DEVELOPING SCHEDULER TEST CASES TO VERIFY SCHEDULER IMPLEMENTATIONS IN TIME-TRIGGERED EMBEDDED SYSTEMS

Mouaaz Nahas[1] and Ricardo Bautista-Quintero[2]

[1] Department of Electrical Engineering, College of Engineering and Islamic Architecture, Umm Al-Qura University, Makkah, KSA
mmnahas@uqu.edu.sa

[2] Department of Mechanical Engineering, Instituto Tecnólogico De Culiacán, Sinaloa, México
r.bautista@unb.ca

ABSTRACT

Despite that there is a "one-to-many" mapping between scheduling algorithms and scheduler implementations, only a few studies have discussed the challenges and consequences of translating between these two system models. There has been an argument that a wide gap exists between scheduling theory and scheduling implementation in practical systems, where such a gap must be bridged to obtain an effective validation of embedded systems. In this paper, we introduce a technique called "Scheduler Test Case" (STC) aimed at bridging the gap between scheduling algorithms and scheduler implementations in single-processor embedded systems implemented using Time-Triggered Co-operative (TTC) architectures. We will demonstrate how the STC technique can provide a simple and systematic way for documenting, verifying (testing) and comparing various TTC scheduler implementations on particular hardware. However, STC is a generic technique that provides a black-box tool for assessing and predicting the behaviour of representative implementation sets of any real-time scheduling algorithm.

KEYWORDS

Scheduler algorithm, scheduler implementation, cyclic executive, time-triggered co-operative scheduler, resource-constrained embedded system, scheduler test cases, predictability, jitter, task overrun.

1. INTRODUCTION

There are numerous ways in which we can describe (and distinguish) the different architectures employed in embedded computer systems. The architecture which forms the focus of this paper is usually described as "time triggered" (TT); as opposed to "even-triggered" (ET) [1]. For an embedded system with a time-triggered architecture, we are supposed to know in advance how the system will behave exactly at every instance during its running time. Knowing the complete behaviour of the system and hence determining whether the system is capable of meeting all its timing constraints can be referred to as "predictability" [2], [3].

Such an ideal TT behaviour is inevitably hard to achieve in real world. However, approximations of this model have been developed and widely used in practice. The best approximation of an ideal TT architecture contains a set of periodic tasks running in a co-operative (or simply "non-pre-emptive") manner. Such a design is referred to as "time-triggered co-operative" (TTC) architecture [4]–[8]. Early versions of this architecture used to be called a "cyclic executive" [9]–[11]. Unlike time-triggered pre-emptive algorithms (e.g. "rate monotonic"), systems with TTC architectures have high predictability in their timing behaviour, i.e. they demonstrate very low levels of task jitter [10] and have the ability to maintain such low-jitter characteristics even when modern techniques are integrated to improve system performance or minimise power consumption (e.g. “dynamic voltage scaling” (DVS) technique [12]).

The principal aim of this paper is to ensure that accurate timing predictions made at the design phase are maintained during the process of implementing (and also maintaining) a practical system. It has been argued that there is a “one-to-many” mapping between scheduling algorithms and scheduler implementations ([9], [13]–[15]). However, the process of transforming between these two system representations has not received widespread attention. In an effort to facilitate a systematic approach for validating an implementation of TT system, we introduce an empirical approach which we call a “scheduler test case” (STC). Such a technique is not intended to test all features of the system (by any means): instead, it is intended to be used with “dummy tasks” solely as a practical means for assessing the scheduler implementation employed in a given system. In this way, we aim to allow those implementing the system to gain a better understanding (during system construction, testing or maintenance) of the way in which a given TTC implementation is expected to behave under a range of normal and abnormal operating conditions. Our objective is that the developed STC method can provide a “black box” evaluation of a given scheduler implementation without the necessity to access (or even comprehend) the underlying source code by the system developer (or user).

Predictability is an important design parameter in real-time resource-constrained embedded systems, particularly in those employing TTC architectures. Thus, it is used here as the essential criteria for assessing the behaviour of a scheduler. To be able to express predictability using qualitative and quantitative measures, the following three criteria are considered: (1) task execution sequence; (2) timing jitter; and (3) the capability of the scheduler to handle unexpected errors. In addition, computational, memory and power overheads resulted from the implementation of each scheduler are used to allow a practical comparison between the various studied schedulers.

The remainder of the paper is organised as follows. In Section 2, we provide a detailed literature review of the work conducted in the same field of this study. In Section 3, we introduce what we call “Scheduler Test Cases” (STCs). In Section 4, we demonstrate how the STC technique is employed in TTC algorithm. In Section 5, the methodology used to attain the empirical results of this study is described. In Section 6, we report the results obtained when the STC is applied to a set of representative TTC scheduler implementations collected from the literature. Finally, we present our conclusions in Section 7.

2. LITERATURE REVIEW

In this section, a detailed literature review of the work carried out in this area is provided.

2.1. Automated code generation

One way to bridge the gap between scheduler algorithms and scheduler implementation is by means of automated code generation. Such techniques help to reduce the time and effort consumed in the implementation process of safety-critical systems, while removing errors that are likely to arise during this stage of development [16], [17]. Various industries – such as aerospace and automotive – have extensively used automatic code generation tools for control and signal processing systems [18]–[20]. TTC systems considered in this study are typically used in such application arenas. It is believed that hundreds of thousands of modern cars rely on codes generated using automatic code generation [18], [21].

Previous work on the “automatic” generation of systems using TTC architecture are presented in [22], [23]. This kind of work helps the developer to utilise a collection of “design patterns” for creating a complete code (with the underlying scheduler) of a TTC system. 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). Incorrect tuning of these parameters may result – at worst – in unschedulability of tasks. However, even if all tasks are ensured to be scheduled, incorrect decisions may still cause degradation to the system performance (e.g. introducing high levels of task jitter) or an increase in the implementation costs (e.g. increasing CPU overhead or system power consumption). To conclude, automated code generation techniques – despite their important role in validating embedded software – do not take into account the possible runtime behaviour that can be resulted from a particular implementation of software code. For example, no feedback process is incorporated in such techniques which allows the designer to examine the consequence of using a particular code on the runtime behaviour of their system and hence modify the source code accordingly.

2.2. Scheduler implementation

It is noticed that a great deal of work has been carried out in developing, assessing and modifying (enhancing) scheduling algorithms. Despite that, only a limited amount of work has been found on the process of scheduler implementation and its vulnerability to the decisions made at early stages of the development lifecycle (namely, during scheduler design). The relationship between any scheduling algorithm and its possible implementation options is usually viewed as "one-to-many" [4], [9], [14], [24]–[26], which simply means that the output behaviour of the system would entirely depend on the implementation decisions. To elaborate more on this point, if the source code of a scheduler is implemented using an appropriate collection of "software design patterns" which is a common practice [4], the relationship between a design pattern and its implementation is typically described as "one pattern, many implementations" [27]. This would inevitably cause the scheduling algorithm to have a wide range of scheduler implementations; each corresponds to a different pattern. Figure 1 shows the implementation of a simple TTC algorithm based on a set of different software design patterns, where each pattern represents a different scheduler implementation (e.g. TTC-1, TTC-2, etc.) and uses various Pattern Implementation Examples (PIEs) corresponding to different processor platforms (e.g. 8-bit 8051, 16-bit c167 or 32-bit ARM microprocessor). This figure helps to explain the concept behind the "one-to-many" relationship between a particular scheduling algorithm and its practical implementations in real-time, resource-constrained embedded systems (see [27] for more details).

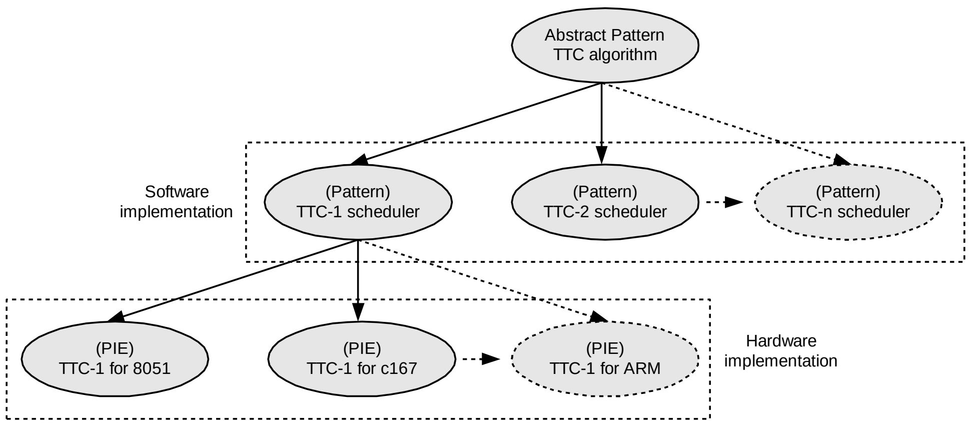

Figure 1. The "one-to-many" mapping between the TTC scheduler and its practical implementations using software design patterns (This figure is reproduced from [27]).

Scheduler implementation is a key problem which faces the developers of real-time embedded systems. To understand this, it is necessary to make a clear distinction between the two terms "scheduling" and "scheduler implementation". The former term describes the process in which the optimum schedule for a set of real-time tasks is determined. In contrast, the latter term describes the process in which the scheduler is physically implemented – either in software or hardware – to execute the designed task-schedule at the system operating time [28]. An early work considering the implementation of TTC architectures using the Ada programming language was conducted in [9]. Later, the work in [29] considered the issues of implementing forms of cyclic executive (i.e. a simple TTC algorithm) in Assembly language. The work outlined in [12] developed techniques to maintain low task jitter when dynamic voltage scaling is applied in TTC algorithm to reduce power consumption of the system. In our previous work [30], [31], we developed various low-jitter TTC schedulers aimed at highly-predictable embedded applications. In [6], [32], [33], we developed and evaluated low-jitter TTC schedulers for multi-processor distributed real-time embedded systems built using Controller Area Network (CAN) protocol. Various ways in which TTC algorithm is implemented in multi-processor embedded systems were described in detail in [34]. The potential impact of scheduler implementation on task jitter for TTC algorithm in both single- and multi-processor designs was studied in detail in [15]. In [35], [36], TTC schedulers with "task guardian" mechanisms were introduced to address the task overrunning problem.

In trying to link scheduling algorithm and scheduler implementation, Katcher et al. [13] argued that a wide gap exists between the theory of scheduling and its practical implementation in operating system cores running on specific hardware platforms. Moreover, they asserted that bridging such a gap is imperative if a meaningful validation of real-time systems is sought to be achieved. The same argument was made by other researchers (e.g. [37], [38]). For example, [13] stated that the implementation of a particular scheduler may introduce costs that must not be overlooked if the timing behaviour of a real-time system is to be validated.

2.3. Software verification and testing

Since our focus is to link scheduling algorithms and scheduler implementations, we should think of a way to ensure a complete matchup (i.e. compliance) between the design and implementation processes of the scheduler (with a particular focus on software implementation). Software engineers refer to this process as "software verification". We have found evidence that confusion exists in using the terms "validation" and "verification" by many people concerned with the development and evaluation of software applications. For example, some people think that "validation" is a synonym to "verification" and vice versa [39], and hence use the two terms interchangeably. Based on a wide range of definitions collected from various dictionaries, a system is considered to be validated (or valid) if its final software product fulfils the requirements of the user (or the customer who pays for it). Therefore, any process involved in testing such a fulfilment is called "validation process". In contrast, verification requires checking each component in the system to ensure that its output matches the specifications of its input and hence maintains the integrity of the user's initial requirements [2]. In [40], it is noted that *"validation usually takes place at the end of the development cycle, and looks at the complete system as opposed to verification, which focuses on smaller sub-systems".* Figure 2 shows one possible way in which we attempt to integrate validation and verification processes in the general model of software development life cycle as illustrated by [39].

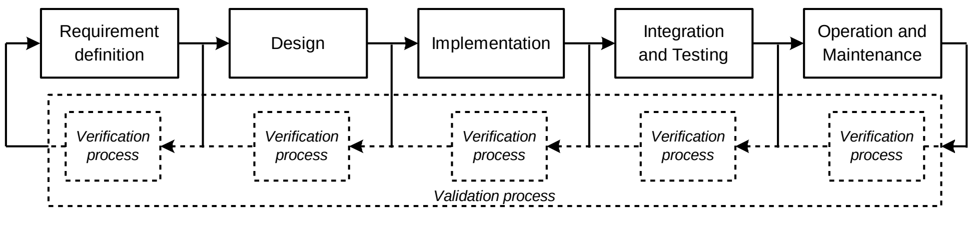

Figure 2. Inserting validation and verification processes in the software development life cycle

For system verification, there are two main approaches: “static” and “dynamic”. Static approach involves *software inspections* and *formal methods*, while dynamic one mainly involves *software testing* [39]. Among these techniques, software testing scores the best in verifying the implementation of a given real-time scheduler, since it allows testing dynamic features which only manifest when the system is operational (this is an important requirement in real-time system where correct timing of results is crucial). A detailed comparison between the various verification techniques is provided in [2]. For any testing process, an appropriate set of test cases has to be designed. In fact, only an effective subset of possible test cases is used. A general model for software testing process is illustrated in Figure 3. The figure reveals the basic elements of any testing process which are: (1) test cases; (2) test data (i.e. test input); and (3) test output. The test output has to be compared with the output estimated at the design process of the test case by developers who possess a complete knowledge about the system and its expected operation at real-time.

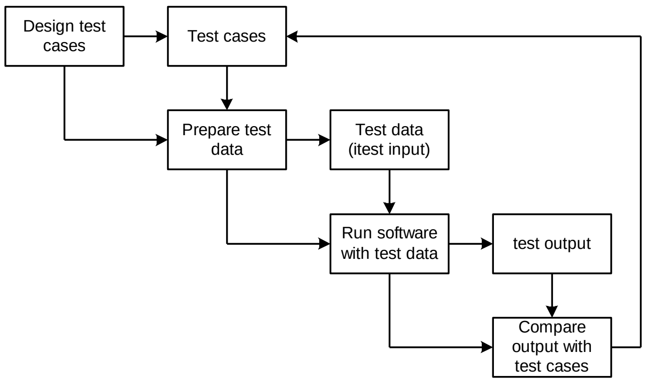

Figure 3: Model of testing process (reproduced from [39])

2.4. Test cases and test-case generation

Many researchers have argued that testing may consume almost half of the total development cost [39], [41]–[46]. Lots of studies have therefore demonstrated that test process automation holds the promise to reduce time, effort and costs (e.g. [47]–[49]). As mentioned in the introduction, the focus in our paper is on a form of test cases.

There has been a great deal of previous work on both test cases and test-case generation. For example, Beck’s work on “extreme programming” has at heart a view that test cases for the system should be produced early in the product life cycle [50]. In [51], it is argued that the most challenging phase in the test process is the selection and execution of test cases. Many studies have, therefore, proposed techniques for automatic test-case generation (see for instance, [43], [47], [49], [51]). Test cases may be created directly from the source code, or other sources such as control flow graphs, design representations and specifications [52]. Many studies have considered the process of generating test cases for real-time computing systems (e.g. [47], [51], [53]–[57]).

In this section, only a small collection of literature has been reviewed to show the interest of researchers in this area. In effect, previous work on testing considered the examination of software functionality or software quality attributes (features). The authors of this paper are not aware of any extensive work which considered the development of test cases solely to study the impact of using a particular source code (i.e. scheduler software) on the output behaviour of the system executing the code, especially when algorithms such as TTC schedulers are implemented.

3. SCHEDULER TEST CASE (STC) TECHNIQUE

As noted before, testing here is not aiming to check the correct functionality of the application software or evaluate its quality attributes. Instead, it is mainly used to evaluate the runtime behaviour of the system as a result of employing a particular software implementation of TTC scheduler using commonly-used processor platforms. As in all testing processes, a suitable collection of test cases are required. Such test cases typically determine the system inputs, predicted outputs, and execution conditions, and aimed to verify the system conformity with specific requirements. Again, note that only a selective subset of possible test cases can be used since inclusive testing is not possible. The feature of the system to be tested must be selected along with the inputs that will execute that feature, and the expected outputs of the test cases must be known in advance. All abovementioned test case components have been integrated in the process of developing the STCs presented here. This is further described as follows.

The STC is a simple technique which employs a collection of test cases to examine the output behaviour of a wide range of TTC scheduler implementations. The STCs developed here are created manually depending on previous experience and full comprehension of the characteristics and constraints of the TTC algorithm [4]. The STC technique employs various scheduling examples using non-real (dummy) tasks where those examples show a different set of behaviour patterns as the scheduler implementation varies. Such task sets represent the test inputs (i.e. test data). Each STC contains a different collection of tasks with different properties. The test item here is the TTC scheduler to which the tasks in each STCs are entered. The scheduler will then execute on the target microcontroller hardware for a sufficient period of time during which the system response is monitored. Then, the output behaviour will be recorded and compared to the predicted behaviour (which was fully documented at the test case design stage). Figure 4 shows the detailed process of STC testing developed in this study.

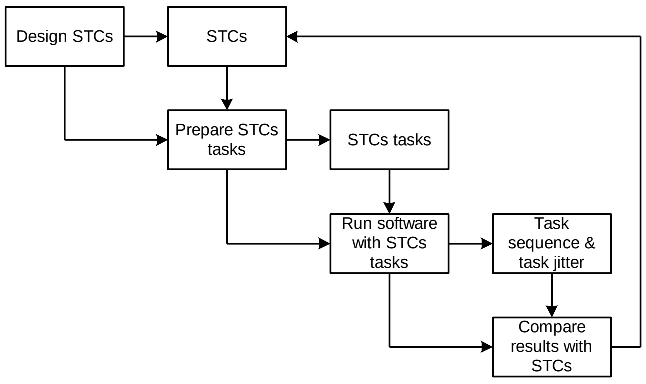
Design STCs
STCs
Prepare STCs tasks
STCs tasks
Run software with STCs tasks
Task sequence & task jitter
Compare results with STCs

Figure 4. Testing process model for the STC technique (this is extracted from the process illustrated in Figure 3)

The STC technique tests the system performance under normal and abnormal operating conditions. By normal conditions, we mean that the scheduler operates in non-existence of errors. In contrast, abnormal conditions refer to the situation where errors occur. The main criterion to define an error mode is that it should represent a recognised problem facing the developers of the system under verification. For example, TTC systems (which form the key focus of this study) suffers from “task overrunning” problem which can deteriorate the overall system performance if not addressed immediately (sometimes, it can spoil the system’s functionality) [35], [36]. Accordingly, task overrunning problem has been selected to represent the error mode in TTC schedulers.

As noted above, the key criteria used to assess the behaviour of each TTC scheduler are: task sequencing, jitter levels and the ability to address task overrunning problem. You can look at these criteria as the main tested features of the TTC scheduling algorithm. Such criteria are used to reflect the level of predictability in the TTC algorithm. The task sequencing criterion checks if the scheduler runs tasks in the required order as determined in the schedule design process. Jitter in the task timing is used to assess the timing performance of the system. In our study, the jitter is measured at the release time of all tasks running in the TTC scheduler. Release jitter describes the deviation of the start time of a task from its release time. It can easily be shown that – in many real-time embedded system designs – reducing jitter on all (or some) tasks has the potential to increase the overall system's predictability.

4. THE SCHEDULER TEST CASES (STCS) FOR TTC ALGORITHM

In this section we present the various "scheduler test cases" (STCs) developed in this study to analyse and test the behaviour of different implementation classes of the TTC scheduler.

4.1 STC A (Task-induced jitter)

STC A investigates the impact of variations in the task execution time on the jitter levels. Figure 5 illustrates the way in which a particular TTC implementation might be expected to execute the set of tasks which we consider in STC A. In this figure (and others in this paper) the vertical arrows represent the points at which periodic timer interrupts (or "ticks") occur. The interval between successive timer interrupts is known as "tick interval". In Figure 5, all tasks execute with a "tick offset" value of 0. This simply means that each task runs for the first time in tick interval number zero.

Task characteristics for this STC are detailed in Table 1. Considering STC A in more detail, the figure illustrates how Task B (and Task C) may suffer from release jitter (since the execution duration of Task A varies from one tick to another).

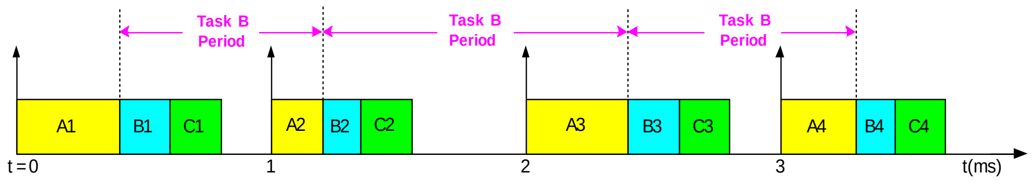

Figure 5. Graphical representation of Example schedule A1

Table 1: Task set for STC A (Major cycle = 1 Tick).

Task Name	**Period (Ticks)**	**Offset (Ticks)**	**Priority (1 = High)**	**ET**[1]	**Allowable jitter in start time of task**
A	1	0	1	ET(A) – variable (0.01 – 0.4 Ticks)	Low
B	1	0	2	ET(B) – variable (0.01 – 0.2 Ticks)	Low
C	1	0	3	ET(C) – variable (0.01 – 0.2 Ticks)	High

[1] ET denotes the actual execution time of a task on a given run (this figure will vary between runs in most cases).

Examples of possible schedules obtained with this task set are given in Table 2 and Table 3.

Table 2: Example schedule A1.

	Start time (after Tick)	**Jitter (Ticks)**
Ax	0	Low (related to Tick jitter & scheduler overhead)
Bx	ET(Ax)	Potentially high (varies with ET of previous task)
Cx	ET(Ax) + ET(Bx)	Potentially high (varies with ET of previous tasks)

Comment:
In a simple scheduler implementation, it is expected to see high levels of jitter in the start times of tasks executed later in the tick interval, if the execution time of the earlier tasks is variable. This is illustrated in Figure 5. Note that such a scheduler implementation is not suitable for use with jitter-sensitive tasks.

Table 3: Example schedule A2.

	Start time (after tick)	**Jitter (Ticks)**
Ax	0	Low (may be related to scheduler overhead)
Bx	WCET[2](A)	Low (may be related to scheduler overhead)
Cx	WCET(Ax) + WCET(Bx)	Low (may be related to scheduler overhead)

Comment:
In a scheduler with low-jitter characteristics, the scheduler can compensate for variations in the execution time of tasks. Thus, tasks with lower priorities will not suffer from high jitter at their release times. This is illustrated in Figure 6.

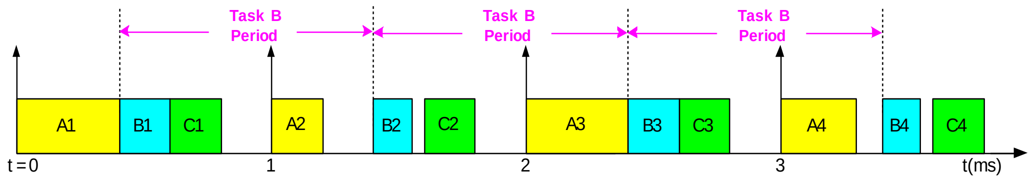

Figure 6. Graphical representation of Example schedule A2

4.2 STC B (Schedule-induced jitter)

Unlike STC A, STC B investigates the impact of variations in the schedule on the jitter levels of tasks. A summary of the task characteristics for this STC is presented in Table 4.

Table 4: Task set for STC B (Major cycle = 2 Ticks).

Task Name	**Period (Ticks)**	**Offset (Ticks)**	**Priority (1 = High)**	**ET**	**Allowable jitter in start time of task**
A	2	0	1	ET(A) – variable (0.01 – 0.4 Ticks)	Low
B	1	0	2	ET(B) – variable (0.01 – 0.2 Ticks)	Low
C	1	0	3	ET(C) – variable (0.01 – 0.2 Ticks)	High

[2] WCET denotes the worst-case (longest) execution time of a task during the period in which it was observed. Estimating the WCET of a program (or function) is a challenging problem that has been under investigation for many years. For more details about the available techniques, refer to [58].

Examples of possible schedules obtained with this task set are given in Table 5 and Table 6.

Table 5: Example schedule B1 (Basic scheduler).

	Start time (after tick)	**Jitter (Ticks)**
Ax	0	Low (related to Tick jitter & scheduler overhead)
Bx	0 or ET(Ax)	High (start time of task varies on alternate Ticks)
Cx	ET(Bx) or ET(Ax) + ET(Bx)	High (start time of task varies on alternate Ticks)

Comment:
Here, the release jitter in Task B is expected to be high as it executes immediately after Task A, where Task A has a variable duration time. This is shown in Figure 7.

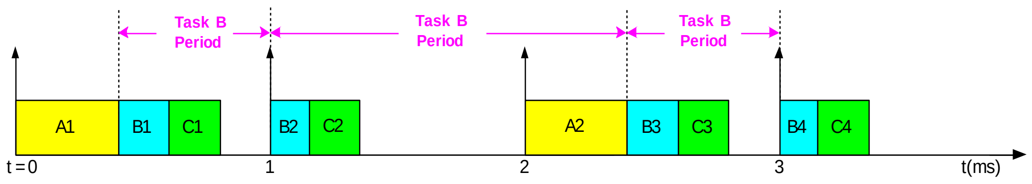

Figure 7. Graphical representation of Example schedule B1

Table 6: Example schedule B2 (TTC scheduler with gap insertion).

	Start time (after tick)	**Jitter (Ticks)**
Ax	0	Low (related to Tick jitter & scheduler overhead)
Bx	WCET(Ax)	Low (if WCET estimates are accurate)
Cx	WCET(Ax) + WCET (B)	Low (if WCET estimates are accurate)

Comment:
This scheduler implementation fulfils the jitter requirements for Task B and Task C by compensating for the variation in the schedule itself (see Figure 8).

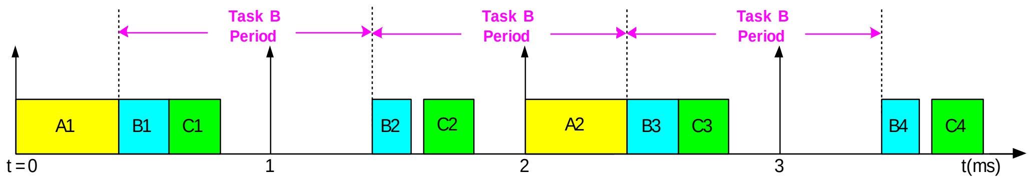

Figure 8. Graphical representation of Example schedule B2

4.3 STC C (Long tasks)

TTC scheduler is typically based on the use of timer interrupt that overflows periodically to generate tick intervals. In STC A and STC B, we assume that all tasks which begin execution in a given tick interval are intended to complete their execution before the next tick occurs. Such a restriction is not a fundamental requirement in TTC systems[3], but may cause a limitation in some TTC scheduler implementations. Therefore, we test the scheduler ability to handle "long tasks" in

[3] A TTC design is co-operative in nature. So task pre-emption is not allowed at all. In case of "long tasks" whose execution time is longer than the tick interval, the task is interrupted by the scheduler (at the arrival of the next tick) but not by another task.

STC C. A summary of the task characteristics for this test is presented in Table 7. Examples of possible schedules obtained with this task set are given in Table 8 to Table 12.

Table 7: Task set for STC C (Major cycle = 4 Ticks).

Task Name	**Period (Ticks)**	**Offset (Ticks)**	**Priority (1 = High)**	**ET**	**Allowable jitter in start time of task**
A	2	1	1	ET(A) – fixed (0.2 Ticks)	Low
B	4	0	2	ET(B) – fixed (2.4 Ticks)	Low
C	2	1	3	ET(C) – fixed (0.2 Ticks)	High

Comment

In this sequence, Task B runs for 2.4 ticks. While executing Task B, Task A (which has a low-jitter requirement) becomes ready to execute. With STC C, we can check how the scheduler will deal with tasks which are deliberately set to run beyond the tick interval. This test also checks how the scheduler will manage the task priorities; for example, Task A has a higher priority than Task C so once Task B finishes executing, Task A should be executed prior to Task C.

Table 8: Example schedule C1 (Basic scheduler).

	Start time (after tick)	**Jitter (Ticks)**
Ax	0 or 0.4 Ticks	High (start time of task varies on alternate Ticks)
Bx	0	Low (related to Tick jitter & scheduler overhead)
Cx	ET(Ax) or ET(Ax) + 0.4 Ticks	High (start time of task varies on alternate Ticks)

Comment

In a basic TTC implementation, all delayed tasks will execute in their predefined order upon the completion of the long task; that is Task B in our case (Figure 9).

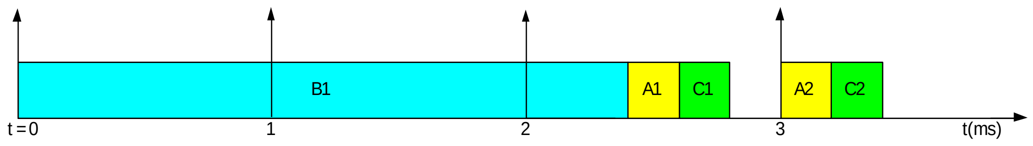

Figure 9. Graphical representation of Example schedule C1

Table 9: Example schedule C2.

	Start time (after tick)	**Jitter (Ticks)**
Ax	0 or ET(Cx) + 0.4 Ticks	High (start time of task varies on alternate Ticks)
Bx	0	Low (related to Tick jitter & scheduler overhead)
Cx	ET(Ax) or 0.4 Ticks	High (start time of task varies on alternate Ticks)

Comment

In many TTC implementations, the schedule checks the status of the task next to the long one in the task sequence. If it is due to run, the schedule will execute it regardless of whether higher priority tasks are waiting to execute or not. In the example shown in Figure 10, Task C will be executed immediately after Task B and before Task A (which has the highest priority in the whole task list).

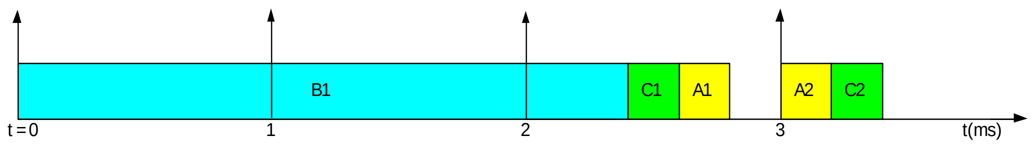

Figure 10. Graphical representation of Example schedule C2

Table 10: Example schedule C3.

	Start time (after tick)	**Jitter (Ticks)**
Ax	0	Low (related to Tick jitter & scheduler overhead)
Bx	0	Low (related to Tick jitter & scheduler overhead)
Cx	ET(Ax)	Low (since ET(Ax) is fixed)

Comment

In each major cycle, the first execution of both Task A and Task B is omitted (see Figure 11).

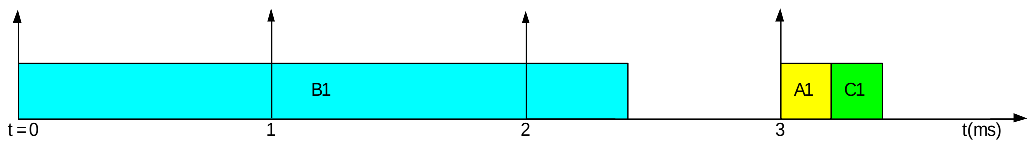

Figure 11. Graphical representation of Example schedule C3

Table 11: Example schedule C4.

	Start time (after tick)	**Jitter (Ticks)**
Ax	0 or ET(Ax)	High (task runs twice in the same Tick at different start times)
Bx	0	Low (related to Tick jitter & scheduler overhead)
Cx	0.4 Tick or 2ET(Ax)	High (start time of task varies on alternate Ticks)

Comment

Here, Task A will be first executed in the last tick of the whole cycle, as depicted in Figure 12.

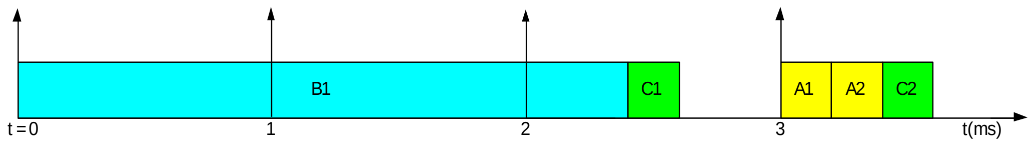

Figure 12. Graphical representation of Example schedule C4

Table 12: Example schedule C5.

	Start time (after tick)	**Jitter (Ticks)**
Ax	0	Low (related to Tick jitter & scheduler overhead)
Bx	0	Low (related to Tick jitter & scheduler overhead)
Cx	ET(Ax)	Low (since ET(Ax) is fixed)

Comment

The scheduler enforces any long task to shut down as soon as the next tick arrives (Figure 13).

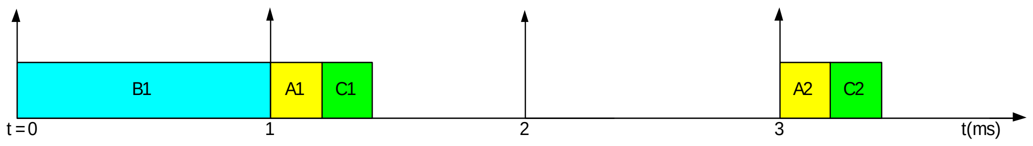

Figure 13. Graphical representation of Example schedule C5

4.4 STC D (Task overruns)

All previous STCs assume that the system operates normally without errors. STC D is specifically designed to investigate the impact of unplanned task overruns. Task characteristics for this test are presented in Table 13. Examples of possible schedules obtained with this task set are given in Table 14.

Table 13: Task set for STC D (Major cycle = 20 Ticks).

Task Name	Period (Ticks)	Offset (Ticks)	Priority (1 = High)	ET	Overrun duration (in Ticks)
A	20	0	1	ET(A) – fixed (0.2 Ticks)	10
B	1	0	2	ET(B) – fixed (0.2 Ticks)	0

Comment

In this sequence, Task A is designed to run for the duration of 0.2 Tick. Due to an error, Task A overruns by 10 Ticks. This is illustrated in Figure 14.

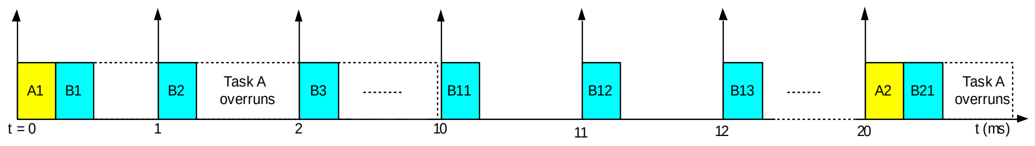

Figure 14. Summary of STC D (Task A overruns by 10 ticks)

Table 14: Example schedule D1a, D1b, D2a, D2b, D3a and D3b.

Schedule Name	Shut down time (from Ticks)	Backup task	Comment
D1a	---	Not applicable	Overrunning task is not shut down, and the number of elapsed ticks - during overrun - is not counted, hence tasks due to run in these ticks are ignored completely.
D1b	---	Not applicable	Overrunning task is not shut down, but the number of elapsed ticks - during overrun - is counted. The due tasks in these ticks are run immediately after overrun ends.
D2a	1 Tick	Not available	Overrunning task is detected at the arrival of the next tick and hence shut down.
D2b	1 Tick	Available – BK(A)	Overrunning task is detected at the arrival of the next tick and hence shut down. Moreover, a backup task is inserted in the schedule to replace the overrunning task.
D3a	WCET(Ax)	Not available	Overrunning task is shut down immediately after it exceeds its estimated WCET.
D3b	WCET(Ax)	Available – BK(A)	Overrunning task is shut down immediately after it exceeds its estimated WCET. A backup task is inserted in the schedule to replace the overrunning task.

4.5 CPU, memory and power overheads

This paper is concerned with the type of embedded systems that can be built on “commercial off-the-shelf” (COTS) microcontrollers. Such small and cheaply-available hardware usually have limited resources (i.e. CPU, memory and power). We will therefore report CPU, memory and power requirements (overheads) for all schedulers considered in this paper. Note that in many embedded system applications, average power consumption is a key concern, as this is related to the system battery lifetime.

5. METHODOLOGY

In this section, we provide an overview of the methodology used to obtain the empirical results presented in this paper.

5.1 Representative examples of TTC implementations

In fact, it is impossible to cover all possible implementation options for a simple TTC scheduler in a single study. Hence, we apply the proposed STC technique on a range of TTC schedulers which form representative examples of the broad range of TTC implementations developed previously. The selected implementations are all based on the use of periodic timer interrupts[4] and have been gathered from a wide range of different projects carried out in the Embedded Systems Laboratory (ESL) at the University of Leicester, UK. Such TTC implementations are: TTC-ISR [4], TTC-Dispatch [4], [32], TTC-DVS [12], TTC-TG [36], TTC-ISR [30] and TTC-Adaptive [60] schedulers. For further details about these schedulers, please refer to the references listed in this section.

5.2 Hardware platform

As noted before, the work conducted here is based on the use of low-cost embedded systems based on small microcontroller hardware such as: 8051, Infineon C16x, Philips LPC2xxx, or PH Processor. For programming the hardware (i.e. writing the software program including the scheduler code), the C programming language has been found to be competitive among many other surveyed languages (see [61] for more details).

The empirical studies presented here were carried out using Ashling LPC2000 evaluation board with built-in Philips LPC2106 processor [62]. The LPC2106 is 32-bit microcontroller with an ARM7 core that uses an on-chip “phase-locked loop” (PLL) to run the processor at frequencies ranging from 12 MHz to 60 MHz [63]. The oscillator and CPU frequencies used throughout this study (except for TTC-DVS) are 12 MHz and 60 MHz, respectively. The compiler used is the GCC ARM 4.1.1 where the used simulator is the Keil ARM development kit (v3.12).

5.3 Scheduler behaviour test

In each scheduler implementation, the scheduler behaviour (e.g. task sequencing) was measured directly from the simulator by using breakpoints in each task to observe the order (and the time) at which the tasks execute. The measurements were made over a number of successive major cycles. The results obtained when executing each STC were then reported and compared with the example schedules discussed in Section 4.

[4] See [30], [31], [59] for some further implementation options.

5.4 Jitter tests

To obtain a meaningful set of task jitter results, Task A, Task B and Task C had variable durations in STC A and STC B. This is in order to facilitate a detailed study of the impact of varying task duration on the release jitter of other tasks scheduled in the same ticks. In STC C, we explored the impact of long tasks on the tick and tasks jitter. It should be noted that the jitter levels were only considered when the scheduler operates in normal conditions (i.e. jitter levels were not assessed with STC D which is originally designed to test the system under abnormal conditions).

To measure the jitter on the tick and tasks using practical means, we set a pin high at the start of the tick or task (for a short period of time) and then measured the periods between each two consecutive rising edges. We recorded 5000 samples in each experiment. The periods were measured using a National Instruments data acquisition card 'NI PCI-6035E' [64] along with LabVIEW 7.1 software [65]. To assess the jitter levels, two values can be considered: "average jitter" and "difference jitter". The difference jitter is the difference between the maximum and the minimum period in the measurements. This kind of jitter is also described as "absolute jitter" [66]. The average jitter is taken as the standard deviation from the average period. Here, we reported the difference jitter only. Of course, there can be many other ways to measure jitter in practice, but such methods have been found to be appropriate for our study.

5.5 CPU, memory and power test

The CPU overhead is one of the cost parameters that have been used here to differentiate between the various scheduler forms. To get practical CPU overhead measurements for each scheduler, we ran the STC A for the interval of 25 seconds and measured the total time required to run the scheduler by the performance analyser tool in the Keil simulator. The CPU overhead results were then presented in percentages of the total computational resources.

For memory requirements, we recorded the code (ROM) and data (RAM) memory values required to implement STC A for each scheduler considered in this study. We performed this task by using the ".map" file created when the STC is compiled. For data memory overhead (i.e. RAM requirements), we also measured the STACK usage by filling the data memory with a dummy code and recording the number of changed (overwritten) bytes after the system is executed for a sufficient period of time.

In order to obtain representative results of power consumption, we measured the input current and voltage to the LPC2106 CPU core while executing STC A and STC B. The sampling frequency of 10 kHz was used over 5000 major cycles (the need for this specific frequency is explained more in [12]). The values of currents and voltages were then multiplied and averaged out.

6. RESULTS

In this section, we provide the output results from the application of the STC technique on all selected TTC implementations. As we have attempted to make clear, even the small (and by no means exhaustive) selection of TTC scheduler implementations demonstrate a wide range of different patterns of behaviour, and a "one size fits all" implementation does not exist. We suggest that the results obtained by applying the scheduler test cases (summarised in Table 15) provide a simple, concise way of distinguishing between the different implementation options. The first column in the results table shows the list of TTC scheduler implementations which have been tested in this study. The second four columns summarise the output task sequencing results from all STCs in each TTC implementation. The sixth column then contains CPU overheads in percentages. Columns 7 to 9 include: Tick jitter levels from STC C, the release jitter levels for Task A and the release jitter levels for Task B, respectively, using STC B. It is worth noting that these are the most appropriate measures for showing the impact on jitter as a result of using various scheduler structures. Finally, the last three columns present the memory and power overhead results for comparison purposes.

Table 15: Summary of results obtained in this paper.

Scheduler	**STC A**	**STC B**	**STC C**	**STC D**	**CPU %**	**Tick Jitter (µs)**	**Task A Jitter (µs)**	**Task B Jitter (µs)**	**ROM (Bytes)**	**RAM (Bytes)**	**Power (mW)**
TTC-ISR	A1	B1	C1	D1a	39.5	9999.5	0.1	4016.7	2256	127	36.4
TTC Dispatch	A1	B1	C1	D1b	39.7	0.5	0.1	4022.7	4012	325	35.7
TTC-DVS	A1	B1	C1	D1b	40.6	0.6	0.1	4192.9	17460	767	16.6
TTC-TG	A1	B1	C5	D2b	39.8	0.3	0.1	4026.2	4296	446	35.7
TTC-MTI	A1	B2	C5	D3a	39.6	0.3	0.1	0.0	3620	514	36.3
TTC-Adaptive	A2	B2	C6	D3b	39.8	0.3	0.1	0.0	5364	510	36.5

Jitter in Task A has been included in the table to allow a comparison with the jitter levels in low-priority tasks. Key jitter results are shown in Figure 15 for comparison purposes. It can be clearly noticed that the TTC-MTI and TTC-Adaptive schedulers had the ability to eliminate release jitter in all tasks running in the system regardless of their order or position (see [30], [60] for further details). The results for CPU, memory and power requirements are presented in Figure 16 to Figure 19 to facilitate a graphical comparison between all schedulers with respect to hardware resource utilisation.

From the presented graphs, we can make the following observations. The CPU overheads in all compared schedulers are approximately the same. However, such results have been presented in the paper to demonstrate that the improvement achieved by some schedulers is not compromised by their computational costs. On the other hand, it can be seen that the code memory required to implement (for example) the TTC-MTI scheduler was even smaller than was used for the majority of other schedulers. Particularly for the TTC-Adaptive scheduler, the little increase in the code memory as compared to other schedulers is outweighed by the significant improvement achieved in the scheduler predictability [60]. Moreover, it is so clear that the TTC-DVS has an advantage over all other implementations in term of power requirement. This is due to the incorporation of DVS technique which was intended to reduce the CPU power consumption in TTC scheduler [12].

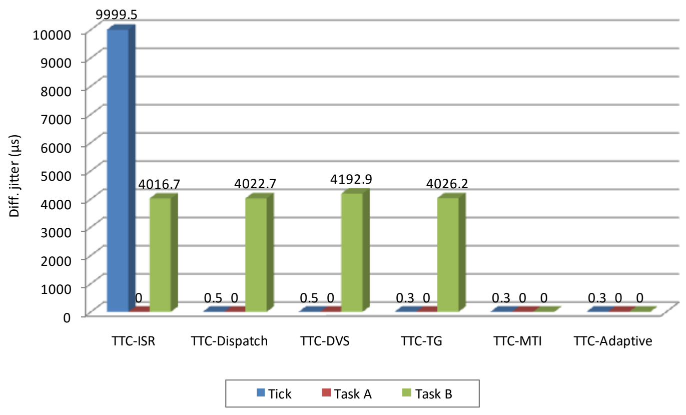

Figure 15. Key jitter results in all TTC implementations

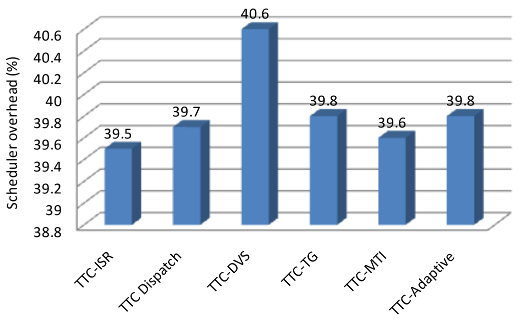

Figure 16. CPU requirements in all TTC implementations

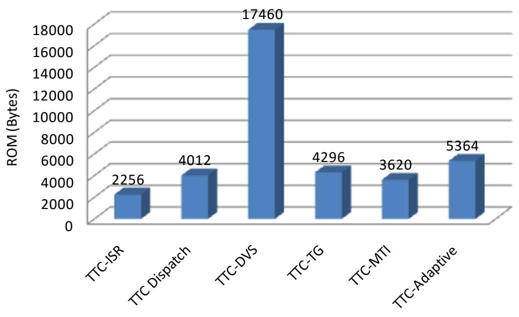

Figure 17. ROM requirements in all TTC implementations

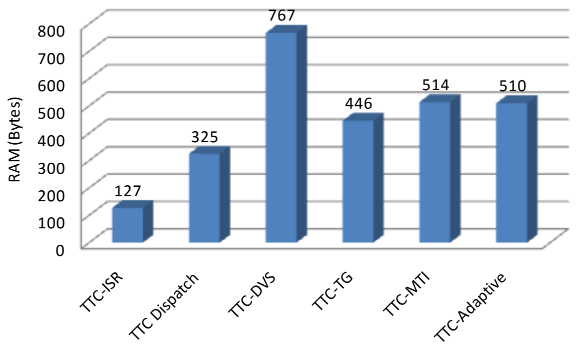

Figure 18. RAM requirements in all TTC implementations

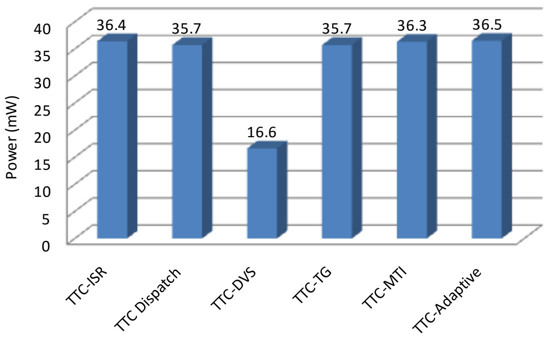

Figure 19. Power requirements in all TTC implementations

7. Conclusions

While there has been a great deal of interest in the development, assessment and refinement of real-time scheduling algorithms, only limited amount of work has considered ways for bridging the gap between scheduling algorithms and scheduler implementations in real-time, resource-constrained embedded systems. This is unfortunate because there is always a "one-to-many" mapping between scheduler algorithms and practical scheduler implementations. Therefore, it is expected that decisions made at the implementation stage will have a measurable impact on the runtime behaviour of the embedded application.

In the study presented in this paper, we first sought to identify a set of representative implementation classes for a TTC scheduler and then introduced the concept of "scheduler test cases" (STCs) as a means of assessing the behaviour of these different scheduler forms. The aim with these STCs was to facilitate empirical "black box" comparisons of the different scheduler implementations and be able to distinguish the behaviour of such implementations without necessitating access to (or understanding of) the underlying source code. Through the employment of these STCs, we were able to fully document and compare the behaviour of TTC implementations against some empirical measures: i.e. task behaviour (under normal conditions and in the event of errors), tick and tasks release jitter, CPU and memory overheads, and finally the CPU power consumption.

Please note that, while we believe that STCs provide a useful way of making quantitative empirical comparisons of different TTC scheduler implementations, further work remains to be done in this area. For example, our current STCs consider only system behaviour and do not take into account factors such as ease of use. It may be that empirical techniques – such as the Small Group Methodology [67] – could be integrated with STCs in order to take such factors into account. Despite this limitation, we still believe that the STC approach has the great potential to apply on a wide range of scheduling algorithms other than TTC. Certainly, if the complexity of an algorithm grows, the number of possible implementation options of that algorithm would naturally grow. This would simply imply that the developed STC technique is more required to verify (and possibly compare) the behaviour of the various implementations of the tested algorithm. For example, alternative real-time scheduling algorithms such as rate monotonic, earliest deadline first, least laxity first and priority ceiling protocol can be considered as potential targets for the proposed STC technique. Since such algorithms are generally more complicated than the TTC, designing (hence implementing) an appropriate set of test cases would inevitably be a challenging process. Moreover, considering the application of STC technique in multi-processor embedded systems employing TTC scheduling algorithms (along with shared-clock protocol for message scheduling) can be a good suggestion for future publication.

ACKNOWLEDGEMENTS

Authors of this paper would like to thank Professor Michael Pont for supervising this project, performed in the Embedded Systems Laboratory (ESL) research group, University of Leicester, United Kingdom.

REFERENCES

[1] H. Kopetz, *Real-Time Systems*. Boston, MA: Springer US, 2011.

[2] M. Nahas, “Bridging the gap between scheduling algorithms and scheduler implementations in time-triggered embedded systems.,” Thesis, University of Leicester, 2009.

[3] B. Sun, X. Li, B. Wan, C. Wang, X. Zhou, and X. Chen, “Definitions of predictability for Cyber Physical Systems,” *Journal of Systems Architecture*, vol. 63, pp. 48–60, Feb. 2016.

[4] M. J. Pont, *Patterns for time-triggered embedded systems: building reliable applications with the 8051 family of microcontrollers*. Harlow: Addison-Wesley, 2001.

[5] M. Short, “Analysis and redesign of the ‘TTC’ and ‘TTH’ schedulers,” *Journal of Systems Architecture*, vol. 58, no. 1, pp. 38–47, Jan. 2012.

[6] M. Nahas, M. J. Pont, and M. Short, “Reducing message-length variations in resource-constrained embedded systems implemented using the Controller Area Network (CAN) protocol,” *Journal of Systems Architecture*, vol. 55, no. 5–6, pp. 344–354, May 2009.

[7] M. A. Hanif, “Design and evaluation of flexible time-triggered task schedulers for dynamic control applications,” Thesis, Department of Engineering, 2013.

[8] M. Nahas and A. M., “Ways for Implementing Highly-Predictable Embedded Systems Using Time-Triggered Co-Operative (TTC) Architectures,” in *Embedded Systems - Theory and Design Methodology*, K. Tanaka, Ed. InTech, 2012.

[9] T. P. Baker and A. Shaw, “The cyclic executive model and Ada,” *Real-Time Syst*, vol. 1, no. 1, pp. 7–25, Jun. 1989.

[10] C. D. Locke, “Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives,” *The Journal of Real-Time Systems*, vol. 4, no. 1, pp. 37–53, Mar. 1992.

[11] G. Langelier, A. Dury, A. Petrenko, S. Ramesh, and T. Assaf, “Building an interactive test development environment for cyclic executive systems,” in *Industrial Embedded Systems (SIES), 2015 10th IEEE International Symposium on*, 2015, pp. 1–9.

[12] T. Phatrapornnant and M. J. Pont, “Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling,” *IEEE Transactions on Computers*, vol. 55, no. 2, pp. 113–124, Feb. 2006.

[13] D. I. Katcher, H. Arakawa, and J. K. Strosnider, “Engineering and analysis of fixed priority schedulers,” *IEEE Transactions on Software Engineering*, vol. 19, no. 9, pp. 920–934, Sep. 1993.

[14] B. Koch, “The Theory of Task Scheduling in Real-Time Systems: Compilation and Systematization of the Main Results,” Studies Thesis, University of Hamburg, 1999.

[15] M. Nahas, “Studying the Impact of Scheduler Implementation on Task Jitter in Real-Time Resource-Constrained Embedded Systems,” *Journal of Embedded Systems*, vol. 2, no. 3, pp. 39–52, 2014.

[16] M. W. Whalen and M. P. E. Heimdahl, “On the requirements of high-integrity code generation,” in *4th IEEE International Symposium on High-Assurance Systems Engineering, 1999. Proceedings*, 1999, pp. 217–224.

[17] E. Estévez, A. Sánchez-García, J. Gámez-García, J. Gómez-Ortega, and S. Satorres-Martínez, "A novel model-driven approach to support development cycle of robotic systems," *Int J Adv Manuf Technol*, vol. 82, no. 1–4, pp. 737–751, Jun. 2015.
[18] P. Marsh, "Models of control," *Electronics Systems and Software*, vol. 1, no. 6, pp. 16–19, Dec. 2003.
[19] C. O'Halloran, "Issues for the automatic generation of safety critical software," in *ase*, 2000, p. 277.
[20] B. Schätz, T. Hain, F. Houdek, W. Prenninger, M. Rappl, J. Romberg, O. Slotosch, M. Strecker, and A. Wisspeintner, "CASE tools for embedded systems," *Tum-i, TU München*, 2003.
[21] K. Michels, "Trends in the Development of Drive Components for Electric and Hybrid Vehicles," *ATZelektronik worldwide*, vol. 10, no. 4, pp. 4–7, 2015.
[22] C. Mwelwa, K. Athaide, D. Mearns, M. J. Pont, and D. Ward, "Rapid software development for reliable embedded systems using a pattern-based code generation tool," SAE Technical Paper, 2006.
[23] C. Mwelwa, M. J. Pont, and D. Ward, "Towards a CASE tool to support the development of reliable embedded systems using design patterns," in *Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering*, 2003, pp. 67–80.
[24] S. K. Baruah, "The Non-preemptive Scheduling of Periodic Tasks upon Multiprocessors," *Real-Time Systems*, vol. 32, no. 1–2, pp. 9–20, Feb. 2006.
[25] T. Phatrapornnant, "Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling," Engineering, 2007.
[26] M. J. Pont, S. Kurian, H. Wang, and T. Phatrapornnant, "Selecting an appropriate scheduler for use with time-triggered embedded systems.," in *Proceedings of the 12th European Conference on Pattern Languages of Programs (EuroPLoP '2007)*, Irsee, Germany, 2007, pp. 595–618.
[27] C. Mwelwa, "Development and Assessment of a Tool to Support Pattern-Based Code Generation of Time-Triggered (TT) Embedded Systems," PhD Thesis, University of Leicester, Leicester, UK, 2006.
[28] Y. Cho, S. Yoo, K. Choi, N.-E. Zergainoh, and A. A. Jerraya, "Scheduler implementation in MP SoC design," in *Design Automation Conference, 2005. Proceedings of the ASP-DAC 2005. Asia and South Pacific*, 2005, vol. 1, pp. 151–156 Vol. 1.
[29] S. Key, M. J. Pon, and S. Edwards, "Implementing Low-cost TTCS Systems using Assembly Language.," in *Proceedings of the Eighth European conference on Pattern Languages of Programs (EuroPLoP 2003)*, Germany, 2003, pp. 667–690.
[30] M. Nahas, "Employing Two 'Sandwich Delay' Mechanisms to Enhance Predictability of Embedded Systems Which Use Time-Triggered Co-Operative Architectures," *Journal of Software Engineering and Applications*, vol. 04, no. 07, pp. 417–425, 2011.
[31] M. Nahas, "Implementation of highly-predictable time-triggered cooperative scheduler using simple super loop architecture," *International Journal of Electrical & Computer Sciences*, vol. 11, pp. 33–38, 2011.
[32] M. Nahas, M. J. Pont, and A. Jain, "Reducing task jitter in shared-clock embedded systems using CAN," in *Proceedings of the UK Embedded Forum 2004*, Birmingham, UK, 2004, pp. 184–194.
[33] M. Nahas, "Applying Eight-to-Eleven Modulation to reduce message-length variations in distributed embedded systems using the Controller Area Network (CAN) protocol," *Canadian Journal on Electrical and Electronics Engineering*, vol. 2, no. 7, pp. 282–293, 2011.
[34] M. Nahas, "Developing a Novel Shared-Clock Scheduling Protocol for Highly-Predictable Distributed Real-Time Embedded Systems," *American Journal of Intelligent Systems*, vol. 2, no. 5, pp. 118–128, Dec. 2012.
[35] Z. H. Hughes and M. J. Pont, "Design and test of a task guardian for use in TTCS embedded systems," in *Proceedings of the UK Embedded Forum 2004*, Birmingham, UK, 2004, pp. 16–25.
[36] Z. M. Hughes and M. J. Pont, "Reducing the impact of task overruns in resource-constrained embedded systems in which a time-triggered software architecture is employed," *Transactions of the Institute of Measurement and Control*, vol. 30, no. 5, pp. 427–450, Dec. 2008.
[37] R. L. Burdett and E. Kozan, "Techniques to effectively buffer schedules in the face of uncertainties," *Computers & Industrial Engineering*, vol. 87, pp. 16–29, 2015.
[38] F. Werner, "Scheduling under uncertainty," *unpublished document*, 2012.
[39] I. Sommerville, *Software Engineering*. Addison-Wesley, 2007.
[40] E. Tran, "Verification/validation/certification," *Topics in Dependable Embedded Systems. Carnegie Mellon University*, 1999.

[41] Z. Liu, N. Gu, and G. Yang, “An automate test case generation approach: using match technique,” in *Computer and Information Technology, 2005. CIT 2005. The Fifth International Conference on*, 2005, pp. 922–926.
[42] P. Pringsulaka and J. Daengdej, “Coverall algorithm for test case reduction,” in *Aerospace Conference, 2006 IEEE*, 2006, p. 8–pp.
[43] H. Singh, M. Conrad, and S. Sadeghipour, “Test case design based on Z and the classification-tree method,” in *Formal Engineering Methods., 1997. Proceedings., First IEEE International Conference on*, 1997, pp. 81–90.
[44] K. Tahera, C. Earl, and C. Eckert, “The role of testing in the engineering product development process,” 2012.
[45] T. Arts, J. Hughes, U. Norell, and H. Svensson, “Testing AUTOSAR software with QuickCheck,” in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, 2015, pp. 1–4.
[46] T. Kos, M. Mernik, and T. Kosar, “Test automation of a measurement system using a domain-specific modelling language,” *Journal of Systems and Software*, vol. 111, pp. 74–88, 2016.
[47] S. J. Cunning and J. W. Rozenblit, “Automatic test case generation from requirements specifications for real-time embedded systems,” in *Systems, Man, and Cybernetics, 1999. IEEE SMC’99 Conference Proceedings. 1999 IEEE International Conference on*, 1999, vol. 5, pp. 784–789.
[48] H. Do and G. Rothermel, “On the use of mutation faults in empirical assessments of test case prioritization techniques,” *Software Engineering, IEEE Transactions on*, vol. 32, no. 9, pp. 733–752, 2006.
[49] W. T. Tsai, L. Yu, X. X. Liu, A. Saimi, and Y. Xiao, “Scenario-based test case generation for state-based embedded systems,” in *Performance, Computing, and Communications Conference, 2003. Conference Proceedings of the 2003 IEEE International*, 2003, pp. 335–342.
[50] K. Beck, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.
[51] A. Hessel, “Model-based test case generation for real-time systems,” 2007.
[52] A. J. Offutt and S. Liu, “Generating test data from SOFL specifications,” *Journal of Systems and Software*, vol. 49, no. 1, pp. 49–62, 1999.
[53] A. Banerjee, S. Chattopadhyay, and A. Roychoudhury, “On Testing Embedded Software,” *Advances in Computers*, 2016.
[54] M. M. Jaghoori, F. de Boer, D. Longuet, T. Chothia, and M. Sirjani, “Compositional schedulability analysis of real-time actor-based systems,” *Acta Informatica*, pp. 1–36, 2016.
[55] W. Hasanain, Y. Labiche, and S. Gheorghe, “Automated state-based online testing real-time embedded software with RTEdge,” in *Model-Driven Engineering and Software Development (MODELSWARD), 2015 3rd International Conference on*, 2015, pp. 294–302.
[56] M. Z. Iqbal, A. Arcuri, and L. Briand, “Environment modeling and simulation for automated testing of soft real-time embedded software,” *Software & Systems Modeling*, vol. 14, no. 1, pp. 483–524, 2015.
[57] J. Guan and J. Offutt, “A model-based testing technique for component-based real-time embedded systems,” in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, 2015, pp. 1–10.
[58] V. P. Kozyrev, “Estimation of the execution time in real-time systems,” *Programming and Computer Software*, vol. 42, no. 1, pp. 41–48, 2016.
[59] M. J. Pont, S. Kurian, and R. Bautista-Quintero, “Meeting Real-Time Constraints Using ‘Sandwich Delays,’” in *Transactions on Pattern Languages of Programming I*, J. Noble and R. Johnson, Eds. Springer Berlin Heidelberg, 2009, pp. 94–102.
[60] M. Nahas and R. Bautista-Quintero, “Implementing adaptive time-triggered co-operative scheduling framework for highly-predictable embedded systems,” *American Journal of Embedded Systems and Applications*, vol. 2, no. 4, pp. 38–50, 2014.
[61] M. Nahas and A. Maait, “Choosing Appropriate Programming Language to Implement Software for Real-Time Resource- Constrained Embedded Systems,” in *Embedded Systems - Theory and Design Methodology*, K. Tanaka, Ed. InTech, 2012.
[62] “LPC2100 Datasheet, PDF - Alldatasheet.” [Online]. Available: http://www.alldatasheet.com/view.jsp?Searchword=Lpc2100. [Accessed: 01-Jun-2016].
[63] I. CIRCUITS, “LPC2106/2105/2104 USER MANUAL,” 2003.
[64] “Multifunction Devices - National Instruments.” [Online]. Available: http://www.ni.com/data-acquisition/multifunction/. [Accessed: 01-Jun-2016].

[65] "LabVIEW System Design Software - National Instruments." [Online]. Available: http://www.ni.com/labview/. [Accessed: 01-Jun-2016].
[66] G. C. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. New York: Springer, 2005.
[67] D. Ayavoo, "The development of reliable x-by-wire systems: assessingt he effectiveness of a'simulation first'approach," Engineering, 2006.

Authors

Mouaaz Nahas was born in UK on 1977. He received the B.Sc. degree (Electrical Engineering) from Jordan University of Science and Technology, Jordan, in 2001, the M.Sc. degree (Communications Engineering) from Loughborough University, UK, in 2002, and the Ph.D. degree (Embedded Systems) from University of Leicester, UK, in 2009. He is currently an Assistant Professor in the Department of Electrical Engineering at Umm Al-Qura University, Makkah, Saudi Arabia. His main research interest is in the development of cost-effective techniques for maximising the reliability of real-time, resource-constrained embedded systems. He also has research interest in electromagnetism and wireless communications.

Ricardo Bautista-Quintero was born in Mexico City on 1972. He received the B.Sc. degree (Electronic Engineering) from Instituto Politécnico Nacional, Mexico, in 1995, the M.Sc. degree (Microelectronics Engineering) from Instituto Politécnico Nacional, Mexico in 2001, and the Ph.D. degree (Embedded Systems) from University of Leicester, UK, in 2009. He is currently a Researcher Professor in the Department of Mechanical Engineering at Instituto Tecnológico de Culiacan, Sinaloa, México. His main research interest is in the development of reliable robust control implementations on resource-constrained embedded systems.