Combined Scheduling of Time-Triggered Plans and Priority Scheduled Task Sets[*]

Jorge Real†, Sergio Sáez‡, Alfons Crespo†
† Instituto de Automática e Informática Industrial
‡ Instituto Tecnológico de Informática
Universitat Politècnica de València
Camino de vera, s/n, 46022 Valencia, Spain

E-mail: {jorge,ssaez,alfons}@disca.upv.es

Abstract

Preemptive, priority-based scheduling on the one hand, and time-triggered scheduling on the other, are the two major techniques in use for development of real-time and embedded software. Both have their advantages and drawbacks with respect to the other, and are commonly adopted in mutual exclusion.

In a previous paper, we proposed a software architecture that enables the combined and controlled execution of time-triggered plans and priority-scheduled tasks. The goal was to take advantage of the best of both approaches by providing deterministic, jitter-controlled execution of time-triggered tasks (e.g., control tasks), coexisting with a set of priority-scheduled tasks, with less demanding jitter requirements.

In this paper, we briefly describe the approach, in which the time-triggered plan is executed at the highest priority level, controlled by scheduling decisions taken only at particular points in time, signalled by recurrent timing events. The rest of priority levels are used by a set of concurrent tasks scheduled by static or dynamic priorities. We also discuss several open issues such as schedulability analysis, use of the approach in multiprocessor architectures, usability in mixed-criticality systems and needed changes to make this approach Ravenscar compliant.

1 Introduction

Despite the advantages of priority-based scheduling (PB) for real-time applications, a known issue is that such schedulers introduce irregular and difficult to analise delay patterns at run time, that translate into variable release *jitter*, i.e., the difference between the tasks' theoretical release time and their actual start of execution. Variable jitter degrades the performance of real-time applications such as digital control systems and synchronous communication protocols. Although several techniques have been proposed to reduce or mitigate the effects of variable jitter in these areas [2, 4, 5, 6], they involve changes to the task set design, or to the tasks' timing parameters, or they introduce additional run time overhead.

On the other hand, time-triggered scheduling (TT) uses an offline predefined plan, in which the designer identifies the exact points in time when every planned activity should start, and the times by which they should have finished. Since all scheduling decisions are planned in advance, a TT scheduler introduces very small runtime overhead, and therefore less –and easier to bound– jitter. The main drawback attributed to the TT approach is that plans are difficult to design, specially when they involve a large number of TT activities.

Our aim is to explore an alternative that combines both schemes for the same application. Activities imposing strict jitter requirements (such as digital regulators) are scheduled according to a TT plan, whereas the rest of tasks

[*]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).

are scheduled by the rules of a preemptive, priority-based scheduler. The whole set of tasks (both jitter-sensitive and jitter-tolerant tasks) will be resting on the same preemptive, priority scheduler, but the TT schedule will run at the highest priority to ensure minimal, controlled release jitter. The rest of tasks execute at lower priority levels, where they can be all scheduled under the same dispatching policy (preemptive, non-preemptive, EDF) or a combination of several dispatching policies, e.g. by using Ada's priority specific dispatching.

In this position paper, we refer to the system model described in [11], which we summarise in Section 2. Section 3 defines the actions taken by the TT scheduler, with Section 4 briefly describing the TT scheduler's API and several patterns for TT activities. In Section 5 we discuss extensions to the model and Section 6 concludes the paper.

2 System Model

In our system model, an offline, static TT plan coexists with a set of concurrent PB tasks. Since all TT activity is given the highest priority, the particular priority assignment and scheduling policy used for PB tasks will not affect the execution of the plan. The PB workload will find time to execute only when there is no TT activity running, and will be preempted by the arrival of TT events. In other words, PB tasks never preempt TT tasks. We therefore limit our description to the system model for the TT plan.

A TT plan is described by an ordered sequence of *time slots*. Each slot is characterised by two parameters:

***work* identifier** (*Work_Id*), an integer value ultimately referring to either a piece of user-provided application code or a predefined scheduler action;

slot duration, a time interval that must accommodate the execution of the work denoted by the slot's work identifier.

The whole plan sequence starts at a given time and each slot starts right after the end of its predecessor in the plan. The plan is repeated cyclically. 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 discuss overrun handling later on.

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 TT activity will be executed. This is useful for inserting gaps in the plan where they are needed. These gaps in the plan can be used by PB tasks, which will also use any spare time left by TT tasks taking less than their allocated slot duration to complete. Empty slots use the special *Work_Id* value zero.

- A ***mode-change slot*** defines a point in time where it is possible to substitute the current plan with a new one. 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 inclusion of this type of slot provides a flexible way to specify in which points of the plan a mode change can be enforced. Mode change slots are identified by $Work\_Id = -1$.

Each slot accommodates at most *one* TT activity, as opposed to the classic *cyclic executive* [1, 7]. This is because we want to have the highest possible control over release jitter of TT activities, which cannot be accomplished if several activities of varying execution times share the same slot. Another advantage is that, with only one activity per slot, the TT scheduler only needs to check one activity at a time for potential overrun (see next Section). This simplifies the implementation of the TT scheduler, which in turn helps keeping release jitter small.

3 The time-triggered scheduler

The TT scheduler ensures the timely execution of the plan. This implies not only releasing the activities at the right moments, 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 taking longer than their allocated slot duration. The TT scheduler must also give support to mode changes, as described previous section. Contrary to the case of PB schedulers, all decisions and actions of the TT scheduler are enforced at predefined points in time; in our case, at the start of each slot.

At the start of **any slot**, the scheduler will first detect whether there is still pending work from the previous slot, which is equivalent to detecting an overrun. There are several possible ways to handle this situation. One possibility is to lead the system to a fail-safe state (as in [9]), which can be achieved by means of a mode change. For the rest of this paper, we will take a *more tolerant* approach and allow the offending activity to continue executing at a lower priority, thereby limiting its interference to only tasks with an even lower priority, if any. The particular *demoted priority* can be specified for each activity. After this overrun check, the TT scheduler will proceed 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 makes sure it is assigned the TT level priority.

Empty slot: No other user or scheduler activity will be executed until the arrival of the next slot.

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 and the next slot to be processed is set to the first slot of the new plan. The actual implementation details of mode changes will depend on the concrete hardware platform. In some cases, the system has enough memory to allocate all the required TT tasks for all modes. On platforms with scarce resources, it could be necessary to delete old-mode tasks and load new-mode tasks to memory. The time needed for these operations can be absorbed by the mode-change slot duration, as well as the time needed for the scheduler to actually enforce the mode change and start scheduling the new mode.

The TT scheduler's activity always coincides with the start of a new slot. In our implementation we use a recurrent timing event to signal the start of each slot. The handler for this timing event encapsulates the scheduler's actions.

4 Application Program Interface and Task Patterns

An application program interface (API) for TT plans is described in [11]. What follows is a minimal summary of the most important features of that API, and a set of proposed task patterns using the TT scheduler. The API is provided by the Ada package Time_Triggered_Scheduling, which includes data types for representing the TT plan itself, as well as a TT scheduler supported by a protected type Time_Triggered_Scheduler. The scheduler offers:

- A protected procedure Set_Plan, to establish the plan to execute. This procedure comes in two versions to determine either an absolute or a relative starting time for the plan being set.
- A protected entry Wait_For_Activation, where TT tasks wait for their next slot, according to the current TT plan, as set by Set_Plan.
- A protected procedure Leave_TT_Level that allows a TT task to abandon the TT priority level and continue execution at a lower priority. This is useful for supporting the initial-optional-final pattern described below. The priority can be lowered to a predetermined value or a value passed as a parameter to Leave_TT_Level and the TT level priority is obtained again at the next slot assigned to the calling *Work_Id*.
- Three getter protected functions to obtain relevant timing information from the TT plan, which enable the implementation of richer task patterns, as described later. In particular:
  - Get_Last_Release returns the absolute release time of the last slot of a given *Work_Id*.

– Get_Last_Slot_Duration returns the duration of the last slot of a given *Work_Id*.
– Get_Next_Slot_Separation returns the separation between the start of the last slot and the next slot of a given *Work_Id*.

Based on this API, the simplest pattern for TT tasks is shown in Listing 1, which includes the specification and body of the task type Simple_Worker. Worker tasks are created by instantiation of this task type. 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_Id. The discriminant Prio specifies the demoted priority, i.e., the priority to which the task will be demoted in case of overrun. During normal operation, time-triggered tasks run at the highest priority. We have used the CPU aspect in our experiments to set the affinity of all time-triggered tasks to the same processor, although this is not in principle compulsory. Adapting this architecture to multiprocessor systems is the subject of ongoing work (see Section 5).

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.

```
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;    -- We limit ourselves to a single processor

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;
```

Listing 1. Simplest pattern for a time-triggered task

More complex task patterns are also possible, still relying essentially on the same simple scheduler described so far. In particular, we propose the following patterns, beyond the Simple_Worker pattern:

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 Simple_Worker pattern in Listing 1 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 it would cause overrun.

To obtain the time of this absolute delay, the pattern uses the TT scheduler's functions Get_Last_Release, that returns the last release time of a given Work_Id, and Get_Last_Slot_Duration that returns the duration of the last slot of a given work. Adding these two values gives the end time of the current slot. 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.

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

To avoid these issues during the optional phase, this pattern will first execute the initial part until completion. Then it will inform the TT scheduler that it wants to leave the TT priority level and be demoted to a given non-disturbing priority, to execute the optional phase at that priority. This functionality is supported by procedure Leave_TT_Level, that takes as parameters the *Work_Id* and the demoted priority, the latter defaulting to a value Demoted_Priority given in the work's definition. When the optional part completes, the task will wait again for activation until the arrival of the final part slot, where the work's task will execute the final part using the best result obtained during the optional part. If the optional part had not finished by the time when starting the final part is due, then the optional phase will be aborted (as in the *Worker_With_Cancellation* pattern). Listing 2 gives the pattern implementation.

Figure 1 shows the execution of a system composed by a TT plan with three works (numbered 1, 2 and 3) and two PB tasks (T4 and T5) scheduled under deadline monotonic. Work 1 follows the simple worker pattern, work 2 uses initial-final, and work 3 is initial-optional-final. 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 the demoted priority, between T4 and T5, in competition with the rest of tasks. When the optional part 3_O 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 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 run time. To avoid this overhead on the scheduler, we have added a field to the Slot_Type record, named Next_Slot_Separation. 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_Speration returns precisely this value for a given *Work_Id*.

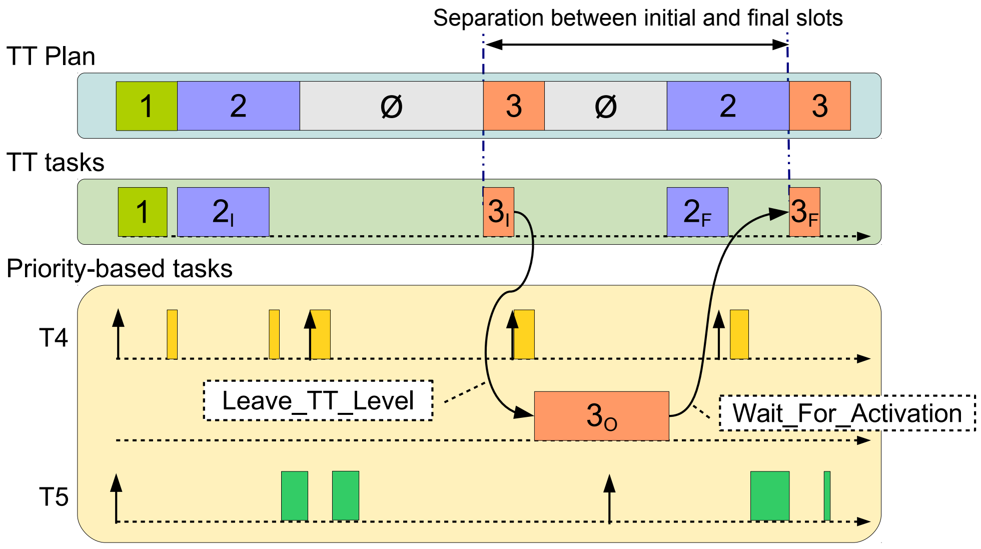

Figure 1. Execution of a *Worker_With_Initial_Optional_Final* pattern.

```
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 ;
```

Listing 2. Pattern for control tasks with initial, optional and final parts

5 Discussion

In the following subsections, we discuss various aspects related to the proposed architecture.

5.1 Schedulability analysis

A schedulability analysis is needed to assess the feasibility of the full task set, including both the TT plan and the PB 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. Although there exists abundant literature addressing schedulability analysis of priority-based systems under different assumptions, we need to incorporate the interference produced by the time-triggered level by characterising the plan workload in a way that it can be accounted for by a system-wide schedulability analysis.

One possibility is to consider the plan as a real-time *transaction*, as defined in the computational model of [8]. 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. As shown in Figure 2, adjacent time slots can be considered as a single task in the resulting transaction. This transaction has the highest priority and the interference introduced in lower priority levels can be computed as described in the above mentioned work [8].

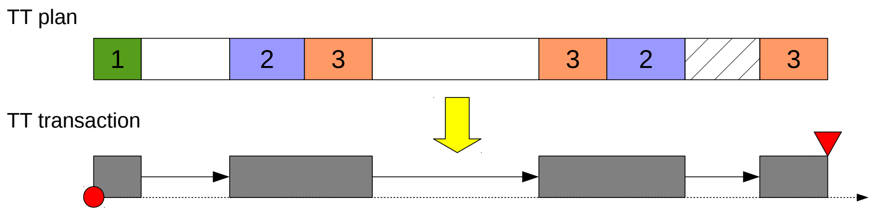

Figure 2. Transformation of a time-triggered plan into a transaction, for schedulability analysis.

5.2 Use in multiprocessor platforms

This combined approach of coexisting TT and PB tasks is also applicable on multiprocessor platforms, where the advantage of increased performance can be exploited. However, it makes sense to apply certain constraints in order to be able to assert timing guarantees. One is that plans have affinity: each plan runs on a different CPU. It makes little sense to apply global multiprocessor scheduling (jobs can migrate at any time of their execution), but we don't need to go to the other extreme with full partitioning (nothing migrates). Controlled forms of migration could be applied for cases where a single processor is insufficient to guarantee compliance with jitter restrictions. Figure 3 illustrates this idea: there are two plans, running on two different processors. The TT task with *Work_Id* 1 has an initial and a final part (it follows the *Worker_With_Initial_Final* pattern described above). Given the need to schedule works 2 and 3 on processor 0 when work 1_F is due to be released, this final part of work 1 can migrate to a different processor (processor 1 in this case) to find the CPU time that processor 0 cannot grant. This flexibility however is not attainable without restrictions in the design of all plans: they need to be executed synchronously in all CPUs (use a common clock and epoch), and all plans must have durations that are exact multiples of each other.

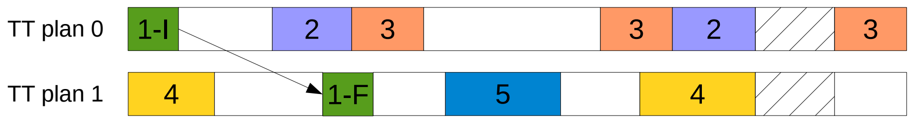

Figure 3. The final part of work 1 is executed on a different processor than its initial part.

In order to enforce this association between plans and processors, we need control over tasks' affinities. We think the set of currently available facilities in Ada (via dispatching domains) is sufficiently rich. But for complete control, affinity of timing events is also desirable. This feature is currently missing in Ada, although its usefulness and implementability are the subject of ongoing work [13].

5.3 Ravenscar implementation

Our proposal assumes that there is an underlying priority scheduler in charge of ultimately scheduling all the workload, both TT and PB tasks. Although the TT workload is given the highest priority, and its scheduling points are handled by (ideally fast and efficient) timing events, the scheduler will necessarily incur an extra overhead with respect to the extremely simple scheduler of a pure TT scheduler. If the TT scheduler and task patterns were implemented in Ravenscar, then the TT level would benefit from relying on an efficient and certifiable Ravenscar run-time system.

Certification is a major concern in high-integrity, safety critical systems. TT scheduling finds a natural niche in certified software because it is more amenable to certification than general priority scheduling. However, Ravenscar also leads to certifiable implementations. So what would be the use of combining TT plans with Ravenscar tasks, when one could implement everything in Ravenscar and eventually obtain a certifiable system? In our view, the potential advantages are twofold: one is, the control *and reduction* of release jitter of TT tasks; the other is the possibility for direct reuse of previously designed TT, including cyclic executives, which can be easily transformed into a plan that is compliant with the model presented in Section 2.

Several aspects should be changed in our current implementation [11] to make it Ravenscar compliant. One refers to the overrun control policy, since tasks cannot be demoted in Ravenscar. Task demotion should therefore be reconsidered as a viable overrun handling policy. An alternative overrun handling policy could be a mode change to enter a fail-safe mode, as in [9]. Or to let overrunning tasks to catch up by using successive slots in the plan. Use of asynchronous transfer of control is also forbidden in Ravenscar, hence patterns that involve abortion (such as the *Worker_With_Initial_Optional_Final* pattern) would need to be reconsidered as well. Other aspects of the interface should also be adapted to Ravenscar, such as eliminating an entry family we are currently using in the implementation of the TT scheduler. The release mechanism for TT tasks could instead be based on suspension objects, rather than a protected entry family.

5.4 Use in mixed criticality systems

In mixed criticality systems, real-time tasks with different criticality levels share the same execution platform. In such systems the run-time support has to guarantee a certain degree of time isolation between tasks with different criticality levels. At any given moment the system is in a *system criticality level* that specifies which tasks are active and which are not. When a task executes for longer than its allocated execution time at the current system criticality level, a change of criticality level is due. Depending on the mixed criticality model, each task in the system has a given amount of execution time allocated for each possible criticality level [3]. It is common that high criticality tasks are given increasing execution slots in higher criticality levels. These different execution times are normally due to the use of more conservative WCET estimation methods for higher criticality levels, or due to probabilistic estimation of execution times, with higher criticality levels granting higher confidence on the estimation. On the contrary, lower criticality tasks normally see their allocated execution time reduced as the criticality level increases, which forces low criticality tasks to use simpler and simpler algorithms. In the extreme case, low criticality tasks will never execute when the system criticality level is high enough.

On priority-scheduled systems, each criticality model can also use several approaches to determine the priority of each task at a given system criticality level. However, under TT scheduling, each task executes during its allocated time slots independently of its criticality level. Even so, different plans can be built for each system criticality level, indicating which tasks have to be executed in a given criticality level and for how long they have to execute.

When a TT task overruns its time slot, the scheduler automatically demotes the offending task. This is in order to ensure time isolation among TT tasks and also to keep the interference on PB tasks within the assumptions of the schedulability analysis. In a mixed criticality context, an overrun can trigger a change of the system criticality level. Depending on the mixed criticality model, this could be supported by a plan replacement. Since the current plan will only be changed at the next *mode change slot*, there will be a transient period where the system criticality level has changed, but the active plan is still the previous one, corresponding to the previous criticality level. To minimise the latency of the transition between plans, the scheduler can perform some on-line optimizations, such as e.g. to shrink (or ignore) the time slots of the lower criticality tasks that require shorter (or null) execution times in the incoming plan; and/or to enlarge the time slots belonging to high criticality tasks that require more execution time in the new plan, so long as the current plan allows it.

6 Conclusions

Combining time-triggered (TT) and priority-based (PB) scheduling is a promising approach for applications including jitter-sensitive activities and more jitter-tolerant tasks. A time-triggered schedule can be designed for only the most demanding tasks (in terms of release jitter) whereas the rest of tasks in the system can be scheduled by priorities. The experimental results presented in [11] show that release jitter can be confined to low values for TT tasks, compared with the much higher variability of PB tasks.

In this paper we have referenced the approach presented in [11] and discussed several potential areas for extensions of the basic model, including schedulability analysis of the combined TT-PB system; considerations on the use of this approach in multiprocessor platforms; changes needed to adapt our current Ada implementation to Ravenscar; and the integration of these ideas in the context of mixed criticality systems.

There are other aspects of the proposal, some of which are the subject of current and future work. They include the development of tools to aid in building plans (with automatic code generation as a useful add-on) and to integrate this architecture in the framework proposed in [10, 12, 14].

References

[1] T. P. Baker and A. Shaw. The cyclic executive model and Ada. In *Proceedings IEEE Real Time Systems Symposium 1988, Huntsville, Alabama*, pages 120–129, 1988.

[2] P. Balbastre, I. Ripoll, J. Vidal, and A. Crespo. A Task Model to Reduce Control Delays. *Real-Time Systems*, 27(3):215–236, September 2004.

[3] A. Burns and R. Davis. Mixed Criticality Systems - A Review. Technical report, Depatment of Computer Science, University of York, 2013.

[4] A. Cervin. *Integrated Control and Real-Time Scheduling.* PhD thesis, Lund Institute of Technology, April 2003.

[5] R. Dobrin. *Combining Off-line Schedule Construction and Fixed Priority Scheduling in Real-Time Computer Systems*. PhD thesis, Mälardalen University, 2005.

[6] S. Hong, X. Hu, and M. Lemmon. Reducing Delay Jitter of Real-Time Control Tasks through Adaptive Deadline Adjustments. In IEEE Computer Society, editor, *22nd Euromicro Conference on Real-Time Systems – ECRTS*, pages 229–238, 2010.

[7] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall Inc., 2000.

[8] J. Palencia and M. González-Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *9th IEEE Real-Time Systems Symposium*, 1998.

[9] M. J. Pont. *The Engineering of Reliable Embedded Systems: LPC1769 edition*. Number ISBN: 978-0-9930355-0-0. SafeTTy Systems Limited, 2014.

[10] J. Real and A. Crespo. Incorporating Operating Modes to an Ada Real-Time Framework. *Ada Letters*, 30(1):73–85, April 2010.

[11] J. Real, S. Sáez, and A. Crespo. Combining time-triggered plans with priority scheduled task sets. In M. Bertogna and L. M. Pinho, editors, *Reliable Software Technologies – Ada-Europe 2016*, volume 9695 of *Lecture Notes in Computer Science*. Springer, June 2016.

[12] S. Sáez, J. Real, and A. Crespo. An integrated framework for multiprocessor, multimoded real-time applications. In M. Brorsson and L. Pinho, editors, *Reliable Software Technologies – Ada-Europe 2012*, volume 7308, pages 18–34. Springer-Verlag, June 2012.

[13] S. Sáez, J. Real, and A. Crespo. Implementation of Timing-Event Affinities in Ada/Linux. *Ada Letters*, 35(1), April 2015.

[14] A. J. Wellings and A. Burns. A Framework for Real-Time Utilities for Ada 2005. *Ada Letters*, XXVII(2), August 2007.