Automating the processes of selecting an appropriate scheduling algorithm and configuring the scheduler implementation for time-triggered embedded systems

Ayman K. Gendy and Michael J. Pont

Embedded Systems Laboratory,
University of Leicester,
University Road, Leicester LE1 7RH, UK
{akg14, M.Pont}@le.ac.uk

Abstract. Predictable system behaviour is a necessary (but not sufficient) condition when creating safety-critical and safety-related embedded systems. At the heart of such systems there is usually a form of scheduler: the use of time-triggered schedulers is of particular concern in this paper. It has been demonstrated in previous studies that the problem of determining the task parameters for such a scheduler is NP-hard. We have previously described an algorithm (“TTSA1”) which is intended to address this problem. This paper describes an extended version of this algorithm (“TTSA2”) which employs task segmentation to increase schedulability. We show that the TTSA2 algorithm is highly efficient when compared with alternative “branch and bound” search schemes.

Keywords: safety-related embedded systems, automatic code generation, scheduler, time triggered.

1 Introduction

Developers creating software for use in the majority of “desktop” applications face a very different set of challenges from those creating embedded software. For example, the time interval within which the desktop system responds to a command may vary significantly without causing a major problem: by contrast, even small variations in timing behaviour (milliseconds or much less) in embedded systems may prove life threatening in (for example) an industrial, automotive or medical system.

There are two common approaches used in building real-time embedded systems: event-triggered (ET) and time-triggered (TT). The ET approach may prove cost effective in cases where the system must handle many aperiodic and sporadic events ([4], [5]), since the conversion of such events to periodic events may reduce the system utilisation. On the other hand “*time-triggered systems are to be preferred with respect to fault tolerance*” [4] and are also considered as best match for supporting safety critical applications ([1], [2], [3], [4]). In addition, it is widely recognised that “*Very safety critical systems, like X-by-wire require fault-tolerance and redundancy.*”

The implementation of such systems probably will fail without the framework of time-triggered architectures” [5]. For these reasons, this paper focuses on systems with a TT architecture.

In most TT designs, an “offline” (also known as “pre-runtime”, or “static”) schedule is considered the best choice ([6], [7], [12], [14], [3]). It has been demonstrated in previous studies that the problem of testing the schedulability and determining the scheduler and task parameters for a set of tasks for such a system is NP-hard ([7], [8], [9], [10]). As part of an effort to address these problems we previously introduced a novel two-stage heuristic search technique (“TTSA1”) which is intended to support the configuration of TT schedulers for use with resource-constrained embedded systems which employ a single processor [11]. In this paper, we extend our TTSA1 technique. Our goal is to show that, with appropriate (static) task execution behaviour, tasks may be cleanly segmented, allowing an increase in schedulability, while meeting the constraints of a set of periodic tasks for use with reliable embedded systems.

The remainder of this paper is organised as follows. In Section 2, we review previous work in scheduler design and selection. In Section 3, we introduce and describe a modified scheduling algorithm (“TTSA2”) which is used to automate the process of scheduler selection and configuration. In Section 4, we describe the process used to assess the TTSA2 algorithm and present the results obtained from this assessment. Finally, in Section 5, we discuss the results, present our conclusions and make some suggestions for future work.

2 Related Work

In this section we review previous work in this area.

2.1 Scheduling safety critical resource-constrained embedded systems

A wide range of software architectures can be used for real-time systems, ranging from a simple scheduler to a full real-time operating system (RTOS).

For resource-constrained embedded systems, which have a very limited memory and CPU performance, a simple form of “time triggered co-operative” (TTC) – a form of cyclic executive – scheduler ([2], [8], [12], [14], [15]), “*which has low run-time overhead*”[14], is often used. For safety-critical applications which have hard real-time constraints, such as low jitter requirements, TTC architectures demonstrate very low levels of task jitter [16], and can maintain their low-jitter characteristics even when techniques such as dynamic voltage scaling (DVS) are employed to reduce system power consumption [17].

2.2 Time triggered co-operative scheduler (TTC)

The TTC implementation discussed in this paper executes each task in a predefined time intervals which is derived from a scheduler “tick”. The scheduler tick is usually

signalled by an interrupt associated with the (periodic) overflow of a hardware timer. At each tick the status of each task is updated and tasks which are due to run are dispatched. The processor is then often placed in an “idle” mode, where it will remain until the next tick (in order to reduce the system power consumption).

2.3 Time triggered hybrid scheduler (TTH)

Despite some attractive features, a TTC solution is not always appropriate. For example a TTC system cannot respond to a critical external event while executing specific task: this presents problems if the required response time is shorter than the worst case execution time, “WCET”, of any of the system tasks [18].

In these circumstances, the TTC architecture can be replaced with a fully pre-emptive architecture (for example, a rate monotonic or the earliest deadline first, architecture [22]). Such an approach provides flexibility (and, possibly, portability), but it will also tend to increase the system complexity and overhead when compared to pre-run-time scheduling ([6], [7]).

In some designs the system responsiveness can be increased while maintaining the minimal resource requirements, by allowing a limited level of pre-emption in the system. This can be done by employing what we call a “time-triggered hybrid” (TTH) scheduler ([3], [19]), and others have called a “multi-rate executive with interrupts” [20]. The TTH can be seen as a rate-monotonic scheduler that supports a single, short, high priority, pre-empting task, and a collection of co-operative tasks (which have equal priority which is less than that of the pre-empting task).

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 systems [13], and applications which involve data sampling and Fast-Fourier transforms (FFTs) or similar techniques: see, for example, the work by Schlindwein et al. [21].

2.4 Scheduler design and configuration

When implementing a TTC or TTH scheduler, the system designer has to determine some parameters (including the length of the tick interval, the order in which tasks must be dispatched, and the initial delay - or phase - of each task). Inappropriate choice of these parameters may affect systems reliability (by violating task constraints) or 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], [8], [9], [10]).

In order to cope with this challenge, schedulability analysis and scheduler design have been studied extensively over many years: see, for example, work by Liu and Layland [22] through to work by Xu [7]. Researchers have proposed solutions based on simulated annealing techniques [9], constraint programming heuristics [30], branch and bound algorithm ([28]), and genetic algorithms [29].

None of the work summarised above relates directly to TTC / TTH architectures: instead, most previous studies have tended to focus on “conventional” RT operating

systems (e.g. VxWorks: [29]). Such operating systems exceed the resource requirements available in the types of processor considered in this study.

2.5 The TTSA1 scheduling algorithm

In an effort to support creation of TTC / TTH designs we have previously introduced a novel two-stage heuristic search technique, “TTSA1”, which is intended to support the configuration of these time-triggered schedulers for use with resource-constrained embedded systems which employ a single processor [11].

As noted above the TTSA1 algorithm helps to automate the process of both scheduler selection and configuration. If a suitable scheduler is identified for a given task set, TTSA1 attempts to determine the suitable scheduler parameters (the tick interval) and task parameters (such as the task order and task offset). In determining these parameters, TTSA1 aims to ensure that: (i) task constraints are met; (ii) power consumption is “as low as possible”; (iii) a fully co-operative scheduler architecture is employed whenever possible.

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 this scheduler, the process is repeated using the TTH scheduler. The algorithm calculates a suitable tick interval, along with the task order and the required offset value for each task if all the tasks are schedulable; otherwise a list of the schedulable tasks is generated.

To achieve this result, TTSA1 begins by sorting the tasks according to two criteria: a) task precedence, b) task deadline, laxity, period, WCET, or jitter. 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).

3 The TTSA2 scheduling algorithm

In this section, we describe a modified version of the TTSA1 algorithm (“TTSA2”). TTSA2 employs task segmentation to increase the number of task sets which can be scheduled.

3.1 Overview

Despite its attractive features, the TTSA1 algorithm fails to find a suitable schedule for a set of tasks in some cases. For example assume that for a given set all tasks have deadlines equal to their periods. Assume also that this set includes two short tasks (Task S1 and Tasks S2), and at least one long task (Task L).

Table 1. Task specifications for task set that cannot be scheduled with TTC/TTH.

Task	WCET (ms)	Deadline (ms)	Period (ms)
A	10	50	50
B	1	10	10
C	1	10	10

Table 2. Task specifications for task set that can be scheduled with TTC/TTH.

Task	WCET (ms)	Deadline (ms)	Period (ms)
SA ₁	5	45	50
SA ₂	5	50	50
SB ₁	1	10	10
SC ₁	1	10	10

The TTSA1 algorithm fails to find a suitable schedule for this set if:

$$\text{Deadline (S1)} < \text{WCET (S1)} + \text{WCET (L)}$$

and

$$\text{Deadline (S2)} < \text{WCET (S2)} + \text{WCET (L)}$$

For example Task B and / or Task C shown in **Table 1** will miss their deadlines every time Task A runs if these three tasks are scheduled using TTC / TTH. To overcome this situation, while still using a TTC / TTH architecture, long task(s) can be divided into multiple short tasks ([3],[8], [16]): for example Task A can be divided into two segments, Segment SA₁ and Segment SA₂, as shown in **Table 2**.

As previously indicated, testing the schedulability of a set of tasks and finding a suitable scheduler for them (if any) is an NP-hard problem. The problem becomes more complex if some parts of some tasks are required to exclude parts of other tasks in the set. For example, it may be that Segment SA₂ in Task A excludes Segment SB₃ and Segment SC₂ in Task B and Task C respectively, and Segment SB₁ in Task B excludes Segment SC₁ in Task C.

In this section we extend our previous TTSA1 algorithm to deal with such situations. We call the resulting algorithm TTSA2. The input to TTSA2 is a list of task specifications and constraints, including critical-section boundaries.

The TTSA2 algorithm tests the schedulability of the given task set, first using the TTC scheduler, if possible, otherwise it will try the TTH, considering each task as a single segment. If the task set cannot be scheduled the process is repeated after dividing one or more long tasks into two or more shorter tasks. The algorithm calculates a suitable tick interval, the task order, the smallest number of task segments along with the required offset value for each task and task segment if all the tasks are schedulable; otherwise a list of the schedulable tasks and task segments is generated.

To achieve this result, TTSA2 begins (like TTSA1) by sorting the tasks according to two criteria: a) task precedence, b) task deadline, laxity, period, WCET, or jitter. It is then assumed that the first task will run as one segment with zero offset and the algorithm tries to find a suitable offset for the second task (in one segment), using the

longest possible tick interval (the greatest common divisor of the task periods). 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). If a schedule cannot be found at any stage the last task added to the design is removed and divided into two segments. After adding the segmentation overhead and updating the segment deadlines (as explained in the next subsections), the search proceeds (Fig. 1).

```

START
Arrange tasks;
// 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];
        Add the first task as one segment with zero offset;
        i = 1; Sched_Tasks = 1;
        DO{
            i++;
            Add the next task, one segment at a time;
            start segment with zero offset;
            DO{
                Sched[i] = Check_Sched(i, Tick_index,
                                      Sched_Strategy_Index);
                IF (Sched[i] = TRUE)
                { Sched_Tasks ++ ;}
                ELSE
                {
                    Increment offset of the last added segment;
                    Add the segmentation overhead;
                }
            } WHILE ((constraints violated) and
                    (offset <= Period));
        } WHILE (i < total number of tasks);
        IF ( Sched_Tasks = total number of tasks )
        {
            Print task offsets, tick interval,
            scheduler type;
            EXIT;
        }
        ELSE
        { Tick_index++;}
    } WHILE (Tick_index <= m )
    Sched_Strategy++; // Try the TTH strategy
} WHILE (Sched_Strategy_Index <= 2);
Print list of scheduled and unscheduled tasks;
END

```

Fig. 1. Pseudo code for the TTSA2 algorithm.

Please note that this search process is not exhaustive, and might be described as “best characteristics first” approach: for example, it starts with a long tick interval (which is known to reduce power consumption) and it gradually reduces the tick interval until it matches 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.

3.2 Adjusting the segment deadline

If Task T is divided into n segments, $ST_1, ST_2, \dots, ST_n$, then the TTSA2 algorithm calculates the deadline of each segment as follows:

$$\text{Deadline}(ST_n) = \text{Deadline}(T) . \quad (1)$$

$$\text{Deadline}(ST_{i-1}) = \text{Deadline}(ST_i) - \text{WCET}(ST_i), \text{ where } i = n, n-1, n-2, \dots, 2 . \quad (2)$$

Please notice the deadline of Segment SA_1 in **Table 2** as an example.

To be able to divide long tasks into multiple short tasks accurate information about the task WCET and the points at which the task can / cannot be pre-empted (the critical sections boundaries) must be specified. This can be done using techniques such as the “single path programming paradigm” ([24], [25]) or code balancing techniques [23].

3.3 Adding segmentation overhead

If a task is divided into two or more segments, the TTSA2 algorithm takes segmentation overhead into account. This overhead represents the time needed to save the context of the current segment and the time needed to restore this context when the next segment becomes ready to run. The time required for saving the context (*Context_Saving_overhead*) may not be the same as that required for loading the context (*Context_Loading_overhead*).

If Task T is divided into n segments, $ST_1, ST_2, \dots, ST_n$, then the TTSA2 algorithm updates the segments WCETs to reflect this overhead, as follows:

$$\text{WCET}(ST_1) = \text{WCET}(ST_1) + \text{Context_Saving_overhead} . \quad (3)$$

$$\text{WCET}(ST_i) = \text{Context_Loading_overhead} + \text{WCET}(ST_i) + \text{Context_Saving_overhead}, \text{ where } i = 2, 3, \dots, n-1. \quad (4)$$

$$\text{WCET}(ST_n) = \text{Context_Loading_overhead} + \text{WCET}(ST_n) . \quad (5)$$

4 The effectiveness of the TTSA2 scheduling algorithm

In this section the complexity and the effectiveness of the TTSA2 is evaluated. We compare the performance of the TTSA2 with that of the “branch and bound” algorithm (BaB), a standard benchmark which has been used previously to test the effectiveness of heuristic algorithms[26].

4.1 Algorithm complexity

Assume we have a set of n independent tasks and that each consists of s segments. Investigating the schedulability of these tasks by means of a BaB algorithm requires 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”[27].

As noted elsewhere [11], the offset of each task can be any value in the range $[0, \text{Period}]$, in ticks. Taking all possible offset combinations (t^n), where t is the period, and considering each task as set of s segments, each may have different offset, the complexity will increase to $O(t^{n.s}.n!)$.

By contrast, the TTSA2 algorithm does not try all paths. In addition, it does not change the task or / and segment offset of a given task once it has been added successfully to the schedule (that is, added without causing violation of the constraints of any of the tasks and segments which have been included in the schedule previously). The complexity of this algorithm is $O(n.s.t)$.

4.2 Algorithm performance

An empirical test was carried out to explore the performance of the TTSA2 algorithm. The procedure and the results of this test are discussed in this section.

4.2.1 The test tools

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 measurements of the scheduler overhead and the segmentation overhead were carried out using this microcontroller. This overhead was taken into account while performing the scheduling selection and configurations.

To compare the performance of the TTSA2 with that of the BaB a simple (custom) scheduler simulator was executed on a desktop PC.

4.2.2 Task constraints

The constraints considered in this study are described in this section.

Jitter calculation

As far as we are concerned in this paper, a task's jitter is the variation in the interval between the start times of the task. The starting time of each task is recorded so that the jitter statistics can be estimated. In the experiment discussed in the present paper, the upper bound of task jitter is (pseudo) randomly generated according to Equation 6.

$$0 \leq \text{Jitter} \leq P(i), \text{ where } P(i) \text{ is the period of Task } i. \quad (6)$$

Precedence

If it is required that Task A precedes Task B, then, in any tick, Task B is allowed to start its execution only after Task A completes its execution (e.g. see [28]).

In the current study, the precedence relation between any two tasks, A and B, is (pseudo) randomly generated iff

$$P(A) = P(B). \quad (7)$$

and

$$P(A) \geq (WCET(A) + WCET(B)). \quad (8)$$

Exclusion

If it is required that Segment SA_2 in Task A excludes Segment SB_3 in Task B, then, at any tick, Segment SA_2 is not allowed to pre-empt Segment SB_3 and vice versa [28].

In the current study some tasks segments in each set were (pseudo) randomly selected to have an exclusion relation between them.

Distance

The distance relation between any two tasks, A and B, can be defined as the minimum distance between the completion of Task A and the start of Task B [29].

In the current study the precedence relation between two tasks was (pseudo) randomly generated according to Equation 9.

$$0 \leq \text{Distance}(A, B) \leq P(A) - (WCET(A) + WCET(B)). \quad (9)$$

given that:

$$P(A) = P(B) \text{ and } P(A) \geq (WCET(A) + WCET(B)) \quad (10)$$

Latency

The latency relation between any two tasks, A and B, can be defined as the maximum distance between the completion of Task B and the start of Task A [29].

In the current study the latency relation between two tasks was (pseudo) randomly generated as follows:

If there are no distances constraint between Task A and Task B then:

$$(WCET(A) + WCET(B)) \leq \text{Latency}(A, B) \leq \text{Max}(P(A), P(B)) \quad (11)$$

given that:

$$\text{Max}(P(A), P(B)) \geq (WCET(A) + WCET(B)). \quad (12)$$

Otherwise:

$$(\text{WCET}(A) + \text{WCET}(B) + \text{Distance}(A,B)) \leq \text{Latency}(A, B) \leq P(A) . \quad (13)$$

4.2.3 Dataset used

To explore the effectiveness of this algorithm, 1000 sets of tasks were (pseudo) randomly generated. Each set consisted of 3, 4 and 5 tasks specified by WCETs, deadlines and periods. These specifications were generated according to the following criteria:

$$0 < \text{WCET}(i) \leq 2000 \mu\text{s} . \quad (14)$$

$$\text{WCET}(i) < P(i) \leq 10000 \mu\text{s} . \quad (15)$$

$$\text{WCET}(i) \leq D(i) \leq P(i) . \quad (16)$$

In order to simplify the calculations, task periods were (pseudo) randomly generated at multiples of 1 ms (constrained by (15)). Task constraints of precedence, exclusion, distance, latency, and upper bound of jitter were also (pseudo) randomly generated and were in line with the findings from previous studies (e.g. see [28], [29]) and are (pseudo) randomly generated constrained by Equation 6 – Equation 16.

4.2.4 Results (small task sets):

We tested the effectiveness of the TTSA2 algorithm when scheduling small sets of tasks (each contains 3, 4, or 5 tasks) and compared the results with those from the TTSA1 and the BaB algorithms. The results obtained from the BaB algorithm with / without using task segmentation are recorded as BaB1 and BaB2 respectively.

Fig. 2 to Fig. 4. show the number of task sets that was found to be schedulable using TTSA1, TTSA2, BaB1, and BaB2. Please note that the results obtained by combining the (unique) results from TTSAx-EDF, TTSAx-LLF, TTSAx-Jitter, TTSAx-RM, and TTSAx-SJF are shown in these figures as TTSAx-ALL, where x equals 1 or 2 for TTSA1 and TTSA2. The number of trials until each algorithm identified the set of tasks as schedulable/unschedulable and the total time are also shown in **Table 3**.

From the results obtained it was noted that:

- TTSA2 found a suitable scheduler for more sets than TTSA1.
- Because TTSA2 tries to find a suitable (TTC or TTH) scheduler using the lowest number of task segments, the results obtained from TTSA1 are found to be a subset of the complete list of valid schedules identified by TTSA2. This means that all the schedulers identified by both TTSA1 and TTSA2 have the same scheduling overhead and power consumption.
- The results obtained from TTSA1 and TTSA2 (when overheads are taken into account) are found to be a subset of the complete list of valid schedules identified by BaB1 and BaB2, respectively. In addition, although TTSA1 and TTSA2 test the schedulability using a subset of all the possible offset combinations, they produce results which are similar to those obtained with the BaB1 and BaB2 methods.

Table 3. Number of trials and the total time

3-task set	TTC				TTH			
	TTSA1	TTSA2	BaB1	BaB2	TTSA1	TTSA2	BaB1	BaB2
Minimum number of trials	2.00E+00	2.00E+00	2.00E+00	2.00E+00	2.00E+00	2.00E+00	2.00E+00	2.00E+00
Maximum number of trials	2.70E+02	5.80E+02	2.97E+03	4.66E+06	1.65E+02	3.30E+02	2.83E+03	1.99E+05
Average number of trials	2.46E+01	4.63E+01	3.51E+02	1.75E+04	1.59E+01	3.04E+01	1.97E+02	2.08E+03
Total number of trials	2.46E+04	4.63E+04	3.51E+05	1.75E+07	1.59E+04	3.04E+04	1.97E+05	2.08E+06
Total time (s)	4.00E+00	4.00E+00	3.90E+01	8.49E+02	2.50E+00	2.50E+00	2.50E+01	1.17E+02
4-task set	TTC				TTH			
	TTSA1	TTSA2	BaB1	BaB2	TTSA1	TTSA2	BaB1	BaB2
Minimum number of trials	3.00E+00	3.00E+00	3.00E+00	3.00E+00	3.00E+00	3.00E+00	3.00E+00	3.00E+00
Maximum number of trials	2.65E+02	5.30E+02	7.23E+04	1.60E+08	2.65E+02	5.30E+02	4.21E+04	2.78E+07
Average number of trials	3.49E+01	7.01E+01	5.45E+03	5.30E+05	2.49E+01	4.99E+01	3.24E+03	1.13E+05
Total number of trials	3.49E+04	7.01E+04	5.44E+06	5.29E+08	2.49E+04	4.99E+04	3.23E+06	1.13E+08
Total time (s)	3.50E+00	4.50E+00	2.22E+02	1.73E+04	2.50E+00	4.50E+00	1.11E+02	4.17E+03
5-task set	TTC				TTH			
	TTSA1	TTSA2	BaB1	BaB2	TTSA1	TTSA2	BaB1	BaB2
Minimum number of trials	4.00E+00	4.00E+00	4.00E+00	4.00E+00	4.00E+00	4.00E+00	4.00E+00	4.00E+00
Maximum number of trials	1.40E+02	3.14E+02	1.03E+06	1.21E+09	1.02E+02	2.26E+02	1.33E+06	1.98E+08
Average number of trials	4.04E+01	8.58E+01	8.84E+04	2.04E+07	2.93E+01	6.34E+01	5.41E+04	5.09E+06
Total number of trials	4.04E+04	8.58E+04	8.84E+07	2.04E+10	2.93E+04	6.34E+04	5.41E+07	5.09E+09
Total time (s)	2.50E+00	6.00E+00	3.69E+03	2.94E+06	2.50E+00	3.50E+00	5.26E+03	3.79E+05

- The criteria used for adding the tasks to the TTSA1 and TTSA2 have an impact on the schedulability of the set (different criteria may give different results).
- Combining results from the variations of TTSA1 and variations of TTSA2 together gives results which are very close to those obtained from the BaB1 and BaB2 respectively while requiring a much lower number of trials, and hence less time (see **Table 3**).

4.2.5 Results (large task set):

The results shown in Fig. 2. to Fig. 4. 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 TTSA2 on larger problems, 1000 new data sets were created. Each data set consisted of 50 tasks, each with a maximum execution time of 2 ms and maximum period of 200 ms. The task sets were (pseudo) randomly created according to the constraints described previously. To reduce the length of the major cycle, task periods were (pseudo) randomly generated as a multiple of 20 ms.

The results from this test are shown in Fig. 5. It took approximately 1 minute to complete the schedulability test for one set of 50 tasks using TTSA2-EDF, and a total of approximately 6 minutes to complete the test for TTSA2-All. It was not possible to complete this search using a BaB approach as this would have required performing a huge number of trials.

Fig. 2. Number of Scheduled task sets (3 interdependent tasks in each set).

Fig. 4. Number of Scheduled task sets (5 interdependent tasks in each set).

Fig. 3. Number of Scheduled task sets (4 interdependent tasks in each set).

Fig. 5. Number of Scheduled task sets (50 interdependent tasks in each set).

5 Discussion and conclusions

It has been previously demonstrated that using offline, or pre-runtime, scheduling helps in reducing the complexity of inspecting and verifying the timing behaviour of safety critical embedded systems ([6], [7]).

In this paper we introduced a new offline scheduling algorithm, TTSA2, which helps to 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. The TTSA2 algorithm tries to find a suitable scheduler for the set

of tasks by dividing each task into two or more segments. While searching for a workable scheduler the proposed scheduling algorithm ensures that the CPU power consumption is “as low as possible” (by choosing the longest possible tick interval), and the task constraints as well as individual segment constraints are met (by adjusting the segment offsets, tick interval, and task orders), using the lowest number of segments. If the tasks cannot all be scheduled (for example, if some timing constraints of one or more tasks cannot be met), a list of the schedulable/unschedulable tasks is generated. The algorithm improves on the performance of both a BaB search and a previous version of this algorithm (TTSA1).

The algorithm can be used as part of a tool for automatic code generation for safety-critical, resource-constrained embedded systems. Using such a tool will not only reduce the time and effort required to develop such systems but will reduce the probability of scheduling errors, which may cause serious damage (see [31] for an example), as well.

In the current work we assume that a task can be divided into two or more segments only at points of time (the critical segment boundaries). Future work needs to be done to determine more efficient way to find the points of time at which task can be divided. In addition, other methods are needed to explore ways for choosing the pre-empting task in the TTH scheduler, in order to improve the results.

References

1. Domaratsky, Y., Perevozchikov, M.: Highly dependable time-triggered operating system-static scheduling approach and effective run-time implementations, *Dedicated Systems Magazine*, 77 - 84 (Oct.-Dec. 2000)
2. Kopetz, H.: *Real-Time Systems, Design Principles for Distributed Embedded Applications*, Kluwer Academic (1997).
3. Pont, M. J.: *Patterns For Time-Triggered Embedded Systems*, Addison-Wesley (2001)
4. Scheler, F., Schröder-Preikschat, W.: Time-Triggered vs. Event-Triggered: A matter of configuration?, In: Dulz, Winfried ; Schröder-Preikschat, Wolfgang : *MMB Workshop Proceedings (GI/ITG Workshop on Non-Functional Properties of Embedded Systems Nuremberg, Berlin : VDE Verlag , 107 - 112, ISBN 978-3-8007-2956-2 (2006)*
5. Albert, A.: Comparison of Event-Triggered and Time-Triggered Concepts with Regard to Distributed Control Systems, *Embedded World 2004, Nuremberg, WEKA Verlag*, 235 - 252 (2004)
6. Xu, J.: Making Software Timing Properties Easier to Inspect and Verify, *IEEE Software*, vol. 20, n. 4, 34 - 41 (July/August, 2003)
7. Xu, J., Parnas, D. L.: Priority Scheduling Versus Pre-Run-Time Scheduling, *Int. Journal of Time-Critical Systems*, Kluwer Academic Publishers, vol. 18, 7 - 23, (2000)
8. Baker, T. P., Shaw, A.: The Cyclic Executive Model and Ada, *Real-Time Systems*, vol. 1, vo. 1, 7 - 25 (1989)
9. Tindell, K., Burns, A., Wellings, A.: Allocating Hard Real-Time Tasks: An NP-Hard Problem Made Easy, *Real-Time Systems*, vol. 4, no. 2, 145 – 165 (1992)
10. Baruah, S. K.: The Non-Preemptive Scheduling of Periodic Tasks Upon Multiprocessors, *Real-Time Systems*, vol. 32, no.1-2, 9 - 20 (2006)
11. Gendy, A.K. and Pont, M.J.: Automatically Configuring Time-Triggered Schedulers for Use with Resource-Constrained, Single-Processor Embedded Systems, *IEEE Trans. on Industrial Informatics*, vol. 4, no.1, 37 - 46 (2008)

12. Gangoiti, U., Marcos, M., Estévez, E.: 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, 4785-3790 (2005)
13. Buttazzo, G. C.: Rate monotonic vs. EDF: Judgement day", Real-Time Systems, vol. 29, no. 1, 5 - 26 (2005)
14. Huang, C., Chang, L., Kuo, T.: A Cyclic-Executive-Based QoS Guarantee over USB, IEEE 9th Real-Time and Embedded Technology and Applications Symposium, Toronto, Canada, 88-95 (May 27-30, 2003)
15. Burns, A.: Generating Feasible Cyclic Schedules, Control Engineering Practice, vol. 3, no. 2, 151 - 162 (1995)
16. Locke, C. D.: Software Architecture for Hard Real-Time Applications: Cyclic Executives Vs. Fixed Priority Executives, Real-Time Systems, vol. 4, no. 1, 37 - 52 (1992)
17. Phatrapornnant, T., Pont, M. J.: 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, 113 - 124 (2006)
18. Allworth, S. T.: An Introduction to Real-Time Software Design, London, Macmillan (1981)
19. Maaita, A., Pont, M. J.: Using 'Planned Pre-Emption' to Reduce Levels of Task Jitter in a Time-Triggered Hybrid Scheduler, UK Embedded Forum, Birmingham, UK, University of Newcastle (2005)
20. Kalinsky, D.: Context Switch, Embedded Systems Programming, vol. 14, no. 1, 94 - 105 (2001)
21. Schlindwein, F. S. Smith, M. J., Evans, D. H.: 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, 228 - 232 (1988)
22. Liu, C. L., Layland, J. W.: Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment, Journal of the ACM, vol. 20, no. 1, 40-61 (1973)
23. Gendy, A.K. and Pont, M.J.: Towards a Generic "Single-Path Programming" Solution with Reduced Power Consumption, Proceedings of the ASME 2007 International Design Engineering Technical Conferences & Computers and Information in Engineering Conference (IDETC/CIE 2007), Las Vegas, Nevada, USA (September 4-7, 2007)
24. Puschner, P. and Burns, A.: Writing Temporally Predictable Code, In Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, 85 - 91 (Jan. 2002)
25. Puschner, P.: The Single-Path Approach Towards WCET-Analysable Software, In Proc. IEEE International Conference on Industrial Technology, 699 - 704 (Dec. 2003)
26. Cucu, L., Sorel, Y.: 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)
27. Buttazzo, G.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Kluwer Academic (1997)
28. Xu, J., Parnas, D. L.: Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations, IEEE Transactions on Software Engineering, vol. 16, no. 3, 360 - 369 (1990)
29. Sandström, K., Norström, C.: Managing Complex Temporal Requirements in Real-Time Control Systems, 9th IEEE Conf. Engineering of Computer-Based Systems, IEEE, Sweden (2002)
30. Ekelin, C., Jonsson, J.: Evaluation of Search Heuristics for Embedded System Scheduling Problems, in Proc. Int. Conf. Principles and Practice of Constraint Programming, Paphos, Cyprus, 640 - 654 (2001)
31. Reeves, G.: What Really Happened on Mars?, - Authoritative Account, research.microsoft.com, (1997)