Automatically Configuring Time-Triggered Schedulers for Use with Resource-Constrained, Single-Processor Embedded Systems

Ayman K. Gendy and Michael J. Pont, *Member, IEEE*

***Abstract*— This paper describes a novel two-stage search technique which is intended to support the configuration of time-triggered schedulers for use with resource-constrained embedded systems which employ a single processor. Our overall goal is to identify a scheduler implementation which will ensure that: (i) all task constraints are met; (ii) CPU power consumption is "as low as possible"; (iii) a fully co-operative scheduler architecture is employed whenever possible. Our search process is not exhaustive, and might be described as "best characteristics first" approach. We proceed iteratively, stopping the search when we have identified the first workable solution. We assume that - because we have begun the search with "best characteristics" - any schedule identified will represent a good (but not necessarily completely optimal) solution. We show that the proposed configuration algorithm is highly effective. We also demonstrate that the algorithm has much lower complexity than alternative "branch and bound" search schemes. We conclude by making some suggestions for future work in this area.**

***Index Terms*— automatic code generation, cooperative, cyclic executive, embedded system, hybrid, non pre-emptive, pre-emptive, scheduler, time triggered**

I. Introduction

THIS paper describes a novel two-stage search technique which is intended to support the configuration of schedulers for use with resource-constrained embedded systems. The specific implementation options which we consider are a time-triggered co-operative (TTC) scheduler (a form of cyclic executive: e.g. [1]), and a time-triggered "hybrid" (TTH) scheduler (sometimes referred to as a "multi-rate executive with interrupts": [2]). Such architectures are employed frequently in low-cost control systems (e.g. automotive control: [3]) and in condition-monitoring / fault diagnosis systems (e.g. [4]). Other recent uses of such simple schedulers can also be noted. For example Gangoiti et al [16] used a fixed-polling binary tree for USB bandwidth scheduling using a cyclic-executive-based approach "*which has low run-time overhead*" [16]. They showed that this method helped to guarantee the quality of service requirements for the device. In another recent study, Huang et al [17] used co-simulation of different tools employed in PLC programming and process modelling and simulation. They use a cyclic executive design to mark the time instants in which data exchange must be performed for each control loop, in order to facilitate the design of the simulation steps in both tools.

The type of TTC scheduler implementation discussed in this paper is usually implemented using a hardware timer, which is set to generate interrupts on a periodic basis (with "tick intervals" of around 1 ms being typical). In most cases, the tasks will be executed from a "dispatcher" (function), invoked after every scheduler tick. The dispatcher examines each task in its list and executes (in priority order) any tasks which are due to run in this tick interval. The scheduler then places the processor into an "idle" (power saving) mode, where it will remain until the next tick. Fig. 1 shows an example of three tasks run with TTC scheduler with a tick interval of 1 ms. Please note that the offsets (the time, measured from the start of the schedule, at which the task first starts execution) of Task A and Task B are zero while the offset of Task C is 1 ms.

Whether a TTC or TTH implementation is used, a number

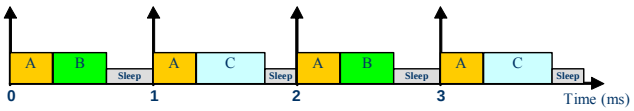

Fig. 1. Illustrating the operation of a typical (interrupt-driven) TTC scheduler implementation.

of key scheduler parameters must be determined (including the tick interval, task order, and initial delay - or phase - of each task). Inappropriate choices may mean that a given task set cannot be scheduled (at all). Where the parameter set does ensure that all tasks are scheduled, inappropriate decisions may still lead to unnecessarily high levels of task jitter and / or to increased system power consumption. It has been demonstrated in previous studies that the problem of determining these parameters is NP-hard, ([7] - [14]).

Our goal in this paper is to automate the process of parameter selection for this important class of low-resource schedulers. In determining these parameters, we aim to ensure that: (i) task constraints are met; (ii) power consumption is "as

Manuscript received May 28, 2007.

The authors are with the Embedded Systems Laboratory, University of Leicester, University Road, Leicester LE1 7RH, UK.
E-mail: {akg14, m.pont}@le.ac.uk.

low as possible"; (iii) a fully co-operative scheduler architecture is employed whenever possible.

The remainder of this paper is organised as follows. In Section II, we review previous work in scheduler design and selection. In Section III, we introduce and describe a scheduling algorithm ("TTSA1") which is used to automate the process of scheduler selection and configuration. In Section IV, we describe the process used to assess this algorithm and present the results obtained from this assessment. Finally, in Section V, we discuss the results, present our conclusions and make some suggestions for future work.

II. Related work

In this section, we review previous work in this area.

A. *Time-triggered software architectures for resource-constrained systems*

This paper is concerned with the development of software for an important class of embedded systems in which there are two (sometimes conflicting) constraints. First, (in order to reduce costs) we wish to implement the design on a low-cost microcontroller which has – compared to a desktop computer – very limited memory and CPU performance. Second, we wish to produce a system with low levels of task jitter (typically around 1 µs), because the application may involve periodic data sampling.

In order to minimise costs and maximise predictability in this type of application, we wish to keep the software architecture as simple as possible. As a consequence, instead of a full RTOS, some form of simple scheduler is generally used. For example, a cyclic executive ([7], [15]) is a form of co-operative (or "non pre-emptive") scheduler which has a "time triggered" [8] - as opposed to event-triggered [1] - architecture. Provided that an appropriate implementation is used, a time-triggered, co-operative (TTC) architecture is a good match for a wide range of low-cost, resource-constrained applications. TTC architectures also demonstrate very low levels of task jitter [15], and can maintain their low-jitter characteristics even when techniques such as dynamic voltage scaling (DVS) are employed to reduce system power consumption [6].

Despite some attractive features, such a TTC solution is not always appropriate. As Allworth has noted: "[The] main drawback with this [co-operative] approach is that while the current process is running, the system is not responsive to changes in the environment. Therefore, system processes must be extremely brief if the real-time response [of the] system is not to be impaired."[18]. In this case a pure co-operative scheduler will not generally be suitable. In such circumstances, it is tempting to opt immediately for a fully pre-emptive design. Indeed, some studies seem to suggest that this is the only viable alternative (e.g. [15],[19]). However, there are other design options available. For example, a single, time-triggered, pre-empting task can be added to a TTC architecture, to give what we have called a "time-triggered hybrid" (TTH) scheduler [20], [21] and others have called a "multi-rate executive with interrupts" [2]. Use of a TTH scheduler allows the system designer to create a static schedule made up of (i) a collection of tasks which operate co-operatively and (ii) a single – short - pre-empting task, see Fig. 2. Please note that in this schedule, the co-operative tasks all have the same priority (Priority C) while the pre-empting task has Priority P. We assume that Priority P > Priority C. Please also note that all tasks are periodic. This is in contrast to architectures investigated in some previous studies (e.g. [22]) which have sought to integrate time-triggered task scheduling with the response to aperiodic (event related) interrupts.

In systems employing a TTH architecture, the pre-empting task may be used for periodic data acquisition, typically by means of an analogue-to-digital converter or similar device. Such requirements are common in, for example, control

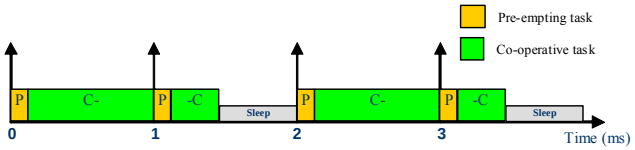

Fig. 2. Illustrating the operation of a typical TTH scheduler implementation (adapted from [21], Figure 1).

systems [23], and applications which involve data sampling and Fast-Fourier transforms (FFTs) or similar techniques: see, for example, the work by Schlindwein et al. [4].

Please note that it is not our intention to imply that a TTH architecture has – in terms of its scheduling behaviour – any particularly novel characteristics. Indeed, in many cases, a TTH architecture will be used to implement a common "rate-monotonic" schedule: this type of schedule has been extensively studied over a number of years: see, for example, work by Liu and Layland [24] through to work by Buttazzo [23]. In addition, it should be emphasised that we support in this architecture only a single pre-empting task (since this is all we require). As a consequence, in terms of a theoretical scheduling analysis, this type of scheduler is of limited interest. However, in a resource-constrained embedded system, it is a very attractive proposition because it allows us to create a scheduler with minimal resource requirements which is precisely matched to the needs of many practical applications.

B. *Challenges with simple TT architecture*

Two key challenges facing the developers of simple TTC and TTH designs are the possibility of task overruns (at run time) and the schedule fragility (at design time). We consider these challenges in this section.

1) *Impact of long tasks during system execution:* As we discussed in the previous sub-section, TTH architectures allow a designer to execute one or more tasks of Worst Case Execution Time (WCET) e and also respond within an interval t to external events, such that $t < (e +$ *execution time of the task that handles the event*$)$. This solution can be effective, for many designs, if the WCET of every task is

known at design time. Unfortunately, as many researchers have observed ([24]- [32]), determining the WCET of tasks is rarely straightforward.

Lack of knowledge about WCETs is a problem which faces the developers of many embedded systems (not just those based on TTC / TTH designs). For example, as Gergeleit and Nett have noted: "Nearly all known real-time scheduling approaches rely on the knowledge of WCETs for all tasks of the system." [29]. Nonetheless, the fact that a TTC / TTH architectures employs static scheduling (and, even in the case of TTH, a very limited degree of pre-emption) means that – in the event of a task overrun – the problem may not even be detected (let alone resolved). This may have a serious impact on the system behaviour. For example, as Buttazzo has noted: "[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" [23].

One simple solution to this problem is to err on the side of caution when employing WCET estimates, thereby reducing the chances that an overrun will occur. Typical "safety margins" used in this way are said to be around 20% [33]. Such an approach is simple and can be effective, but it 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 run time) any overrunning tasks can be shut down, and / or the schedule can be adjusted. This can be done by employing some form of "watchdog timer" (e.g. [34]) in a "scheduler watchdog" design (e.g. [35]). Alternatively, greater control over the system behaviour can be obtained by using a "task guardian" [36].

2) *The fragility of TTH and TTC designs:* Using mechanisms such as those outlined in the previous section, we can obtain highly predictable behaviour from TTC and TTH designs, even in the event of task overruns (due to error situations or design problems). However, it remains the case that – during the design process – TTC / TTH designs are "fragile": that is, small changes to the timing of particular tasks can mean that the developer has to make substantial changes to the whole schedule (e.g. see [1]). Our focus in this paper is therefore on ways in which the process of configuring TT schedulers for use in single-processor embedded systems can be automated.

In general, automated code generation holds the promise of reducing the time and effort required to implement safety-critical systems, while at the same time eliminating errors introduced in this stage of development [37]. Industries such as aerospace and automotive have made extensive use of automatic code generation tools aimed at control and signal processing systems ([38]-[40]). They are used first to model systems and then to generate code. Originally, code was generated automatically for prototyping platforms or PCs. More recently, code generation has become a more practical means of generating production code for embedded hardware. It is thought that many thousands of cars now rely on code generated using these techniques [38].

Automatic generation of schedules and schedulers is less common than "general purpose" code generation, but some work has been done in this area too. For example Tindell et al [9] used the simulated annealing technique to solve the problem of allocating a number of tasks to a number of processors in a distributed hard real time architecture. Ekelin and Jonsson [10] addressed the problem of task allocation and scheduling using constraint programming heuristics. Xu and Parnas [41] presented a branch and bound algorithm that finds the optimal schedule (if one exists), for a set of processes. The scheduler considered supports a limited degree of pre-emption. Xu [42] extended the previous study to find a feasible co-operative scheduler (if one exists) in a multi processor environment. Kovalyov and Xu [43] went on to further refine this algorithm. Sandström and Norström [44] used genetic algorithms to assign attributes such as priorities and offsets to a set of tasks that have complex timing constraints in a pre-emptive priority based run-time systems (using off-the-shelf operating systems). Dobrin and Fohler [45] also describe some useful techniques which are intended to help lower the overheads (in systems involving time-triggered scheduling and task pre-emption) by reducing the number of task pre-emptions which occur.

All of this previous work is relevant to a discussion about tool support for scheduler design. However, none of this previous work relates directly to TTC / TTH architectures: instead, such previous studies have tended to focus on "conventional" RT operating systems (e.g. VxWorks: [44]). Such operating systems exceed the resource requirements available in the types of processor considered in this study.

While previous studies on scheduler parameter selection do not relate directly to the work presented in this paper, there has been considerable work on the "automatic" creation of systems with a TTC architecture (e.g. [46], [47]). Such work supports the creation of code for complete TTC systems (including the system scheduler) using a collection of "design patterns". Such tools do not (so far) support the creation of systems with a TTH architecture [5]. In addition, even with TTC architectures, the user still needs to "hand tune" some task parameters (like the offset) and scheduler parameters (like the tick interval). The work presented in this paper seeks to address the problem of choosing between TTC and TTH schedulers and – for the chosen scheduler – determining an appropriate set of task parameters.

III. THE TTSA1 SCHEDULING ALGORITHM

In Section II, we considered the need for tools which will help to automate the process of developing TTC and TTH schedules. In the remainder of this paper we present and assess a novel algorithm which addresses this need. For ease of reference, we call this algorithm "time-triggered scheduling algorithm 1" (TTSA1). We describe TTSA1 in this section.

A. Overview

The pseudo code shown in Fig. 3 describes TTSA1. The input to TTSA1 is a list of task specifications and constraints. The algorithm tests the schedulability of the given task set, first using the TTC scheduler. If the task set cannot be scheduled with the TTC scheduler, the process is repeated using the TTH scheduler (see Fig. 4).

```
START
Arrange tasks in order according to their deadlines (EDF);
// Common divisors of task periods in descending order
GCD[t] = {GCD1, GCD2, …, GCDm}, t=1, 2,,.., m;
Sched_Strategy = {TTC, TTH};

// First check schedulability using TTC strategy
Sched_Strategy_Index = 1;
DO
   {
   Tick_Index = 1;
   DO
      {
      Tick_Interval = GCD[Tick_index];
      i = 1; Offset[i] = 0;
      Sched[i] = TRUE;   Sched_Tasks = 1;
      DO
         {
         i++; Offset[i] = 0;
         DO
            {
            Length_of_Major_Cycle = LCM(Period[k]),
                                    k=1,2..,i ;
            Max_Offset = Max(Offset[k]), k=1,2..,I;
            Test_Period = 2* Length_of_Major_Cycle +
                          Max_Offset;
            Sched[i] = Check_Sched(i, Test_Period,
                       Tick_Interval, Sched_Strategy_Index);
            IF (Sched[i] = TRUE)
               { Sched_Tasks ++ ;}
            ELSE
               { Offset[i]++ ;}
            } WHILE ( (Offset[i]<Period[i]) and
                      (Sched[i] = FALSE) );
         } WHILE ( i < n );
      IF ( Sched_Tasks = n )
         { print task offsets, tick interval, scheduler
                type; EXIT; }
      ELSE
         { Tick_index++;}
      } WHILE (Tick_index <= m )
   Sched_Strategy++;       // Try the TTH strategy
   } WHILE (Sched_Strategy <= 2);
Print list of scheduled and unscheduled tasks;
END
```

Fig. 3. Pseudo code for the TTSA1 algorithm.

```
START
Current_Tick_Num= 1; j=1; Current_Time=0; Temp_Tick_Num=0;
DO {
   DO {
      IF (Task[j] is due to run)
         {
         IF (Sched_Strategy[Sched_Strategy_Index] = TTC)
            {
             // are we in the start of a new Tick?
            If (Current_Time % Tick_Interval = 0)
                // add the scheduling overhead to
                // Current_Time
               { Current_Time += TTC_Overhead}
            Temp_WCET = WCET[j];
            WHILE (Temp_WCET > (Tick_Interval - Current_Time
                                % Tick_Interval))
               { //Temp_WCET is longer than reminder time in
                 //current tick
               Temp_WCET -= (Tick_Interval - Current_Time %
                             Tick_Interval);
               Current_Time += (Tick_Interval - Current_Time
                              %Tick_Interval) + TTC_Overhead;
               }
            Current_Time += Temp_WCET;
            }
         ELSE  //use TTH
            {// is it the pre-emptive tasks?
            IF ((j = p) and (Current_Tick_Num <>
              Temp_Tick_Num))
               { Current_Time += WCET[j] + TTH_Overhead;}
            ELSE
               {//it is a co-operative task, add its WCET to
                //the current
               Temp_WCET = WCET[j];
               Temp_Tick_Num = Current_Tick_Num;
               WHILE (Temp_WCET > (Tick_Interval –
                      Current_Time % Tick_Interval))
                  { //Temp_WCET is longer than reminder time
                    //in current tick
                  Temp_WCET -= (Tick_Interval - Current_Time
                               % Tick_Interval);
                  Current_Time += (Tick_Interval –
                                Current_Time % Tick_Interval)
                                 + TTH_Overhead;
                  Temp_Tick_Num++;
                  If (Task[p] is due to run in Temp_Tick_Num)
                     { //check exclusion relation
                     If (Exclusion[j][p] = 1)
                        {return (FALSE);}
                     ELSE
                        {Current_Time += WCET[p];}
                     }
                  }
               Current_Time += Temp_WCET;
               }
            }
         IF (D-line, Jit, Prec, Dist, or Lat constraint of
             the tasks added so far is not met)
            {return(FALSE);}
         }
      j++;
   } WHILE (j <= i);
   // Update the Current Time and Current tick count
   IF ((Current_Time < (Current_Tick_Num * Tick_Interval))
      { Current_Time += Current_Tick_Num * Tick_Interval;}
   Current_Tick_Num ++;
   } WHILE (Current_Tick_Num <= Test_Period);
RETURN TRUE;
END
```

Fig. 4. Pseudo code of the Check_Sched() function of TTSA1 scheduler.

The output from the algorithm depends on the results of each schedulability test, as follows:

- If all the tasks are schedulable, a suitable tick interval is calculated, along with the task order and the required offset value for each task.
- If the tasks cannot all be scheduled, a list of the schedulable tasks is generated[1].

To achieve this result, TTSA1 begins by sorting the tasks according to two criteria: a) task precedence, b) task deadline (earliest deadline first). It is then assumed that the first task will run with zero offset and the algorithm tries to find a suitable offset for the second task, using the longest possible tick interval. If such an offset is identified (and the constraints of both tasks are met), a third task is added to the system and the process is repeated. We carry on in this way until all tasks have been scheduled (if this proves possible), see Fig. 4.

Please note that our search process is not exhaustive, and might be described as "best characteristics first" approach: for example, we start with a long tick interval (which is known to reduce power consumption: see Appendix A) and we gradually reduce the tick interval until we match the timing needs of the application (if ever). We proceed iteratively, stopping the search when we have identified the first workable solution. We assume that - because we have begun the search with "best characteristics" - any schedule identified will represent a good (but not necessarily completely optimal) solution.

B. *Tick interval*

We have previously stated that an inappropriate choice of tick interval may mean that a given task set cannot be scheduled (at all). We illustrate this situation here with a simple example.

Suppose that we employ a tick interval of 2 ms with the task set shown in Table I (assuming zero offsets), Task B will always run after Task A (in the same tick) and can miss its deadline. However, using a tick interval of 1 ms and appropriate task offsets, changing the offset of Task B to 1 ms (1 tick), means that all tasks meet their deadlines.

Where the parameter set does ensure that all tasks are scheduled, inappropriate choice of tick interval may still lead – for example - to increased system power consumption. We have not been able to find evidence in the literature to back up this observation. Results from a small empirical study illustrating this result are therefore presented in Appendix A.

To find the most suitable (that is, longest possible) tick interval, the algorithm checks the schedulability using all the common divisors of the task periods, starting with the Greatest Common Divisor (GCD) for the best results in power reduction. The algorithm stops at the largest possible tick interval with which all the tasks meet their deadlines (if such an interval exists).

[1] This is intended to help the user to modify tasks which cannot be scheduled (for example, by dividing a long task into more than one short task [7], [15], [20]).

C. *Offset*

Choice of offset can have a significant impact on the levels of task jitter in the system.

As an example of the impact of an inappropriate offset choice, please see Table II. In this example, the task set cannot all be scheduled because the sum of the WCETs means that Task C cannot meet its deadline (and there will also be significant jitter in the start times of Task A).

By using a suitable tick interval and adjusting the task offsets we can often achieve a workable schedule (e.g. see [48], [49]). For example, the tasks in Table II can be scheduled if we use a tick interval of 5 ms and adjust the offset of Task C to 5 ms (1 tick).

D. *Test period*

Choosing a suitable offset for each task may require that we test the schedule (using different offset combinations) over a period of time long enough to determine that all the tasks will meet their deadlines (or not). Since all tasks are periodic, we need to test for schedulability over the "major cycle" (a period of time equal to the Least Common Multiple – LCM - of the task periods: e.g. [7]).

In addition, since each task may have a different offset, the full schedule will not necessarily begin immediately: instead, the algorithm must therefore test the schedule for one complete cycle, measured from the time that the last task to be added to the schedule is executed for the first time. Finally, we may also need to consider the task behaviour at the boundary between the end of one (major) cycle and the start of the next.

As a result, for a given tick interval and set of offsets, the testing period used in this paper is represented by (1), the units here are "ticks".

$$\text{Test\_period} = 2 \times \text{Length\_of\_Major\_cycle} + \text{Maximum\_offset} \quad (1)$$

We further assume (for the purposes of this paper) that the offset of a task is in the range of [0, period], assuming that values of offset and period are expressed in ticks.

So for a set of *n* tasks the longest test period can be

TABLE I
TICK SPECIFICATIONS FOR A SYSTEM IN WHICH TICK INTERVAL AFFECTS TASK SCHEDULABILITY

Task	WCET (ms)	Deadline (ms)	Period (ms)
A	0.3	0.5	2
B	0.4	0.5	2

TABLE II
TASK SPECIFICATIONS FOR A SYSTEM IN WHICH TASK OFFSETS AFFECT TASK SCHEDULABILITY

Task	WCET (ms)	Deadline (ms)	Period (ms)	Inappropriate Offset (ms)	Appropriate Offset (ms)
A	1	5	5	0	0
B	1.5	5	10	0	0
C	3	5	10	0	5

TABLE III
SAMPLE OF TASK SPECIFICATIONS AND CONSTRAINTS (SET OF 3 TASKS)

Task	WCET (*μs*)	Deadline (*μs*)	Period (*μs*)	Jitter (*μs*)	Exclusion	Precedence	Distance (*μs*)	Latency (*μs*)
A	496	3964	4000	1618	Task A Excludes Task C	Task A Precedes Task C	Distance between Task A & Task C is 3335	Latency between Task A & Task C is 3921
B	828	4711	10000	9488				
C	64	3673	4000	67				

TABLE IV
NUMBER OF TRIAL AND THE TOTAL TIME

	3-tasks sets				4-tasks sets				5-tasks sets			
	TTC		TTH		TTC		TTH		TTC		TTH	
	TTSA1	BaB	TTSA1	BaB	TTSA1	BaB	TTSA1	BaB	TTSA1	BaB	TTSA1	BaB
Minimum number of trials	2	2	2	2	3	3	3	3	4	4	4	4
Maximum number of trials	85	2966	75	2966	125	33571	64	35072	170	1585571	87	879901
Average number of trials	16.3	162.0	11.4	159.8	31.7	2561.7	17.4	2544.2	59.6	56283.7	23.7	46575.6
Total number of trials	16285	161962	11360	159823	31655	2561690	17360	2544241	59596	5.6E+07	23652	4.7E+07
Total time (s)	1	2	1.5	3	1.5	88	2	184	3	3091	3.5	4924

calculated form (2).

Test_period = 2 x LCM (P[1], P[2], .., P[n]) + Max (P[1], P[2],…,P[n]) (2)

It can be seen from (2) that – in theory – the LCM of the task periods and hence the test period could be very long (particularly in large task sets with co-prime periods). However, in practice, there may be some flexibility in the choice of task periods ([20], [50]). As an example Gerber et al [51], [52] present a design methodology in which the end-to-end timing constraints (which is initially defined in such a way like: the car dynamics, such as speed, must be updates, based on the input throttle position, within period of 5 ms) are transformed into a set of intermediate rate constraints. They introduce an algorithm that solves these constraints by minimising the CPU utilisation. They show that a feasible solution for task constraints (like the period) can found by considering the period harmonicity relationship of that task with all its successors. Kim et al [53] go further to improve and automate this period calibration method.

E. Task starting time

At any time, task Task[i] is considered to be due to run at tick 'Tick_Num' if the condition represented by (3) is true.

(Tick_Num - Offset[i]) % Period[i] = 0 (3)

F. Deadline checking

The task deadline is the time, measured from the start of the period, before which the task must finish its execution (sometimes called the "relative deadline": e.g. [44]).

Assuming that a specific segment of Task[i], which has deadline D[i] that is less than or equal to P[i], begin its execution at time 'Starting_Time' and finishes its execution at time 'Finishing_Time' this task is considered to have met its deadline if the condition in (4) is satisfied for all its segments:

(Finishing_Time – Starting_Time) ≤ D[i] (4)

G. Taking scheduler overheads into account

The scheduler overhead may have a considerable impact on the schedulability of the task set. This overhead arises from the time spent in handling the tick interrupt, the time spent in updating and testing the delay of each task in turn (in order to check which task should run next), and the time spent in saving/resuming the state of pre-empted tasks in TTH designs. The level of this overhead depends on many factors including the number of tasks in the system, the scheduler type, and the speed of the hardware used to implement the system.

Previous work has been conducted in this area, for example Sandström et al [22] handle the interrupt overhead in an efficient, non-pessimistic, way. In this paper we introduce an alternative way of representing the overall scheduler overhead for a given number of tasks. We assume that the scheduler overhead can be represented by adding a dummy task to the set of tasks to be scheduled. This additional task is included in our schedule calculations at every tick and has a WCET equal to the actual scheduler overhead. This effect is shown in the `Check_Sched()` function (Fig. 4).

Of course, we need to determine the WCET value for this "overhead" task. We cannot predict this value (without conducting an extensive – and expensive – modelling process). We therefore note that the maximum scheduling overhead will occur when all the tasks run in the same tick (if ever).

Assuming that we have *n* tasks and that the scheduler enters "sleep" mode after running all the ready tasks in each tick (if there is time left), then the scheduling overhead is given by (5).

$$overhead = Tick\_Interval - (time\_spent\_in\_sleep \bmod e + \sum_{i=1}^{n} WCET[i]) \quad (5)$$

The overhead can be determined empirically, using a scheduler with the same number of "dummy" (empty) tasks

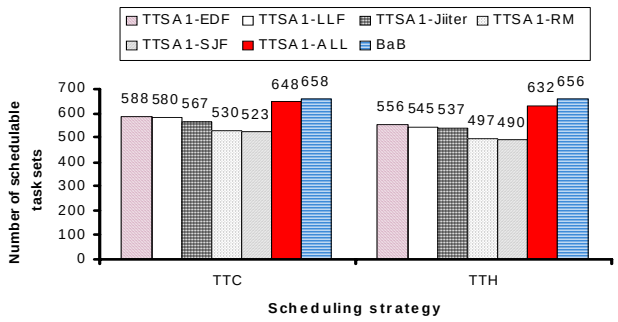

Fig. 5. Number of scheduled task sets (3 interdependent tasks in each set).

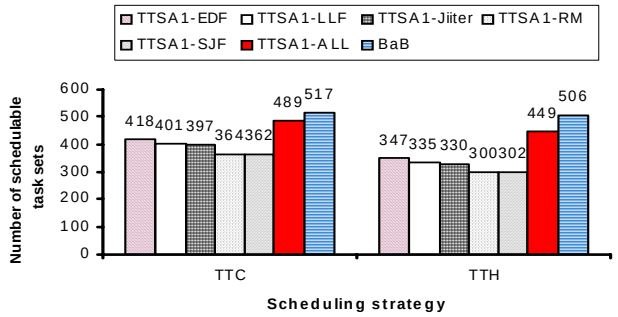

Fig. 6. Number of scheduled task sets (4 interdependent tasks in each set).

that will be employed in the final system. In this case, the last term in (5), $\sum_{i=1}^{n} WCET\ [i]$, can be assumed to be 0, and a single set of measurements will be required for a given hardware platform, regardless of the particular system being implemented

Determining the overhead in this way may seem to be unduly pessimistic for a static schedule. However, this measure of the maximum scheduler load is easily obtained (one single measurement, rather than having to make numerous measurements as we experiment with different schedules). In addition making a precise measurement of this load is – in practice – not straightforward. We therefore choose to accept a slight risk that the scheduling decision made will be altered by the inaccuracy of this overhead measurement (indeed, we assume that any loss of accuracy that results from this approach is likely to be smaller than the errors which results from WCET approximations for the tasks: see Section II).

Please note that we can determine the value of *time_spent_in_sleep_mode* either through the use of a hardware simulator or by making direct measurements from the hardware.

IV. Evaluating the TTSA1 algorithm

We evaluate the TTSA1 algorithm in this section. The "branch and bound" algorithm (BaB) was chosen previously as a benchmark to test the effectives of other heuristic algorithms [14]. The same algorithm is adapted here to evaluate the effectiveness of the TTSA1 algorithm.

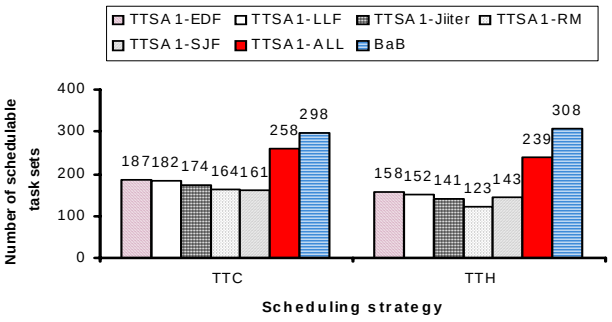

Fig. 7. Number of scheduled task sets (5 interdependent tasks in each set).

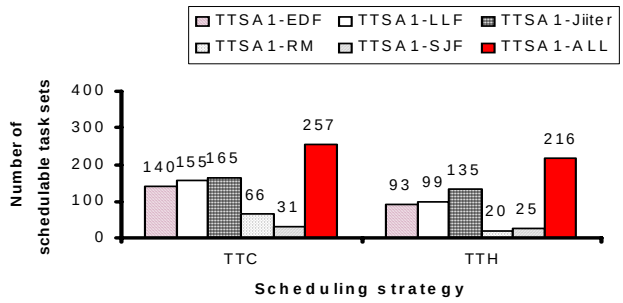

Fig. 8. Number of scheduled task sets (50 interdependent tasks in each set).

A. *Algorithm complexity*

Consider a set of *n* independent tasks, Task[1], Task[2], …, Task[n], with periods P[1], P[2], .., P[n], respectively. As previously discussed, the offset O[i] of task Task[i] is assumed to take any value from zero to P[i]. Choosing a suitable set of offsets may require testing schedulability over the period defined by (1).

Using the BaB search algorithm a partial schedule is constructed by adding tasks one by one to the system (trying all possible offsets of this task). A branch is terminated if the constraints of any added task, or the task under test, are violated. Ignoring the possible task offsets, in the worst case this will require testing *n* paths each of length *n*!; this has a complexity of $O(n.n!)$ which is "*computationally intractable and cannot be used in practical systems when the number of tasks is high*" [54]. In this case the longest testing period will therefore be given by (6).

(number of offsets combinations) x (number of possible execution orders) x (test period)

$$= \prod_{i=1}^{n} (P[i] \times (i!) \times (Max\,(O[1], O[2],.., O[i]) + 2 \times LCM\ (P[1], p[2],.., p[i])) \quad (6)$$

This problem has an order of complexity $O(t^n.n!)$, where *t* is the period (in ticks).

By contrast, the TTSA1 algorithm tries only a subset of the possible offset combinations. In this case, the longest testing period will be given by (7).

$$\sum_{2}^{n} (P[i] \times (Max\ (O[2], O[3],.., O[i]) + 2 \times LCM\ (P[1], P[2],.., P[i]))) \quad (7)$$

The complexity of this algorithm is $O((n\text{-}1)t)$ or simply $O(n.t)$.

Please note that summation in (7) starts from index 2, (rather than index 1). This is because the TTSA1 algorithm assumes that, after sorting the set of tasks, the first task is added to the system with offset 0. The offsets of subsequent tasks are determined at the time they are added to the system (one by one). Once an offset for a given task is identified, this is "fixed".

Please also note that these calculations ignore the effort required to determine the scheduler overhead (for both the BaB and TTSA1 calculations).

B. Algorithm performance

An empirical test of the performance of the TTSA1 algorithm was carried out. The procedure and results obtained by applying the algorithm to a set of interdependent tasks are detailed in this section.

1) *Method:* The schedulability of the task sets was assessed using the BaB search. The results were then compared with those obtained using the TTSA1 algorithm.

The chosen hardware platform was an NXP (formerly Philips) LPC2129 microcontroller running on a small evaluation board. The LPC2129 is based on an ARM7TDMI core and is typical of modern (low cost) embedded processors. The tests were conducted as follows:

- The measurements of scheduler load were carried out using the NXP board.
- The BaB and the TTSA1 algorithm schedulability tests were carried out using a simple (custom) schedule simulator, running on a desktop PC (making use of the load information obtained from the NXP board).

2) *Dataset used:* To explore the effectiveness of this algorithm, 1000 sets of tasks were randomly generated. Each set consisted of 3, 4 and 5 tasks specified by WCET, deadline and period. These specifications were generated according to the following criteria:

- $0 < WCET(i) \leq 1000\ \mu s$ (8)
- $WCET(i) < P(i) \leq 10000\ \mu s$ (9)
- $WCET(i) \leq D(i) \leq P(i)$ (10)

Task constraints of precedence, exclusion, distance, latency, and upper bound of jitter were also randomly generated and were in line with the findings from previous studies (e.g. see [41], [44]).

In order to simplify the calculations, task periods were (pseudo) randomly generated at multiples of 1 ms (constrained by (9)). Table III shows an example of a set of 3 tasks generated according to the above constraints.

3) Extending the basic algorithm: Variations on the original TTSA1 algorithm were also investigated in this trial. In the original algorithm (henceforth referred to as "TTSA1-EDF"), the tasks are added to the schedule "earliest deadline first".

We explored variations on this algorithm so that tasks were added:

- According to their slack - or laxity - time (least laxity first). This is "TTSA1-LLF" and is based on a "least laxity first" scheduling algorithm [55].
- According to their periods (shortest period first). This "TTSA1-RM" is related to a rate monotonic scheduling strategy [24].
- According to their WCET (shortest WCET first). This is referred here as "TTSA1-SJF" and is related to a "shortest job first" scheduling strategy [56].
- According to their upper bound of jitter (shortest jitter first). This is referred here as "TTSA1-Jitter"

TABLEA-1
AVERAGE POWER CONSUMPTION (MW) USING DIFFERENT TICK INTERVALS

Tick interval (ms)	TTC	TTH
1	16.6725	17.5583
2	16.3807	16.5104
5	16.1999	16.2332
10	16.1262	16.1524

4) *Results (small task sets):* The numbers of identified task sets that found to be scheduled using the TTSA1 and BaB are shown in Fig. 5 to Fig. 7. Please note that the results obtained by combining the (unique) results from TTSA1-EDF, TTSA1-LLF, TTSA1-RM, and TTSA1-SJF are shown in these figures as TTSA1-ALL. The number of trials until each of the two algorithms identified the set of tasks as scheduled/unscheduled and the total time is also shown in Table IV.

From the results obtained it was noted that:

- For both the TTC and TTH schedulers the results obtained from TTSA1 (when overheads are taken into account) are found to be a subset of the complete list of valid schedules identified by the BaB search. In addition, although TTSA1 tests the schedulability using a subset of all the possible offset combinations, it produces results which are similar to those obtained with the BaB method.
- The criteria used for adding the tasks have an impact on the schedulability of the set (different criteria may give different results).
- Combining results from the variations of TTSA1 together gives results which are very close to those obtained from the BaB search while requiring a much lower number of trials, and hence less time (see Section IV-A).

5) *Results (large task set):* The results shown in Fig. 5 to Fig. 7 consider a maximum of 5 tasks. This is not an unrealistic number for the resource-constrained systems we are concerned with in this paper. However, this task set does not fully test the algorithm. In order to explore the performance of TTSA1 on larger problems, 1000 new data sets were created. Each data set consisted of 50 tasks, each with a maximum execution time of 1 ms and maximum period of 100 ms. The task sets were randomly created according to the constraints described previously. To reduce the length of the major cycle, task periods were randomly generated as a multiple of 10 ms. The results from this test are shown in Fig. 8. It took approximately 10 seconds to complete the schedulability test for one set of 50 tasks using TTSA1-EDF,

and a total of approximately 50 ms to complete the test for TTSA1-All. It was not possible to complete this search using a BaB approach as this would have required performing a huge number of trials.

V. Discussion and conclusions

Our aim in this paper has been to help automate the process of determining the parameters required to schedule a given set of tasks in a resource-constrained embedded system employing a TTC or TTH architecture. We believe that we have achieved this aim through the use of a novel algorithm which – while it does not perform an exhaustive search – does provide results close to those obtained in the BaB search, in a fraction of time. While searching for a workable scheduler the proposed scheduling algorithm ensures the CPU power consumption is "as low as possible" (by choosing the longest possible tick interval), and that task constraints are met (by adjusting the tasks' offsets, tick interval, and task orders).

The results, while useful, still have scope for improvement. For example, we note that the match between TTSA1 and BaB is better for the TTC schedules than it is for the TTH schedules. As noted, the TTH designs support a single pre-empting task: in this study, we assume that this task should be the one with the shortest deadline, laxity, period, WCET, or jitter (for the TTSA1-EDF, TTSA1-LLF, TTSA1-RM, TTSA1-SJF, and TTSA1-Jitter algorithms, respectively). This choice may be unduly restrictive, not least when exclusion relations restrict the behaviour of the pre-empting task (thereby, in many cases, marking the set as "unschedulable"). So choosing the pre-empting task amongst all the other tasks in the set has considerable effect on the results. Further work is required to explore this.

Appendix A: Power consumption vs. tick interval

The simple time triggered architectures which are discussed in this paper (TTC / TTH) are built on the idea of executing the tasks from a (dispatcher) function which is invoked every scheduler tick. This function updates the state of each task, runs "ready" tasks, then it places the processor into a power-saving mode until the next tick. If the tick interval employed is shorter than necessary, there may be some empty ticks (ticks in which there is no tasks ready to run) during which the system has to come out of the power saving mode to execute the dispatcher before the system goes back to the power-saving mode. This will, inevitably, increase power consumption when compared to a design with an "optimal" tick interval.

To illustrate the effect of tick interval on system power consumption an empirical experiment was carried out. In this experiment a set of 3 dummy tasks was used and run on an NXP LPC2106 microcontroller. The period of all the three tasks was set at 10 ms. The power consumption of the core microcontroller was measured using a range of different tick intervals. In each case, the results of several runs were averaged using both TTC and TTH architectures (see Table A-1).

It can be noticed from Table A-1 that, for both TTC and TTH, choosing the largest possible tick interval reduces the power consumption.

Acknowledgment

The authors would like to thank M. Nahas (ESL, Leicester) for his help in making the measurements of the scheduler overhead and K. L. Chan (ESL, Leicester) for his help in making the power measurements.

References

[1] A.C. Shaw, Real-Time Systems and Software, John Wiley, New York, 2001.

[2] D. Kalinsky, "Context switch," *Embedded Systems Programming*, Vol. 14, No. 1, 2001, pp. 94-105.

[3] D. Ayavoo, Development of a Tool to Support the Design of Real-Time Embedded Control Systems for X-By-Wire Applications, PhD thesis, Embedded Systems Laboratory, University of Leicester, 2006.

[4] F. S. Schlindwein, M. J. Smith, and D. H. Evans, "Spectral analysis of Doppler signals and computation of the normalized first moment in real time using a digital signal processor," *Medical & Biological Engineering & Computing*, Vol. 26, 1988, pp. 228-232.

[5] C. Mwelwa, Development and Assessment of a CASE Tool to Support the Design and Implementation of Time-Triggered Embedded Systems, PhD thesis, Embedded Systems Laboratory, University of Leicester, 2006.

[6] 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 (Special Issue on Design and Test of Systems-On-a-Chip)*, Vol. 55, No. 2, 2006, pp. 113-124.

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

[8] H. Kopetz, Real-Time Systems, Design Principles for Distributed Embedded Applications, Kluwer Academic, 1997.

[9] K. Tindell, A. Burns, and A. Wellings, "Allocating hard real-time tasks: An NP-hard problem made easy," *Real-Time Systems*, Vol. 4, No. 2, 1992, pp. 145–165.

[10] C. Ekelin and J. Jonsson, "Evaluation of search heuristics for embedded system scheduling problems," *in Proc. Int. Conf. Principles and Practice of Constraint Programming, Paphos*, Cyprus, 2001, pp. 640 – 654.

[11] P. Brucker, M. R. Garey, and D. S. Johnson, "Scheduling equal-length tasks under treelike precedence constraints to minimize maximum lateness, "*Mathematics of Operations Research*, Vol. 2, No. 3, Aug. 1977, pp. 275-284.

[12] J. Xu and D. L. Parnas, "Pre-run time scheduling processes with exclusion relations on nested or overlapping critical sections," *11th IEEE Int. Phoenix Conf. Computers and Communications*, Scottsdale, AZ, USA, 1992, pp. 774-782.

[13] S. K. Baruah, "The non-preemptive scheduling of periodic tasks upon multiprocessors," *Real-Time Systems*, Vol. 32, No.1-2, Feb. 2006, pp.9-20,

[14] L. Cucu and Y. Sorel, "Non-preemptive multiprocessor scheduling for strict periodic systems with precedence constraints," *In Proc. 23rd Annual Workshop of the UK Planning and Scheduling Special Interest Group*, PLANSIG'04, Cork, Ireland, Dec. 2004.

[15] 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. 37-52.

[16] U. Gangoiti, M. Marcos, and E. Estévez, "Using cyclic executives for achieving closed loop co-simulation," *Proc. of the Joint 44th IEEE Control and Decision Conference and European Control Conference CDC-ECC'2005*, Sevilla, ISSN: 0-7803-9568-9, pp. 4785-3790.

[17] C. Huang, L. Chang, and T. Kuo, "A Cyclic-Executive-Based QoS Guarantee over USB", *in IEEE 9th Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, May 27-30, 2003, pp 88-95.

[18] S. T. Allworth, An Introduction to Real-Time Software Design, London, Macmillan, 1981.
[19] I. J. Bate, Scheduling and Timing Analysis of Safety Critical Hard Real-Time Systems, PhD thesis, Department of Computer Science, University of York, 1998.
[20] M. J. Pont, Patterns For Time-Triggered Embedded Systems, Addison-Wesley, 2001.
[21] A. Maaita and M. J. Pont, "Using 'planned pre-emption' to reduce levels of task jitter in a time-triggered hybrid," *UK Embedded Forum, Birmingham, UK, University of Newcastle,* 2005.
[22] K. Sandström, C. Norström, and G. Fohler, "Handling interrupts with static scheduling in an automotive vehicle control system," *In Proc. 5th Int. Conf. on Real-Time Computing Systems and Applications*, IEEE Computer Society, 1998, pp. 158-165.
[23] G. C. Buttazzo, "Rate monotonic vs. EDF: Judgement day," *Real-Time Systems*, Vol. 29, No. 1, 2005, pp. 5-26.
[24] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, Vol. 20, No. 1, 1973, pp. 40–61.
[25] L. B. Becker, E. Nett, S. Schemmer, and M. Gergeleit, "Robust scheduling in team-robotics," *11th Int. Workshop on Parallel and Distributed Real-Time Systems*, Nice, France, 2003.
[26] L.B. Becker and M. Gergeleit, " Execution environment for dynamically scheduling real-time tasks," RTSS 2001, *22nd IEEE Real-Time Systems Symposium*, London, 2001.
[27] Y. Domaratsky and M. Perevozchikov, "Highly dependable time-triggered operating system," *Dedicated Systems Magazine*, Oct.-Dec. 2000, pp. 77-84.
[28] J. A. Engblom, A. Ermedahl, *et al*., "Worst-case execution-time analysis for embedded real-time systems," *Journal of Software Tools for Technology Transfer*, Vol. 4, No. 4,2001, pp. 437-455.
[29] M. Gergeleit and E. Nett, "Scheduling Transient Overload with the TAFT Scheduler," *GI/ITG specialized group of operating systems*, Berlin, 2002.
[30] R. Kirner and P. Puschner, "Discussion of misconceptions about worst-case execution-time analysis," *3rd Euromicro International Workshop on WCET Analysis*, 2003.
[31] E. Nett, H. Streich, *et al*., "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.
[32] 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.
[33] K. S. Vallerio and N. K. Jha, "Task graph extraction for embedded system synthesis," *Proc. 16th Int. Conf. on VLSI Design concurrently with the 2nd Int. Conf. on Embedded Systems Design*, 2003, pp. 480-486.
[34] J. Ganssle, The Art of Programming Embedded Systems, Academic Press, San Diego, USA, 1992.
[35] M. J. Pont and R. H. L. Ong, "Using watchdog timers to improve the reliability of single-processor embedded systems: Seven new patterns and a case study," *in: Hruby, P. and Soressen, K. E. [Eds.] Proceedings of the First Nordic Conference on Pattern Languages of Programs*, 2002, pp. 159-200.
[36] Z. M. Hughes and M. J. Pont, "Design and test of a task guardian for use in TTCS embedded systems," *UK Embedded Forum, Birmingham*, UK, University of Newcastle, 2004.
[37] M. W. Whalen and M. P. E. Heimdahl, "On the requirements of high-integrity code generation," *Proc. of the 4th High Assurance in Systems Engineering Workshop*, 1999.
[38] P. Marsh, "Models of control," *IEE Electronics Systems and Software*, Vol. 1, No. 6, 2003, pp. 16-19.
[39] C. O'Halloran, "Issues for the automatic generation of safety critical software," *15th IEEE Int. Conf. Automated Software Engineering*, Grenoble, France, 2000.
[40] B. Schatz, T. Hain, *et al*., "CASE tools for embedded systems," Technical Report, Technical University of Munich, Reference No. TUM-I0309, 2003.
[41] J. Xu and D. L. Parnas, "Scheduling processes with release times, deadlines, precedence and exclusion relations," *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, 1990, pp. 360-369.
[42] J. Xu, "Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations," *IEEE Transactions on Software Engineering*, Vol. 19. No. 2, 1993, pp. 139-154.
[43] M. Kovalyov and J. Xu, "Uniform processor scheduling with release times, deadlines, precedence and exclusion relations International," *Workshop Discrete optimization methods in scheduling and computer-aided design*, Minsk, Belarus, 2000.
[44] K. Sandström and C. Norström, "Managing complex temporal requirements in real-time control systems," *9th IEEE Conf. Engineering of Computer-Based Systems*, IEEE, Sweden, 2002.
[45] R. Dobrin and G. Fohler, "Reducing the number of pre-emptions in fixed priority scheduling," *in Proc. 16th Euromicro Conference on Real-Time Systems*, 2004, pp.144-152.
[46] C. Mwelwa, M. J. Pont, and D. Ward, "Towards a CASE tool to support the development of reliable embedded systems using design patterns," paper presented at *the workshop "Quality of Service in Component-Based Software Engineering*, Toulouse, France, 2003.
[47] C. Mwelwa, K. Athaide, *et al*. , "Rapid software development for reliable embedded systems using a pattern-based code generation tool", *SAE Transactions: Journal of Passenger Cars (Electronic and Electrical Systems)*, Vol. 115, No. 7, pp. 795-803.
[48] J. Goossens and R. Devillers, "The non-optimality of the monotonic priority assignments for hard real-time offset free systems," *Journal of Real-Time Systems*, Vol. 19, No.2, 1997, pp. 107-126.
[49] J. Xu and D. L. Parnas, "Priority scheduling versus pre-run-time scheduling," *Int. Journal of Time-Critical Systems*, Vol. 18, pp. 7-23, Kluwer Academic Publishers, 2000.
[50] J. Xu and D. L. Parnas, "On satisfying timing constraints in hard - real - time systems," *IEEE Transactions on Software Engineering*, Vol. 19, No. 1, 1993, pp. 70 - 84.
[51] R. Gerber, S. Hong, and M. Saksena, "Guaranteeing end-to-end timing constraints by calibrating intermediate processes, "*In Proc. IEEE Real-Time Systems Symposium*, 1994, pp. 192–203. IEEE Computer Society Press.
[52] R. Gerber, S. Hong, and M. Saksena.," Guaranteeing real-time requirements with resource-based calibration of periodic processes," *IEEE Transactions on Software Engineering*, Vol. 21, No. 7, 1995, pp. 579–592.
[53] N. Kim, M. Ryu, *et al*., "Experimental Assessment of the Period Calibration Method: A Case Study," *Real-Time Systems*, Vol. 17, No. 1, July 1999, pp. 41-64.
[54] G. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Kluwer Academic, 1997.
[55] A. Chen, Real- Time Systems, Scheduling, Analysis, and Verfications, John Wiley & Sons, 2002.
[56] J. Stankovic and K. Ramamritham, "The design of the spring kernel," *Proc. of the IEEE real-Time Systems Symposium*, 1987.

Ayman K. Gendy received the BSc degree (Electrical Engineering) from Assiut University, Egypt, in 1995 and the MSc degree (Electrical Engineering) from Suez Canal University, Egypt, in 2001. He has been working at these universities since 1996. He is currently a PhD student at the Embedded Systems Laboratory, University of Leicester, UK. His primary research interests include, scheduling design, automatic code generation, and reliability in embedded systems.

Michael J. Pont holds a BSc from the University of Glasgow and a PhD from the University of Southampton. He worked at the University of Southampton and then University of Sheffield before joining the University of Leicester as a Lecturer in 1992. He is currently a Reader in Embedded Systems and Head of the Embedded Systems Laboratory at the University of Leicester; he is also Founder and CEO of TTE Systems Ltd. Michael's main research focus is on the development of techniques and tools which support the design and implementation of embedded systems: he is particularly interested in the links between system architecture and key characteristics such as temporal predictability and power consumption. Michael is author or co-author of more than 100 technical papers, and is the author of three books. Michael is a Member of the IEEE, a Member of the SAE, a Member of the IET and a Member of the BCS.