and logical elements of real-time applications [2]. 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.

MAST offers a graphical user interface but it's still immature to be completely usable. Too many frequent bugs.

2   MAST Model

For a reference to the MAST syntax, visit `https://mast.unican.es/mast_description.pdf`. Since we are dealing with hard real-time analysis, we define only the WCET to the operations.

```
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);
```

We have a board with only one CPU and interrupt ranges are taken from `System` package. Task priorities range from 1 to 240, whereas interrupt priorities go from 241 to 255. So we have at maximum 240 task priorities, if we ever would need more priorities we could use the technique described in [4]. The Interrupt Service Routine overhead is measured as the time taken to execute `Interrupt_Handler` in System.BB.Board_Support package, apart from the execution time of the user-defined interrupt handler. It takes into account the overhead of managing the Task and Interrupt Execution Clocks [8].

The system timer used by the board is Tick Scheduling [5], which is accounted in the analysis using the technique described in [6]. The period of the tick is 1ms, as defined in `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 `Sys_Tick` trap.

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

We only have one primary scheduler in the system, no hierarchical scheduling. The context switch overhead is measured as time to trigger the context switch interrupt `Pend_SV` and the execution time of `Pend_SV_Handler` in `System.BB.CPU_Primitives.Context_Switch_Trigger` package, to save the active context and load the new one.

```
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 );

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

Each task is a Scheduling Server, whereas `interrupt_server` models the runtime which runs the Interrupt Service Routine using the technique described in [7].

```
Shared_Resource (
   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);
```

Protected objects are modeled as Shared Resourses which use the Immediate Priority Ceiling Protocol, the same defined as `Priority Ceiling Locking` in Ada Reference Manual D.3.

```
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 ));
```

Protected methods are modeled as simple operations which lock and unlock the protected object resourse. The execution time is measured from the first line of the method to the last one, so it doesn't include the runtime overhead associated with invoking protected methods.

```
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 ));
```

For each protected method there is an Enclosing operation which takes into account the overhead associated with invoking protected methods. Sometimes it's already a method defined by GEE, other times it's defined in the model on purpose. By doing so we can define the more complex methods as Composite operations, which have the execution time as the sum of the execution times of the comprised operations.

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

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 ));
```

Therefore, by changing the worload parameter of `Small_Whetstone` in the application implementation, 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. If we wanted to try what happens by increasing the load of factor 10, we would just increase the WCET to `0.19363`, without the need to measure again all the Enclosing operations, since all the methods which use the Whetstone are defined as Composite. However we have been careful to avoid forgetting any overhead in a Composite method and make sure they are not impacted by any change of the Whetstone workload. For instance, if we had defined `regular_producer` as Enclosing we could have defined as composed of only `rp_operation` and then measure its WCET, which will implicitly count also overrun detection and delay queue overhead. By instead defining it as Composite, we have been careful to define the simple operations `overrun_detection` and `delay_until` to include their execution time, using the technique described in [6].

```
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 a periodic external event, with period of 1s. The event is handled by the `regular_producer` operation by the task of the same name. The Event Handler is of type `System_Timed_Activity` to take into account the jitter and the overhead caused by the tick scheduling

```
Transaction (
    Type             => regular ,
    Name             => ocp_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. Similar modeling has been done for the `Activation Log Reader` sporadic task.

```
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 blue button interrupt event stream is modeled as a triggered by a sporadic event of 5s as minimum interarrival time and it's first handled by the `interrupt_server` which runs the ISR at interrupt priority level and then by the user-defined `external_event_server` at task priority-level.

3 Overrun detection

— Overrun.ads

```ada
with Ada.Real_Time ;
with Ada.Execution_Time ;

package Overrun is
    type Limits_Array is array (0 .. 2) of Ada.Execution_Time.CPU_Time;

    procedure Start (Index : Natural; Budget : Ada.Real_Time.Time_Span);
    procedure Check (Index : Natural);
end Overrun;
```

—— Overrun . adb

```
with Ada . Real_Time ; use Ada . Real_Time ;
with Ada . Execution_Time ; use Ada . Execution_Time ;

package body Overrun is
    use Ada . Real_Time ;
    use Ada . Execution_Time ;

    Limits : Limits_Array := ( CPU_Time_First , CPU_Time_First , CPU_Time_First );

    procedure Start ( Index : Natural ; Budget : Time_Span ) is
    begin
        Limits ( Index ) := Ada . Execution_Time . Clock + Budget ;
    end Start ;

    procedure Check ( Index : Natural ) is
    begin
        if Ada . Execution_Time . Clock > Limits ( Index ) then
            raise Program_Error with " Detected␣overrun ";
        end if ;
    end Check ;
end Overrun ;
```

Ispiration from [3]. The measured execution times include overrun detection overhead for Regular Producer, On Call Producer and Activation Log Reader.

4   FPS analysis

Transaction	Worst case response time (s)	Slack	Worst blocking time (s)
rp_transaction	0.020393	2477.0%	2.000E-06
ocp_transaction	0.026525	10852.0%	1.000E-06
alr_transaction	0.030109	27088.7%	0.00
event_queue_interrupt	1.100E-05	N/A	1.000E-06
event_queue_interrupt	3.818E-05	N/A	1.000E-06

Table 1: Holistic analysis results for FPS

The system slack is 2401.2% and total utilisation 2.59%.

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] 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.

[3] 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.

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

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

[6] 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.

[7] Klein, M., Ralya, Th., Pollak, B., Obenza, R., Harbour, M.G. . Use an Interrupt Handler. In *A Practitioner's Handbook for Real-Time Analysis* . chapter 5.3.5.1. 1993.

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