A Framework for Scheduling in Safety-Critical Embedded Control Systems

I. Bate and A. Burns
Department of Computer Science
University of York
York, United Kingdom
e-mail: {iain.bate,alan.burns}@cs.york.ac.uk

Abstract

This paper describes an approach that has been developed over a number of years for the task of scheduling systems and providing evidence that the timing requirements are met. The approach has been targeted at the safety-critical systems domain, and more specifically the development of jet engine control systems. The work has resulted in a computational model that supports the reuse of legacy systems whilst providing a powerful computational model. In addition, timing analysis has been developed that features low pessimism, low computational complexity and that is robust to change. The aim of this paper is to summarise and bring together all the facets of the work. This is, in part, achieved through a case study that is used as a threaded example through the course of the paper.

1 Introduction

Safety-critical systems are different to other systems because a failure to meet a requirement may lead to a catastrophic effect, e.g. an accident leading to loss of life. For this reason, there is a greater emphasis on verification and validation of software in safety-critical systems than with other types of system. An evaluation of industrial practice for avionic systems confirms that the majority of safety-critical systems still employ a cyclic scheduler. Locke [1], amongst others, describe a number of problems with the cyclic scheduler, including having a restricted computational model (i.e. supporting only periodic tasks with a limited range of iteration rates) and poor maintainability.

This paper describes work to support two transitions: from a cyclic scheduler to a fixed priority scheduler, and then from a uniprocessor system to a distributed system. The approach developed is not intended to be general purpose but specifically tackles the following three objectives:

1. the ability to reuse as much of legacy systems as possible and the associated development process,
2. to be able to certify the end product, and
3. to be able to implement the system's timing requirements as efficiently and effectively as possible.

The paper briefly describes the typical characteristics of the systems being considered and how an appropriate computational model is chosen to support reuse and certification (in section 2). Then, an overview is provided of how task attributes can be assigned and analysis carried out. This is performed for the two cases of interest, when the system is based on a uniprocessor architecture (in section 3) and then when the system is migrated to a distributed architecture (in section 4).

2 Background

There are two principal parts to this section: to assess the existing work that is published on fixed priority scheduling in order to establish the areas of work that need to be addressed so that systems can be practically developed, and to identify the characteristics of the system to be scheduled in order to tailor the eventual solution to our needs.

2.1 Review of the Existing Work and Identification of its Shortcomings

Most texts on fixed priority scheduling generally assume the following computational model: software is implemented using a preemptive flow of control, support is provided for both periodic and sporadic tasks, overheads are zero, faults do not occur, and all task attributes (periods, deadlines and offsets) are provided.

For systems with this computational model there is a vast amount of literature available. However, these assumptions are considered unrealistic in practice. For fixed priority scheduling to be adopted by industry, these assumptions were reviewed. A number of threads of work were identified to compensate for inaccuracies in the assumptions when the domain and applications of interest were considered. This paper discusses the work performed related to the computational model, task attribute assignment and schedulability analysis. The problems with the assumption that a preemptive scheduler is used are that:

1. the verification of the timing characteristics is a relatively small job compared to that of verifying the functional characteristics of the system, and a preemptive scheduler vastly increases the number of paths through the software and hence the verification effort,
2. the computational model of the cyclic scheduler is based on a non-preemptive flow of control where a task is always executed to completion. Since a clean sheet of paper is rarely available for any system design (instead the design must reuse legacy components) and most existing systems are based on forms of cyclic scheduling, then a transition from non-preemptive to preemptive scheduling would require a lot of work.

For these reasons, our work has been based on a non-preemptive scheduling model. Also, due the reluctance of certification authorities to have any features that are consider "difficult" to predict, the work is based on periodic tasks. However, all the theory presented in this paper is equally applicable to a preemptive scheduling model. In [2] the "standard" schedulability analysis is enhanced (and proven) to suit this computational model resulting in the recurrence equations (1) - (2) which terminates when $R_i^{n+1} = R_i^n$, or $R_i^{n+1} > D_i$, and blocking equation (3).

$$R_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n + J_j - C_i + \Delta}{T_j} \right\rceil C_j \quad (1)$$

where i is a task/message in the set of tasks/messages
$hp(i)$ is the set of tasks with higher priority than i
R_i is the worst case response time of i and $R_i^0 = C_i$,
C_i is the worst case execution time of i
D_i is the deadline of i,
T_i is the period of i
B_i is the blocking time for i
J_j is the release jitter for j, and
Δ is one clock cycle

When the analysis converges, i.e. $R_i^{n+1} = R_i^n$, the final value for the worst case response time is given as:

$$R_i = R_i^{n+1} + J_j \quad (2)$$

For tasks executed non-preemptively and released periodically, the blocking time is equivalent to the maximum worst-case execution time from the set of tasks that can cause blocking. These tasks are those that are not always released simultaneously as the task that is being analysed and the tasks released simultaneously but whose completion time is greater than the period of the task being analysed. This is expressed in equation (3).

$$\begin{aligned} B_i &= max_{\forall k}(C_k) \\ \text{where } k &\in lp(i) \cdot \forall n : \mathbb{N} | (T_k \neq nT_i \wedge T_i \neq nT_k) \\ &\vee \quad O_k \neq O_i \vee R_k > T_i \end{aligned} \quad (3)$$

where $lp(i)$ is the set of lower priority tasks than task i, and
O_i is the offset of task i.

The most important assumption is the fact that tasks' attributes are complete, which is often not the case. This is particularly important because there has been little work on this issue, and the problem and eventual solution can be made largely independent of the actual computational model used. The basic problem can be stated as taking a set of timing requirements and breaking these down into task attributes. There are two main approaches to the problem of uniprocessor task attribute assignment presented in literature, by Gerber et al [3] and Yerrabilli [4]. These approaches suffer from similar common problems, which are an assumption is made that the task attributes are completely flexible (i.e. existing periods and/or interactions with other tasks are not considered) and the approaches are not intuitive (i.e. they would be difficult to explain to non-specialists and then perform by hand - this is essential for both technology transfer and certification reasons).

This assumption is dealt with in the remainder of this paper. The problem is broken down into two parts: the uniprocessor scheduling problem (section 3) and the distributed scheduling problem (section 4). The paper provides an overview of the techniques due to space considerations. A more complete discussion of the techniques, their origins, how other timing requirements such as separation can be satisfied, and the approach's pros and cons are in [2].

2.2 Typical Characteristics of an Engine Controller

This section is to identify the typical timing requirements for safety-critical systems, which can be divided into two categories, those associated with *tasks*, and those associated with *transactions*.

To give context to the work in this paper, a representative example is used through the course of this paper. The example that is used is the set of requirements illustrated in Tables 1 and 2. Table 1 gives the timing characteristics and requirements of the tasks and Table 2 the timing requirements of the transactions for both a uniprocessor architecture and a distributed architecture. In the case of the distributed architecture, the system is one with three processors all of which are directly connected to a single databus, i.e. there are no gateways. Clearly, the message information presented in the tables only applies to the distributed case.

Each transaction is given an identifier, a precedence order (i.e. 1st task followed by 2nd task followed by 3rd task), and an end-to-end deadline requirement. For example, transaction A refers to a transaction requirement that task 10 is followed by task 00 followed by task 20 and all should be completed within a time of 25000. For the distributed case, the details of the messages carrying the necessary information are also provided by the ordering of the columns in the table, i.e. message M10 carries the data between tasks 10 and 20.

Id	N	T	D	J	C
00	0	25000	25000	0	5406
01	0	500000	500000	0	6864
02	0	800000	800000	0	2904
03	0	50000	50000	0	3095
10	1	25000	25000	12500	3405
11	1	500000	500000	0	6187
12	1	800000	800000	0	1840
13	1	250000	250000	0	2303
20	2	25000	25000	12500	5687
21	2	500000	500000	0	2520
22	2	400000	400000	0	314
23	2	650000	650000	0	226
M10	M	25000	25000	0	200
M11	M	500000	500000	0	265
M20	M	25000	25000	0	24
M21	M	500000	500000	0	179

Table 1: Task Set's Characteristics

Trans. Id	1st Task	1st Message	2nd Task	2nd Message	3rd Task	Trans. Deadline
A	10	M10	00	M20	20	25000
B	21	M11	01	M21	11	500000

Table 2: Transaction Characteristics

The example in Tables 1 and 2 is considered to be realistic because it features constraints that are relevant to embedded systems, which are:

1. *End-to-End Transactions* consisting of a number of tasks with a defined precedence order, a period and an end-to-end deadline.

2. *Constraints on Input and Output Jitter* in order to help control the system's stability.

3 Task Attribute Assignment in the Uniprocessor Case

The purpose of this section is to explain our approach to task attribute assignment with particular reference to the set of requirements outlined in section 2.2.

The approach to meeting the timing requirements and providing evidence that they are met consists of two independent parts. These are: deriving the task attributes (without knowledge of the tasks' worst-case execution times) so that the timing requirements may be met, and then using the tasks' worst-case execution times to verify that the timing requirements are met. This section provides the first part of the approach (i.e. the task attribute assignment), and the second part (i.e. the schedulability analysis) has already been briefly covered in section 2.

The proposal is that the necessary timing requirements can be handled by just setting deadlines and offsets to appropriate values. The system timing requirements are then verified by simply proving the tasks' offsets are enforced in the scheduler and using schedulability analysis to show the tasks' deadlines are met.

3.1 Meeting Transaction Deadlines in Uniprocessor Systems

The approach to meeting the transaction requirements is based on reducing the tasks' deadlines in a systematic manner until the requirement is met and so that the tasks' deadlines are reduced by the minimum possible.

A characteristic of the transaction's requirement is a task's period could be longer than the period of the task that follows it in the transaction. In this case, it may not be necessary or possible to reduce all the deadlines so that perfect precedence between the tasks of the transaction is maintained. For example, consider the transaction requirements of: task precedence constraint is task A followed by task B followed by task C followed by task D, an iteration rate of 100 and an end-to-end deadline of 150. The periods of tasks A, B, C and D are 50, 100, 100 and 50 respectively. In this case, task B cannot always follow task A because task B has a slower update rate.

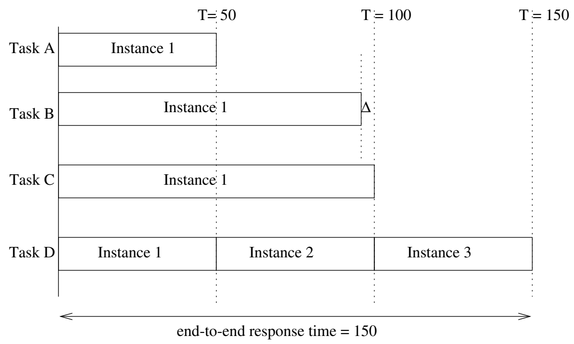

Figure 1: Worst-Case Transaction Response With Attributes Modified using Algorithm 1

Without altering tasks' deadlines, the requirement is not met because the transaction's response time is 250. By an approach based on enforcing precedence by setting deadlines (i.e. $D_A = 50 - 3\Delta$, $D_B = 50 - 2\Delta$, $D_C = 50 - \Delta$ and $D_D = 50$), the transaction requirements could be met with a worst-case transaction response time of 50. However in this case, the task attributes are sub-optimal because the timing requirements could be met with larger deadlines (i.e. $D_A = 50$, $D_B = 100 - \Delta$, $D_C = 100$ and $D_D = 50$) as shown in Figure 1. Having larger deadlines means the tasks have a lower priority. The benefit of this is that the impact of the change in deadlines on the rest of the system is minimised and hence the task set is more likely to be schedulable.

The task attributes of Figure 1 can be generated using Algorithm 1. The algorithm does not simply rely on the tasks' deadlines being reduced so that perfect precedence is maintained. Instead, while the transaction requirement is not met, the highest deadline is reduced by Δ, and if this causes any of the preceding sequence of tasks in the transaction to have the same deadline, then its deadline is also reduced by Δ. The benefit of the approach in Algorithm 1 is that if the deadlines are larger, then there is a greater chance of a schedulable solution. The algorithm also contains a manipulation of the *TDAC - Task_Deadlines_Are_Changing* flag. The use of this flag becomes apparent later.

Algorithm 1 - *for Dealing with Transaction Deadlines*

for each task i in the transaction
  if the following task j has an equivalent deadline then
    reduce the task i's deadline by Δ
    assign the value of false to the TDAC flag
  while the transaction deadline is not met
    assign the value of true to the TDAC flag
    take the longest deadline and reduce it by Δ
    for each task in the transaction
      if the task j has an equivalent deadline then
        reduce the task's deadline by Δ
        assign the value of false to the TDAC flag

Using the approach given in Algorithm 1 on the timing requirements in Tables 1 and 2 leads to tasks 10, 00, 21 and 01 having their deadlines altered (the rest of the tasks' deadlines are unchanged) to 24998, 24999, 499998, and 499999. The set of deadlines result in the tasks having priorities (assigned using DMPO) that enforce the necessary precedence order. Therefore, as long as the tasks meet their individual deadlines, then all the requirements are met.

3.2 Jitter

A prime driver for this work is to control jitter (where it matters) and enable the schedulability analysis for tasks to also verify the jitter requirements. Therefore, the jitter requirement should be handled using deadlines rather than priorities. If a task has a deadline equal to the jitter requirement, then the requirement is met because the window of allowed execution is constrained between the critical instant and the time allowed for the jitter requirement. Using this approach would be pessimistic since there is a minimum processing time for each task that adds no variability, i.e. the Best Case Response Time (BCRT). As an alternative, the deadline can be calculated using equation (4), which allows for the maximum invariant in the task's response time - the BCRT may be found using a technique in [5].

$$D_i = J_i + BCRT_i \qquad (4)$$

where $BCRT_i$ is the best case response time of task i.

The problem with the approach in equation (4) is if a jitter requirement placed on the last task in the transaction, then this would lead to all the preceding tasks having a shorter deadline than D_i, which could significantly affect schedulability. The problem is solved by constraining the variation in the last task's execution at the end of the allowed execution time of the transaction by using an appropriate offset. The offset is calculated by subtracting the value obtained through equation (4) from the transaction's deadline. Precedence of the other tasks in the transaction is enforced by making the other tasks execute before the last task is released. Therefore, the next to last task is given a deadline equal to the offset of the last task and then Algorithm 1 is applied to the tasks preceding the last task. The approach for choosing tasks' deadlines and offsets to meet the jitter requirements can be represented by Algorithm 2.

Algorithm 2 - *for Dealing With Jitter*

for each task (denoted i) in the system
  if task i has a jitter requirement then
    if task i is not the last task in the transaction then
      $D_i = J_i + BCRT_i$
    else
      if the transaction deadline $>$ *transaction period then*
        $D_i = J_i + BCRT_i$
      else
        if $D_i >$ *transaction deadline then*
          task deadline = transaction deadline
        $O_i = D_i - (J_i + BCRT_i)$
        for all preceding tasks (denoted j) in the transaction
          if transaction deadline is not met then
            if $D_j >$ *transaction deadline -* $(J_i + BCRT_i)$ *then*
              D_j *= transaction deadline -* $(J_i + BCRT_i)$

When Algorithm 2 is applied to the requirements in Tables 1 and 2, the attributes of tasks with jitter requirements change to the following (for simplicity it is assumed the best-case execution times of the tasks is not known):

task 10 has an offset of 0 and a deadline of 12500
task 00 has an offset of 12500 and a deadline of 25000

3.3 Overall Task Attribute Assignment

The algorithms presented in the previous subsections cannot be used on their own. The reason is tasks may have more than one requirement placed on them, for example tasks 10 and 20 from Tables 1 and 2 are both part of a transaction and also have a jitter requirement. To prevent anomalies caused by the attributes trying to solve more than one requirement, the algorithms have to be combined so that a consensus is obtained. The result of this is that the algorithms must be integrated in a fashion that means all the tasks' attributes are refined until convergence is achieved.

Combining the various techniques discussed so far in this paper results in Algorithm 3.

If the attributes of the tasks contributing to transaction A (from Table 2) are calculated, then the offsets (column O_S, which is the offset for the uniprocessor case, in Table 3) and deadlines (column D_S in Table 3) result in the jitter and transaction deadlines being met. Therefore as long as the tasks are schedulable the transaction and jitter requirements are also met. The results of the schedulability analysis for the requirements given in Tables 1 and 2 are presented in column R_S in Table 3.

The overall method for priority assignment is not considered to be optimal because of some particularly obscure events that are discussed in [2]. However, use has shown that the approach is sufficient for the typical system timing requirements encountered.

Algorithm 3 - *Overall Algorithm for Priority Assignment*

initialise TDAC flag to true
apply algorithm 2
while the TDAC flag is true
  assign the value of false to the TDAC flag
  for each transaction in the set of transactions
    apply algorithm 1
apply priorities to the tasks by DMPO
perform schedulability analysis of the task set

4 Distributed Scheduling

The production and verification of distributed real-time systems is very complex, particularly for safety critical systems with the need to certify the product. There are a number of technical issues associated with distributed systems. These issues include: allocation, task attribute assignment, increasing robustness to change, and timing analysis. This section concentrates on the last three of these issues, making the assumption that the allocation of functionality to processors is largely based on the physical position of devices. The real challenge with distributed scheduling is the efficient implementation and verification of transactions involving more than one processor. This is because the scheduling of a particular task is no longer influenced by the other tasks executing on the processor it is resident upon. Instead, the influence of other processors becomes relevant, which makes the synthesis, change control, and maintenance problems more complex [6].

The aim of this section is to demonstrate some of the issues involved in scheduling for distributed systems and how a smooth transition can be made from an uniprocessor to a distributed architecture. A key component of this work is trying to maximise the reuse of the techniques developed for the uniprocessor systems. The reasons for this are:

1. There may be a great deal of time, money and effort invested in the existing theory and tools. It would be beneficial to allow the tools to be reused.
2. Treating each processor individually eases the synthesis and maintenance effort by facilitating a modular approach to design.

4.1 Earlier Approaches to Distributed Scheduling

The purpose of this section is to briefly survey the existing literature available on fixed priority scheduling and analysis for distributed systems.

4.1.1 Event-Driven Distributed Scheduling

The early work on an event-driven model for scheduling of distributed systems was performed by Tindell and Clark [7]. Their approach uses periodic tasks for the first task in the transaction. Subsequent tasks or messages are triggered as sporadic tasks when the preceding task has been completed. For the purpose of the analysis, sporadic tasks are modelled as periodic tasks, with period equal to their minimum inter-arrival time. To account for the variability of a sporadic task's release, each task is given a release jitter equal to the worst-case response time of the preceding task. The advantage of the release jitter approach is the existing uniprocessor analysis may be used. However, there are four principal disadvantages of the release jitter approach, which are:

1. Sporadic tasks are frowned upon in safety critical systems due to the difficulty in analysing their behaviour.
2. The use of sporadic tasks means the tasks later in the transaction's order have more cumulative release jitter which is prohibitive for applications where the stability of a control system is important. In addition, the amount of release jitter on the tasks later in the transaction may cause the task set to be unschedulable.
3. Pessimism in the analysis due to interference that cannot occur in practice because of the tasks' phasing [6].
4. The release jitter approach is not robust to change. Small changes on one processor can easily lead to the whole system requiring re-verification.

4.1.2 Distributed Scheduling Using Offsets

The original work on this approach was performed by Liu and Sun [8], and was referred to as the Phase Modification Approach. By giving a task an offset such that its dispatch (or release for simplicity) is always greater than the worst case response time of the event trigger (i.e. the worst case arrival time of the message) then precedence is maintained, even across a distributed system. The technique using offsets is applied in a similar fashion to the release jitter method, where timing analysis is performed for each individual node separately, but this time using exact analysis [9]. The exact analysis involves showing that every instance of every task on a particular processor is schedulable over the period [Maximum Offset of Any Task, 2$\times$Least Common Multiple of the Task Periods + Maximum Offset of Any Task). A problem with this approach is the exact analysis can be an intractable problem. The next section demonstrates our solution.

4.2 Inexact Analysis Using Composite Tasks

In this section, it is proposed that offsets are used to control precedence but in a way that is tractable, this requires the existing uniprocessor schedulability analysis to be extended to account for offsets. Figure 2 gives an approach that combines the offset-based approach to distributed scheduling with the uniprocessor task attribute assignment. Basically at a uniprocessor level, the task attributes are calculated using Algorithm 3, however some tasks will already have attributes (offsets and deadlines) brought about from distributed transactions. In these cases, the following rules are applied when performing uniprocessor task attribute assignment: *the offset of a task can never be reduced and the deadline of a task can never be increased*. Figure 2 also shows the three main features related to distributed task attribute assignment, which are:

1. expansion of the approach to allow for multiple components, i.e. more than one processor and databus,
2. the feedback of schedulability analysis results back into the uniprocessor attribute assignment, and
3. that the distributed schedulability analysis is carried out until convergence is achieved.

4.2.1 Defining the Composite Task

The motivation behind the composite approach is that if a composite task with zero offset can represent the tasks with non-zero offsets and the same period, then the analysis presented in section 2.1 may be used. The benefit of the composite task approach is that the computational complexity is kept sufficiently low, while allowing the resource to be spread through time. The following are the steps followed to derive each composite task needed.

1. A composite task is created for each period where there are tasks with non-zero offsets.
2. Define a set ST that consists of the tasks with the same period and non-zero offsets, and a maximum of one arbitrarily chosen task (if one exists) with the same period and zero offset.
3. Define a set ST1 that consists of the tasks with the same period and non-zero offsets. If the composite task represents a task with zero offset, then a task is added to set ST1 with the same period as the other tasks and an offset equal to its period.
4. A worst-case execution time is assigned equal to the maximum execution time of any task in the set ST.
5. A set ST2 is defined which is an ordered (by increasing value of offset) version of set ST1.
6. A set ST3 is defined with the same order as set ST2, but with value equal to the offset of the member in set ST2 divided by the index (range 1..N) into the set.
7. The composite task is given a period equal to the minimum value of any member in the set ST3.
8. The composite task is given a deadline equal to the minimum relative deadline for any task in the set ST. The minimum relative deadline is classed as the task's deadline minus the task's offset.
9. Replace the tasks that form the composite task in the task set to be analysed (i.e. the members of set ST) with the composite task.

One of the principal benefits of the technique is the fact the existing analysis can be used with a limited amount of pre- and post-processing. There are a number of benefits of being able to reuse the analysis, including the fact existing tools and training can still be used as well as only a limited amount of extra certification evidence is needed.

4.2.2 Use of The Free Variable Argument

The rigorous treatment of offsets is a new area that allows scope to optimise the chosen timing constraints. Consider a distributed transaction requirement specified for two tasks. As long as the necessary precedence is maintained for the two tasks and the second task meets its required deadline, then the size of the offset may be increased. Therefore, it is proposed that to some extent offsets and deadlines are treated as free variables to allow the analysis to be improved.

The approach that could be taken is to treat the tasks with offsets in a similar fashion to allocating tasks in the cyclic schedule. For N tasks on the same processor (each having a period of T) that require offsets, the following offsets would be assigned to tasks:

$$0, \frac{T}{N}, \frac{2T}{N}, \cdots, \frac{(N-1)T}{N}$$

The free variable argument could result in tasks having greater offsets than necessary. However this could provide better analysis results because the composite offset analysis can be particularly pessimistic if tasks have small non-zero offsets. A constraint over the assignment of offsets to tasks is that the offsets should only ever be increased. In the event that the available offset values provided are not sufficient (i.e. the constraint that offsets can't be reduced is broken) to map the timing requirements to tasks then in effect the number of slots could be increased. Therefore, the choice of offsets is as below, where M is greater than N:

$$0, \frac{T}{M}, \frac{2T}{M}, \cdots, \frac{(M-1)T}{M}$$

If the messages on the databus given in Table 1 have the free variable argument applied with M=N=2, this would result in the messages M10 and M20 having offsets of 0 and 12500 respectively. The choice of deadline for the composite task

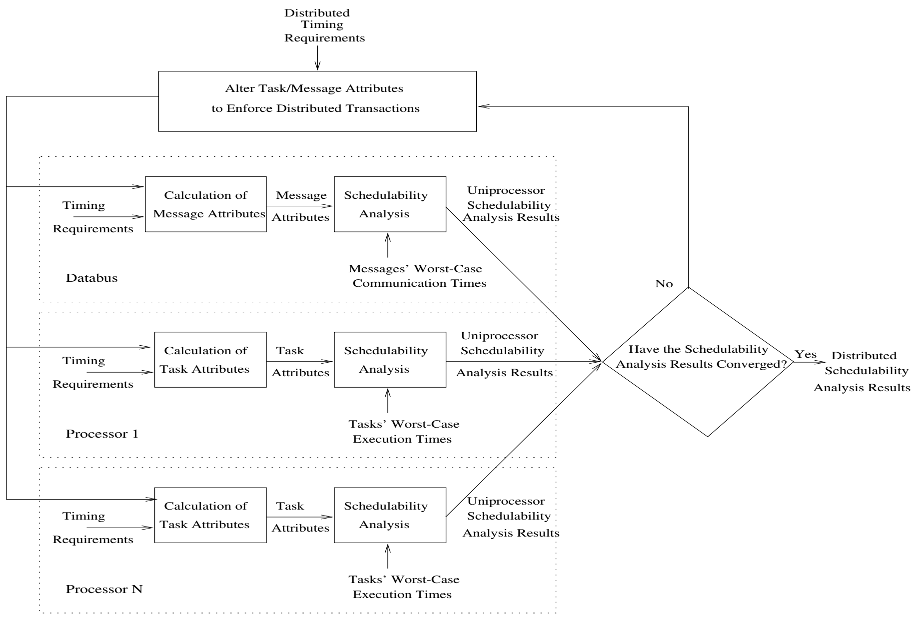

Figure 2: The Approach to Meeting the Timing Requirements in Distributed Systems

is maximised if a task's deadline is equal to the offset of the task with the next largest offset in the transaction. It should be noted that, coincidently the offsets and deadlines chosen with the free variable argument for tasks 10 and 20 are the same as those needed to satisfy their jitter requirements, i.e. $O'_{10} = 0$, $D'_{10} = 12500$, $O'_{20} = 12500$ and $D'_{20} = 25000$.

In our scheduling scheme, the free variable argument is also applied at the distributed systems level. Therefore for transaction A, $M = N = 5$ which would lead to $O''_{10} = 0$, $D''_{10} = 25000/5$, $O''_{M10} = 25000/5$, $D''_{M10} = 2 * (25000/5)$ and so on. The final values of the offsets and deadlines chosen must comply with the rule that offsets are only ever increased and deadlines decreased, therefore the larger values are chosen, i.e. $O^{final}_{10} = 0$, $D^{final}_{10} = 25000/5$ and so on.

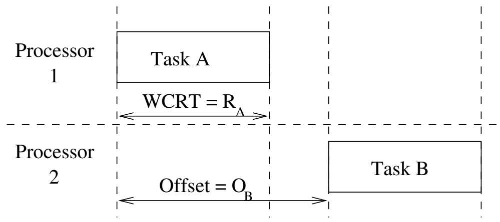

Figure 3: Phasing of Tasks on Different Processors

One of the benefits of the free variable argument is the robustness to change, which enhances maintainability by saving verification time. The reason is that within defined bounds a change on one processor does not necessitate a system wide re-verification.

Figure 3 illustrates how the timing characteristics of processor 1 may be modified until $R_A \geq O_B$, without affecting the scheduling of processor 2, since $D_A = O_B$. Therefore, if the software on one processor is modified, then only that processor needs to be re-analysed as long as its timing requirements are met. When the timing requirements are no longer met, the schedulability analysis and task attribute assignment for the whole system is repeated. Therefore, the partial integration of task attribute assignment and schedulability analysis has helped meet our reduction in pessimism and robust analysis objectives.

Examples and extensive simulation studies have been carried out that demonstrate the approach described in this paper [2]. It is shown that the approach provides significant improvements in effectiveness (i.e. task sets are up to 20% more likely to be schedulable) when compared with the release jitter approach.

4.3 Moving the Example Set of Requirements from Section 2.2 to a Distributed Architecture

The purpose of this section is to show how the timing requirements presented in Tables 1 and 2 can be met in the distributed case. Table 3 presents the results of the analysis for a number of cases, which are: the uniprocessor case, exact analysis, the release jitter approach and the composite approach. When, the schedulability analysis is performed using the exact analysis (column R_E in Table 3) and composite offset (column R_C) approaches, the results show that after the transition to a distributed architecture the system's timing requirements are still schedulable. However, when the release jitter approach is used to analyse the system, the

Id	N	O_S	D_S	R_S	R_J	R_E	R_C
00	0	0	24999	10259	22086	11405	16140
01	0	0	499999	27074	27684	18269	27481
02	0	0	800000	71002	23675	5999	23675
03	0	0	50000	12466	8501	3095	8501
10	1	0	12500	5479	9592	3405	3405
11	1	0	500000	42353	41822	24635	41681
12	1	0	800000	72842	13735	7548	13735
13	1	0	250000	14490	7548	5708	7548
20	2	12500	25000	23666	**27976**	17116	24812
21	2	0	499998	50279	14434	2834	8747
22	2	0	400000	20210	14208	314	8521
23	2	0	650000	62692	14434	3060	8747
M10	N	N/A	N/A	N/A	9995	3605	16605
M11	N	N/A	N/A	N/A	9415	8786	27946
M20	N	N/A	N/A	N/A	22289	11429	3870
M21	N	N/A	N/A	N/A	28087	18448	9212

Table 3: Tasks' Characteristics and Analysis Results

results (column R_J) indicate that task 20 of the system is no longer schedulable, since $R_{20} > D_{20}$.

During the move to a distributed system it has been shown, through example, that introducing more processors can make the same processor load unschedulable with current approaches. However with our approach the example was schedulable.

5 Conclusions

During the course of this paper, techniques have been discussed that are aimed at supporting smooth transitions from cyclically scheduled systems to fixed priority scheduled systems and from uniprocessor to distributed architectures. Further details of the work are contained in [2]. The work has been tailored in order to fulfill a number of objectives. These objectives include:

1. the ability to reuse as much of legacy systems as possible and the associated development process - *periodic task release and non-preemptive scheduling has been used so that the actual software and the process of producing it changes by the minimum possible*,
2. to be able to certify the end product - *the approach comes with analysis that is acceptable to industry and the certification authorities [10]*, and
3. to be able to implement the system's timing requirements as efficiently and effectively as possible - *a method for task attribute assignment has been developed that is tailored for the needs of the application. The resulting analysis has acceptable levels of pessimism, computational complexity and release jitter. For large scale systems development, a greater benefit may be the fact the scheduling analysis is resilient to change within defined bounds.*

Finally the work has achieved its primary aim of being useful to industry. This is demonstrated by Rolls-Royce having used it successfully on a project [10]. As part of their work, tools have been produced that perform the task attribute assignment and schedulability analysis.

References

[1] C. Locke, "Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives," *Real-Time Systems*, vol. 4, pp. 37–53, March 1992. Real-Time Syst. (Netherlands).

[2] I. Bate, *Scheduling and Timing Analysis for Safety-Critical Systems*. PhD thesis, Department of Computer Science, University of York, November 1998.

[3] R. Gerber, S. Hong, and M. Sabsena, "Guaranteeing end-to-end timing constraints by calibrating intermediate processes," *IEEE Real-Time Systems Symposium*, 1994.

[4] R. Yerraballi, *Scalability in Real-Time Systems*. PhD thesis, Computer Science Department, Old Dominion University, August 1996.

[5] J. Gutierrez, J. Garcia, and M. Harbour, "Best-case analysis for improving the worst-case schedulability test for distributed hard real-time systems," in *10th Euromicro Workshop on Real-Time Systems*, pp. 35–44, 1998.

[6] A. Burns, N. Audsley, and A. Wellings, "Real-time distributed computing," *Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, pp. 34–40, August 1995.

[7] J. A. Clark and K. Tindell, "Holistic schedulability analysis for distributed hard real time systems," *Microprocessing & Microprogramming*, vol. 50, pp. 117–134, April 1994.

[8] J. Sun and J. Liu, "Synchronization protocols in distributed real-time systems," in *In The 16th International Conference on Distributed Computing Systems*, May 1996.

[9] J. Y. T. Leung, "A note on preemptive scheduling of periodic real-time tasks," *Information processing Letters*, vol. 11, November 1980.

[10] S. Hutchesson and N. Hayes, "Technology transfer and certification issues in safety critical real-time systems," in *Digest of the IEE Colloquium on Real-Time Systems*, no. 98/306, April 1998.