

# Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System

Kristian Sandström, Christer Eriksson, and Gerhard Fohler  
Department of Computer Engineering  
Mälardalen University, P.O Box 883, 721 23 Västerås, Sweden  
{ksm,cen,gfr}@mdh.se

## Abstract

*The requirements of industrial applications only rarely permit the exclusive use of single paradigms in the development of real-time systems. Product cost, reuse of existing solutions, and efficiency require diverse, or even opposing methods to coexist or to be integrated. In this paper, we deal with one problem encountered during the development of a real-time system for motion control in automotive vehicles, the integration of static scheduling and interrupts. The user mandates pre run-time scheduling for a number of reasons, e.g., predictability, testability and low run-time overhead. However, the interrupt overhead can not be ignored in a safety critical system, and therefor has to be accounted for when creating a static schedule. We propose a method that combines static scheduling and run-time interrupts by applying standard static scheduling techniques and exact analysis. The appropriateness of this method is underlined by successful industrial deployment.*

## 1. Introduction

Requirements of industrial applications only rarely permit the exclusive use of single paradigms in the development of real-time components. Product cost, reuse of existing solutions, and efficiency require diverse, or even opposing methods to coexist or to be integrated. In this paper, we deal with one problem encountered during the development of a real-time system for motion control in automotive vehicles, the integration of static scheduling and interrupts.

This method of managing interrupt and static scheduling has been applied in industry, for development of the control system for automotive vehicles. The real-time system managing the motion of the automotive vehicle consists of two single micro controller units and redundant busses. The application tasks are assumed to be precedence constrained, allow additional synchronisation via mutual exclusion, and to communicate with tasks of the same and different period

times. Communication between interrupt routines and application tasks is indirect via memory only, i.e., there is no direct, synchronised communication. Minimum inter-arrival times for interrupts are known but exact points in time for arrivals are unknown. The particular problem to be solved is to construct static schedules, i.e., start times and completion times for tasks in such a way, that interrupts are accounted for in a safe and efficient way. We will discuss the concrete industrial requirements leading to this problem and our general solution in this paper.

The user mandates pre run-time scheduling for a number of reasons, e.g., predictability, testability and low run-time overhead. However, various hardware architectures rely on the use of interrupts, e.g., to read sensor data and manage bus messages. This company had used a time-triggered architecture for many years. As the constructed systems were quite small and simple, the pre run-time schedules were hand crafted. In this new version of the control system the functionality and complexity has increased dramatically. The consequence of this was that the engineers needed the help of tools for schedule construction. The first version of the tool created the schedules but did not take the effects of interrupts into account. The interrupt overhead was non-negligible and as a result there were a lot of deadline violations during run-time. The first attempt to solve this problem by the engineers was to increase the worst case execution time (WCET) of the tasks by multiplying it by a constant. The result was that fewer, but not all, tasks missed their deadlines. It was not possible to increase this constant so that all tasks met their deadline and still have a schedulable system. The second attempt was to increase the WCET for each task by the worst case interrupt interference. No task would miss its deadline with this approach, but it was impossible to find a feasible schedule for the system due to the pessimistic assumption. Therefor an algorithm that more exactly calculate the worst-case interrupt interference for a given task set was required.

Typically, static scheduling does not account for interrupts, assuming their execution can be ignored or incorporated into task execution times. In a number of applications, the interrupts are, however, non-negligible and inclusion in task execution is too pessimistic and inefficient. Furthermore, as inter-arrival and execution times of interrupts are smaller than the granularity of the online dispatcher and the arrival times are unknown, interrupt-handling routines cannot be modelled as pre scheduled tasks. The application of server algorithms, e.g., sporadic server [4] total bandwidth server [6], and slack stealing [5] are not feasible due to the short response times that are required. The same holds for slot shifting [1], which, in addition, is only reactive, i.e., it is applied to existing schedules, instead of providing guarantees along with schedule construction. Ramamritham has proposed an algorithm that distributes resources during schedule construction [2], but does not give guarantees either.

The key issue for static scheduling accounting for interrupts is the consideration of the overhead. If interrupts occur at run-time, interrupt-handling routines are executed. The delay this poses on task execution is accounted for feasibly in the schedule beforehand. Evidently, an inherent, minimum amount - the worst case penalty - to handle a worst case scenario has to be reserved, according to minimum inter-arrival times and execution times. Any amount exceeding this, however, is overhead imposed by the used method. It is this overhead that has to be kept small for efficient utilisation of the processor. The naive approach of adding a worst case penalty to every task, each may be "hit" by a worst case interrupt arrival, is overly pessimistic. In this industrial application, the naive approach results in a utilisation of **140.1 %**. Instead, our algorithm eliminates consideration of unnecessary penalties by utilising information about task execution behaviour at run-time for a given schedule. We provide analysis to be used during schedule construction, as well as, being applied to already constructed schedules.

Our algorithm enables the co-existence of the seemingly adverse paradigms of static scheduling and interrupts. It determines interrupt overhead during schedule construction in an efficient way and allows the analysis of existing schedules for feasible interrupt handling. The described algorithm is successfully deployed as part of a commercial toolkit for design of embedded real-time systems.

The rest of this paper is organised as follows: section 2 describes the industrial application and its requirements and section 3 describes the task model and

the representation of a schedule. The proposed algorithm and its applicability are described in section 4. Section 5 summarises the paper.

## 2. Application characteristics

The application is a vehicle control system with high demands on safety, reliability, and timeliness. The hardware in the system consists of a number of nodes that are connected via redundant buses. The application contains tasks, running at different period times, which collaborate to perform a certain control function. The system contains about 80 tasks with well-defined functionality allocated to two nodes. Each node is very I/O intensive. The complete system has about 150 I/O channels connected to it.

The application contains tasks with three different period times: 10 milliseconds, 50 milliseconds, and 100 milliseconds. A few tasks require longer period times, in these cases a task is executed with a period time of 100 milliseconds and the actual period time is controlled by an internal counter. The reason for this construction has been to keep the size of the schedule at an appropriate level. The execution times of the tasks in the application range from about 10  $\mu$ s to 1 millisecond. To be able to fulfil the jitter requirements a clock tick resolution of one millisecond is used. The requirements on jitter for I/O are between 2 and 20 milliseconds depending on the rate of data. Due to the fact that a majority of all tasks have an execution time that is less than one clock tick it is of vital importance that the scheduler can schedule several tasks within one clock tick. As a consequence the dispatcher has to support switching between these tasks without the occurrence of a clock tick (see Section 3).

The application is, due to the construction of the hardware, quite interrupt intensive. As this application has a number of interrupts and the effect these interrupts have on the timing of the tasks has to be taken into account when constructing the schedule. Hence, the minimum inter-arrival time and duration of the interrupts are specified in the design. The minimum inter-arrival times for the interrupts are 250  $\mu$ s, 500  $\mu$ s, and 1 millisecond.

The worst case utilisation of the processors for the critical part is around 80%. Divided into 35% for interrupts and 45% for application tasks. The spare capacity left is used by soft real-time tasks. During run-time the spare capacity will be more than the remaining 20% if the load is less than the worst case.

### 3. Task model and run-time representation

The task model allows a number of requirements to be expressed. For each task the following temporal requirements have to be specified:

- Period time
- Release time (relative the start of the period)
- Deadline (relative the start of the period).
- Worst case execution time

Relations between tasks can be specified by:

- Precedence relationships
- Mutual exclusion relationships (shared resources)
- Communication
  - Synchronous communication (communication between task that have same period time)
  - Asynchronous communication, i.e., communication between tasks running with different period times. The semantics of asynchronous communication is that messages are transferred with the lowest frequency of the sender and the receiver

In addition, all interrupts have to be specified by minimum inter arrival time and worst case execution time of the interrupt routine.

When creating a schedule, not only these requirements have to be taken into account, but also the run-time representation of the schedule. A common representation of a static schedule is a vector, where one position in the vector represents a discrete point in time at which the execution of a task can start. The granularity of time has to be matched with the frequency of the periodic clock that drives the dispatcher, which will execute the tasks according to the schedule. If the execution time of a task is less than this granularity, or if it exceeds a multiple of the granularity with a small fraction, then the utilisation of the CPU resource will decrease. This because there will be time intervals that can not be used to execute tasks. An apparent solution to this is to increase the granularity (frequency) of the periodic clock. However, with a higher frequency of the clock the dispatcher will instead use more of the CPU resources, since it will execute more often.

Another way of representing a schedule is as a matrix, where each row represents a point in time at which the dispatcher is to start the execution of a sequence of several tasks. The first task in this sequence,

or *chain*, is started at the given point in time. All other tasks in the chain are started as soon as the preceding task in the sequence has completed its execution, without need for the clock to trigger the dispatcher. This representation will allow several tasks to be executed during an interval less than the period time of the dispatcher clock. Hence, the dispatcher overhead can be kept low at the same time as the utilisation of the CPU resource is high.

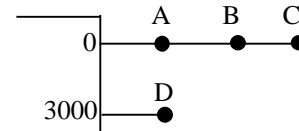


Figure 3.1. The representation of a schedule.

The chains in the matrix are ordered with ascending start times. In Figure 3.1 we see two chains: one starting at time zero including task A, B and, C, and the other starting at time 3000, including only task D. Task B and C is defined as *chain successors* to A. If pre-emption of tasks is allowed, the schedule can be constructed so that a chain can be pre-empted by another chain that has a later start time. As soon as the last task in that chain completes, the pre-empted chain will resume its execution. It is the task of the static scheduler to construct chains that unconditionally will fulfil all the requirements put on the task set.

The latter representation is used by the run-time system that is used together with the method described in this paper.

The following rules apply to the construction of chains:

- The start time of a chain has to be a multiple of the OS clock tick.
- Consecutive chains have ascending start times.
- The earliest start time of a task is the start time of the chain. The reason for this is that the minimal execution times are unknown, and therefor is assumed to be zero.
- The latest completion time of a task  $T$  is the sum of the worst case execution time ( $WCET$ ) of all chain predecessors of the task, plus the  $WCET$  of the task itself. If the task  $T$  or any of its chain predecessors is pre-empted by another chain, the  $WCET$  of all the tasks of that chain have to be taken into account when calculating the scheduled completion time for task  $T$ .
- A task  $T$  might be pre-empted if the scheduled completion time of task  $T$  is greater than the start time of a succeeding chain. A succeeding chain is a

chain that has a start time greater than the start time of the chain that task  $T$  is part of.

- If pre-emption is not desired the schedule is constructed in such way that the last task of a chain always has a scheduled completion time that is less or equal to the succeeding chain's start time and thus pre-emption can not occur.

#### 4. Algorithm

The algorithm has to introduce as little additional overhead as possible. One way of doing this, instead of penalising each task, is to make use of the fact that tasks are executed in chains. For example, if  $n$  tasks are executed after each other in the same chain they can, from an algorithmic perspective, be viewed as one single task. We can then calculate the completion time of the  $n$ 'th task with exact analysis [3] for one task having an execution time of the sum of the  $n$  tasks. The critical instant will be at the start time of the chain and the interrupts is regarded as higher priority tasks. This is repeated for task  $n-1$  disregarding task  $n$  and so on. The advantage of this approach is that we can have a single critical instant for all the  $n$  tasks. In this way, the total penalty for all tasks will be lower than the naive approach in most cases (never higher).

If pre-emption is considered things become more complicated. For a pre-empted chain we have to consider the delay introduced by the pre-emption. We also have to take into account the interference, from interrupts, put upon both the pre-empting chain and the pre-empted chain. With a critical instant, as before, at the start time of the chain, we can calculate which task that will be pre-empted, and where during its execution. We then insert the pre-empting chain at this point and continue calculating the interference on the rest of the chain including the pre-empting chain.

More precisely, the effect that the interference of an interrupt has on a specific task  $T$  can be split into two different cases that are of interest:

1. *The interrupt hits the task  $T$  or a task that is a chain predecessor.*

Assume that task  $T$  is hit by an interrupt. The effect this will have on task  $T$ 's completion time, will be the same as if task  $T$ 's execution time would be prolonged with the execution time of the interrupt routine. As a consequence, the completion time of all tasks that are chain successors, to the task that were hit, will be delayed with the same amount. Conclusion; if task  $T$  or a task that is a chain predecessor is hit by an interrupt the *direct* worst

case delay is the *WCET* of the interrupt routine. See Figure 4.1.

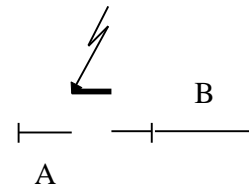


Figure 4.1. The interference of interrupts.

2. *The interrupt hits a task that is executing in a chain that has pre-empted task  $T$  or a chain predecessor to task  $T$ .*

The chain that is hit by the interrupt will be effected according to case 1. This will delay the resume point of the pre-empted chain. Therefore task  $T$  and all its chain successors will be delayed by the *WCET* of the interrupt routine. Conclusion: the *direct* delay of task  $T$  will be the *WCET* of the interrupt routine.

The term *direct* delay is used because a delay of a task could lead to a pre-emption that would not normally occur if the task were not delayed. See Figure 4.2.

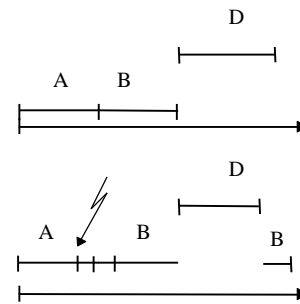


Figure 4.2. An interrupt may cause pre-emption.

It should be noted that in case 2, the pre-empting chain could in itself be pre-empted. Though, all such cases could be explored with a combination of case one and two. In all other cases, the task, which we considered, will not be affected.

#### 4.1. Calculation of worst case completion time

Assuming that chains are enumerated and that tasks in a chain are enumerated in ascending order, with the first task in the chain numbered one. For a given task  $i$  in a chain  $ch$  the worst case completion time, relative the start time of the chain, is then given by:

$$R_i = \sum_{n=1}^i C_n^{ch} + \sum_{\forall p \ni ic(R_i)} \sum_{m=1}^{nofTask(p)} C_m^p + \sum_{\forall interrupt} \left[ \frac{R_i}{T_{interrupt}} \right] C_{interrupt}$$

Where:

- $C_b^a$  denotes the WCET of task  $b$  in chain  $a$
- $ic(R_i)$  denotes all chains  $p$  that conforms to:  $(p \neq ch) \wedge startTime(ch) < startTime(p) \wedge (R_i + startTime(ch))$   
This is all chains that pre-empt task  $i$  or any of its chain predecessors.
- $nofTask(p)$  is the number of tasks in chain  $p$ .

To show whether this analysis is correct, a general chain configuration could be expressed as a set of tasks conforming to exact analysis. If, for this task set, the exact analysis could be written as the equation above, then this equation can rely on the proof of exact analysis [3]. This proof is available in the appendix.

#### 4.2. Applying the algorithm

Below is a pseudo code description of an implementation of the algorithm. The objective is to calculate the completion time for a task  $i$ , residing in a chain  $ch$  with start time  $t$ . Given is a set of interrupts with minimal inter arrival time and WCET.

Let  $TaskInterference$  be the sum of the WCET of all tasks that interfere with task  $i$ . Tasks in chains with start time less than  $t$  are of no interest, because they do not interfere with task  $i$ .

Let  $hChains$  be all chains that have a start time greater than  $t$ .

1.  $TaskInterference = WCET_i + \sum WCET$  of all tasks that precede  $task_i$  in chain  $ch$ .
2.  $R_i^0 = TaskInterference$ .
3. **For** every chain,  $h_{chain}$ , in  $hChains$  **do**  
**if**  $R_i^n + t > startTime(h_{chain})$  **then**  
 $TaskInterference = TaskInterference + \sum WCET$  of all tasks in  $h_{chain}$   
 $R_i^n = R_i^n + \sum WCET$  of all tasks in  $h_{chain}$ .  
Remove  $h_{chain}$  from  $hChains$   
**EndIf**

4.  $R_i^{n+1} = TaskInterference + \sum_{\forall interrupt} \left[ \frac{R_i}{T_{interrupt}} \right] C_{interrupt}$   
**if**  $R_i^{n+1} + t_{chain} > deadline(task_i)$  **then** abort.  
**Else if**  $R_i^{n+1} = R_i^n$  **and**  $R_i^{n+1} + t_{chain} < start$  time of all chains in  $hChain$  **then**  
 $R_i^{n+1} + t$  is the latest completion time of task  $i$ .  
**Else** go to step 3.

When the algorithm is incorporated into the scheduling phase, this algorithm is used while constructing the schedule. This means that some of the chains in step 3 might not exist from start when calculating the completion time of  $task_i$ . Instead they will "appear" as the scheduling proceeds. If a new chain, that pre-empts  $task_i$ , is created the algorithm has to be applied to that chain in a hierarchical fashion. The scheduler tries to put as many tasks as possible in the same chain, as this will decrease the run-time system overhead.

As an example, assume the following task set:

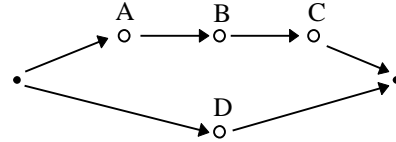


Figure 4.3. Precedence graph of the example.

The arrows in between tasks, in Figure 4.3, denote precedence relationships, the filled circles denotes start and end of the graph.

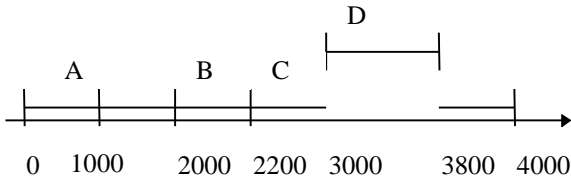
| Task | WCET | Release time | Deadline | Period Time |
|------|------|--------------|----------|-------------|
| A    | 2000 | 0            | 5000     | 5000        |
| B    | 200  | 0            | 5000     | 5000        |
| C    | 1000 | 0            | 5000     | 5000        |
| D    | 800  | 3000         | 4000     | 5000        |

Table 4.1. Specification of tasks.

| Interrupt  | WCET | Minimum inter arrival time |
|------------|------|----------------------------|
| Interrupt1 | 100  | 1000                       |
| Interrupt2 | 100  | 3000                       |

**Table 4.2. Specification of interrupts.**

Assume that the scheduler uses pre-emption and earliest deadline as search heuristic. The clock tick is 1000. A schedule that does not consider the interrupt interference will then look like Figure 4.4.



**Figure 4.4. The schedule for the example.**

If we calculate the worst case completion time for each task, with the analysis presented, the completion times of the tasks will be:

$$R_A^1 = 2000 + 0 + \left\lceil \frac{2000}{1000} \right\rceil 100 + \left\lceil \frac{2000}{3000} \right\rceil 100 = 2300$$

$$R_A^2 = 2000 + 0 + \left\lceil \frac{2300}{1000} \right\rceil 100 + \left\lceil \frac{2300}{3000} \right\rceil 100 = 2400$$

$$R_A^3 = 2000 + 0 + \left\lceil \frac{2400}{1000} \right\rceil 100 + \left\lceil \frac{2400}{3000} \right\rceil 100 = 2400$$

Task A is the first task in the chain and the first term will therefore be equal to the wcet of A. There is no task pre-empting A, hence, the second term is zero. The total worst case interference from the interrupts is 400 and the worst-case completion time of A is 2400.

$$R_B^1 = (2000 + 200) + 0 + \left\lceil \frac{2200}{1000} \right\rceil 100 + \left\lceil \frac{2200}{3000} \right\rceil 100 = 2600$$

$$R_B^2 = (2000 + 200) + 0 + \left\lceil \frac{2600}{1000} \right\rceil 100 + \left\lceil \frac{2600}{3000} \right\rceil 100 = 2600$$

The first term for task B is the wcet of task A and B. Note that task B is not “affected” by any interference from the interrupts. This does not mean that an interrupt could not pre-empt B, but if it does, the completion time

of A will be less than the worst case and therefore task B will have an earlier start time.

$$R_C^1 = (2000 + 200 + 1000) + 800 + \left\lceil \frac{4000}{1000} \right\rceil 100 + \left\lceil \frac{4000}{3000} \right\rceil 100 = 4600$$

$$R_C^2 = (2000 + 200 + 1000) + 800 + \left\lceil \frac{4600}{1000} \right\rceil 100 + \left\lceil \frac{4600}{3000} \right\rceil 100 = 4700$$

$$R_C^3 = (2000 + 200 + 1000) + 800 + \left\lceil \frac{4700}{1000} \right\rceil 100 + \left\lceil \frac{4700}{3000} \right\rceil 100 = 4700$$

Since task C clearly will be pre-empted by D, the second term will be the sum of wcet of all tasks in the pre-empting chain, i.e., the wcet of task D.

$$R_D^1 = 800 + 0 + \left\lceil \frac{800}{1000} \right\rceil 100 + \left\lceil \frac{800}{3000} \right\rceil 100 = 1000$$

$$R_D^2 = 800 + 0 + \left\lceil \frac{1000}{1000} \right\rceil 100 + \left\lceil \frac{1000}{3000} \right\rceil 100 = 1000$$

Task D is not affected by any interrupt interfering with the chain ABC, since its start time is absolute and not dependent on A, B, and C. The worst case completion time of D is  $1000 + 3000 = 4000$ .

The utilisation in this example is 94%. The naive approach of including interrupt overhead into task execution times will result in a utilisation of 102%.

## 5. Conclusions

In this paper we presented methods to combine static scheduling and online interrupt handling in the real-time system controlling the motion of vehicles. We have described the real-world application and derived specific requirements. Meeting these and consideration of cost and efficiency necessitate the use of interrupts.

We propose analysis that allows the processing demand of online interrupt requests to be taken into account during schedule construction, i.e., into the timing of task chains. The naive approach of including interrupt overhead into task execution times is prohibitively inefficient. Rather, our analysis limits the amount of penalty to be included for runtime interrupt handling, by identifying task chains affected by worst case interrupt arrival. In this industrial application, our methods result in a schedule size<sup>1</sup> of **74.5 %** of LCM, to be compared with **140.1 %** using the naive approach. A

<sup>1</sup> Schedule size is defined as the percentage of time of the LCM schedule that is allocated to the tasks. Interrupts that occur at instants during the LCM where no tasks are scheduled are not included in the schedule size. Therefore the schedule size can be lower than the utilisation is not strange.

lower bound would be to summarise the *WCET* of all tasks and perform exact analysis on that sum. In this case we would get a schedule size of **72.9** % of *LCM*. This is the number that our method would result in if all tasks would execute in one single chain. Though, this is not possible with the actual specification.

The resulting static schedules allow the coexistence of the seemingly conflicting paradigms of offline schedule construction and online interrupt handling in an efficient way. The appropriateness of our approach is underlined by its successful use in automotive vehicles.

## References

1. G.Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In Proc. 16<sup>th</sup> Real-Time Systems Symposium, Pisa, Italy, Dec. 1995.
2. K. Ramamritham, G. Fohler, and J.-M. Adan. Issues in the static allocation and scheduling of complex periodic tasks, In Proc. 10<sup>th</sup> IEEE Workshop on Real-Time Operating Systems and Software, NY, USA, May 1993.
3. M. Joseph and P.K. Pandya. Finding response times in a real-time system. Comp. J., 29(5). 1986.
4. B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic Task Scheduling for Hard-Real-Time Systems. The Journal of Real-Time Systems 1, 27-60 (1989)
5. J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft aperiodic tasks in fixed priority preemptive systems. In Proc. IEEE Real-Time Systems Symposium. Dec. 1992.
6. M. Spuri and G. C. Buttazzo. Efficient Aperiodic Service under Earliest Deadline Scheduling. In Proc. IEEE Real-Time Systems Symposium. Dec. 1994.

## Appendix

In this appendix we show how a general chain structure can be expressed as a set of tasks conforming to the exact analysis theory. For this task set we will show that the exact analysis can be re-written as the presented formula for analysing interrupt interference.

**Definition:** A chain predecessor to a task *i* is any task that is scheduled to execute before task *i* and is allocated to the same chain.

To calculate the worst case response time for a task *i*, assume the following:

1. A critical instant at the start of the chain that task, *i*, reside within. Any task in a chain is *ready* to execute at the chain start time. If all chain predecessors and all pre-empting tasks execute for

zero amount of time, then the task will start its execution at the start time of the chain.

2. The interference that we have to consider is
  - tasks that precedes task *i* in the same chain.
  - tasks that pre-empt task *i* or any of the chain predecessors to task *i*.
  - Interrupts.
3. All tasks have a period time equal to *LCM*. Task that have a shorter period time, in the specification, than *LCM* will be replaced by *LCM*/period time number of instances of that task. Each instance in considered on its own in the schedule. These instances will be referred to as separate tasks.
4. All task have a deadline less or equal to *LCM* (because all task have deadline less or equal to the period time)
5. Interrupts have arbitrary period times, with known minimal inter arrival time.

To calculate the worst case response time for a task *i*, using exact analysis, we have to know which tasks that have a higher priority than task *i*, *hp(i)*. For task *i*, let:

- A. All chain predecessors to task *i* be in the set of higher priority tasks. Exact analysis assumes a critical instant with all tasks ready at a specific point in time [3]. Hence, the chain predecessors could be modelled as higher priority task in descending order of priority.
- B. All tasks that might pre-empt task *i* or any of its chain predecessors be in the set of higher priority tasks. The critical instant is the worst case, i.e., if a task *i* will fulfil its deadline for a critical instant it will meet its deadline for all other cases to [3]. Thus, if a higher priority task will be ready at a later time, e.g., at the time of the start of a pre-empting chain, task *i* will still meet its deadline. Therefore all of these tasks can be modelled as higher priority tasks.
- C. All interrupts be in the set of higher priority tasks. Interrupt can be modelled as tasks with higher priority than tasks in A and B and with a period time equal to the minimal inter arrival time of the interrupt.

Exact analysis gives us:

$$(1) R_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Equation (2) divides the sum of all higher priority tasks in to two sums. Where  $task(i)$  is the tasks relating to A and B above, the second is the interrupts.

$$(2) \quad \sum_{\forall j \in hp(i)} \left\lfloor \frac{R_i}{T_j} \right\rfloor C_j \leftrightarrow \sum_{\forall j \in task(i)} \left\lfloor \frac{R_i}{T_j} \right\rfloor C_j + \sum_{\forall k \in interrupt} \left\lfloor \frac{R_i}{T_k} \right\rfloor C_k$$

Consider the first sum in equation (2),  $\sum_{\forall j \in task(i)} \left\lfloor \frac{R_i}{T_j} \right\rfloor C_j$ .

This contains all task that apply to A and B. According to assumption 3, all these tasks have a period time equal to  $LCM$ . All tasks have a deadline less or equal to  $LCM$ , i.e., assumption 4. From this follows that the response time has to be less or equal to  $LCM$ . If the response time is greater than  $LCM$  we can stop the calculation<sup>2</sup>, because the deadline is missed. This gives:

$$(3) \quad T_j = LCM \wedge R_i \leq LCM \Rightarrow \frac{R_i}{T_j} \leq 1 \Rightarrow$$

$$\Rightarrow \left\lfloor \frac{R_i}{T_j} \right\rfloor = 1 \Rightarrow \left\lfloor \frac{R_i}{T_j} \right\rfloor C_j = C_j \Rightarrow$$

$$\Rightarrow \sum_{\forall j \in task(i)} \left\lfloor \frac{R_i}{T_j} \right\rfloor C_j = \sum_{\forall j \in task(i)} C_j$$

This simply says that the tasks will only execute once during  $LCM$ , which is what was stated in assumption 3. Inserting the result of equation (2) and equation (3) in equation (1) yield:

$$(4) \quad R_i = C_i + \sum_{\forall j \in task(i)} C_j + \sum_{\forall k \in interrupt} \left\lfloor \frac{R_i}{T_k} \right\rfloor C_k$$

The sum of all tasks can be divided in to two sums. The sums of all chain predecessors to task  $i$ , according to A, and the sum of all tasks that pre-empt task  $i$  or any of its chain predecessors, according to B.

$$\sum_{\forall j \in task(i)} C_j \leftrightarrow \sum_{\forall l \in pred(i)} C_l + \sum_{\forall m \in prem(i)} C_m$$

Where  $pred(i)$  is the chain predecessors according to A and  $prem(i)$  is the tasks in B. If this is substituted in equation (4), this leads us to:

<sup>2</sup> In many cases the calculation could be stopped before the response time reaches  $LCM$ . This because the deadline can be smaller than  $LCM$  and a start time greater than zero.

(5)

$$R_i = C_i + \sum_{\forall l \in pred(i)} C_l + \sum_{\forall m \in prem(i)} C_m + \sum_{\forall k \in interrupt} \left\lfloor \frac{R_i}{T_k} \right\rfloor C_k$$

We will show that the worst-case completion time analysis formula, presented in section 5.1, is equivalent to equation 5.

(6)

$$R_i = \sum_{n=1}^i C_n^{ch} + \sum_{\forall p \in ic(R_i)} \sum_{m=1}^{nofTask(p)} C_m^p + \sum_{\forall interrupt} \left\lfloor \frac{R_i}{T_{interrupt}} \right\rfloor C_{interrupt}$$

The two first terms in equation (5) describes all chain predecessors to task  $i$  and task  $i$ . This is equal to the first term in equation (6).

$$(7) \quad C_i + \sum_{\forall l \in pred(i)} C_l \leftrightarrow \sum_{n=1}^i C_n^{ch}$$

The third term in equation (5) is all tasks that pre-empt task  $i$  or any chain predecessor of task  $i$ . The second term in equation (6) is all tasks in all succeeding chains that has start time less than the response time of task  $i$ , i.e., all tasks that pre-empt  $task\ i$  or any of its chain predecessors. Thus, the two terms are equal.

$$(8) \quad \sum_{\forall m \in prem(i)} C_m \leftrightarrow \sum_{\forall p \in ic(R_i)} \sum_{m=1}^{nofTask(p)} C_m^p$$

The last term in equation (5) and the last term in equation (6) are the same.

$$(9) \quad \sum_{\forall k \in interrupt} \left\lfloor \frac{R_i}{T_k} \right\rfloor C_k \leftrightarrow \sum_{\forall interrupt} \left\lfloor \frac{R_i}{T_{interrupt}} \right\rfloor C_{interrupt}$$

If the terms in equation (5) that are equal to terms in equation (7) to equation (9) is substituted with these, then we have the following equation.

$$R_i = \sum_{n=1}^i C_n^{ch} + \sum_{\forall p \in ic(R_i)} \sum_{m=1}^{nofTask(p)} C_m^p + \sum_{\forall interrupt} \left\lfloor \frac{R_i}{T_{interrupt}} \right\rfloor C_{interrupt}$$

That is, equation (6). This gives that equation (5) and equation (6) are equal.

Relying on the proof [3] of exact analysis the analysis in equation (6) is correct.