ABSTRACT
Knowledge of task execution time is a key requirement when determining the most appropriate scheduler algorithm (and scheduler parameters) for use with embedded systems. Unfortunately, determining task execution times (ETs) can be a challenging process. This paper introduces a novel system architecture which is based on two components (i) the main processor (MP) platform, containing the time-triggered (co-operative) scheduler and task code, and (ii) a second processor, executing a “scheduler agent” (SA). In the experiments described in this paper, the MP contains an instrumented scheduler and, during a “tuning” phase, the SA measures – on line – the ET of each task as it runs. The measured values are then used to fine tune the task schedule in an attempt to ensure that (i) all task constraints - such as deadline and jitter - are met (ii) power consumption is reduced. After the tuning phase is completed the SA continues to monitor the MP and can take appropriate action in case of errors. In the paper, the effectiveness of the proposed architecture is demonstrated empirically by applying it to a set of tasks that represent a typical embedded control system.

1. INTRODUCTION
As embedded designs become more widespread and increasingly complex they tend to use faster processors and systems-on-chip implementations [1]. Such platforms often employ performance-enhancement features such as pipelines, caches, and branch predictors: these features make it difficult to predict the internal state of the system [2]. As a consequence analysing the timing of the system and estimating the worst case execution time (WCET) will require significant effort even for very simple systems [3].

Our goal in this paper is to introduce a novel system architecture which can be used to monitor – on line – certain key task parameters (such as WCET) in embedded systems. Our particular focus is on control systems, implemented using simple “time-triggered co-operative” (TTC) schedulers. The specific TTC scheduler we use here is sometimes described as a form of “time-driven cyclic executive” [4].

In the paper we discuss techniques for the measurement of “best case execution time” (BCET), WCET and scheduler overhead. The measured values are then used to fine tune the task schedule in an attempt to ensure that (i) all task constraints - such as deadline and jitter - are met (ii) power consumption is reduced. After this tuning phase is complete, the proposed architecture is also able to monitor the system while it is running and take appropriate action (such as reseting the systems) in the event of faults. The effectiveness of the proposed architecture is tested empirically by applying it to a set of tasks that represent a typical embedded control application.

The remainder of this paper is organised as follows. Section 2 discusses previous work in this area. Section 3 describes the proposed system architecture in detail. Section 4 studies the performance and effectiveness of the proposed architecture. Finally, in Section 5, we present our conclusions.
2. PREVIOUS WORK

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

2.1. TT Software architectures

This paper is concerned with the development of software for control systems implemented (for cost reasons) on resource-constrained embedded processors. In order to minimise costs and maximise predictability in this type of application, we wish to keep the software architecture as simple as possible. As a consequence, instead of a full RTOS, some form of simple scheduler is generally used. For example, a cyclic executive [5], [6] is a form of co-operative (or “non-preemptive”) scheduler which has a “time triggered” [7] - as opposed to event-triggered [8] - architecture.

Provided that an appropriate implementation is used, a time-triggered, co-operative (TTC) architecture is a good match for a wide range of low-cost, resource-constrained control applications. TTC architectures also demonstrate very low levels of task jitter [6], and can maintain their low-jitter characteristics even when techniques such as dynamic voltage scaling (DVS) are employed to reduce system power consumption [9].

The type of TTC scheduler implementation discussed in this paper is usually implemented using a hardware timer, which is set to generate interrupts on a periodic basis (with “tick intervals” of around 1 ms being typical). In most cases, the tasks will be executed from a “dispatcher” (function), invoked after every scheduler tick. The dispatcher examines each task in its list and executes (in priority order) any tasks which are due to run in this tick interval. The scheduler then places the processor into an “idle” (power saving) mode, where it will remain until the next tick. Figure 1 shows an example of two tasks run with TTC scheduler with a tick interval of 1 ms. This “tick” is derived from a timer overflow: drift or jitter in this timing is, in large part, dependent on the associated computer hardware.

![Figure 1. Illustrating the operation of a typical (interrupt-driven) TTC scheduler implementation.](image)

2.2. The need for accurate WCET predictions

TTC architectures employ static scheduling and no task pre-emption. The schedule is calculated based on estimates of task “worst case execution time” (WCET). If such estimates prove to be incorrect, the problem may not even be detected in a TTC implementation (let alone resolved). This may have a serious impact on the system behaviour. For example, as Buttazzo has noted: “[Co-operative] scheduling is fragile during overload situations, since a task exceeding its predicted execution time could generate (if not aborted) a domino effect on the subsequent tasks” [10].

As many researchers have observed [11-19], determining the WCET of tasks is rarely straightforward.

A large number of different studies have been conducted in connection with WCET and we can only mention a small sample here. For example: Engblom and Jonsson, [20] used a mathematical models to examine the timing of instructions on in-order single-issue pipelined processors; Puschner and Burns [21] proposed a “single path programming paradigm” for reducing variations in the execution times of programs; Devere and Puaut [2] measured the execution time of different program segments (using generated test data) in order to obtain the WCET of the whole program.

2.3. Dealing with WCET errors

The complexity of WCET estimation means that - if employing a TTC implementation - the user needs to appreciate this potential risk, and understand precisely how the implementation will behave if such an error occurs.1

One simple solution to this problem is to err on the side of caution when employing WCET estimates, thereby reducing the chances that an overrun will occur. Typical “safety margins” used in this way are said to be around 20% [22].

Such an approach is simple and can be effective, but it inevitably adds to costs. An alternative is to be slightly more conservative when estimating WCET values (e.g. add 5% to accurate estimates) and then extend the scheduler (or add additional hardware) in such a way that (at run time) any overrunning tasks can be shut down, and/or the schedule can be adjusted. Such an approach may also allow us to deal with error-related overruns (for example, tasks which overrun because of a hardware-related error). In these circumstances, the problem can be addressed (at least in part) by employing some form of “watchdog timer” (e.g. [23]) in a “scheduler watchdog” design (e.g. [24]). Alternatively, greater control over the system behaviour can be obtained by using a “task guardian” [25].

2.4. Other TT architectures

The monitoring approach in another TT architecture should also be mentioned here. The high-level Giotto language [26] splits the system into two components, an “Embedded Machine” and a “Scheduling Machine”. In the case of Giotto, the two components execute (as virtual machines) on the same CPU.

---

1 We should note that lack of knowledge about WCETs is a problem which faces the developers of many embedded systems (not just those based on TTC). For example, as Gergeleit and Nett have noted: “Nearly all known real-time scheduling approaches rely on the knowledge of WCETs for all tasks of the system.” [15].
3. THE MP-SA ARCHITECTURE

In Section 2 we considered the need for information about task WCET when developing schedules. In the remainder of this paper we present and assess a novel system architecture that can be used to address this need. We describe the architecture in this section.

3.1. Overview

An overview of the proposed system architecture is given in Figure 2.

![Figure 2. An overview of the MP-SA architecture.](image)

The proposed architecture employs an additional microcontroller (the Scheduling Agent, SA) to measure the BCET and WCET of each task while they are running on the main processor (MP) for a specified period of time. The SA also measures the scheduler overhead. The information gathered in this way is used in an attempt to fine tune task parameters (such as the task offsets, or “initial delay”) and scheduler parameters (such as the tick interval) in order that all task constraints are met and the power consumption is reduced. After the fine-tuning phase is finished the SA continues to monitor the MP while running the tasks with the new parameters and take the appropriate action in case of errors.

For ease of reference, we refer to this two-processor arrangement as the “MP-SA architecture” throughout this paper.

3.2. How does the MP-SA architecture operate?

The MP-SA architecture operates as follows:

i) At compile time, the task specifications (estimated BCET, estimated WCET, deadline, period, and upper bound of jitter) are provided to the MP.

ii) When the system starts, the MP sends task specifications to the SA.

iii) The SA asks the MP to schedule a number of dummy (empty) tasks, equal to the total number of the real tasks. The SA measures the scheduler overhead (by measuring the time spent by the scheduler out of “idle” mode as it executes the dummy tasks) for a pre-specified period of (“overhead”) time.

iv) The SA calculates the initial task order, task offsets, and the scheduler tick interval based on the measured scheduler overhead and the given task parameters. In doing this the SA assumes that the overhead can be represented as an additional task that runs at every tick with BCET equal to zero and WCET equal to the scheduler overhead measured in Step iii. The SA sends these parameters to the MP and asks it to begin running the real tasks for a specified period of (“tuning”) time.

v) While the MP is running the real tasks, the SA measures the actual BCET and WCET of each task.

vi) The SA repeats Step iv to fine tune the scheduler based on the actual measured BCET and WCET of each task.

vii) If a set of parameters is found so that all task constraints are met then the SA sends these parameters to the MP to restart the scheduler according to these new parameters. (If a set of such parameters cannot be found then the SA tells the MP to “sail silently”).

viii) If a suitable set of parameters have been found, the SA monitors the system during the normal program execution (functioning as a form of “task watchdog”).

Please note that in Step i and Step ii we assume that the task specification will be (initially) stored in the MP. This allows us to create a generic SA (suitable for use for a wide range of different MPs).

3.3. Calculating task and scheduler parameters

Given a set of tasks each described by (BCET,WCET, deadline, period, jitter), the SA tries to calculate the appropriate task orders, task offsets, and tick interval so that task constraints (deadline and jitter in this study) are met and the power consumption is reduced. In doing this, the SA does not attempt to complete an exhaustive search. Instead, the SA employs a “best characteristics first” approach and stops trying parameter combinations when it finds the first working solution (if ever). To achieve this, the SA begins by sorting the tasks according to two criteria: task precedence and task deadline (earliest deadline first). It then assumes that the first task will run with zero offset and tries to find a suitable offset for the second task (starting with zero offset), using the longest possible tick interval (usually the greatest common divisor of task periods). If such an offset is identified (and both tasks meet their constraints), a third task is added to the system and the process is repeated.

At each step it is assumed that the constraints of all tasks are tested for a duration which is long enough to ensure full coverage of the task set. Since all tasks are assumed to be periodic, the test duration period is given by the “major cycle” (a period of time equal to the Least Common Multiple – LCM - of the task periods: e.g. [5]) plus the maximum offset².

We carry on in this way until all tasks have been scheduled (if this proves possible). If a feasible schedule cannot be found the SA repeats the above process with a shorter tick interval.

Please note that scheduling the tasks with the longest possible tick interval can help in reducing the power consumption (because the system spends more time in “idle” mode).

² In this study, task offsets are expressed in ticks and are assumed to fall within the range [0, period].
We also emphasise that this is a “best characteristics first” approach: that is we start with an “ideal” design and proceed iteratively, stopping the search when we have identified the first workable solution.

4. THE MP-SA PERFORMANCE

An empirical test was carried out to study the performance of the MP-SA architecture.

The procedure and results obtained by using this architecture for a system with a set of 3 tasks are detailed in this section. A time triggered co-operative (TTC) scheduler described previously [27] was used to schedule the tasks

4.1. Task set

To explore the effectiveness of the MP-SA architecture a set of 3 tasks (Task Sa, Task Co, Task Ac) was used: these were intend to be representative of those used in a typical embedded control system. In such a system, Task Sa would be the first task to run and would be used for data sampling. Task Co would then execute the control algorithm and – finally - Task Ac would control the actuator(s). In this study, we assume that the tasks run co-operatively, in this sequence.

The specifications of the tasks (BCET, WCET, deadline, period, and the maximum allowed jitter) are shown in Table 1. Please note that there are two values for the BCET and another two for the WCET of each task indicated in this table. The first value is the value estimated by the designer and the second value is the value which measured by the SA while the tasks run on the target MP.

The chosen hardware platforms for both the SA and the MP were 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 communications between the SA and the MP was carried out via a CAN bus in this design.

4.2. Task scheduling without the MP-SA architecture

One possible way to schedule the task set described Table 1 is shown in Figure 3. This schedule is based on the estimated values for the BCET and the WCET along with the other constraints for the periods, deadlines, and the upper bound of jitter. The tick interval of this scheduler is chosen to be 100 ms length and the offsets of all the tasks are set to 0.

This schedule has several potential drawbacks which can be summarised as follows:

i) Estimated values of BCET and WCET may not be accurate (Table 1). Building the scheduler based only on the estimated values may cause some tasks to miss their deadlines and / or encounter high level of jitter (Task Ac in this example: Figure 4).

ii) Ignoring the scheduler overhead can lead to similar effects (such as missed deadlines and / or increase the level of jitter): see Figure 5.

iii) Using a short tick interval, such as the 100 ms tick interval that was used here, instead of using the longest possible tick interval (which is 400 ms in the current case), will increase the system power consumption.

4.3. Task scheduling with the MP-SA architecture

The impact of the MP-SA architecture is illustrated in Figure 5.

In this case, the SA has adapted the task schedule based on the measured values of the task BCET and WCET. One consequence is that the SA has identified a “compromise” tick interval (200 ms), which could be expected to reduce the power consumption (compared with the original value of 100 ms) while also ensuring low levels of jitter in both of the time-sensitive tasks (Task Sa and Task Ac). In this case, the jitter in the original schedule was 0.014 ms and 8.001 ms for Task Sa and Task Ac (respectively): after use of the SA, the jitter became 0.014 ms for both the tasks.

Please note that in addition to adjusting the tick interval to 200 ms, the SA adjusted the offset of Task Ac to 1 tick (rather than 0).

4.4. System behaviour in case of faults

The effectiveness of the MP-SA architecture on the system behaviour in the “normal” operating mode was also tested. During this test, a fault was injected in the system which changed the characteristics of Task Sa (the BCET became 37 ms and the WCET became 50 ms respectively). This error was assumed to represent the impact of a hardware fault

Under these circumstances, without using the MP-SA architecture, the system ran without sensing the violated deadline and jitter constraints. In this test, the measured jitter of Task Co was 25.35 ms.

When the test was repeated in a system using the MP-SA architecture, the SA sensed the violated constraints and it forced the MP to restart (in an effort to recover from this fault).

Please note that other recovery behaviour (e.g. backup tasks) could also be implemented in response to the detected errors.
Table 1. Task specifications

<table>
<thead>
<tr>
<th>Task</th>
<th>Estimated</th>
<th>Measured</th>
<th>Deadline</th>
<th>Period</th>
<th>Jitter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BCET</td>
<td>WCET</td>
<td>BCET</td>
<td>WCET</td>
<td></td>
</tr>
<tr>
<td>Sa</td>
<td>37</td>
<td>38</td>
<td>37</td>
<td>40</td>
<td>50</td>
</tr>
<tr>
<td>Co</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td>11</td>
<td>65</td>
</tr>
<tr>
<td>Ac</td>
<td>20</td>
<td>21</td>
<td>20</td>
<td>22</td>
<td>70</td>
</tr>
</tbody>
</table>

Figure 3. A simple schedule based on the estimated BCET and WCET with 100 ms tick interval.

Figure 4. Effect of inaccurate estimations of BCET and WCET on task behaviour.

Figure 5. Effect of scheduler overhead on task behaviour.

Figure 6. Task behaviour with the scheduler produced by the MP-SA architecture.
4.5. Extended task set

To test the effectiveness of the MP-SA architecture with a slightly more complex system, two additional tasks (Task EXT1 and Task EXT2) were added. Table 2 shows the specifications of the additional tasks (BCET, WCET, deadline, period, and the maximum allowed jitter).

The same hardware platforms were used in this study.

In this case, the SA set the tick interval to 200 ms and the offsets of the Task Ac, Task EXT1 and Task EXT2 to 1 tick.

Using the SA, the measured values of the jitter from the Task Sa, Task Co, Task Ac, Task EXT1 and Task EXT2 was found to be 0.014 ms, 6.008 ms, 0.014 ms, 4.012 ms and 5.995 ms respectively. The jitter constraints for all 5 tasks were met.

5. CONCLUDING REMARKS

In this paper we have introduced a novel architecture that can be used in time triggered embedded systems to:

[1] Measure the BCET, WCET, and scheduler overhead during “normal” system operation.

[2] Fine tune the scheduler so that task constraints (such as deadline and jitter) are met and power consumption is reduced.

[3] Monitor the system and take the appropriate action in the event of faults.

These results were achieved at the expenses of using an additional microcontroller in the system.

The effectiveness of the proposed architecture was demonstrated using a small empirical study.

6 REFERENCES


on Object-oriented Real-time Dependable Systems, Laguna Beach, California, U.S.A.


