American Journal of Embedded Systems and Applications
2014; 2(4): 38-50
Published online September 30, 2014 (http://www.sciencepublishinggroup.com/j/ajesa)
doi: 10.11648/j.ajesa.20140204.12

Implementing adaptive time-triggered co-operative scheduling framework for highly-predictable embedded systems

Mouaaz Nahas[1, *], Ricardo Bautista-Quintero[2]

[1]Department of Electrical Engineering, College of Engineering and Islamic Architecture, Umm Al-Qura University, Makkah, KSA
[2]Department of Mechanical Engineering, Instituto Tecnologico De Culiacan, Sinaloa, Mexico

Email address:
mmnahas@uqu.edu.sa (M. Nahas), ricardo.bquintero@gmail.com (R. Bautista-Quintero)

To cite this article:
Mouaaz Nahas, Ricardo Bautista-Quintero. Implementing Adaptive Time-Triggered Co-Operative Scheduling Framework for Highly-Predictable Embedded Systems. *American Journal of Embedded Systems and Applications.* Vol. 2, No. 4, 2014, pp. 38-50.
doi: 10.11648/j.ajesa.20140204.12

Abstract: For many real-time embedded systems, Time-Triggered Co-operative (TTC) scheduling algorithms provide simple and reliable solution at low cost. Previous work in this area has focused on the development of a wide range of TTC implementations for various purposes (e.g. for achieving low-jitter characteristics, reducing CPU power consumption or dealing with task-overruns). Despite the great deal of work in this area, it can be said that each previous scheduler implementation was created to address only one particular problem in TTC algorithm. For applications which require extremely high degree of reliability, a combinational TTC architecture – that incorporates multiple features – can be an appropriate solution. This paper describes the implementation of an adaptive, highly-predictable TTC scheduler that addresses both jitter and task-overrun problems simultaneously. Furthermore, the presented scheduler incorporates an online technique for measuring the practical "worst-case execution time" for each task during system runtime. The behavior of the proposed scheduler is compared with a set of previously developed schedulers in terms of timing jitter, task-overrun handling capability and resource requirements for practical real-time implementations.

Keywords: Time-Triggered, Co-Operative, Cyclic Executive, Jitter, Worst Case Execution Time, Multiple Timer Interrupts, Task-Overrun, Task Guardian, Adaptive Scheduler

1. Introduction

Embedded systems are often implemented as a collection of communicating tasks [1]. For example, if the tasks are invoked as a response to aperiodic events, the system architecture is described as "event-triggered" [2],[3]. Alternatively, if the tasks are invoked periodically under the control of timer, the system architecture is described as "time-triggered" [4],[3]. Moreover, if the tasks – once invoked – can pre-empt (interrupt) other tasks, then the system is described as "pre-emptive": if, instead, tasks cannot be interrupted, the system is described as "non pre-emptive" or "co-operative".

Cyclic executive [5],[6] is a form of co-operative scheduler that has a time-triggered architecture. Such Time-Triggered Co-operative (TTC) schedulers can be a good match for a broad range of embedded applications, even those which have hard real-time requirements [5]-[11] . Since all tasks in TTC scheduler run regularly according to their predefined order, such schedulers demonstrate very low levels of task jitter (see for instance [6],[13] and [14]). Moreover, they can maintain their low-jitter characteristics even when complex techniques, such as Dynamic Voltage Scaling (DVS), are employed to reduce system power consumption [15].

Nonetheless, implementing the software code of TTC algorithms, with less care, can result in demonstrating high levels of task jitter [16],[17]. The presence of jitter can have a detrimental impact on the performance of many embedded applications. For example, [18] show that – during data acquisition tasks – jitter rates of 10% or more can introduce errors which are so significant that any subsequent interpretation of the sampled signal may be rendered meaningless. Similarly [19] discusses the serious impact of jitter on applications such as spectrum analysis and filtering. In embedded control systems, jitter can greatly degrade the performance by varying the sampling period [20],[21].

Impact of sampling jitter on real-time adaptive embedded control system is explored in detail in [22].

In addition to jitter problem, a pure TTC architecture has a failure mode which has the potential to impair the system performance: this mode relates to task overruns (see [23] and [24] for more details). Briefly, task-overrun describes a situation when one or more tasks exceed their pre-determined "worst-case execution time" (WCET)[1]. In the most severe circumstances, overrun could mean that a high-priority task attempts to execute "forever", denying lower-priority tasks access to the CPU. Buttazzo [14] has noted that: *"[Co-operative] scheduling is fragile during overload situations, since a task exceeding its predicted execution time could generate (if not aborted) a domino effect on the subsequent tasks"*.

As many researchers have observed, determining the WCET of tasks is rarely straightforward [25]-[34]. Therefore, when implementing a TTC scheduler the user needs to appreciate this potential risk and understand precisely how the scheduler will behave in the presence of task overrun. It should be noted that lack of knowledge about WCET is a problem which faces the developers of many embedded systems (not just those based on TTC). For example, as Gergeleit and Nett [30] have noted: *"Nearly all known real-time scheduling approaches rely on the knowledge of WCETs for all tasks of the system"*. For further details about WCET analysis, see [35].

One simple solution to this problem is to err on the side of caution when employing WCET estimates, thereby reducing the chances of an overrun occurrence. Typical "safety margins", used in this approach, are around 20% [36]. Such technique is simple and can be effective, but inevitably adds to costs. An alternative is to be slightly more conservative when estimating WCET values (e.g. add 5% to accurate estimates) and then extend the scheduler (or add additional hardware) in such a way that (at runtime) any overrunning tasks can be shut down, and/or the schedule can be adjusted [37]. Such an approach also allows dealing with error-related overruns (for example, tasks which overrun because of a hardware-related error). In these circumstances, the problem can be addressed (at least in part) by employing some form of "watchdog timer" (e.g. [38]) in a "scheduler watchdog" design (e.g. [39]). Alternatively, greater control over the system behavior can be obtained by using a "task guardian" [7],[23],[40]. The concept of allowance for handling task overrun is introduced in [41],[42].

The present study is concerned with implementing highly-predictable embedded system. Predictability is one of the most important objectives of real-time embedded systems [43]-[45],[14]. It simply reflects the ability to determine, in advance, exactly what the system will do at every moment of time in which it is running and hence determines whether the system is capable of meeting all its timing constraints. One way in which predictable behavior manifests itself is in low levels of task jitter and the ability to deal with any task-overrun.

The particular focus of this paper is on addressing the problem of task jitter and task overruns in attempting to increase the overall system predictability in TTC schedulers. Previous work in this area has led to the development of low-jitter TTC schedulers. For example, [17] introduced the TTC-SD and TTC-MTI schedulers which both had the potential to reduce the amount of task jitter significantly at the cost of little increases in memory overheads. Moreover, [23] and [46] developed the TTC-TG scheduler implementation which employs a wide range of task guardian mechanisms to address the task-overrun problem, thereby improving the real-time performance of TTC system.

This paper describes and evaluates a flexible (adaptive) TTC architecture that provides extremely predictable task scheduling. This is aimed towards implementing a "perfect" TTC scheduler which satisfies all requirements for which it was initially intended (for example, high reliability, predictability, efficiency and determinism). The new implementation is called "TTC-Adaptive" scheduler which has mainly been developed from the concepts employed in the TTC-MTI and TTC-TG schedulers described elsewhere (see [17] and [46], respectively). The developed scheduler is simply based on a runtime measurement of the tasks' WCETs before applying jitter-reduction and task guardian techniques which would depend on the results obtained by the WCET measurements.

The remainder of this paper is laid out as follows: Section 2 describes the concept and impact of task-overrun in TTC system. Section 3 introduces the TTC-Adaptive scheduler and describes its implementation in low-cost embedded system. Section 4 provides the experimental methodology used to assess the behavior of the TTC-Adaptive scheduler and the results obtained from this scheduler. The results are presented in the form of comparison between the TTC-Adaptive scheduler and three previously developed TTC schedulers against a set of criteria including: task jitter, ability to deal with task-overrun and implementation costs. The overall paper conclusion is finally presented in Section 5.

2. Task-Overrun in TTC System

In systems employing TTC architectures, task timing should be highly predictable. To guarantee such levels of predictability, the WCETs of all tasks need to be known in advance. If, during runtime, a certain task executes in time longer than its known WCET, the task is said to have overrun.

Fig. 1 (a) shows the schedule of two tasks A and B. Task A has a higher-priority with 1 ms period and 0.5 ms WCET. Task B has a lower-priority with the period equals to 5 ms. In Fig. 1 (b), Task A has executed in 5.5 ms (it has overrun). In this case, Task B is delayed by 5 ms. In practice, the situation may be much worse if Task A never completes, as this would cause Task B to be entirely missed throughout system runtime.

[1] WCET is the longest time taken by the CPU to execute a task without pre-emption [14].

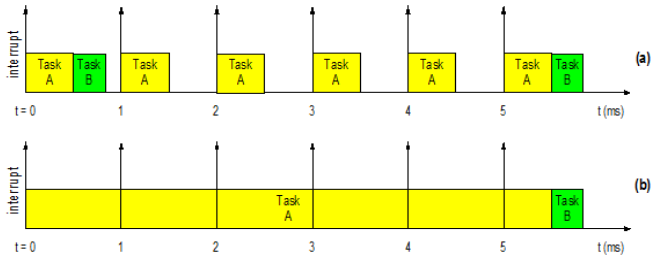

Figure 1. *The impact of task-overrun on a TTC scheduler.*

Consider the original TTC scheduler [8] with function call tree shown in Figure 2. If a task overruns, then – instead of 'Sleep' (i.e. idle mode) being interrupted by the Interrupt Service Routine (ISR) (e.g. Update function) – the overrunning task is interrupted and all other co-operative tasks are blocked as control from the interrupt is passed straight back to the overrunning task: this process is illustrated in Figure 3.

Figure 2. *Function call tree for the original TTC scheduler (normal operation).*

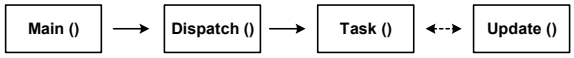

Figure 3. *Function call tree for the original TTC scheduler (with task-overrun).*

Note that the most basic TTC scheduler (as in [8]) assumes that – if a task overruns – all subsequent tasks will be delayed. This is fine, in theory, but it seems unlikely that any practical TTC implementation can ever achieve this. For example, if a task overruns for a week – or a year – then, in theory, the TTC scheduler should keep track of all "missing" tasks and execute them "immediately" when the overrunning task completes. Providing full support for such a mechanism requires a large memory capacity (potentially an infinite memory capacity).

3. TTC-Adaptive Scheduler

This section describes the design and implementation of the TTC-Adaptive scheduler. The developed scheduler framework is intended to provide (generic) software-based mechanisms for dealing with overruns in the co-operative tasks, while maintaining very low-levels of jitter at task release times. The framework also incorporates a simple, but effective, mechanism for calculating practical WCETs of the co-operative tasks during runtime. Our work, as noted previously, is based on the use of task guardians. Note that estimating the tasks' WCETs is performed in this study by employing a runtime measurement method.

The main software architecture of the proposed scheduler is taken from the TTC-MTI scheduler (for operating mode) whereas new task guardian mechanism is implemented here. Also note that the WCET value computed by this scheduler represents the longest possible execution time of the task which is obtained during the measurement period (this does not necessarily represent the actual WCET which is accepted by many researches as a non-straightforward process to calculate).

The reason for describing the proposed scheduler as "adaptive" is that – unlike all previous TTC schedulers – it is self-adapted to changes in task execution times during system runtime.

3.1. Overview

The architecture of this TTC scheduler was based on that used in our previously developed TTC-MTI scheduler [17]. In particular, with this implementation, two interrupts are used: "Tick interrupt" and "Task interrupt". The Tick interrupt is used to generate the scheduler periodic tick while the Task interrupt is used to trigger the execution of tasks within the tick interval. The function call tree of the MTI scheduler is shown in Fig. Figure **4**. This helps to control jitter levels at all task release times. Moreover, the present scheduler employs a simple mechanism for calculating the WCET of each task during the system operation. In all previously developed TTC scheduler implementations, WCET information is input to the system by the user (for more details, see [17]).

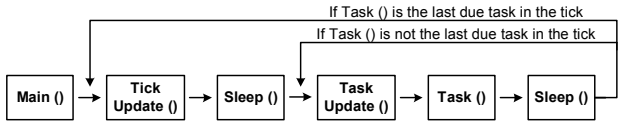

Figure 4. *Function call tree for the TTC-MTI scheduler (in normal conditions).*

Overall, there are two different modes in which the system can operate: Calculating Mode (CM) and Operating mode (OM). Each of these modes is described in the following section.

3.2. Scheduler Modes

The proposed scheduler framework consists of two basic modes as follows:

3.2.1. Calculating Mode (CM)

The system runs the calculating mode for a short period of time, allowing the scheduler to perform an online calculation of the WCET for each co-operative task, and the required release time at which the task must start its execution. That is, once the system starts (power is up), the scheduler takes short time to measure the WCETs and release times of all tasks before switching into a normal operating mode. The calculating time period must be defined by the user in "number of ticks", based on system specifications. Note that during this period, the system tasks execute normally but task jitter might be at very high levels. This is fine and should not jeopardize the whole system operation assuming that the calculating mode takes a short but enough time to execute.

The scheduler structure, described in Section 10, is used here but with some modification (Fig. Figure **5**).

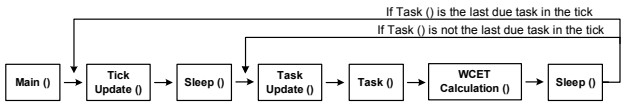

Figure 5. *Function call tree for the TTC-Adaptive scheduler (calculating mode).*

In this process, after the task is executed, SCH_WCET() function is called to calculate the WCET of the completed task and its release time required for low-jitter characteristics. The WCET of a task is measured by recording the time just before and after the task execution (using, for example, the Timer Control Register "TCR": see [47]). The WCET is then calculated, in the "SCH_WCET()" function, by subtracting the stop time from the start time. In the same way, release time of a task is measured by recoding the time just after the Task Update() function begins to execute. The SCH_WCET() stores the maximum WCET and the maximum release time for each task in the task array. Note that the release time of the first task in the system is based on the worst case duration of the Tick Update() function. After calculating the WCET of the current task, the processor is placed in the idle (Sleep) mode for a very short period before the next Task interrupt occurs (see void SCH_WCET(void)

```
{
   tLong Duration;

   // Record Stop time
   Stop_Time = T1TC;

   // Calculate duration for no overrun
   Duration = Stop_Time - Start_Time;

   // Calculate duration of Task Update
   Task_Update_Duration  = Start_Time - Release_Time;

      // If index is larger than 0
      if (Index_G)
      {
         // If the measured WCET is larger than recorded
         if (SCH_tasks_G[runme[Index_G - 1]].WCET < Duration)
     {
            // Modifiy the recorded WCET
            SCH_tasks_G[runme[Index_G - 1]].WCET = Duration+1;
         }

         // If release time is less than the tasks start time
         if (SCH_tasks_G[runme[Index_G - 1]].Req_Rls_Tm <
         (Release_Time))
         {
            // Modify the release time
            SCH_tasks_G[runme[Index_G - 1]].Req_Rls_Tm =
            Release_Time+2;
         }

         // set the match register to current time plus little margin: this is
     // because we want the Task_Update to be called immediately
         if (runme[Index_G] != SCH_MAX_TASKS)
         {
            // Set the timer to interrupt almost immediately so we can run next
task
            // Set timer match register to current time + 4
             T1MR1 = T1TC + 4;
         }
      }

   // Disable any interrupt and send the scheduler to sleep
   SCH_End_Task();
}
```

Listing 1).

Please recall that the WCET value computed in this algorithm is basically the longest possible execution time of the task obtained during the measurement period. As many researchers have observed, determining the accurate WCET of a particular activity is often a very complicated process (see [46] for more details).

```
void SCH_WCET(void)
{
   tLong Duration;

   // Record Stop time
   Stop_Time = T1TC;

   // Calculate duration for no overrun
   Duration = Stop_Time - Start_Time;

   // Calculate duration of Task Update
   Task_Update_Duration  = Start_Time - Release_Time;

      // If index is larger than 0
      if (Index_G)
      {
         // If the measured WCET is larger than recorded
         if (SCH_tasks_G[runme[Index_G - 1]].WCET < Duration)
     {
            // Modifiy the recorded WCET
            SCH_tasks_G[runme[Index_G - 1]].WCET = Duration+1;
         }

         // If release time is less than the tasks start time
         if (SCH_tasks_G[runme[Index_G - 1]].Req_Rls_Tm <
         (Release_Time))
         {
            // Modify the release time
            SCH_tasks_G[runme[Index_G - 1]].Req_Rls_Tm =
            Release_Time+2;
         }

         // set the match register to current time plus little margin: this is
     // because we want the Task_Update to be called immediately
         if (runme[Index_G] != SCH_MAX_TASKS)
         {
            // Set the timer to interrupt almost immediately so we can run next
task
            // Set timer match register to current time + 4
             T1MR1 = T1TC + 4;
         }
      }

   // Disable any interrupt and send the scheduler to sleep
   SCH_End_Task();
}
```

Listing 1. *WCET-calculation function in the TTC-Adaptive scheduler.*

3.2.2. Operating Mode (OM)

This relates to the normal operation mode of the scheduler. It is assumed here that the user has set the duration of the calculating mode long enough to obtain a correct set of WCET values: this must be estimated by the user based on prior knowledge about the system specifications. Once the calculation time completes, the system is switched into the operating mode during which scheduled tasks run in their allotted time "slots" with no release jitter.

The function call tree for the operating mode is identical to those illustrated in Figure 4. Note that, without any addition to the design, the system is expected to behave in the same way as the TTC-MTI scheduler [17]. This means that a very simple task guardian mechanism is employed in which the scheduler allows an overrunning task to run until the next task (or tick) interrupt. This solution will be called here 'Option 1'. Here, the last task in that tick has a chance to overrun for the rest of tick interval (which is relatively large as compared to task slots) causing the CPU to consume large amount of unnecessary power.

Therefore, a more effective task guardian solution is still required. One suggested way is to employ a mechanism which detects the overrun once occurred and shutdown the overrunning task immediately whether or not there are scheduled tasks to run afterwards in the same tick interval. This solution will be called 'Option 2'. In this solution, the scheduler employs three interrupts: "Tick" interrupt and "Task" interrupt (as before) and a third interrupt called "Task Overrun" interrupt. The ISR functions for the Tick and Task interrupts (i.e. Tick Update() and Task Update(), respectively) are very similar to those used in the TTC-MTI scheduler. However, the Tick Update() function here keeps track of the number of ticks for the calculating mode. Once the calculation time (defined by the user) is over, the scheduler switches into operating mode.

In addition to setting the match register of the task timer to be equal to the release time of the next due task, the Task Update() function also sets the match register of the "task-overrun" timer to be equal to the task release time plus the task WCET plus the duration of the task update function. This simply implies that if a task exceeds its measured WCET, it will be interrupted immediately by a Task_Overrun_Update() function which is linked to the "Task Overrun" timer interrupt. This function reports the overrun and sends the scheduler to 'Sleep'. If everything goes well and no overrun occurs, an End_Task() function is called after the completion of each task which will simply disable the task-overrun timer interrupt and send the scheduler to 'Sleep'. Note that the Tick Update() function sets the return address after each task to be for the End_Task() function.

Figure 6 and Figure 7 illustrate the sequence of functions in 'Option 2' implementation with and without overrun.

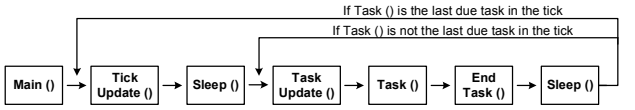

Figure 6. *Function call tree for the TTC-Adaptive scheduler 'Option 2' (normal operation).*

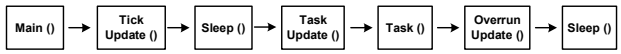

Figure 7. *Function call tree for the TTC-Adaptive scheduler 'Option 2' (with task-overrun).*

In order to provide a complete task guardian mechanism, a third solution which includes support for backup tasks has been proposed: this is called 'Option 3'. In this solution, once an overrun is detected, the function referred to as Task_Overrun_Update() will report the overrun, set "backup" task to be the next due task to run and then send the scheduler to 'Sleep'. In the next Tick interrupt, the scheduler executes the backup task before continuing to execute the following tasks (if any). Note that the tasks that have already been executed in the tick interval – in which the overrun took place – will not be re-executed in the following tick. Overall, with this approach, the scheduler imposes a one-tick delay for the whole scheduler. This can still maintain a high determinism assuming that overruns occur very occasionally. The sequence of functions in 'Option 3' implementation is illustrated in Figure 8.

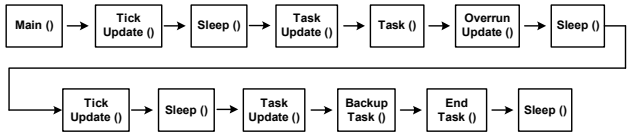

Figure 8. *Function call tree for the TTC-Adaptive scheduler 'Option 3' (with task-overrun).*

The code for the TTC-Adaptive scheduler is shown in

```
void SCH_Tick_Update(void)
{
tByte i = 0;
tByte Index;
static tWord Tick_Count = 0;

// If tick is not paused (no overruns)
if (!PauseTick)
   {
   // Go through the task array
   for (Index = 0; Index < SCH_MAX_TASKS - 1; Index++)
      {
            // Check if there is a task at this location
      if (SCH_tasks_G[Index].pTask)
         {
         if (--SCH_tasks_G[Index].Delay == 0)
            {
               // indicate the task is to be run
               runme[i++] = Index;

               if (SCH_tasks_G[Index].Period != 0)
                  {
                  // Schedule period tasks to run again
```

```
                        SCH_tasks_G[Index].Delay =
SCH_tasks_G[Index].Period;
                        }
                      else
                        {
                        // Delete one-shot tasks
                        SCH_tasks_G[Index].pTask   = 0;
                        }
                      }
                    }
        }

                // Indicate no more tasks in runme queue
                runme[i] = SCH_MAX_TASKS;

                /* If there are tasks in current tick interval */
                if (runme[0] != SCH_MAX_TASKS)
                        {
                        // If task is 0
                        if (runme[0] == 0)
                          {
                           // If release time is less than current time + 3
                            if (SCH_tasks_G[0].Req_Rls_Tm <
                            (Tick_Update_Duration))
                            {
                            // Modify release time to be current + 3
                            SCH_tasks_G[0].Req_Rls_Tm =
                            Tick_Update_Duration+3;
                            }
                          }

                        // Setup Match Register 1 - interrupt in uS from tick
                        T1MR1 = SCH_tasks_G[runme[0]].Req_Rls_Tm;

                         // Interrupt on match 1
                        T1MCR |= 0x08;
                        }
            // Reset the task index
        Index_G = 0;
        }

    // If tick is paused, set release time to backup task so that the backup task runs
    // first and then the next tasks in the schedule can carry on as normal
          else
            {
            // Setup Match Register 1 - interrupt in uS from tick
            T1MR1 = SCH_tasks_G[runme[Index_G]].Req_Rls_Tm;

            // Interrupt on match 1
            T1MCR |= 0x08;

            // Enable tick to run next time
            PauseTick=0;
            }

    // Return to sleep
    cTask = SCH_Go_To_Sleep;

  // Keep track of the number of ticks for the calculating mode.
  // Once the calculation time (defined by the user) completes, the scheduler goes
to
  // operating (normal) mode.
    if (Mode_G == CALCULATING_MODE)
      {
         // If ticks is larger than calculation time
         if (Tick_Count++ > CALCULATION_TIME)
           {
           // Change mode to operating mode
            Mode_G = OPERATING_MODE;
            }
          }

     // If the scheduler goes into the operating mode
        if (Mode_G == OPERATING_MODE)
        {
            // Run End_Task after evry task
        mTask = SCH_End_Task;
        }

    // Record the duation of the Tick Update
    Tick_Update_Duration = T1TC;
    }
```

Listing 2 to

```
 void SCH_Task_Overrun_Update(void)
{
        // Goto sleep after ISR
        cTask = SCH_Go_To_Sleep;

        // Increment task overrun flag
        SCH_tasks_G[Index_G-1].Overrun++;

        // If there exists a backup task
        if (SCH_tasks_G[Index_G-1].bTask)

        {
           // Disable task interrupt on match 1
           T1MCR &= 0xFFFFFFF7;

           // Set backup task to run
           SCH_tasks_G[Index_G-1].pTask = SCH_tasks_G[Index_G-1].bTask;

           // Point index back to overruning task
           Index_G--;

           // Pause the next tick
           PauseTick = 1;
        }
}
```

Listing 5.

```
void SCH_Tick_Update(void)
    {
    tByte i = 0;
    tByte Index;
    static tWord Tick_Count = 0;

    // If tick is not paused (no overruns)
    if (!PauseTick)
        {
        // Go through the task array
        for (Index = 0; Index < SCH_MAX_TASKS - 1; Index++)
            {
                  // Check if there is a task at this location
           if (SCH_tasks_G[Index].pTask)
               {
               if (--SCH_tasks_G[Index].Delay == 0)
                   {
                        // indicate the task is to be run
                        runme[i++] = Index;

                        if (SCH_tasks_G[Index].Period != 0)
                          {
                         // Schedule period tasks to run again
                           SCH_tasks_G[Index].Delay =
SCH_tasks_G[Index].Period;
                          }
```

```
                else
                    {
                    // Delete one-shot tasks
                    SCH_tasks_G[Index].pTask   = 0;
                    }
                }
            }
        }

            // Indicate no more tasks in runme queue
            runme[i] = SCH_MAX_TASKS;

            /* If there are tasks in current tick interval */
            if (runme[0] != SCH_MAX_TASKS)
                {
                // If task is 0
                if (runme[0] == 0)
                  {
                   // If release time is less than current time + 3
                    if (SCH_tasks_G[0].Req_Rls_Tm <
                    (Tick_Update_Duration))
                    {
                    // Modify release time to be current + 3
                    SCH_tasks_G[0].Req_Rls_Tm =
                    Tick_Update_Duration+3;
                    }
                  }

                // Setup Match Register 1 - interrupt in uS from tick
                T1MR1 = SCH_tasks_G[runme[0]].Req_Rls_Tm;

                 // Interrupt on match 1
                T1MCR |= 0x08;
                }
          // Reset the task index
       Index_G = 0;
       }

    // If tick is paused, set release time to backup task so that the backup task runs
    // first and then the next tasks in the schedule can carry on as normal
         else
           {
           // Setup Match Register 1 - interrupt in uS from tick
           T1MR1 = SCH_tasks_G[runme[Index_G]].Req_Rls_Tm;

           // Interrupt on match 1
           T1MCR |= 0x08;

           // Enable tick to run next time
           PauseTick=0;
           }

    // Return to sleep
    cTask = SCH_Go_To_Sleep;

  // Keep track of the number of ticks for the calculating mode.
  // Once the calculation time (defined by the user) completes, the scheduler goes
to
  // operating (normal) mode.
    if (Mode_G == CALCULATING_MODE)
      {
         // If ticks is larger than calculation time
         if (Tick_Count++ > CALCULATION_TIME)
           {
           // Change mode to operating mode
           Mode_G = OPERATING_MODE;
           }
         }

      // If the scheduler goes into the operating mode
       if (Mode_G == OPERATING_MODE)
       {
           // Run End_Task after evry task
       mTask = SCH_End_Task;
       }

     // Record the duation of the Tick Update
     Tick_Update_Duration = T1TC;
     }
```

Listing 2. *"Update" ISR of the Tick-Timer-Interrupt in the TTC-Adaptive scheduler.*

```
void SCH_Task_Update(void)
{
   Release_Time = T1TC;

   // Run task after this function
   cTask = SCH_tasks_G[runme[Index_G]].pTask;

   // Setup Match Register 1 - for the next task
   T1MR1 = SCH_tasks_G[runme[Index_G+1]].Req_Rls_Tm;

   // Setup Match Register 2 - for WCET for end task
   T1MR2 = SCH_tasks_G[runme[Index_G]].Req_Rls_Tm +
   SCH_tasks_G[runme[Index_G]].WCET + Task_Update_Duration + 4;

   // Increment task index
   Index_G++;

   // Disable Interrupt on match 1
   T1MCR &= 0xFFFFFFF7;

   // Enable Interrupt on match 1
   T1MCR |= (1 & (tLong) (runme[Index_G] != SCH_MAX_TASKS)) << 3;

   // Disable Interrupt on match 2
   T1MCR &= 0xFFFFFFBF;

   // Enable WCET end_task interrupt for current task
   T1MCR |= (1 & (tLong) (Mode_G == OPERATING_MODE)) << 6;

   // Record start time
   Start_Time = T1TC;
}
```

Listing 3. *"Update" ISR of the Task-Timer-Interrupt in the TTC-Adaptive scheduler.*

```
void SCH_End_Task(void)
{
 // Disable Interrupt on match 2
   T1MCR &= 0xFFFFFFBF;

 // Goto Sleep
 SCH_Go_To_Sleep();
}
```

Listing 4. *End-Task function in the TTC-Adaptive scheduler.*

```
void SCH_Task_Overrun_Update(void)
{
        // Goto sleep after ISR
        cTask = SCH_Go_To_Sleep;

        // Increment task overrun flag
        SCH_tasks_G[Index_G-1].Overrun++;
```

```
    // If there exists a backup task
    if (SCH_tasks_G[Index_G-1].bTask)

    {
       // Disable task interrupt on match 1
       T1MCR &= 0xFFFFFFF7;

       // Set backup task to run
       SCH_tasks_G[Index_G-1].pTask = SCH_tasks_G[Index_G-1].bTask;

       // Point index back to overruning task
       Index_G--;

       // Pause the next tick
       PauseTick = 1;
    }
}
```

Listing 5. *"Update" ISR of the Task-Overrun-Interrupt in the TTC-Adaptive scheduler.*

4. Evaluation of the TTC-Adaptive Scheduler

This section first outlines the experimental methodology used in this study to evaluate the TTC-Adaptive scheduler described in the previous section. It then presents the output results in terms of release task jitter, task-overrun handling capability and implementation costs. Note that the results obtained from the TTC-Adaptive scheduler are compared with those obtained from the modified version of the original TTC scheduler (as described in [16]), TTC-TG and TTC-MTI schedulers to highlight the benefits of using such a new scheduler implementation in systems requiring high degree of predictability.

4.1. Experimental Methodology

We first outline the experimental methodology used to obtain the results presented in this section.

4.1.1. Hardware Platform

It is assumed in this project that the target platform for the embedded system will be a small microcontroller (e.g. 8051, Infineon C16x, Philips LPC2xxx, or PH Processor: [48]) which will be programmed in C language.

In particular, the empirical studies reported in this study for the single-processor systems were conducted using Ashling LPC2000 evaluation board supporting Philips LPC2106 processor [49]. The LPC2106 is a modern 32-bit microcontroller with an ARM7 core which can run – under control of an on-chip PLL – at frequencies from 12 MHz to 60 MHz [47]. The single-processor studies outlined in this paper used an oscillator frequency of 12 MHz, and a CPU frequency of 60 MHz. The compiler used was the GCC ARM 4.1.1 operating in Windows by means of Cygwin (a Linux emulator for windows). The IDE and simulator used was the Keil ARM development kit (v3.12).

4.1.2. Jitter Test

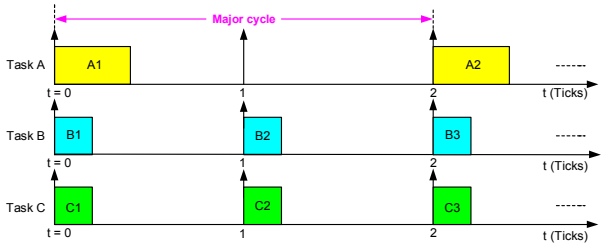

Figure 9. *Graphical representation of the task set in jitter test.*

In order to distinguish between the jitter behavior of the compared TTC scheduler implementations, the following task set is used (Fig. Figure **9**). To allow exploring the impact of schedule-induced jitter, Task A is scheduled to run every two ticks. Moreover, all tasks have variable execution durations: this is to allow exploring the impact of task-induced jitter.

Jitter is measured at the release time of each task as well as the scheduler tick. To measure the jitter experimentally, we set a pin high at the beginning of the tick or task (for a short time) and then measure the periods between every two successive rising edges. We recorded 5000 samples in each experiment. The periods were measured using a National Instruments data acquisition card 'NI PCI-6035E' [50], used in conjunction with appropriate software LabVIEW 7.1 [51].

To assess the jitter levels, we report two values: the average jitter and the difference jitter. The difference jitter is the difference between the minimum period and the maximum period obtained from the measurements in the sample set. This jitter is sometimes referred to as "absolute jitter" [14]. The average jitter is represented by the standard deviation in the measure of average periods. Note that there are many other measures that can be used to represent the levels of task jitter, but these measures were felt to be appropriate for this study.

4.1.3. Task-Overrun Test

In order to check the ability of the scheduler to deal with a task-overrun, we have used the following task set (Figure 10). Here, Task A is scheduled to run once every 20 ticks whereas Task B runs every tick. However, Task A is set to overrun by 10 ticks.

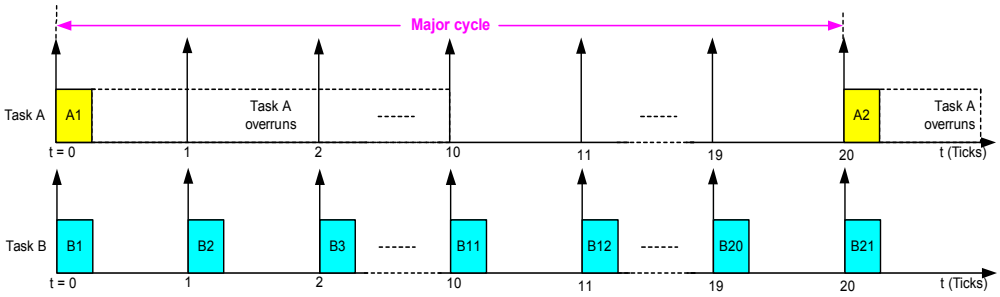

Figure 10. *Graphical representation of the task set in task-overrun test.*

4.1.4. CPU Overhead Test

In order to obtain CPU overhead measurements for each scheduler, we run the scheduler for 25 seconds and then, by using the performance analyzer supported by Keil simulator, the total time required for the scheduler to run throughout the measurement period was obtained. The percentage of the recorded CPU time was then reported to indicate the overhead (i.e. computational cost) required for each scheduler implementation.

4.1.5. Memory Overhead Test

In this test, the CODE and DATA memory values required to implement each scheduler were recorded. Memory values were obtained using the ".map" file which is created when the source code is compiled.

The STACK usage was also measured (as DATA memory overhead) by initially filling the data memory with 'DEAD CODE' and then reporting the number of memory bytes that had been overwritten after running the scheduler for sufficient period.

4.2. Jitter Measurements

Table 1 shows the periods and jitter measurements for the tasks in the original TTC, TTC-TG, TTC-MTI and TTC-Adaptive schedulers (for comparison purposes).

When comparing the original TTC and the TTC-TG schedulers, it can be seen that jitter characteristics are not improved by employing TG mechanisms. On the other hand, like the TTC-MTI scheduler, the TTC-Adaptive scheduler provides very low jitter at the release time of all tasks running in the system. Remember that in the TTC-Adaptive scheduler, users are not requested to enter estimates of the tasks' WCETs prior to system execution (as in the TTC-MTI scheduler). Also, the TTC-Adaptive scheduler has better capability to deal with task-overrun problem. This is further illustrated in the following section.4.3. Task-Overrun Behavior

Table 1. *Task jitter from the original TTC, TTC-TG, TTC-MTI and TTC-Adaptive schedulers (all values in µs).*

Scheduler		TaskA	Task B	Task C
Original TTC scheduler	Min Period	9999.4	2988.4	2164.3
	Max Period	9999.5	7011.1	7864.1
	Average Period	9999.5	4882.0	4799.3
	Diff. Jitter	0.1	4022.7	5699.8
	Avg. Jitter	0.0	1172.7	1226.9
TTC-TG scheduler	Min Period	9999.4	2985.5	2096.2
	Max Period	9999.5	7011.7	7848.1
	Average Period	9999.5	4922.7	4595.6
	Diff. Jitter	0.1	4026.2	5751.9
	Avg. Jitter	0.0	1175.3	1203.3
TTC-MTI scheduler	Min Period	9999.4	4999.7	4999.7
	Max Period	9999.5	4999.7	4999.7
	Average Period	9999.5	4999.7	4999.7
	Diff. Jitter	0.1	0.0	0.0
	Avg. Jitter	0.0	0.0	0.0
TTC-Adaptive scheduler	Min Period	9999.4	4999.7	4999.7
	Max Period	9999.5	4999.7	4999.7
	Average Period	9999.5	4999.7	4999.7
	Diff. Jitter	0.1	0.0	0.0
	Avg. Jitter	0.0	0.0	0.0

By monitoring the behavior of each scheduler, we found that in the original TTC scheduler, when an overrun takes place, the scheduler cannot prevent it. However, the architecture used in the design of this scheduler allows the system to keep track of the number of elapsed ticks during the overrun, and – once the overrunning task (Task A in Figure 10) completes – the scheduler performs all missing executions for Task B (in this case, 10 executions), before continuing to serve the tasks in the following ticks. This means that the scheduler has the potential to "catch up" in the event of certain (infrequent and temporary) errors: see Figure 11.

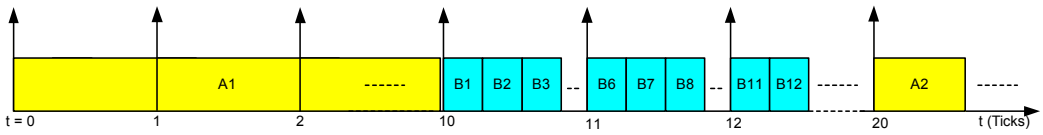

Figure 11. *The behavior of original TTC scheduler with when overrun occurs.*

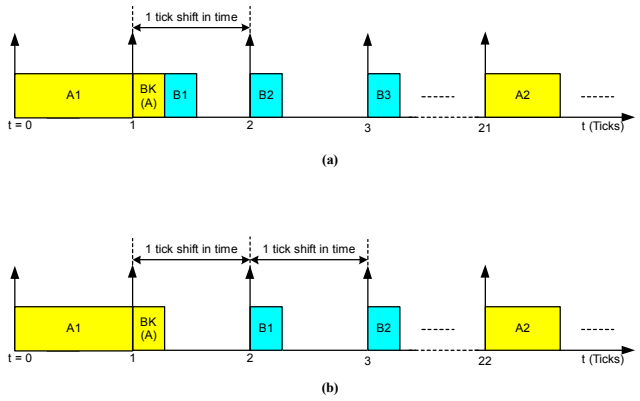

Figure 12. *The behavior of TTC-TG scheduler when overrun occurs: (a) running BK(A) and B1 tasks in the same tick, (b) running BK(A) and B1 tasks in different ticks to avoid domino effect.*

With the TTC-TG scheduler, the scheduler detects and hence terminates the overrunning task (Task A) at the beginning of the tick following the one in which Task A overruns. Moreover, the scheduler allows running a backup task BK(A) to replace Task A in the same tick in which the overrun is detected and hence continues to run the following tasks (Figure 12 (a)). This means that one tick shift is added to the whole schedule. However, in some cases where (for example) the schedule is heavily loaded with tasks, the insertion of a backup task in the next tick of overrun may cause a domino effect. To reduce the impact of such a problem, the whole schedule can be extended for one tick to allow the backup task to complete before the scheduler goes back to its normal operation. With the tasks arrangement used in this study (Figure 10), the whole schedule will be extended for two ticks: one for the backup task and one to run the missed task B1 (Figure 12 (b)).

In contrast, the TTC-Adaptive scheduler has also been designed to provide an efficient solution to task overrun problem. For example, such an implementation detects the overrun immediately and shutdown the overrunning task: this is similar to the behavior observed with the TTC-MTI. However, unlike the TTC-MTI scheduler, the TTC-Adaptive scheduler provides a support for backup task that will replace the overrunning task once shut down. In this scheduler, there can be three different options:

1) If it is not dependent on the output from Task A, Task B1 can still be scheduled to run in the same tick as Task A1 and before BK(A) executes (Figure 13 (a)).
2) If it is dependent on the output from Task A, Task B1 must be scheduled to run in the next tick after task BK(A) completes execution (Figure 13 (b)). This will obviously add one tick shift to the whole schedule.
3) To avoid any possibility for a domino effect to take place, the whole schedule can be extended for one more tick to allow a completion of BK(A) before returning to the normal schedule (Figure 13 (c)). The figure shows that, for the task set considered in Figure 10, two tick shifts will be added to the whole schedule.

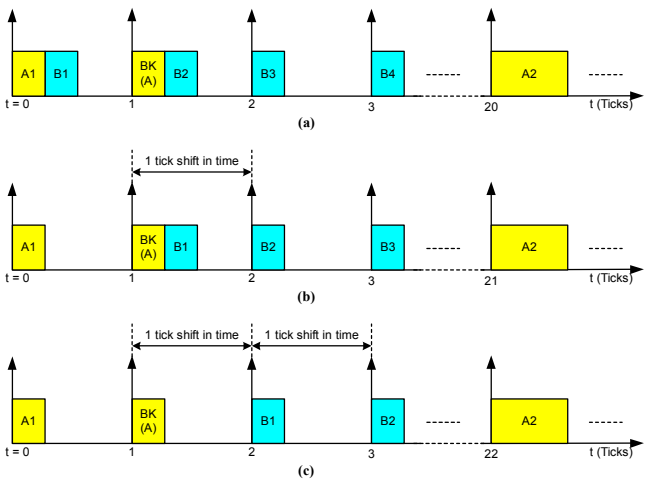

Figure 13. *The behavior of TTC-Adaptive scheduler when overrun occurs.*

Note that the TTC-Adaptive implementation presented in this paper considered the second option (Figure 13 (b)). Such a behavior has been checked through the IDE and simulator used. However, the scheduler framework developed has been made so flexible that the user can – with a little modification – adopt any of the three proposed solutions.

Remember that, in addition to low-jitter provision and overrun prevention, the most advantageous feature of the TTC-Adaptive scheduler is its ability to control the timing behavior of tasks based on real-time measurements (not estimations) of their WCETs.

4.4. CPU and Memory Overheads

Table 2 shows the CPU overhead for the original TTC, TTC-TG, TTC-MTI and TTC-Adaptive schedulers (for comparison purposes).

Table 2. *CPU overhead for the original TTC, TTC-TG, TTC-MTI and TTC-Adaptive schedulers*

Scheduler	Scheduler time (s):	Total time (s):	Overhead %
Original TTC scheduler	9.93	25.01	39.7
TTC-TG scheduler	9.95	25.03	39.8
TTC-MTI scheduler	9.90	25.01	39.6
TTC-Adaptive scheduler	9.95	25.01	39.8

The results in the table show that the implementation of the TTC-Adaptive scheduler requires no additional processing time as compared to the previous schedulers. This means that the developed scheduler is computationally cost-effective.

Table 3 shows the memory overheads for the original TTC, TTC-TG, TTC-MTI and TTC-Adaptive schedulers (for comparison purposes). It can clearly be seen that the code memory overhead in the TTC-Adaptive scheduler is 48% larger than that in the TTC-MTI scheduler. Such a difference

is resulted from the integration of the WCET measurement technique and the task guardian mechanism in the scheduler source code. This increase is around 34% and 25% when comparing the TTC-Adaptive with the original TTC and TTC-TG schedulers, respectively. Such an increase in the memory overhead is outweighed by the improvement achieved in the scheduler behavior.

Table 3. *Memory requirements (RAM and ROM) for the original TTC, TTC-TG, TTC-MTI and TTC-Adaptive schedulers*

Scheduler	ROM requirements (Bytes)	RAM requirements (Bytes)
Original TTC scheduler	4012	325
TTC-TG scheduler	4296	446
TTC-MTI scheduler	3620	514
TTC-Adaptive scheduler	5364	510

5. Conclusions

This paper suggested a useful addition to the range of TTC schedulers that have previously been developed. To deal with task-overrun problem while maintaining low levels of task jitter, TTC-Adaptive scheduler has been introduced and evaluated. As noted in the paper, addressing task overrun and task release jitter at the same time requires knowledge about the tasks' WCETs; which is accepted to be a very complicated process. In previous TTC implementations, it was assumed that such values are estimated and provided to the scheduler by the user. The TTC-Adaptive scheduler was aimed at offering a flexible implementation where the user needs not to estimate the tasks' WCETs during the design stage which in many cases cannot be accurate and may hence cause a significant degradation in the timing performance of the system.

As discussed in the paper, the TTC-Adaptive scheduler employs an online measurement method to calculate the WCETs for all tasks over a sufficient period of time. Such values are then used by the scheduler to adjust the timing of tasks and protect (guard) any task from overrunning. It is worth reminding that the WCET value computed by this scheduler for a particular task is the longest possible execution time of the task during the measurement period (this is not the actual WCET which, as observed in many studies, might require further sophisticated techniques to compute/estimate).

Since it was adapted from the TTC-MTI scheduler developed previously, TTC-Adaptive scheduler also has low resource requirements (for example, low code memory is required). Again, decision to employ the TTC-Adaptive scheduler in a given system would need to consider the system requirements in terms of timing as well as implementation costs.

It is important to note that the TTC-Adaptive scheduler was aimed towards a perfect TTC implementation as it provided effective solutions to jitter and overrun problems. However, a perfect TTC scheduler can be achieved if more features are considered. For example, future work suggests that techniques such as DVS [15] can be incorporated in the scheduler framework to achieve low-power characteristics at zero jitter. Such a modification would require a substantial amount of underlying work in order to avoid any conflicts between timer configurations.

Acknowledgements

The work presented in this paper was carried out in the Embedded Systems Laboratory (ESL) at University of Leicester, UK, under the supervision of Professor Michael Pont, to whom authors are thankful. This project was supported in part by the UK Government (EPSRC-DTA award). Authors would also like to thank Dr Zemian Hughes for providing assistance in writing the software code for the TTC-Adaptive scheduler.

References

[1] A.C. Shaw, Real-time systems and software, New York, John Wiley & Sons Inc, 2001.

[2] N. Nissanke, Realtime Systems, Prentice-Hall, 1997.

[3] A. Albert, "Comparison of event-triggered and time-triggered concepts with regard to distributed control systems," in Proceedings of Embedded World, Nurnberg, Germany, 17-19 Feb, 2004. pp. 235-252

[4] H. Kopetz, Real-time systems: Design principles for distributed embedded applications, Kluwer Academic, 1997.

[5] T. P. Baker, and A. Shaw, "The cyclic executive model and Ada". Real-Time Systems, Vol. 1, No. 1, 1989, pp. 7-25.

[6] C. D. Locke, "Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives." Real-Time Systems, Vol. 4, No. 1, 1992, pp. 37-53.

[7] M. Short, "Analysis and redesign of the 'TTC' and 'TTH' schedulers". Journal of Systems Architecture, Vol. 58, No. 1, 2012, pp. 38-47.

[8] M.J. Pont, Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers, ACM Press / Addison-Wesley, 2001.

[9] M. J. Pont, and M. Banner, "Designing embedded systems using patterns: A case study", Journal of Systems and Software, Vol. 71, No. 3, 2004, 2004, pp. 201-213.

[10] S. Kurian, and M.J. Pont, "Maintenance and evolution of resource-constrained embedded systems created using design patterns". Journal of Systems and Software, Vol. 80, No. 1, 2007, pp. 32-41

[11] E. Anbarasi, N. Karthik, and R. Prabakaran, "Analysis of time triggered schedulers in embedded system". In Electronics Computer Technology (ICECT), 2011 3rd International Conference on, Vol. 1, IEEE, 2011, pp. 134-137.

[12] M. Nahas, and A.M. Nahhas, "Ways for Implementing Highly-Predictable Embedded Systems Using Time-Triggered Cooperative (TTC) Architectures". In Dr. Kiyofumi Tanaka (Ed.), Embedded Systems - Theory and Design Methodology, ISBN: 978-953-51-0167-3, InTech, 2012.

[13] I.J. Bate, "Scheduling and Timing Analysis for Safety Critical Real-Time Systems", PhD dissertation, Department of Computer Science, University of York, 1998.

[14] G. Buttazzo, Hard real-time computing systems: predictable scheduling algorithms and applications, Second Edition, Springer, 2005.

[15] T. Phatrapornnant, and M.J. Pont, "Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling", IEEE Transactions on Computers, Vol. 55, No. 2, 2006, pp. 113-124.

[16] M. Nahas, Pont, M.J., and A. Jain, "Reducing task jitter in shared-clock embedded systems using CAN", In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004, Birmingham, UK, October 2004, pp. 184-194. Published by University of Newcastle upon Tyne.

[17] M. Nahas, "Employing two 'sandwich delay' mechanisms to enhance predictability of embedded systems which use time-triggered co-operative architectures", International Journal of Software Engineering and Applications, Vol. 4, No. 7, 2011, pp. 417-425.

[18] F. Cottet, and L. David, "A solution to the time jitter removal in deadline based scheduling of real-time applications", 5th IEEE Real-Time Technology and Applications Symposium - WIP, Vancouver, Canada, 1999, pp. 33-38.

[19] A.J. Jerri, "The Shannon sampling theorem: its various extensions and applications a tutorial review", Proceeding of the IEEE, Vol. 65, 1977, pp. 1565-1596.

[20] M. Torngren, "Fundamentals of implementing real-time control applications in distributed computer systems", Real-Time Systems, Vol. 14, 1998, pp. 219-250.

[21] P. Marti, J.M., Fuertes, R. Villà, and G. Fohler, "On Real-Time Control Tasks Schedulability", European Control Conference (ECC01), Porto, Portugal, 2001, pp. 2227-2232

[22] F. Abugchem, M. Short, and D. Xu, "An experimental HIL study on the jitter sensitivity of an adaptive control system", Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on, Cagliari, Italy, 2013, pp. 1-8.

[23] Z.H. Hughes, and M.J. Pont, "Design and test of a task guardian for use in TTCS embedded systems", In: Koelmans, A., Bystrov, A. and Pont, M.J. (Eds.) Proceedings of the UK Embedded Forum 2004, Birmingham, UK, October 2004, pp. 16-25

[24] K. L. Chan, and M. J. Pont, "Real-time non-invasive detection of timing-constraint violations in time-triggered embedded systems", In Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on, Bradford, UK, 2010, pp. 1978-1986.

[25] C.L. Liu, and J.W. Layland, "Scheduling algorithms for multi-programming in a hard real-time environment", Journal of the AVM 20, Vol. 1, 1973, pp. 40-61.

[26] E. Nett, H. Streich, P. Bizzarri, A. Bondavalli and F. Tarini, "Adaptive Software Fault Tolerance Policies with Dynamic Real-Time Guarantees". WORDS 96, IEEE Second Int. Workshop on Object oriented Real-time Dependable Systems, Laguna Beach, California, U.S.A., 1996.

[27] Y. Domaratsky, and M. Perevozchikov, "Highly Dependable Time-Triggered Operating System", Dedicated Systems Magazine, Vol. 4, 2000, pp. 77-80.

[28] L.B. Becker, and M. Gergeleit, "Execution Environment for Dynamically Scheduling Real-Time Tasks". RTSS 2001, 22nd IEEE Real-Time Systems Symposium, 2001.

[29] J. Engblom, A. Ermedahl, M. Sjoedin, J. Gustafsson, and H. Hansson, "Worst-Case Execution-Time Analysis for Embedded Real-Time Systems". Journal of Software Tools for Technology Transfer, 2001.

[30] M. Gergeleit, and E. Nett, "Scheduling Transient Overload with the TAFT Scheduler". GI/ITG specialized group of operating systems, Berlin, 2002.

[31] P. Puschner, "Is WCET Analysis a Non-Problem? - Towards New Software and Hardware Architectures". 2nd Intl. Workshop on Worst Case Execution Time Analysis, Vienna, Austria, 2002.

[32] L.B. Becker, E. Nett, S. Schemmer, and M. Gergeleit, "Robust scheduling in team-robotics". 11th International Workshop on Parallel and Distributed Real-Time Systems, Nice, France, 2003.

[33] R. Kirner, and P. Puschner, "Discussion of Misconceptions about Worst-Case Execution-Time Analysis". 3rd Euromicro International Workshop on WCET Analysis, 2003.

[34] J. Lelli, D. Faggioli, T. Cucinotta, and G. Lipari, "An experimental comparison of different real-time schedulers on multicore systems", Journal of Systems and Software, Vol. 85, No. 10, 2012, pp. 2405-2416.

[35] R. Wilhelm and D. Grund, "Computation Takes Time, but How Much?," Communications of the ACM, Vol. 57, no. 2, pp. 94–103, Feb. 2014.

[36] K.S. Vallerio, and N.K. Jha, "Task graph extraction for embedded system synthesis", Proceedings 16th International Conference on VLSI Design concurrently with the 2nd International Conference on Embedded Systems Design, 2003, pp. 480-486

[37] A.K. Gendy, and M.J. Pont, "Automatically configuring time-triggered schedulers for use with resource-constrained, single-processor embedded systems", IEEE Transactions on Industrial Informatics, Vol. 4, No. 1, 2008, pp. 37-46.

[38] J. Ganssle, The art of programming embedded systems, Academic Press, San Diego, USA, 1992.

[39] M.J. Pont, and H.L.R. Ong, "Using watchdog timers to improve the reliability of TTCS embedded systems". in Hruby, P. and Soressen, K. E. [Eds.]Proceedings of the First Nordic Conference on Pattern Languages of Programs, September, 2002 pp.159-200. Published by Micrsoft Business Solutions.

[40] F. Lakhani and M.J. Pont, "Applying Design Patterns to Improve the Reliability of Embedded Systems through a Process of Architecture Migration", High Performance Computing and Communication & 2012 IEEE 9th

International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on, 25-27 June, 2012, pp. 1563-1570.

[41] L. Bougueroua, L. George, and S. Midonnet, "An Execution Overrun Management Mechanism for the Temporal Robustness of Java Real-time Systems," in Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems, New York, NY, USA, 2006, pp. 188–195.

[42] F. Fauberteau, S. Midonnet, and L. George, "Robust Partitioned Scheduling for Real-Time Multiprocessor Systems," in Distributed, Parallel and Biologically Inspired Systems, M. Hinchey, B. Kleinjohann, L. Kleinjohann, P. A. Lindsay, F. J. Rammig, J. Timmis, and M. Wolf, Eds. Springer Berlin Heidelberg, 2010, pp. 193–204.

[43] R.E. Kontak, "Applicability of Ada tasking for avionics executives", Proceedings of the IEEE 1988 National Aerospace and Electronics Conference (NAECON), 23-27 May, Vol. 2, 1988, pp. 739-746.

[44] J.A. Stankovic, "Misconceptions about real-time computing", IEEE Computers, Vol. 21, No. 10, 1988.

[45] W.A. Halang, and A.D. Stoyenko, "Comparative evaluation of high-level real-time programming languages", Real-Time Systems, Vol. 2, No. 4, 1990, pp. 365-382.

[46] Z.M. Hughes, and M.J. Pont, "Reducing the impact of task overruns in resource-constrained embedded systems in which a time-triggered software architecture is employed", Transactions of the Institute of Measurement and Control, Vol. 30, 2008, pp.427 – 450.

[47] Philips Semiconductors, LPC2106/2105/2104 USER MANUAL, 2003, available online (Last accessed: September 2014) http://www.standardics.nxp.com/products/lpc2000/datasheet/lpc2104.lpc2105.lpc2106.pdf

[48] Z.M. Hughes, M.J. Pont, and H.L.R. Ong, "The PH Processor: A soft embedded core for use in university research and teaching In: Koelmans, A., Bystrov, A., Pont, M.J., Ong, R. and Brown, A. (Eds.), Proceedings of the Second UK Embedded Forum, Birmingham, UK, October 2005,pp. 224-245. Published by University of Newcastle upon Tyne.

[49] Ashling Microsystems LPC2000 Evaluation and Development Kits datasheet, 2007, available online (Last accessed: September 2014) http://www.ashling.com/pdf_datasheets/DS266-EvKit2000.pdf

[50] National Instruments, Low-Cost E Series Multifunction DAQ – 12 or 16-Bit, 200 kS/s, 16 Analog Inputs, 2006, available online (Last accessed: September 2014): http://www.ni.com/pdf/products/us/4daqsc202-204_ETC_212-213.pdf

[51] LabVIEW (2007) "LabVIEW 7.1 Documentation Resources", WWW website (Last accessed: September 2014): http://digital.ni.com/public.nsf/allkb/06572E936282C0E486256EB0006B70B4