Handling Temporal Faults in Ada 2005*

José A. Pulido, Santiago Urueña, Juan Zamorano, and Juan A. de la Puente

Universidad Politécnica de Madrid (UPM), E28040 Madrid, Spain
{pulido,suruena,jzamorano,jpuente}@dit.upm.es

Abstract. Hard real-time systems have stringent deadline requirements, which can be guaranteed at system design time by restricting the computational model so that a careful analysis of execution-time budgets and response-time values can be performed. However, design-time guarantees are not enough in many high-integrity systems, in which some degree of run-time fault-tolerance has to be implemented as well. This paper deals with run-time mechanisms for temporal fault detection and recovery, based on some of the new features available in Ada 2005. Fault detection mechanisms are based on execution-time clocks and timers, and timing events. Fault recovery schemes are application-dependent, but some basic patterns are proposed that can be used to develop such kinds of mechanisms.

1 Introduction

The distinctive characteristic of hard real-time systems is to have strict temporal requirements, usually in the form of hard deadlines relative to the activation time of tasks [1]. Most such systems have also high integrity requirements, and thus must be shown to be fully predictable in operation and have all the properties required from them. This is usually achieved by applying static analysis to the software before it is deployed for operation, in addition to the more common approach of performing dynamic tests on the application code. Response-time analysis (RTA) methods [2,3] are particularly useful for analysing the temporal behaviour of hard real-time systems.

Static analysis methods are not always compatible with the full expressiveness of computer languages. In the case of Ada, an ISO report [4] provides guidelines for restricting the language in order to apply different kinds of static analysis techniques. The Ravenscar profile [5,6] (RP) defines a restricted tasking model that can be enforced at compilation time and enables response-time analysis of Ada programs. The profile is part of the new Ada 2005 standard [7, D.13.1]. Available experience supports the claim that the profile is expressive enough for useful hard real-time applications to be developed [8,9] and it provides a solid base for implementing high-integrity real-time systems [10].

* This work has been funded in part by the Spanish Ministry of Education, project no. TIC2005-08665-C03-01 (THREAD), and by the IST Programme of the European Commission under project IST-004033 (ASSERT).

N. Abdennahder, F. Kordon (Eds.): Ada-Europe 2007, LNCS 4498, pp. 15–28, 2007.

However, in some applications with stringent dependability requirements static analysis is not enough, and additional run-time mechanisms have to be provided in order to ensure appropriate levels of fault tolerance at execution time. In particular, appropriate mechanisms for detecting and handling violations of the specified temporal behaviour of the system are of major importance for highly critical real-time systems.

While Ada 95 provided only some basic mechanisms for detecting real-time temporal faults, Ada 2005 has been enhanced with a full set of run-time mechanisms aimed at monitoring the temporal behaviour of a system at run time [7, Annex D]. A previous paper by the authors [11] analysed the usefulness of some of the new mechanisms, and made some suggestions about the proposals that had been discussed for Ada 2005 at the time of writing it. This paper completes that preliminary analysis, taking into account the final definition of time monitoring mechanisms in the Ada 2005 standard, and refining and systematizing error detection and handling strategies based on them. Section 2 categorizes the most common kind of temporal faults in hard real-time systems, and describes some techniques for detecting them using the new Ada mechanisms. Section 3 describes some basic strategies for handling temporal faults. The next section (Sect. 4) gives some guidelines for using the proposed techniques, and Section 5 discusses some application and implementation details. Finally, Section 6 summarizes the main conclusions of this work.

2 Detecting Temporal Faults

2.1 Temporal Fault Characterization

The main temporal requirement for real-time tasks is to complete their activity by the specified deadline in each execution cycle. Therefore, the main problem a task can face is *deadline overrun*. While it may be acceptable for soft real-time tasks to occasionally miss a deadline, hard real-time tasks must always fulfill this requirement.

Although rate-monotonic or response-time analysis [3] can be used to guarantee deadlines at system design time, deadline overruns may still happen during operation if some of the static analysis assumptions are violated. Such violations may come from two sources:

- Violations of the task arrival model, in particular *minimum inter-arrival time* (MAT) of sporadic tasks. A sporadic task executing more often than specified results in an increase of the processor load, which may lead to deadline overruns in the misbehaving task or, more often, in lower-priority tasks.
- Violations of the execution-time budget of some tasks. Temporal analysis assumes that the *worst-case execution-time* (WCET) of all code sequences is known. However, the complexity of modern processor architectures may make it difficult or even impossible to accurately estimate WCET values, which in turn may result in one or more tasks consuming more CPU time

than expected. This may result in the faulty task missing its deadline or, more likely, delaying lower-priority tasks and making them miss theirs.

Since deadline overruns come from these kinds of faults, it is desirable to detect or event prevent them, when possible, before any deadline is missed. In this way, errors can be detected in the faulty task, which often is not the same as the task missing its deadline, and there is more time available for corrective actions to be taken.

There may be other temporal anomalies, which are not dangerous as they cannot result in deadline overruns, but can still expose other problems. For example, a task completing its job much earlier than expected (i.e. violating its assumed best-case execution time), may imply that some required functionality is not executed. We are not considering here such rare cases, and will concentrate only on deadline and WCET overruns, and MAT enforcement.

2.2 Timing Mechanisms in Ada 2005

The current revision of the Ada standard provides a choice of time monitoring mechanisms that can be used to detect temporal faults. The Ada.Real_Time package, including a monotonic real-time clock, which was already part of Ada 95, can be used to check real-time related properties, such as minimum inter-arrival times or task deadlines. Real-time timers were not available as such in Ada 95, but delay statements and asynchronous transfer of control (ATC) provided a similar functionality at a higher abstraction level (see e.g. [1]). However, ATC has a complex implementation and is thus excluded from the Ravenscar profile, which makes it difficult to detect deadline overruns in critical systems adhering to the profile. Unfortunately, critical systems are the kind of systems more prone to require temporal fault detection.

This has been solved in Ada 2005 by providing a lower-level mechanism, *timing events*, which can be used with the Ravenscar profile [7, D.15]. Timing events are a light-weight mechanism for specifying an action to be executed at a given time. A timing event can be set to occur at an absolute time or after a real-time interval. A protected procedure handler is executed whenever the event occurs, unless it is cancelled before that time. It should be noticed that only library-level timing events are allowed by the Ravenscar profile.

Ada 2005 also includes mechanisms for measuring and monitoring execution-time, namely *execution-time clocks*, *execution-time timers*, and *group execution-time budgets*. These mechanisms can be used to estimate the execution time of code segments, and also to handle some kinds of aperiodic events, but this paper will concentrate on their use for detecting execution-time related temporal faults. In Ada 2005 each task has an execution-time clock that computes the amount of CPU time it has consumed, including the run-time services invoked by the task. Implementations are also allowed to allocate the time spent in interrupt service routines or global system services (e.g. scheduling) to the currently running task.

Execution-time timers are objects that are associated with a task —and hence with the task execution-time clock— when they are declared. A timer can be armed to expire at an absolute value of that clock or after some execution-time interval. When the timer expires, a protected procedure handler is executed. Setting again the handler cancels any previous temporization that might be in effect. Group execution-time budgets is a similar mechanism, which can be used with a set of tasks instead of a single task. A task can belong to at most one such group. A global budget of execution-time can be allocated to the whole group, and then it decreases as any task in the group consumes execution time. As with timers, a protected procedure handler can be specified to be executed whenever the budget is exhausted (i.e. it reaches zero). The budget can also be replenished at any time.

Execution-time clocks are allowed in the Ravenscar profile, but timers and group budgets are not. However, we believe that these mechanisms can be safely used in high-integrity systems, provided that some restrictions are put in place:

- Timers and groups are declared only at the library level;
- There is at most one execution-time timer per task;
- Ada.Execution_Time.Timers.Min_Handler_Ceiling is equal to the interrupt priority of the hardware-timer which is used to implement the execution-time timers [12]

In the following, we shall assume an extended profile based on Ravenscar augmented with static execution-time timers and group budgets with the above restrictions.

2.3 Fault Detection Patterns

There are two possible approaches when detecting a temporal fault: to detect it *before* or *after* its effects can be propagated. Corrective actions can be performed if the fault is detected before it is propagated (affecting other tasks), but this approach usually introduces more overhead. On the other hand, detecting faults after having been propagated may be useless as there is no time left to do any corrective action.

All the following patterns detect the fault *before* being propagated, except for the minimum inter-arrival time. In this case the pattern directly avoids its effects, and furthermore some additional code is used to detect and report past MAT violations. Of course, a task can detect all the temporal anomalies if the techniques are combined. This is the preferred approach because the patterns do not introduce much overhead as shown in Section 5, but further analysis should be done for tasks with very tight deadlines.

Deadline Overrun. For hard real-time tasks, where no single deadline can be missed, we propose the schemes in patterns 1 (for periodic tasks) and 2 (for sporadic tasks) to detect deadline overruns using a timing event.

If the task does not finish its job before the absolute deadline, the handler will be invoked directly by the hardware timer interrupt routine, in much the

same way as it is done in cyclic executives. Of course, special care must be taken to avoid unbounded non-preemptible actions. Otherwise, if the interrupts of the hardware clock are inhibited during long time intervals, all of the fault-detection techniques described in this paper will fail. As a side note, the ceiling of the handler must be System. Interrupt_Priority 'Last, as specified in the ARM [7, D.15(14/2)].

For periodic tasks it is worth noting that the handler is set *before* going to sleep, and not afterwards, in order to minimize activation jitter. In this case there is no need to make a call to Cancel_Handler because the timing event is rearmed within the call to Set_Handler which is made at the end of the cycle (see pattern 1).

The code for sporadic tasks has some differences. As shown in pattern 2, the call to Set_Handler is placed *after* the task is released, as the absolute deadline is not known until this time. For this reason the timing event must be cancelled at the end of the execution cycle.

The activation time is recorded inside the Release_Event protected object — before opening the Wait barrier in the Signal procedure— instead of setting it in the body of the sporadic task (after the Wait entry returns). In this way, the interference that the sporadic task suffers from higher priority tasks does not disrupt the measurement of the activation, and therefore the absolute deadline is computed with the best precision.

It should be noticed that a deadline overrun cannot happen before arming the timing event —even when the interference is very long— if the response time analysis is correct, and every task with a higher or equal priority uses the fault-detection and isolation techniques described in this section. To simply arm the timing event inside the protected object does not work, due to race conditions. This method would also add some extra blocking time.

Finally, in some cases the timing event must also be cancelled just after completing its job in a periodic task, but this is only needed if additional code (e.g. data logging) is executed at the end of the cycle. Since this kind of code is usually non-critical, the expiration of a timing event within it should not be identified as a deadline overrun.

Minimal Inter-Arrival Time Enforcement. A pattern for preventing a sporadic task from being activated more often than specified by its minimal inter-arrival time is shown as pattern 3. Once the task has completed its job, it suspends itself until the minimum inter-arrival time (counted from the activation time) has elapsed.

This technique is derived from a well known code template [6, §5.7], with two main differences: First, as in pattern 2, the activation time is recorded inside the protected object, so that the next activation is delayed just the required MAT. The second difference is that this pattern also incorporates additional code inside the protected object to detect the MAT violation (see pattern 4). If the barrier is still open when the procedure Signal is invoked, it means the last event was not processed. Then a counter with the current number of missing events is incremented, and the interarrival time is recorded. Therefore, the Wait

Pattern 1. Deadline overrun detection for periodic tasks

```
Overrun : Ada.Real_Time.Timing_Events.Timing_Event;

task body Periodic_Task is
  Next_Activation : Ada.Real_Time.Time := Epoch;
begin
  loop
    Overrun.Set_Handler(At_Time => Next_Activation + My_Deadline,
                        Handler => My_Monitor.Deadline_Handler'Access);
    delay until Next_Activation;
    Do_Actual_Work;
    Next_Activation := Next_Activation + My_Period;
  end loop;
end Periodic_Task;
```

Pattern 2. Deadline overrun detection for sporadic tasks

```
Overrun : Ada.Real_Time.Timing_Events.Timing_Event;

task body Sporadic_Task is
  New_Data        : Data;
  Activation_Time : Time;
  Deadline_Missed : Boolean;
begin
  loop
    Release_Event.Wait (New_Data,  Activation_Time);
    Overrun.Set_Handler(At_Time => Activation_Time + My_Deadline,
                        Handler => My_Monitor.Deadline_Handler'Access);
    Do_Actual_Work (New_Data);
    Overrun.Cancel_Handler (Deadline_Missed);
    -- Log if detected a deadline overrun
  end loop;
end Sporadic_Task;
```

Pattern 3. Minimal inter-arrival time enforcement

```
task body Sporadic_Task is
  Activation_Time     : Time;            New_Data         : Data;
  Interarrival_Time   : Time_Span;       Next_Activation  : Time;
  Num_Missing_Events : Natural;
begin
  loop
    Monitoring_Event.Wait (New_Data,           Activation_Time,
                           Interarrival_Time,  Num_Missing_Events);
    Do_Actual_Work (New_Data);
    -- Log the number of missing events and the inter-arrival time
    Next_Activation := Activation_Time + My_Min_Interarrival_Time;
    delay until Next_Activation;
  end loop;
end Sporadic_Task;
```

entry returns the activation time of the *last invocation* of the Signal procedure, as well as the minimum inter-arrival time and number of missing events since the previous Wait call. The sporadic task can later use this information to report the problem or take an appropriate action.

WCET Overrun. In the same way as timing events can be used to detect missed deadlines, execution-time timers can be used to detect WCET overruns, as shown in pattern 5. This technique is not very intrusive, as it only adds a call to set the timer on each activation. The CPU timer is set just before going to sleep, at the end of the execution cycle, and not after the delay until, in order to minimize the activation jitter. It should also be noticed that there is no need to call Cancel_Handler, as the timer is reset again at the beginning of the loop if there is no WCET overrun. Actually this is the preferred approach because the whole execution cycle should be accounted for, including even the CPU time needed to set the execution-time timer. The pattern for sporadic tasks is similar to this one.

If the task consumes more CPU time than expected, (e.g. due to a programming bug leading to an infinite loop), the timer will expire and corrective actions can be performed, as explained in the next section. Otherwise, the faulty task could interfere with the correct execution of other tasks, making them miss their deadlines. However, if the CPU timer expires when the task is inside a protected operation some of the possible corrective actions (e.g. task abortion or priority lowering) will not be effective until the task leaves the protected object. This is not a problem in Ravenscar because task abortion and dynamic priorities are not allowed, but this can be an issue in full Ada. Therefore, protected operations must be carefully coded, as is always the case, because WCET overruns within them cannot always be immediately corrected. In the worst case, an infinite loop in a protected operation will lead to the whole system being stuck.

The handler will be invoked directly by the hardware timer interrupt routine, as for a timing event. This is the reason for the requirement that the constant Ada.Execution_ Time.Timers.Min_Handler_Ceiling is equal to the hardware-timer interrupt priority.

3 Fault Handling Strategies

Once a temporal anomaly has been detected by means of any of the methods described in Sect. 2, it must be properly handled in order to minimize its impact on the system. While fault handling is a system issue which can be implemented in different ways depending on the application characteristics and the possible consequences of each kind of fault, three basic strategies can be defined: log temporal errors, give the faulty task a second chance, and force a system mode change.

3.1 Error Logging

The first strategy, and the less intrusive one, is just keeping an error log, e.g. for statistical purposes. Obviously, this kind of *soft* strategy, which cannot even be classified as a corrective action, is only valid for systems with a very low level of criticality because, in high-integrity systems, a temporal misbehaviour even from a low criticality task is a potential hazard for the rest of the system and thus this approach is not admissible.

Pattern 4. Release event with inter-arrival time monitoring for sporadic tasks

```
protected Monitoring_Event is
  procedure Signal (New_Data : in Data);
  entry Wait (New_Data          : out Data;
              Activation_Time   : out Time;
              Interarrival_Time : out Time_Span;
              Missing_Events    : out Natural);
private
  pragma Priority (My_Ceiling);
  Current_Data              : Data;
  Signalled                 : Boolean := False;
  Arrival_Time              : Time    := Time_First;
  Current_Interarrival_Time : Time_Span;
  Num_Missing_Events        : Natural;
end Monitoring_Event;

protected body Monitoring_Event is
  procedure Signal (New_Data : in Data) is
     This_Arrival      : constant Time      := Ada.Real_Time.Clock;
     This_Interarrival : constant Time_Span := This_Arrival - Arrival_Time;
  begin
    if Signalled then
      Num_Missing_Events := Num_Missing_Events + 1;
      if This_Interarrival < Current_Interarrival_Time then
        Current_Interarrival_Time := This_Interarrival;
      end if;
    else
      Signalled                 := True;
      Num_Missing_Events        := 0;
      Current_Interarrival_Time := This_Interarrival;
    end if;
    Current_Data := New_Data;
    Arrival_Time := This_Arrival;
  end Signal;

  entry Wait (New_Data          : out Data;
              Activation_Time   : out Time;
              Interarrival_Time : out Time_Span;
              Missing_Events    : out Natural)  when Signalled is
  begin
    New_Data          := Current_Data;
    Activation_Time   := Arrival_Time;
    Missing_Events    := Num_Missing_Events;
    Interarrival_Time := Current_Interarrival_Time;
    Signalled         := False;
  end Wait;
end Monitoring_Event;
```

Pattern 5. WCET overrun detection

```
My_Identity : aliased constant Task_Id := Periodic_Task'Identity;
WCET_Timer  : Ada.Execution_Time.Timers.Timer(My_Identity'Access);

task body Periodic_Task is
  Next_Activation : Ada.Real_Time.Time := Epoch;
begin
  loop
    WCET_Timer.Set_Handler(In_Time=>My_WCET_Budget,
                           Handler=>My_Monitor.Overrun_Handler'Access);
    delay until Next_Activation;
    Do_Actual_Work;
    Next_Activation := Next_Activation + My_Period;
  end loop;
end Periodic_Task;
```

Another possible usage of this strategy is the testing phase, where temporal data, including possible overruns, can be collected in order to get insight into the system behavior.

3.2 Second Chance Algorithm

The second strategy that we propose is to use a *second chance algorithm*. It is a bit more intrusive than the previous one, as it slightly modifies the schedule. The basic idea is to compute the *slack time* of the faulty task, i.e. the amount by which its execution time can be enlarged while still keeping the system schedulable [13]. Slack time can be computed as part of static analysis, as a simple derivation from worst-case response time computation. Then the available slack time can be used to give the faulty task an extra budget in order to let it complete its job.

The justification for using this technique comes from the fact that there are basically two possible causes for an overrun:

- There is a misbehavior (e.g. an infinite loop) which makes the task unable to properly finish its job.
- The WCET value that has been used for static temporal analysis has been miscalculated.

The first kind of failure is really serious because it is likely to prevent the task from ever completing its job, even though its budget is increased. This approach is clearly not suitable for such kind of failure. On the other hand, overruns due to WCET underestimation are the effect of a possibly transitory overload. In this case, giving the task some extra budget may solve a possibly occasional problem without the overhead of more drastic methods such as those discussed in the next section. In other words, provided that there is enough spare processor utilization, it is possible to give a try to the faulty task and check whether it is just a transitory situation that will not happen again, and let the system continue its normal schedule afterwards.

3.3 Mode Change

A last strategy is still possible and essential. When a task is not able to finish its job within the estimated budget, not even after being given additional time with the *second chance algorithm*, it may seriously jeopardize the system integrity. Therefore, corrective actions must be taken in order to prevent undesirable effects.

Acceptable approaches are:

- Go to a safe state and stop the system *(fail-safe)*.
- Change to a safe mode and continue execution with a degraded functionality *(fail-soft)*.

Both are classical examples of fault recovery actions. There is also the possibility of ensuring full fault tolerance, i.e. continuing execution with full system

functionality, but this approach requires a high level of hardware and software redundancy, and we shall not consider it in the following.

A fail-safe behaviour can be implemented in Ada by terminating the faulty task and setting a global termination handler, which is allowed by the Ravenscar profile [7, D.13.1]. The termination handler must include everything which is required to ensure a safe stop. On the other hand, switching to a safe mode requires that the faulty task is disabled or "quarantined" in some way so that it no longer interferes with the rest of the system. This purpose can be achieved by lowering the faulty task priority, so that it can only execute when there are no other ready tasks, using only spare CPU time. Alternatively, the faulty task can be aborted, as it can be supposed that it is not able to do its work any more. However, both approaches are a source of temporal indeterminism, and consequently their temporal behaviour may be difficult to analyse. In fact, both dynamic priority changes and task abortion are forbidden by the Ravenscar profile for this reason.

Moreover, any functional dependence or synchronization with the damaged task must be taken into account because it can induce undesirable collateral effects. For instance, a task may be suspended *ad infinitum* waiting on a barrier that the faulty task should open, or it may return wrong values due to parameters that are never updated by the task. Therefore, just removing a task from the system is only acceptable if there are no other tasks depending on it. Care should also be taken that the task functionality is not vital for the system, so that the rest of the tasks can continue executing without compromising the system integrity. On the other hand, when the functionality of the faulty task is essential for the system, it must be replaced in some way so that the system can continue working, even though its functionality is reduced. This can be accomplished by a mode change.

A mode change involves a deep alteration of the system configuration. As such, the static environment implied by the Ravenscar profile may be altered and temporal analysis may be made difficult or impossible. However, it is possible to implement mode changes in a Ravenscar-compatible way as long as some constraints are in place [14]. Another source of complexity has to do with the number of different modes that have to be planned in order to face all the potential failures of the system. Given that a typical real system can have several dozens of tasks, it is clearly not feasible to envisage a different mode for covering every potential failure of every task. A more achievable solution is to provide a small number of *degraded modes* [15], and switch the system to one of them whenever an important task fails. Typically only basic services are provided in degraded modes. If no degraded mode can be used, then a fail-safe stop can still be performed, provided it is acceptable for the application.

4 Guidelines for Building Monitored Systems

In order to get the maximum profit from modern computing devices it is becoming more and more common to build partitioned systems. These systems execute a number of applications, possibly with different levels of criticality, on a single

computer platform. In order to preserve the integrity of applications from failures occurring in other applications, temporal and spatial isolation mechanisms have to be put in place. Temporal separation can be based on hierarchical scheduling using Ada 2005 priority bands [16], but architectural mechanisms must be supplemented at run-time with monitoring mechanisms for temporal fault detection and handling. The rest of this section provides some guidelines on how to protect the different subsystems of a partitioned system from temporal faults occurring in other subsystems.

The main strategy is to monitor the CPU-time consumption of every individual task. As above discussed, execution-time timers can be used for this purpose, as shown in pattern 5. Group execution-time budgets can also be used in a similar way to account for global overruns affecting a whole subsystem. The next step is to choose a handling strategy among those discussed in Sect. 3. Although the choice is application-dependent, some general guidelines can be given.

The second chance algorithm should be used whenever possible, as it is the best way to face occasional overloads without significantly altering the system configuration. If after a second chance the overload persists, then it can be assumed that there is a serious error and one of the overrun handlers must be used. An annotation in a log may be enough for non-critical tasks with low priority values (possibly monitored by means of group execution-time budgets), as it is the less intrusive technique. If a failure in one of these tasks does not compromise the whole system, it is better to avoid any action outside the Ravenscar profile, leaving more extreme actions for the situations when they are required.

When the overrun occurs in a critical task a mode change is mandatory. In some cases a safe stop is enough, although this may not be acceptable in some kinds of systems. In these cases the last chance to rescue the system consists in changing to a degraded mode, where only the main functionalities can be executed, in order to to minimize as much as possible the effects of a temporal overrun.

5 Implementation and Applications

The Ada 2005 timing mechanisms have been implemented by the authors on GNATforLEON, a compilation system for the LEON2 processor, a radiation-hardened derivative of the SPARCv8 RISC architecture for the space domain. The implementation has been based on a previous experimental implementation on top of the Open Ravenscar kernel [12]. The modified compilation system is being used as the execution platform for the ASSERT project[1]. The fault detection patterns described in this paper have been successfully applied to a comprehensive example of a spacecraft software system including some critical components [17].

[1] ASSERT (Automated proof based System and Software Engineering for Real-Time) is an FP6 Integrated Project coordinated by the European Space Agency. The main goal of the project is to improve the system-and-software development process for critical embedded real-time systems, in the Aerospace and Transportation domains.

Table 1 provides information about the overhead included in a context switch due to the new timing mechanisms (execution-time timers, group budgets, and timing events), comparing the values which have been obtained for GNAT Pro for ERC32, GNATforLEON 1.0 (without the new timing mechanisms), and GNATforLEON 1.3 (including the new timing mechanisms). The values shown in this section have been measured using a pilot application, and therefore should be considered as average values, not as worst-case metrics.

Table 1. Context switch in GNATforLEON

Operation	GNAT Pro ERC32	GNAT LEON 1.0	GNAT LEON 1.3
Context switch	362	405	606

Values are expressed in terms of instructions as the overall timing impact is highly dependant on hardware elements such as caches or registers. The ideal situation is one cycle for instruction, and taking into account that the simulator used in the test runs at 50 MHz, the ideal performance would be 50 MIPS. However, real experiments show that the real performance is about one third of the ideal one.

Table 2. Overhead of timing mechanisms

Operation	Instructions
Ada.Real_Time.Timing_Events.Set_Handler	240
Ada.Execution_Time.Timers.Set_Handler	271
Timing event handler latency	396
Execution-time handler latency	415
Basic_Event.Signal	685
Release_Event.Signal	787
Monitoring_Event.Signal	833

Table 2 shows some measurements, including the cost of signalling a sporadic task by means of different patterns in GNATforLEON 1.3. Basic_Event is the protected object used to release sporadic tasks [6, §5.6], which only opens the barrier (and stores the new data). This object is not used in any of the fault-detection patterns, but its metrics are shown for comparison purposes. The protected object Release_Event is used in pattern 2, and its procedure Signal also reads the clock before opening the barrier. Finally, the protected object Monitoring_Event (pattern 4) reads the clock, detects a minimum inter-arrival time violation when it occurs, and finally opens the barrier. Of course, a sporadic task combining MAT enforcement and deadline-overrun detection uses only one protected object (Monitoring_Event in pattern 2).

Finally, table 3 shows the memory size needed for the different types used in this paper.

Table 3. Memory size

Type	Size (bytes)
Timing_Event	24
Execution_Time.Timers.Timer	20
Basic_Event	32
Activation_Event	48
Monitoring_Event	56

6 Conclusions and Future Work

An analysis of the timing monitoring mechanisms available in Ada 2005 has been carried out, and some patterns for temporal fault detection based on these mechanisms have been developed. Strategies for handling temporal faults have been discussed, taking into account different levels of criticality and real-time requirements. The patterns have been used in a sample on-board embedded system, using a pilot implementation of Ada 2005 timing mechanisms.

Most of the proposed patterns and strategies use some features excluded by the Ravenscar profile definition. The most important ones are execution-time timers and group budgets. An extended profile allowing a limited use of both mechanisms, while still preserving the static nature of the system, can be a solution to this limitation.

Given the fact that system integrity is greatly enhanced by means of the presented protection mechanisms, and the metrics provided show that the overhead produced on the system is within acceptable values, its use in high-integrity systems is highly recommended.

Planned future work includes code instrumentation for obtaining timing metrics of Ravenscar tasks, and modification of the run-time system to gather statistics about relevant temporal values (e.g. jitter, blocking, response time, computation time).

References

1. Burns, A., Wellings, A.J.: Real-Time Systems and Programming Languages, 3rd edn. Addison-Wesley, Reading (2001)
2. Joseph, M., Pandya, P.: Finding response times in real-time systems. BCS Computer Journal 29, 390–395 (1986)
3. Klein, M.H., Ralya, T., Pollack, B., Obenza, R., González-Harbour, M.: A Practitioner's Handbook for Real-Time Analysis. Guide to Rate Monotonic Analysis for Real-Time Systems. Kluwer Academic Publishers, Boston (1993)
4. ISO/IEC: TR 15942:2000 — Guide for the use of the Ada programming language in high integrity systems (2000)
5. Burns, A., Dobbing, B., Romanski, G.: The Ravenscar tasking profile for high integrity real-time programs. In: Asplund, L. (ed.) Ada-Europe 1998. LNCS, vol. 1411, pp. 263–275. Springer, Heidelberg (1998)

6. ISO/IEC: TR 24718:2005 — Guide for the use of the Ada Ravenscar Profile in high integrity systems. (2005) Based on the University of York Technical Report YCS-2003-348 (2003)
7. Taft, S.T., Duff, R.A., Brukardt, R.L., Ploedereder, E., Leroy, P. (eds.): Ada 2005 Reference Manual. LNCS, vol. 4348. Springer, Heidelberg (2006)
8. Dobbing, B., Romanski, G.: The Ravenscar profile: Experience report. Ada. Letters XIX, 28–32 (1999) Proceedings of the 9th International Real-Time Ada Workshop
9. Vardanega, T.: Reflections on the use of the Ravenscar profile. Ada. Letters XXIII, 93–95 (2003) Proceedings of the 12th International Ada Real-Time Workshop (IRTAW12)
10. Vardanega, T.: Development of on-board embedded real-time systems: An engineering approach. Technical Report ESA STR-260, European Space Agency (1999)
11. de la Puente, J.A., Zamorano, J.: Execution-time clocks and Ravenscar kernels. Ada. Letters XXIII, 82–86 (2003) Proceedings of the 12th International Ada Real-Time Workshop (IRTAW12)
12. Zamorano, J., Alonso, A., Pulido, J.A., de la Puente, J.A.: Implementing execution-time clocks for the Ada Ravenscar profile. In: Llamosí, A., Strohmeier, A. (eds.) Ada-Europe 2004. LNCS, vol. 3063, Springer, Heidelberg (2004)
13. Davis, R.I., Tindell, K.W., Burns, A.: Scheduling slack time in fixed priority preemptive systems. In: IEEE Real-Time Systems Symposium (1993)
14. Alonso, A., de la Puente, J.A.: Implementation of mode changes with the Ravenscar profile. Ada Letters. In: Proceedings of the 11th International Real-Time Ada Workshop, vol. XXI (2001)
15. Lundqvist, K., Srinivasan, J., Gorelov, S.: Non-intrusive system level fault-tolerance. In: Vardanega, T., Wellings, A.J. (eds.) Ada-Europe 2005. LNCS, vol. 3555, pp. 156–166. Springer, Heidelberg (2005)
16. Pulido, J.A., Urueña, S., Zamorano, J., Vardanega, T., de la Puente, J.A.: Hierarchical scheduling with Ada 2005. In: Pinho, L.M., González Harbour, M. (eds.) Ada-Europe 2006. LNCS, vol. 4006, Springer, Heidelberg (2006)
17. Dissaux, P., Moretti, R., Barone, M.R., Puri, S., Cancila, D., Bordin, M., Prochazka, M., Najm, E., Hamid, I.: Experience in modelling a general PP problem. Technical report, ASSERT Consortium, D3.1.3-1 I2R0 (2006)