Combining Time-Triggered Plans with Priority Scheduled Task Sets[⋆]

Jorge Real [1], Sergio Sáez [2], and Alfons Crespo [1]

[1] Instituto de Automática e Informática Industrial
[2] Instituto Tecnológico de Informática
Universitat Politècnica de València
Camino de vera, s/n, 46022 Valencia, Spain
{jorge,ssaez,alfons}@disca.upv.es

Abstract. Time-triggered and concurrent priority-based scheduling are the two major approaches in use for real-time and embedded systems. Both approaches have their own advantages and drawbacks. On the one hand, priority-based systems facilitate separation of concerns between functional and timing requirements by relying on an underlying real-time operating system that takes all scheduling decisions at run time. But this is at the cost of indeterminism in the exact timing pattern of execution of activities, namely variable release jitter. On the other hand, time-triggered schedules are more intricate to design since all scheduling decisions must be taken beforehand in the design phase, but their advantage is determinism and more chances for minimisation of release jitter. In this paper we propose a software architecture that enables the combined and controlled execution of time-triggered plans and priority-scheduled tasks. We also describe the implementation of an Ada library supporting it. Our aim is to take advantage of the best of both approaches by providing jitter-controlled execution of time-triggered tasks (e.g., control tasks), coexisting with a set of priority-scheduled tasks, with less demanding jitter requirements.

Keywords: Real-Time Systems. Jitter. Time-Triggered Scheduling. Ada.

1 Introduction

Using concurrent tasks in real-time systems software allows designers to clearly separate functional from timing requirements, letting them focus on functionality and delegating the scheduling of activities on the underlying real-time operating system (RTOS). The RTOS uses a priority scheme to select which task (or tasks, in multiprocessor systems) can execute at a given time [1]. This approach is backed by extensive research results that define the conditions under which this type of systems can be guaranteed to comply with their deadlines at run time.

[⋆] This work has been partly supported by the Spanish Government's project M2C2 (TIN2014-56158-C4-1-P-AR) and the European Commission's project EMC2 (ARTEMIS-JU Call 2013 AIPP-5, Contract 621429).

However, priority scheduling introduces a fundamental issue in tasks with strict release jitter requirements. Release jitter is the difference in time between the theoretical and actual release time of a task. In major application areas, such as automatic control or synchronised distributed communications, excessive release jitter causes performance degradation that needs be avoided. Some amount of jitter is in practice unavoidable, since scheduling decisions take time and that translates into scheduler's overhead that ultimately interferes the whole task set. But in priority scheduled systems, all but the highest-priority task may be additionally interfered by other higher-priority tasks. When this interference straddles a task's theoretical release time, then the task will suffer release jitter until all higher-priority levels become idle.

On the other hand, time-triggered scheduling is based on an offline predefined schedule (a plan) in which the designer identifies the exact points in time when every planned activity must start. From that release time, the activity is granted a time slot to execute, whose duration is, by design, sufficiently large to accommodate its computational needs. No other activity is scheduled until the end of the previous slot. This property translates into small and bounded jitter, because time-triggered activities do not interfere each other and they are solely affected by the scheduler's overhead. In addition, a time-triggered scheduler is comparatively simple, since all scheduling decisions are taken at well-defined points dictated by the static plan. This means small overhead, and therefore, less jitter. The main drawback attributed to time-triggered scheduling is that plans may become difficult to design, specially when they are large and involve a large number of activities.

In this paper we explore an alternative that combines both schemes for the same application. Activities imposing strict jitter requirements are scheduled according to a time-triggered plan, whereas the rest of tasks are scheduled by a priority-based scheduler. The whole set of tasks (time-triggered and priority-scheduled tasks) will be running under the same preemptive, priority scheduler, but time-triggered activities will do it at the highest priority of the whole set. This is in order to ensure minimal latency in their activations, given that they don't suffer interference from (non-existent) higher-priority tasks, and hence their release jitter can be kept controlled and short. The rest of tasks execute at lower priority levels under the same dispatching policy (preemptive, non-preemptive, EDF) or under a combination of several dispatching policies, e.g. by making use of Ada's priority specific dispatching. The Ada programming language is very well suited for our purpose and we will be using it to illustrate our proposal.

Our approach also pays attention to providing temporal isolation to time-triggered activities. A well-designed time-triggered plan guarantees, by construction, temporal isolation among activities. However, run-time guarantees must be provided to cater for potential overruns (execution of an activity could take longer than assumed due to underestimation of its actual worst-case execution time). We want to guarantee that an overrunning time-triggered activity will not jeopardise temporal isolation by executing beyond its allocated slot. This would increase its interference on priority-scheduled tasks and could even make it enter the slot of another time-triggered task and delay its release. There are several ways to handle overruns, their appropriateness being dependent on the application. One possibility is to abort the offending activity, although this may not be an option for some applications. Another way is to take the system to a degraded mode. Our proposed model and implementation support mode changes at the time-triggered level, hence this is always a possibility. But more specific to the overrun issue, our proposal also supports handling overruns by allowing the offending activity to continue executing at a harmless, lower priority level. At that priority level, they may find time to complete by the start of their next allocated time slot, without interfering higher-priority levels. Our approach therefore supports these three models. Which option to take is not imposed by our proposed scheduler, but enforced by the particular pattern implementing the activity. We propose several such patterns in this paper.

Although we confine most of our discussion to uniprocessor platforms, nothing prevents our model to be applied on multiprocessors. This paper, however, focuses on showing how the approach performs in terms of granting a reduced upper bound to the release jitter of time-triggered activities, limiting our study to a uniprocessor example. General considerations for application on multiprocessor systems are given in Section 8.

The rest of this paper is organised as follows. Section 2 presents related work. Section 3 explains our system model for the time-triggered plan. Section 4 describes an interface for the time-triggered scheduler and in Section 5 we propose several patterns for time-triggered activities that make use of the scheduler functionalities. Implementation details are discussed in Section 6. We have conducted several experiments and obtained jitter measurements that are presented and discussed in Section 7. Finally, in Section 8 we give our conclusions and pointers to further work.

2 Related Work

The issue of jitter in control and communication systems has been tackled from different angles. From a Control Engineering perspective, the work in [2] proposes to dynamically adjust the controller's parameters to compensate for the presence of jitter. Our perspective is different, albeit complementary, since our focus is on the minimisation of jitter at run time (while preserving the benefits of priority scheduling for tasks that are more tolerant to variable jitter).

From a scheduling perspective, [3] proposes methods to transform an off-line schedule into an equivalent fixed-priority task set that matches its runtime behaviour. This transformation is however not always possible, in which case the original task set needs be modified by splitting tasks into instances, hence generating a new task set. Our approach does not impose any transformations to the original task set, hence avoiding the need for scheduling artefacts. In [4], the focus is on the control-scheduling co-design of the system. A so called Control Server uses feedback from execution-time measurements and dynamically modifies the sampling periods to optimise control performance. In our proposal, the workload does not need to go through period modifications. Instead, control tasks preserve their timing parameters because they always run at the highest priority, irrespective of lower-priority events, hence experiencing minimal release jitter. Changes to the workload are however possible in our approach by dynamically changing the whole time-triggered plan. In [5], the authors propose to decompose control tasks in three parts: initial, mandatory and final (the IMF model). This decomposition is then used to assign higher priority to the parts that are most sensitive to jitter (initial and final) which in turn reduces the amount of interference they suffer and therefore contributes to reducing their jitter and improving the control performance. In [6], a method is proposed to reduce delay variations caused by overload perturbations. Their task model includes both IMF and non decomposed tasks and their method is to adjust their deadlines dynamically, according to a heuristic algorithm, so that tasks incur less delay. The algorithm is however non trivial and it introduces additional runtime overhead. In our proposed approach, control tasks are scheduled according to a time-triggered plan, hence their release times are clearly identified and deviation from the planned release points can only be caused by the scheduler's overhead, but not from higher-priority interference.

In summary, existing methods that tackle the jitter issue from a scheduling perspective, assume the system uses a priority scheduler and try to minimise the release jitter of selected tasks by finding clever priority assignments and timing parameters for them and by decomposing them into smaller parts. To the best of our knowledge, there is no previous work that tackles this issue by combining the predictability and controlled jitter of time-triggered schedules for jitter-sensitive tasks, with the flexibility of priority scheduling for the rest of tasks, all running under the same priority scheduler but granting the highest priority to the time-triggered plan.

3 System Model

In our system model, an offline, static time-triggered plan coexists with a set of concurrent, priority scheduled tasks. The priority scheme for these tasks can be either fixed per task (e.g., deadline monotonic, DM) or dynamic per task, fixed per job (e.g., earliest deadline first, EDF). These priority-based models have been extensively described in the literature and are fully supported in Ada [7]. Ada also supports the concept of priority-specific dispatching, which makes it possible to have a combination of dispatching policies conveniently spread over priority bands. In the following subsections we describe the system model for the static time-triggered plan, both in regard to what defines a plan and what are the actions taken by the time-triggered scheduler, and when those actions are executed. By assigning the time-triggered plan the highest priority, the set of priority scheduled tasks does not interfere the execution of the plan.

3.1 The Time-Triggered Plan

A time-triggered plan is described by an ordered sequence of *time slots*. Figure 1 shows a 6-slot example plan. Each slot has its own sequence number (a natural number), and is characterised by two parameters: a **work identifier**, (*Work_Id*), an integer value ultimately referring to either a piece of user-provided application code or a predefined scheduler action; the **slot duration**, a time interval after which the next slot starts. All scheduling decisions are made exclusively at the beginning of each slot. When designing the plan, the slot duration should be made large enough to accommodate the execution of the work denoted by the slot's work identifier

For example, *Slot 2* in Figure 1 allocates 300 time units for the execution of *Work 3*. The whole plan sequence starts at a given time (identified here as time 0) and each slot starts right after the end of its predecessor in the plan. In the absence of mode changes (see later), the plan is repeated cyclically. In some cases, and for some types of slot, the slot duration may be zero. We consider three types of slots, depending on the kind of activity that must be executed during the slot duration:

– A ***regular slot*** defines a time interval for the execution of an application-specific activity. It is denoted by a *regular Work_Id* and a strictly positive slot duration. For *regular Work_Id* we mean a positive integer corresponding to a regular work identifier, i.e. one that ultimately refers to a piece of user-supplied application code. The duration of a regular slot must be, by design, sufficient to accommodate the worst-case execution time of that work – we will consider overrun handling in subsection 3.2. In Figure 1, slots 0, 1, 2 and 4 are regular.

The following two types of slots correspond to scheduler actions exclusively and they have no associated application-specific activity.

– An ***empty slot*** defines a time interval during which no user activity is planned. This is useful for inserting gaps in the plan where they are needed, making the CPU available to priority scheduled tasks. Note that, even though there is no application-specific activity to execute during an empty slot, there will be scheduler actions executed at the beginning of the slot, as described in subsection 3.2. Empty slots use the special value zero as $Work\_Id$. Slot 5 in Figure 1 is an empty slot.
– A ***mode-change slot*** defines a point in time where it is possible to substitute the current plan with a new one. This polling approach is consistent with the nature of time-triggered scheduling, although the definition of mode-change slot provides an extra degree of flexibility, since the designer can place these polling points wherever the system can admit a mode change. At the start of a mode-change slot, the scheduler will check whether there is a pending mode-change request to process. If there is one, then the new plan will start executing at the end of the mode-change slot. The change will be immediate if the mode-change slot duration is defined to be zero. The ability to change mode (substitute the current plan with a new one at run time) introduces a degree of flexibility that off-line, static schedules do not possess by nature. The inclusion of mode change slots provides a flexible means to specify in which points of the plan a mode change can be enforced. Mode change slots (such as Slot 3 in Figure 1) are identified by $Work\_Id = -1$.

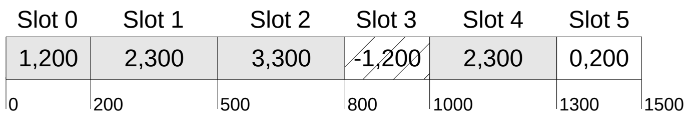

Fig. 1. A time-triggered plan. For each slot, the first number is the work identifier and the second is the slot duration. Slot 3 is a mode change slot. Slot 5 is an empty slot.

Note that each slot can accommodate at most *one* application-specific activity, as opposed to the classic *cyclic executive* [8, 9]. This has several advantages: one is that we want to have the highest possible control over release jitter, which cannot be accomplished if several activities of varying execution times share the same slot; another reason is that, with only one activity per slot, the scheduler only needs to check one activity at a time for potential overrun (see next subsection), which helps keep the scheduler simple, and consequently helps keep release jitter small. Another substantial difference is that each slot may have a different duration, as opposed to the fixed duration of minor frames in the classic cyclic executive.

3.2 The Time-Triggered Scheduler

The time-triggered scheduler is the element of the system that enforces the timely execution of the time-triggered plan. This includes not only releasing the activities at their predefined release times, but also controlling that all activities behave as per the plan's design. In particular, the scheduler must check and take correcting actions for possible overruns, i.e., activities whose actual execution time may exceed their allocated slot duration. The scheduler must also give support to mode changes, as described in the previous subsection. Contrary to the case of priority-based schedulers, where scheduling decisions are taken at arbitrary points in time, all the decisions and actions of the time-triggered scheduler must be taken and executed at predefined points in time; in our case, at the start of each slot. Note that this is not necessarily a periodic event, since slots may have different durations.

At the start of **any slot**, the scheduler checks whether there is still pending work from the previous slot. Since the slot duration must accommodate, by design, the worst-case execution time of its work, continued execution of an activity from the previous slot constitutes an overrun. There are several possible ways to treat this situation. One possibility is to lead the system to a fail-safe state (as in [10]), which can be achieved by means of a mode change as we will show later. For the rest of this paper, we take a *less drastic* approach and allow the offending activity to continue executing at a lower priority, so that it only affects a particular set of tasks in the system (including an empty set, if the overrunning activity is set to *background* priority). The particular *demoted priority* can be specified for each activity as will be shown in the following sections. After this overrun check, the scheduler takes different actions depending on the type of slot:

Regular slot: The scheduler releases the execution of the slot's activity, denoted by its regular *Work_Id*, and assigns it the time-triggered level priority.

Empty slot: No time-triggered activity needs be executed until the arrival of the next slot. During an empty slot, time is fully available for priority scheduled tasks.

Mode change slot: The scheduler checks whether there is a pending mode change request. If there is one, then the current plan is substituted with the new mode plan (which may be totally different) and the next slot will be the first slot of the new plan. Otherwise, the slot duration is also available for priority scheduled tasks.

The actual implementation details of mode changes depend on the concrete platform. Ideally, the hardware includes enough memory resources to allocate all the required time-triggered tasks for all modes. On platforms with scarce resources however, it may be necessary to delete old-mode tasks and load new-mode tasks to memory. The time needed for these operations, as well as any additional overhead incurred to enforce the mode change, can be absorbed by the mode-change slot duration.

4 API for Time-Triggered Plans

We propose an Application Program Interface (API) for Ada programs to use time-triggered plans, possibly in combination with other concurrent, priority scheduled tasks. The API for time-triggered plans is provided via the Ada package Time_Triggered_Scheduling. Listing 1 shows the most relevant aspects of its specification.

The type Any_Work_Id refers to work identifiers in general, including regular and *special* work identifiers (such as empty slots and mode change slots, as described in subsection 3.1). Subtype Special_Work_Id covers the negative range of Any_Work_Id, plus the value zero, whereas subtype Regular_Work_Id refers to strictly positive numbers that correspond to regular work identifiers. Constants Empty_Slot and Mode_Change_Slot (assigned in the private part of the package) identify their corresponding special work identifiers.

The record type Time_Slot encapsulates the two defining elements of time slots: their duration and the work identifier for that slot. Additionally, we include the record field Next_Slot_Separation, whose value is to be supplied at design time, indicating the time separation between the start of the current slot and the next slot in the plan allocated to the same work. The use of this piece of information is further explained in Section 5.

A time-triggered plan is an ordered sequence of time slots, as represented by the array type Time_Triggered_Plan. Additionally, the access type Time_Triggered_Plan_Access provides access to time-triggered plans, so that plans can be efficiently passed as parameters to subprograms.

Listing 1. Time-triggered API (incomplete)

```
-- Context clauses omitted
package Time_Triggered_Scheduling is
   type Any_Work_Id is new Integer;
   subtype Special_Work_Id is Any_Work_Id range Any_Work_Id'First .. 0;
   subtype Regular_Work_Id is Any_Work_Id range 1 .. Any_Work_Id'Last;
   Empty_Slot       : constant Special_Work_Id;
   Mode_Change_Slot : constant Special_Work_Id;

   type Time_Slot is record
      Slot_Duration        : Time_Span;
      Work_Id              : Any_Work_Id;
      Next_Slot_Separation : Time_Span; -- Distance to next slot of same Work_Id
   end record;
   type Time_Triggered_Plan is array (Natural range <>) of Time_Slot;
   type Time_Triggered_Plan_Access is access all Time_Triggered_Plan;

   protected type Time_Triggered_Scheduler (Nr_Of_Work_Ids: Regular_Work_Id)
     with Priority => System.Interrupt_Priority'Last is
      -- Setting a new time-triggered plan
      procedure Set_Plan (TTP : in Time_Triggered_Plan_Access; At_Time : in Time);
      procedure Set_Plan (TTP : in Time_Triggered_Plan_Access; In_Time : in Time_Span);
      -- Time-triggered tasks wait here for their activation
      entry Wait_For_Activation (Work_Id : Regular_Work_Id);

      -- Features for composed task patterns
      -- Continue TT task at default or given demoted priority
      procedure Leave_TT_Level (Work_Id : Regular_Work_Id);
      procedure Leave_TT_Level (Work_Id : Regular_Work_Id; Prio: System.Priority );
      -- Release time of the last slot of a given Work_Id
      function Get_Last_Release (Work_Id : Regular_Work_Id) return Time;
      -- Duration of the last slot of a given Work_Id
      function Get_Last_Slot_Duration (Work_Id : Regular_Work_Id) return Time_Span;
      -- Separation between the start of the last slot and the next slot of a given Work_Id
      function Get_Next_Slot_Separation (Work_Id : Regular_Work_Id) return Time_Span;
   private
      -- ... Further details in listing 4
   end Time_Triggered_Scheduler;

private
   Empty_Slot       : constant Special_Work_Id := 0;
   Mode_Change_Slot : constant Special_Work_Id := -1;
   -- ... Further details in listing 4
end Time_Triggered_Scheduling;
```

Listing 1 continues with the definition of protected type Time_Triggered_Scheduler, which encapsulates all data and subprograms used to implement the time-triggered scheduler. The type has a discriminant (Nr_Of_Work_Ids) to specify the number of regular work identifiers used by all plans in all modes. Based on this number, we define bounded data structures (in the private part of the protected type, not shown here) that are needed for the scheduler's operation. The priority of the time-triggered scheduler is set to the maximum to prevent interference from other parts of the system with its operations.

Care has been taken to implement all the provided operations using constant cost subprograms. The protected procedure Set_Plan sets a new plan (given by parameter TTP) to be started after a given point in time – the two versions of Set_Plan differ only in using a relative or an absolute value for that starting time. A plan set by means of Set_Plan will start executing immediately after that starting time if there was no plan running (it was the first plan to be set) or at the end of the next mode change slot otherwise. The entry Wait_For_Activation suspends the calling task until the work corresponding to its Work_Id must be released, according to the current plan.

The rest of protected subprograms are useful for composed task patterns such as those described in Section 5. With Leave_TT_Level, a time-triggered work requests to continue executing at a default demoted priority, or a particular priority for each invocation. This is useful for works with an optional part that cannot be granted by the plan due to an excessive or unbounded worst-case execution time. The optional part can continue executing in competition with priority scheduled task and calculate the best possible response in the available time, without interfering other planned activities. The three getter functions provide the indicated values: the time when the calling work was last released (Get_Last_Release); the duration of the slot in which the work was last released (Get_Last_Slot_Duration); and the time distance between the work's current slot and the next slot in the plan that is allocated to that work (Get_Next_Slot_Separation). The following Section shows how different patterns can take advantage of these functions.

5 Patterns for Time-Triggered Tasks

Common practice in time-triggered systems is to have all activities implemented by subprograms that are directly called from the scheduler. We have taken a different approach, whereby every activity (work) is executed by its own associated Ada task: there is one task behind each work. Before we justify this implementation decision, note that we are considering time-triggered schedules as part of a more complex system that includes also other priority-scheduled tasks. Hence we are not imposing here a special requirement on the operating system or runtime support: a priority-based, preemptive scheduler is given for granted. Our approach is to implement each time-triggered work with a high-priority task and let the RTOS decide which task to execute at a given time, be it a time-triggered task or a priority-scheduled task. In addition, the implementation must also be prepared for demoting overrunning time-triggered tasks, as explained in Section 3. This feature alone requires that works must be executed by tasks whose priorities can be changed by the scheduler at run time. Hence we use task types to define patterns. We observe however that communication between works requires protected objects (not just shared memory) if we allow overrunning time-triggered tasks to continue executing at a demoted priority beyond their allocated slot, concurrently with other tasks. Using protected objects ensures that priority demotion of overrunning tasks occurs only when data integrity is not compromised ([7], D.5.1).

The API described in Section 4 may be used for implementing time-triggered works of different complexity. Listing 2 shows the simplest pattern for time-triggered tasks we can think of, implemented by task type Simple_Worker. The task first calls the scheduler's entry Wait_For_Activation. The scheduler will then keep the calling task blocked until a slot arrives in which its work identifier is planned to execute. Upon completion of the call to Wait_For_Activation, the task then executes its specific work actions. This is repeated in an infinite loop. Worker tasks are created by instantiation of this task type.[1] Each instance must use a different value for the discriminant Work_Id – this is checked at runtime by the scheduler and the exception Program_Error is raised if a task tries to use another task's work identifier. The discriminant Prio specifies the default demoted priority, i.e., the priority to which the task will be demoted in case of overrun or when it calls Leave_TT_Level without specifying a demoted priority value. We use the CPU aspect here to set the affinity of all time-triggered tasks to the same processor, although this is not compulsory. On a multiprocessor platform, each processor may be running a different plan and each work task must be confined to its respective CPU.

Listing 2. Simple pattern for time-triggered tasks

```
TTS: Time_Triggered_Scheduler(3); -- A scheduler for 3 different works ( arbitrary )

task type Simple_Worker (Work_Id: Regular_Work_Id; Prio: System.Priority )
  with Priority  => Prio, -- Demoted priority in case of overrun
       CPU       => 1;    -- Set task's affinity
task body Simple_Worker is
begin
   loop
      TTS.Wait_For_Activation (Work_Id); -- Block here until my slot arrives
      Do_My_Work (...);                  -- Specific work actions
   end loop;
end Simple_Worker;
```

More elaborated task patterns are also supported by the scheduler described in section 4. In particular, we propose the following additional patterns:

Worker_With_Cancellation Before causing an overrun, a task following this pattern will cancel its activity, instead of following the default behaviour of continuing its execution at a demoted priority level. This pattern is intended for tasks that cannot contribute any value after their allocated slot duration, for example because their result must be applied to a system output immediately.
The pattern modifies the Simpe_Worker by enclosing the Do_My_Work sentence in the abortable part of an Ada asynchronous transfer of control statement. The triggering alternative is an absolute delay until the end of the current slot, hence the work will be aborted before incurring overrun.[2] This time is obtained by adding the slot duration to its corresponding start time (Get_Last_Slot_Duration + Get_Last_Release).

Worker_With_Initial_Final This pattern is conceived for works that require controlled and short jitter both at the beginning and towards the end of their activity. The work is said to have an *initial* part (e.g., sensing a physical environment variable) and a *final* part (e.g., the actuation phase of a control algorithm).
This pattern is a simple duplication of the loop actions of Simple_Pattern: there are two calls to Wait_For_Activation, one preceding the initial part and one preceding the final part of the work. Note that the same effect can in principle be obtained by two works, one for the initial part and one for the final. Using this pattern, however, the advantage is that all communication between the initial and the final

[1] Note that, for general application of the pattern, the actions represented by Do_My_Work are different for each work. We have kept the patterns simple, but in actual systems the task's actions should be determined more flexibly (e.g. by using access to subprogram or generic packages for task patterns).

[2] To be on the safe side, we should subtract the worst-case duration of abort-deferred operations in the work's code. This would avoid the work to cross a slot barrier while executing an abort-deferred operation.

part is immediate since both parts share the common task's stack (no inter-task communication is needed).

Worker_With_Initial_Optional_Final This pattern is for activities with initial and final parts with strict jitter restrictions, plus an optional part between them. The optional part may implement an optimisation algorithm for improving a *quick and dirty* result obtained during the time allocated to the initial part. The execution time of optimisation algorithms may be quite disperse, and hence it is not easy to define their required slot duration: too large a duration would impose delays to other activities; too short and the potential for run-rime overruns increases.

This pattern executes first the initial part until completion. After calling procedure Leave_TT_Level, the task continues with the optional part at a demoted priority. When the optional part is completed, the task will wait again for activation until the arrival of the next slot corresponding to its work identifier. In that slot, the work's task executes the final part using the best result obtained during the optional part. If the optional part has not finished by the time when starting the final part is due, then the optional part is aborted (as in the Worker_With_Cancellation pattern). Listing 3 gives the implementation of pattern Worker_With_Initial_Optional_Final .

Listing 3. Pattern for works with initial, optional and final parts

```
task body Worker_With_Initial_Optional_Final is
  -- Common data to all parts goes here
begin
  loop
    TTS.Wait_For_Activation(Work_Id);
    Initial_Work ;   -- Do initial part
    TTS.Leave_TT_Level(Work_Id,Optional_Part_Prio); -- Prepare to start optional part
    select
      delay until TTS.Get_Last_Release(Work_Id) + TTS.Get_Next_Slot_Separation(Work_Id);
    then abort
      Optional_Work; -- Do optional part
    end select;
    TTS.Wait_For_Activation(Work_Id);
    Final_Work;        -- Do final part
  end loop;
end Worker_With_Initial_Optional_Final ;
```

Figure 2 shows the execution of an example plan with three time-triggered tasks (work tasks 1, 2 and 3) and two priority-based tasks (T4 and T5). Work 1 is a Simple_Worker, work 2 is a Worker_With_Initial_Final and work 3 uses the more elaborated Worker_With_Initial_Optional_Final pattern. T4 and T5 execute at their lower priorities, using the time made available by empty slots and early completion of work tasks. Work 3 starts executing the initial part (marked 3_I), which gets completed before the end of the allocated slot duration. It then calls Leave_TT_Level to continue the execution of the optional part (marked 3_O) at a given priority, in competition with the rest of priority- or time-triggered-scheduled tasks. In this case, the demoted priority is half way between the priorities of T4 and T5. When the optional part completes, it calls Wait_For_Activation to wait for the arrival of the final part slot (marked 3_F). Note that the optional part is abortable, hence it can be forced to not cause overrun. All we need to do is set the delay of the triggering statement to the right value: by the arrival of the next slot allocated to this work. We obtain our last activation time by using the above mentioned extension Get_Last_Release. To this time, we need to add the duration of all slots in between the current slot and the next slot for the current *Work_Id*. This imposes a time cost (traversing the plan) that we do not want to charge on the scheduler at

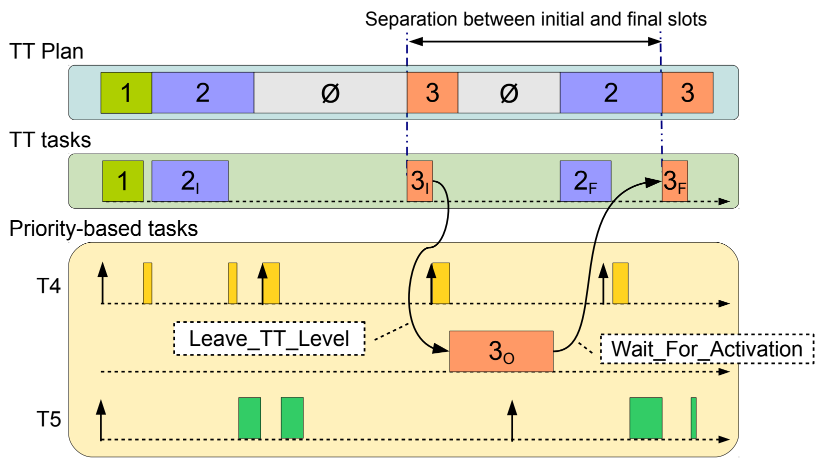

Fig. 2. Execution of a **Worker_With_Initial_Optional_Final** pattern.

run time. To avoid this overhead on the scheduler, we use the Next_Slot_Separation field of Slot_Type record. The separation to my next slot can be easily calculated at design time and stored in the plan using this field, where the scheduler can read it immediately. The API function Get_Next_Slot_Separation returns precisely this value for a given *Work_Id*.

6 Implementation Details

A thorough description of all implementation details is not possible here due to space limitations. We will limit ourselves to the most relevant details in terms of their impact on jitter, i.e., the actions taken by the scheduler to timely enforce the plan. We will omit discussing the implementation details of (much less frequent) mode changes.

As shown in Section 4, a time-triggered scheduler is enclosed in a protected object with the highest priority. This grants mutually exclusive access to it. Listing 4 shows the private parts of package Time_Triggered_Scheduling and protected type Time_Triggered_Scheduler, which were omitted in Listing 1. They include all required types and state variables needed for the time-triggered scheduler to enforce the execution of the plan according to the model described so far.

Listing 4 shows the private details of the time-triggered scheduler. The private part of the protected type includes the entry family Wait_Until_Released, with as many members as work identifiers used in the system (across all modes). When a worker task calls the scheduler's entry Wait_For_Activation, it is ultimately requeued to its corresponding entry family member, where it waits until its specific barrier is open by the scheduler. All barriers are simple booleans stored in Work_Control, one per work identifier – more specifically, the field Allow_Release of the record type Work_Info.

Registration of work tasks in the plan is automatically handled by the scheduler the first time a task calls Wait_For_Activation. Registration consists in taking note of the caller's Task_Id and its default demoted priority, taken from the caller's base priority.

Additional checks are enforced by the scheduler to make sure the calling task is the one that registered for the specified work identifier. Program error is raised otherwise.

Listing 4. Private parts of scheduler package and protected type

```
package Time_Triggered_Scheduling is
   ... -- Types for storing runtime information
   type Work_Info is private;
   type Work_Info_Array is array (Regular_Work_Id range <>) of Work_Info;

   protected type Time_Triggered_Scheduler (Nr_Of_Work_Ids: Regular_Work_Id) ... is
     -- See full spec in Listing 1
   private -- Of protected type
     entry Wait_Until_Released (1 .. Nr_Of_Work_Ids);     -- Entry family: one entry per work
     procedure MC_Handler (Event : in out Timing_Event); -- Handler for mode change timing event
     procedure NS_Handler (Event : in out Timing_Event); -- Handler for new slot timing event
     procedure Change_Plan (At_Time : Time);              -- Enforce plan change
     procedure Update_Slot_Info;                          -- Update indexes and times to new slot
     Current_Plan, Next_Plan : Time_Triggered_Plan_Access := null; -- Current and next plans
     NS_Event, MC_Event      : Timing_Event;              -- New Slot and Mode Change TEs
     Current_Slot_Index, Next_Slot_Index  : Natural:= 0;  -- Relevant indexes and times
     Next_Mode_Release, Next_Slot_Release : Time := Time_Last;
     Work_Control : Work_Info_Array (1.. Nr_Of_Work_Ids); -- Runtime work info
   end Time_Triggered_Scheduler;
private -- Of package
   ...
   type Work_Info is record
      Is_Running: Boolean:= False;          Demoted_Priority: System.Priority := System.Priority'First;
      Work_Task_Id: Task_Id:= Null_Task_Id; Allow_Release: Boolean:= False;
      Last_Release: Time:= Time_Last;       Last_Slot_Index: Natural:= 0;
   end record;
end Time_Triggered_Scheduling;
```

Listing 5. Handler for the new slot timing event

```
procedure NS_Handler (Event : in out Timing_Event) is
   Current_Work_Id : Any_Work_Id;    Now : Time;
begin
   -- Check for overrun (Current_Slot_Index refers to the just expired slot)
   if Current_Slot_Index in Current_Plan'Range then
      Current_Work_Id := Current_Plan (Current_Slot_Index).Work_Id;
      if Current_Work_Id in Regular_Work_Id and then -- Regular work
         Work_Control(Current_Work_Id).Is_Running then -- Still running => demote
            Set_Priority (Work_Control(Current_Work_Id).Demoted_Priority,
                          Work_Control(Current_Work_Id).Work_Task_Id);
      end if;
   end if;
   -- Prepare to process current slot
   Now := Next_Slot_Release;                  -- Start time of current slot
   Update_Slot_Info;                          -- Update Current_Slot_Index and Next_Slot_Release
   Current_Work_Id := Current_Plan(Current_Slot_Index).Work_Id; -- Obtain current Work_Id
   case Current_Work_Id is   -- Process current slot actions
      when Mode_Change_Slot =>
         if Next_Plan /= null and then Next_Mode_Release <= Now then
            Change_Plan (Next_Slot_Release); -- Enforce new plan at the end of MC slot
         else
            NS_Event.Set_Handler (Next_Slot_Release, NS_Handler'Access); -- Reprogram NS_Event
         end if;
      when Empty_Slot =>
         NS_Event.Set_Handler (Next_Slot_Release, NS_Handler'Access); -- Reprogram NS_Event
      when Regular_Work_Id'Range => -- It's a regular slot
         Work_Control(Current_Work_Id).Allow_Release := True;        -- Release the work's task
         NS_Event.Set_Handler (Next_Slot_Release, NS_Handler'Access); -- Reprogram NS_Event
      when others =>
         raise Program_Error with "Undefined Work Id";
   end case;
end NS_Handler;
```

The scheduler may be triggered by two possible timing events: NS_Event, which signals the arrival of a new slot, and MC_Event for mode change events. Their handlers are, respectively, NS_Handler and MC_Handler. As justified above, here we describe only the handling of the new slot event, implemented by protected procedure NS_Handler.

Other objects declared in the private part of protected object Time_Triggered_Scheduler include Change_Plan, which assigns control variables and programs the NS_Event timing event for the first slot of the next plan to switch to; Update_Slot_Info, also a simple procedure that updates control variables when a new slot starts; eight control variables used by the scheduler; and Work_Control, the array of work control blocks.

Listing 5 shows the handler for the timing event signalling the arrival of a new slot (NS_Handler). According to the model described in Section 3, there are different checks to make at the start of every slot. The first one is to detect overrunning work from the just expired slot. The two nested *if* statements check that the task associated to the previous slot is still running, in which case it is demoted to its Demoted_Priority, which is retrieved from its corresponding work control block. The task will continue at a non-disturbing priority level, where its interference is bounded to what is acceptable by the rest of application tasks.

The handler then goes on with processing the just started slot. After updating some indexes and times, the new slot is processed in a *case* statement. If it is a mode change slot and there is a pending mode change request (a revious call to Set_Plan has set a non null value for the Next_Plan control variable), then the new mode is enforced at the end of the current slot if the starting time of the new plan is not in the future – if it happens to be in the future, then the mode change handler (not described here) will take care of changing the plan. If there is no pending plan change to process, then the new slot timing event is reprogrammed for the start time of next slot. In the case of an empty slot, we just need to reprogram the next slot timing event. If it is a regular slot, then the scheduler opens the barrier for the work task, which will be released at the highest priority immediately after completion of the handler, and reprograms the new slot timing event. In the body of each member of Wait_Until_Released, where works wait to be released by the scheduler, the task's priority is set to the time-triggered level (it could have been demoted due to a previous overrun, or a call to Leave_TT_Level, or it could be the first activation of the task). The other two simple operations in Wait_Until_Released are to close again its barrier and to mark the work as *running* in its work control block.

We note that all the required scheduler functionality can be implemented using three types of sentences: simple assignments, setting one task's priority, and setting one timing event. How efficiently these two last operations operations are supported by the runtime is of crucial importance to keep the scheduler's overhead small and hence to cause minimal jitter to work tasks. The time needed to release a work task contributes also to the scheduler's overhead; but that would be the only blocked task in its corresponding Wait_Until_Released member, which contributes to shorten the completion of that protected action.

7 Experimental Results and Discussion

In order to evaluate the performance of the proposed approach, we have conducted experiments to measure release jitter of a combined set of tasks: three time-triggered tasks plus two deadline monotonic tasks. The system matches the one depicted in Figure 2.

The two deadline monotonic tasks, T4 and T5, execute at lower priorities 6 and 4, and have periods of 325 and 500 ms, respectively. The total plan duration is 1200 ms and it contains the following sequence of 7 slots: 100 ms for W1; 200 ms for WI2; an empty slot of 300 ms; 100 ms for WI3; an empty slot of 200 ms; another 200 ms for WF2; and finally 100 ms for WF3.

We compiled this system for MaRTE OS [11] in bare machine configuration and executed it on two hardware platforms, one using a Celeron CPU at 1.8 GHz and the other using an older Pentium III at 800 MHz. Figure 3 shows cumulative frequency histograms of release jitter measured on both platforms for all tasks in the system. Note that the X axes are pseudo-logarithmic and cover the range from 0 to 1 second.

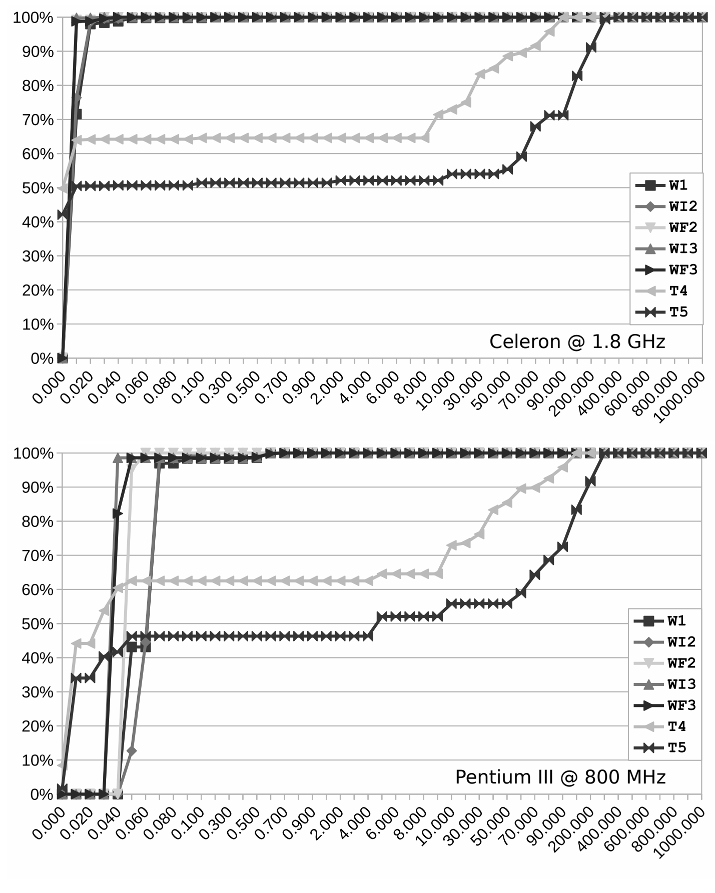

Fig. 3. Cumulative frequency histograms of jitter, measured in ms.

On both platforms the results are comparable in terms of trend. Priority scheduled tasks T4 and T5 experience a wide range of jitter values. In 50 to 60 % of the cases, jitter is comparable to that of time triggered tasks, but then there is a slowly growing trend with release jitters up to 140 ms of T5 on the Celeron and 395 ms on the slower Pentium III. This makes T4 and T5 inappropriate for implementing control algorithms or precisely synchronised communications. Time-triggered tasks experience a maximum jitter of 272 μs on the Celeron and 702 μs on Pentium III. Furthermore, in 98.5 % of the cases, jitter on time-triggered tasks was below 30 μs on the Celeron and 80 μs on the Pentium III. Even considering the totality of cases, the results on maximum jitter are 3 orders of magnitude apart between time-triggered and priority-scheduled tasks.

Looking at minimum jitter values, we observe (more clearly in the Pentium case) that priority-scheduled tasks experience shorter minimum jitter than time-triggered tasks. This occurs when they are released at idle times, when they are free from higher-priority interference. This was expected because releasing a time-triggered task has the additional overhead of the timing event, plus priority promotion, plus completing the protected action implemented by the Wait_Until_Released entry.

8 Conclusions and Future Work

This paper has proposed and explored an approach that allows a time-triggered plan to run under the same priority scheduler where other priority-scheduled tasks are running. By using the highest priority level for the time-triggered schedule, and controlling the scheduler by means of a timing event, the effect is a two-level scheduler that ensures precedence of time-triggered activities over priority-scheduled tasks, which is essential to keep release jitter low for time-triggered activities.

We have also proposed several programming patterns for task time-triggered activities, from the simplest cyclic pattern, to patterns accommodating the structure of decomposed tasks, an approach proposed for control tasks that can also be used for other purposes such as handling communications in networks requiring strict synchronisation (e.g., the CAN bus).

Experimental data indicate that all time-triggered tasks are subject to similar interference, bounded to values that are, in the vast majority of cases, orders of magnitude lower than the release jitter experienced by priority-scheduled tasks. Our approach naturally accepts previously designed time-triggered plans, and facilitates the extension of those plans with additional priority-scheduled tasks. There are other aspects of the proposal, not covered in this paper, that are the subject of current and future work. They include:

Use on multiprocessor platforms Although we have limited our experiments to a single CPU, the approach presented in this paper is applicable to multiprocessor platforms. In a fully partitioned system, each processor executes its own plan and work tasks have their affinity statically assigned. A certain amount of migration is also possible, whereby work tasks can alternate slots of plans supported by different processors, to balance the overall time-triggered workload. Global scheduling of work tasks (i.e. allowing them to migrate at arbitrary points in time) seems not appropriate in this case, since plans assign slots to one and only one work task.

Schedulability analysis A schedulability analysis is needed to assess the feasibility of the full task set, including both the time-triggered plan and the priority-based scheduling levels. The plan can be guaranteed by construction, since it executes at the highest priority level and suffers no interference from priority-based tasks. But the analysis of priority-based tasks needs to take into account the interference caused by the execution of the higher-priority plan. One possibility is to consider the whole plan as a real-time transaction, as defined in the computational model of [12]. The period of the transaction would be the length of the plan and each time slot can be considered as a task of the transaction with a static offset equal to its release time. Adjacent time slots can be considered as a single task in the equivalent transaction. This transaction has the highest priority and the interference introduced in lower priority levels can be computed as described in [12].

Tools and integration with real-time framework Designing a plan can be a difficult task, especially for multiprocessors and with a certain degree of migration. Development of software tools to ease building and analysing these combined systems would be of great value. Additionally, the integration of this approach with existing real-time frameworks (such as those proposed in [13–15]) would facilitate the use of pre-designed periodic tasks patterns, and the independent handling of modes at the two different levels, priority-based and time-triggered. We want to explore the feasibility and properties derived from such integration effort.

References

1. Liu, C., Layland, J.: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. Journal of the ACM **20**(1) (1973) 46–61
2. Martí, P., Fuertes, J., Fohler, G.: Jitter Compensation for Real-Time Control Systems. In: Real-Time Systems Symposium. (2001)
3. Dobrin, R.: Combining Off-line Schedule Construction and Fixed Priority Scheduling in Real-Time Computer Systems. PhD thesis, Mälardalen University (2005)
4. Cervin, A.: Integrated Control and Real-Time Scheduling. PhD thesis, Lund Institute of Technology (April 2003)
5. Balbastre, P., Ripoll, I., Vidal, J., Crespo, A.: A Task Model to Reduce Control Delays. Real-Time Systems **27**(3) (September 2004) 215–236
6. Hong, S., Hu, X., Lemmon, M.: Reducing Delay Jitter of Real-Time Control Tasks through Adaptive Deadline Adjustments. In IEEE Computer Society, ed.: 22nd Euromicro Conference on Real-Time Systems – ECRTS. (2010) 229–238
7. ISO/IEC-JTC1-SC22-WG9: Ada Reference Manual ISO/IEC 8652:2012(E), URL: `http://www.ada-europe.org/manuals/LRM-2012.pdf` (2012)
8. Baker, T.P., Shaw, A.: The cyclic executive model and Ada. In: Proceedings IEEE Real Time Systems Symposium 1988, Huntsville, Alabama. (1988) 120–129
9. Liu, J.W.S.: Real-Time Systems. Prentice-Hall Inc. (2000)
10. Pont, M.J.: The Engineering of Reliable Embedded Systems: LPC1769 edition. Number ISBN: 978-0-9930355-0-0. SafeTTy Systems Limited (2014)
11. Aldea, M., González-Harbour, M.: MaRTE OS: An Ada Kernel for Real-Time Embedded Applications. Reliable Software Technologies - Ada Europe 2001, Lecture Notes in Computer Science **2043** (2001) 305–316
12. Palencia, J., González-Harbour, M.: Schedulability Analysis for Tasks with Static and Dynamic Offsets. In: 9th IEEE Real-Time Systems Symposium. (1998)
13. Wellings, A.J., Burns, A.: A Framework for Real-Time Utilities for Ada 2005. Ada Letters **XXVII**(2) (August 2007)
14. Real, J., Crespo, A.: Incorporating Operating Modes to an Ada Real-Time Framework. Ada Letters **30**(1) (April 2010) 73–85
15. Sáez, S., Terrasa, S., Crespo, A.: A Real-Time Framework for Multiprocessor Platforms Using Ada 2012. In Romanovsky, S., Vardanega, T., eds.: 16th International Conference on Reliable Software technologies – Ada-Europe 2011. Volume 6652., Springer (June 2011)