Towards Efficient Analysis of Interrupts in Real-Time Systems

Jukka Mäki-Turja, Gerhard Fohler, and Kristian Sandtröm
Department of Computer Engineering
Mälardalen University, P.O Box 883, 721 23 Västerås, Sweden
{jukka.maki-turja, gerhard.fohler, kristian.sandstrom}@mdh.se

1. Introduction

Industrial applications require multiple paradigms for the design of real-time systems. In these systems several activities of different criticality and requirements on response times and granularity must co-exist. In this paper, we address issues of efficient analysis of overheads imposed by interrupts on tasks. While being similar in properties to tasks, interrupts cannot be handled at the same level: Interrupt routines typically have execution times significantly less than the granularity of the online dispatcher. Handling them on task level would result in only a small portion of a CPU scheduling slot being used, thus reducing utilisation severely.

Similar reasoning prohibits the use of polling for interrupt events on the task level: Short interarrival times require either a high frequency for the online dispatcher, resulting in increased scheduling overhead, or prevent timely response.

A naive approach to account for interrupt overhead is to include the execution time of interrupt routines in the execution time of tasks. Since interrupt interarrival times are smaller than tasks' execution time, a number of "hits" has to be included in the analysis incurring an unacceptable analytical overhead.

Many of the algorithms presented in literature, e.g., [1], suffer from the above mentioned shortcomings. Our approach is a modification of exact response time analysis [2], including interrupts as high priority tasks [6]. Instead of assuming all tasks to start at the same time, we use the information about release times and deadlines to see which tasks can possibly interfere with each other. A similar approach has been made in [5] and [7] for FPS, our approach however, utilises fixed release times and deadlines of tasks. The only requirement about task priorities we assume here is that they are known at analysis time. Different instances of tasks can have different priorities, but priorities may not change at run-time.

2. Task model and assumptions

The task model we assume is based on fixed (known a priori) release times (*rt*) and deadlines (*dl*). A task that has its release time fulfilled, can only be delayed by higher priority tasks or interrupts. Periodic tasks are transformed into individual jobs, and treated separately. This means that we have to consider the all instances of the tasks up to least common multiple (*LCM)* of the period times. Since we consider each task instance separately we can have dynamic priorities for tasks, i.e., different instances can have different priorities, e.g., *earliest deadline first (EDF)* [3]. Priorities may however not change at run-time.

Interrupts are modelled as high priority tasks, in the analysis, with a minimum inter-arrival time (T_{int}) and a maximum execution time (C_{int}). However interrupts hits the system at run-time anywhere and does not have to wait for next clock tick to execute. Consequently, a task that got interrupted does not have to wait for next clock tick to resume its execution. The OS overhead for the interrupt is considered to be accounted for in C_{int}. Communication between interrupt routines and application tasks is asyn-chronous via memory only, i.e., tasks and interrupts are temporally independent. Minimum interarrival times and execution times of handling routines for interrupts are known but exact invocation times are unknown.

For the schedule to be feasible, every task must have enough execution resources in its execution window [*rt, dl*], even if it experiences the worst case interrupt interference and interference from higher priority tasks.

3. Analysis

The analysis is based on the response time analysis [2]. The worst case, critical instant, for task *i* occurs when it is released at the same time as all tasks with higher priority. If we have fixed execution windows we know tasks may not be released at the same time. Our method reduces analysis overhead by exploiting this information. The algorithm is based on the assumption that the worst case finishing time for task *i* occurs in only one of the following cases:

1) If task *i* can start its execution directly when its predecessors finishes. We define a predecessor to *i* as a task with higher priority than *i* and a shorter release time than *i*.
2) If task *i* is able to start its execution at its own release time and then with no interference from its predecessors.

This is the essence of the analysis approach. In 1) we treat the task as if it had the same release time as its predecessors. In 2) we have a critical instant at the release time of the task and ignore any interference from higher priority

tasks that precede the task at hand. The two cases are depicted in figure 3.1.

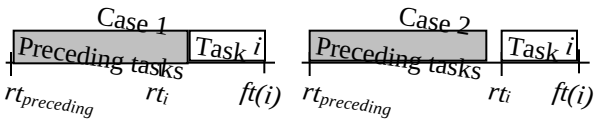

Figure 3.1 Worst case finishing time for task i

Terminology:

rt_i	=	release time of task *i*
dl_i	=	deadline of task *i*
C_i	=	*WCET(i)*
prio(i)	=	priority of task *i* (lower number means higher priority)
ft(i)	=	worst case finishing time of task *i*
$ft_j(i)$	=	finishing time of *i* as if it was released at the release time of task *j* (rt_j)
$R_j(i)$	=	response time for task *i* as if it was released at the release time of task *j* (rt_j)

Analysing task *i*: all tasks with a index less than *i* are those that precede *i*, i.e., $rt_{i-k} < rt_i$ and *prio (i-k) < prio (i)* (that is, *i-k* has higher priority). All tasks with the same release time and higher priority than *i* are regarded as a single task with an execution time equal to the sum of all individual task. This also applies to *i*, i.e., if there are several tasks with the same release time as *i*, those with same or higher priority than *i* are treated as one task. This means that every release time has a unique index. Tasks are ordered in release time order.

$$ft(i) = \max(ft_i(i), ft_{i-1}(i), \cdots, ft_1(i)) \text{ where}$$

$$ft_j(i) = rt_j + R_j(i) \text{ where}$$

$$R_j(i) = C_i + \sum_{int \in interrupt} \left\lceil \frac{R_j(i)}{T_{int}} \right\rceil C_{int} + \overbrace{\sum_{j<k<i} C_k}^{preceding\ term} + \overbrace{\sum_{k \in pre\text{-}empt(R_j(i))} C_k}^{pre\text{-}empting\ term}$$

$$\text{where } pre\text{-}empt(R_j(i)) = \left[k \mid R_j(i) + rt_j > rt_k \quad rt_k > rt_j \quad prio(k) > prio(i)\right]$$

In words, analyse the finishing time (*ft*) for a task *i* by considering every release time of preceding tasks (its own *rt* is included). For each such case, say task *k*, task *i* is considered to be released at rt_k. Interference from predecessors to *k* is ignored. The response time formula states that own execution time, interference from interrupts, predecessors and possible pre-empting tasks is taken into consideration. Pre-empting tasks are those task that have a higher priority than *i*, a later finishing time than the starting point of the analysis (rt_j), and whose release time is less than the finishing time ($R_j(i) + rt_j$) of *i*.

Consider the simplest case where we have only two tasks and some interrupts, as depicted in figure 3.2. *prio(*1*) < prio(*2*),* $rt_1 < rt_2$. *ft(1)* can only be calculated in one way since there are only interference from interrupts (grey areas in the figure), i.e., it has no predecessors. But *ft(2)* is calculated by *max (*$ft_1(2)$*,* $ft_2(2)$*)*, i.e., task 2 is treated as if it had same release time as task *t1* or treating rt_2 as its release but then with no interference from t_1. A sketch of the proof is included in the Appendix.

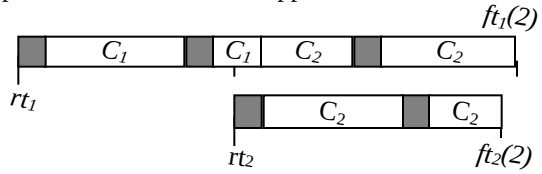

Figure 3.2 Interrupt interference in the two tasks case

If the priority order were reversed neither of the task would have any predecessors. Task t_2 would then possibly be a pre-empting task in the analysis of t_1. If the response time of t_1 goes beyond the release time of t_2, the execution time of t_2 has to be taken into account since it will then pre-empt t_1. The proof still holds since t_2 is taken into account by the pre-empting term instead of the preceding term and the two terms are equal.

3.1. Efficient solving technique

In general, the formula is computationally intractable for very large task sets. In practice, however, one may discard many of the cases, i.e. when calculating *ft(i)*, many $ft_j(i)$ *(j<i)* can be ignored. Furthermore, one does not have to perform the response time analysis from scratch for every $ft_j(i)$, but building on $R_j(i\text{-}1)$ that has already been calculated. $R_j(i)$ is only an extension of $R_j(i\text{-}1)$ with C_i as seen in the formulas below.

$$R_j(i-1) = C_{i-1} + \sum_{int \in interrupt} \left\lceil \frac{R_j(i-1)}{T_{int}} \right\rceil C_{int} + \sum_{j<k<(i-1)} C_k + \sum_{k \in pre\text{-}empt(R_j(i-1))} C_k$$

$$R_j(i) = C_i + \sum_{int \in interrupt} \left\lceil \frac{R_j(i)}{T_{int}} \right\rceil C_{int} + C_{i-1} + \sum_{j<k<(i-1)} C_k + \sum_{k \in pre\text{-}empt(R_j(i))} C_k$$

This means that one can start with $R_j(i\text{-}j) + C_i$ in the first iteration of $R_j(i)$. [7] showed that one can start the iteration with a higher value without jeopardising convergence.

Which cases can be ignored when calculating *ft(i)*? Since we add C_i to $R_j(i\text{-}1)$ for every *j*, we can beforehand predict which cases ($ft_j(i)$) cannot be the maximum in the formula for *ft(i)*. These rules state which cases can be ignored:

(1) If $ft_j(i\text{-}1) \le rt_i$ for every *j*, we only have to consider $ft_i(i)$.

(2) Assume that $ft_j(i\text{-}1)$ is the maximum for *i*-1 for some *j*. We only have to consider those $ft_k(i)$ for which

$$ft_k(i-1) + \sum_{int \in interrupt} C_{int} > ft_j(i-1)$$

$ft_k(i)$ can in the worst case only experience interference from one more instance of every interrupt than $ft_j(i)$, the rate of the interrupts is the same in both cases, only the phasing is different. $ft_k(i)$ can thus not be a candidate for the maximum of *ft(i)*. A detailed proof and discussion can be found in [8].

Method of calculating finishing time for tasks:

1. Calculate the finishing time for all tasks that have no predecessors. We only have to consider the task itself and interrupt interference.

2. Calculate the finishing time for a task for which all predecessors have already been analyzed. Use the response time ($Rj(i\text{-}1)$) previously calculated in the first iteration, i.e., just add C_i to $Rj(i\text{-}1)$. Note that the rules (1) and (2) are used at this stage.
3. If all tasks have been considered and all have a finishing time ≤ deadline, we are done and the schedule is feasible. If a finishing time is later than a deadline we stop and report an infeasible schedule. If there still are tasks to be considered go to 2.

4. Example

Consider the example depicted in 5.1. Execution windows are depicted below the table. Task T_C has a period of 100 so it has two instances in the *LCM*, which is 200.

Task	C_i	T_i	rt_i	dl_i
T_A	20	200	0	140
T_B	25	200	0	140
T_C	20	100	0	50
T_D	15	200	25	75
T_E	50	200	90	200
Int_1	3	25	-	-
Int_2	5	40	-	-

Scheduling strategy: ***EDF***

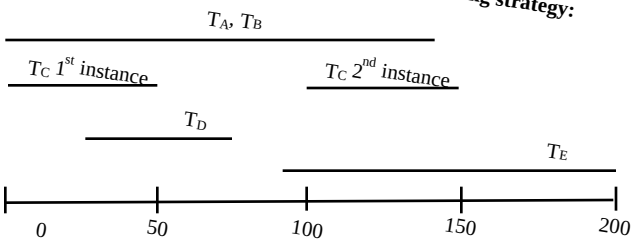

Figure 4.1 Execution windows of all task instances

We start, point 1 in the method, with tasks that have no predecessors: T_A, T_B, and T_C. T_C is treated as a single task since it has highest priority. T_A, T_B are analysed together with T_C, treating them as a single task with execution time $C_A + C_B + C_C$.

$$R_A(C) = C_C + \sum_{int\ interrupt} \left\lceil \frac{R_A(C)}{T_{int}} \right\rceil C_{int} = 20 + 11 = 31$$

$$ft(C) = rt_C + R_C(C) = 31$$

The T_A, T_B and T_C chain starting at $t = 0$ will experience interference from interrupts and be pre-empted by T_D.

$$R_A(A,B,C) = C_A + C_B + C_C + \sum_{int\ interrupt} \left\lceil \frac{R_A(A,B,C)}{T_{int}} \right\rceil C_{int} + C_D = 110$$

$$ft(A) = ft(B) = rt_A + R_A(A,B,C) = 110$$

Point 2. In the method, the only candidate for analysis is now T_D, the only task with all predecessors analysed. We have to consider both $ft_C(D)$ and $ft_D(D)$ since $ft_C(C) - rt_D < 8$ (Sum of all C_{int}).

$$R_C(D) = C_C + \sum_{int\ interrupt} \left\lceil \frac{R_C(D)}{T_{int}} \right\rceil C_{int} + C_D = 20 + 19 + 1 = 54$$

$$ft_C(D) = rt_C + R_C(D) = 0 + 54 = 54$$

$$R_D(D) = C_D + \sum_{int\ interrupt} \left\lceil \frac{R_C(D)}{T_{int}} \right\rceil C_{int} = 15 + 8 = 23$$

$$ft_D(D) = rt_D + R_D(D) = 25 + 23 = 48$$

$$ft(D) = max(ft_D(D), ft_C(D)) = 54$$

Now all remaining tasks (jobs), $T_{C'}$ (' denotes 2nd instance of T_C) and T_E, have all predecessors analysed. The preceding tasks, T_A ,T_B, T_C ,and T_D, finishing time is well beyond $rt_{C'}$, which means we only have to consider $ft_A(C')$. Note that C_C appears twice in the analysis, i.e., both instances.

$$R_i(C') = C_{C'} + \sum_{int\ interrupt} \left\lceil \frac{R_A(C')}{T_{int}} \right\rceil C_{int} + C_A + C_B + C_C + C_D = 138$$

$$ft(C') = rt_A + R_A(C') = 0 + 138 = 138$$

The same thing applies to T_E, with an addition that it will be pre-empted by the 2nd instance of T_C.

$$R_A(E) = C_E + \sum_{int\ interrupt} \left\lceil \frac{R_A(E)}{T_{int}} \right\rceil C_{int} + C_A + C_B + 2*C_C + C_D = 199$$

$$ft(E) = rt_A + R_A(E) = 0 + 199 = 199$$

5. Conclusion

In this paper, we presented methods for efficient analysis of interrupts affecting tasks with fixed release times and deadliness. For analysis purposes, we considered interrupts as high priority tasks and included them in response time analysis. Our analysis reduces analytical overhead by utilising knowledge about release times and deadlines of task to reduce the number of interferences accounted for in the analysis.

We are currently working on improving the efficiency of the analysis further and assess its benefits quantitatively in a concrete industrial application. In particular we plan to find methods to include dynamic arrivals as well.

Acknowledgements

We would like to thank Christer Eriksson and Hans Hansson for reviewing and commenting on the paper. A thanks also goes to Sasi Punnekkat for fruitful discussions.

6. References

1. Kevin Jeffay and Donald L. Stone. *Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems.* IEEE Real-Time Systems Symposium, 1993.
2. M. Jospeh and P. Pandaya. *Finding Response Times in a Real-Time System*. The Computer Journal 29(5), 1986.

3. C. L. Liu and C. L. Layland. *Scheduling Algorithms for Multi-Programming in a Hard Real-Time Environment.* Journal of the Association for Computing Machinery, 1973.
4. A. K. Mok. Fundamental Design Problems of distributed systems for the hard real-time environment. Phd thesis MIT 1983.
5. J.C. Palencia and M. González. *Schedulability Analysis for Tasks with Static and Dynamic Offsets*. IEEE Real-Time Systems Symposium, 1998.
6. K. Sandström, C. Eriksson, and G. Fohler. *Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System*. RTCSA98, Hiroshima, Japan 1998.
7. M. Sjödin an H. Hansson. *Improved response time Analysis calculation*. IEEE Real-Time Systems Symposium, 1998.
8. J. Mäki-Turja and Gerhard Fohler. *Analysis of Interrupts in Real-Time Systems.* Technical report MRTC99RR01, Mälardalen Real-Time research Centre, Mälardalen University, 1999.

7. Appendix

Proof sketch of the two tasks case:

Let K_1 be the maximum possible interrupt interference in $[rt_1, ft_1(2)]$, and K_2 the maximum possible interrupt interference in $[rt_2, ft_2(2)]$.

By the property of response time analysis the following holds:

$$K_1 + C_1 + C_2 = ft_1(2) - rt_1 \quad (1)$$

$$K_2 + C_2 = ft_2(2) - rt_2 \quad (2)$$

In other words, task 1 and 2 exactly fit within $[rt_1, ft_1(2)]$ and task 2 exactly fits within $[rt_2, ft_2(2)]$ if the intervals experiences their worst case hit of interrupts.

We have to prove that task 2 can not finish later than $max(ft_1(2), ft_2(2))$. We divide the proof into 2 cases:

1. $ft_1(2) \geq ft_2(2)$
2. $ft_1(2) < ft_2(2)$

Case 1 $ft_1(2) \geq ft_2(2)$:

How could t_2 finish later than $ft_1(2)$? A task with the execution time $C_1 + C_2$ and release time at rt_1 would always finish before $ft_1(2)$, i.e., if t_1 and t_2 can be considered to have the same release time. But t_2 has an additional constraint: it can not start its execution before rt_2.

So if the interrupts would hit $[rt_1, rt_2]$ so that t_1 is finished before rt_2, t_2 would not be able to start its execution when t_1 finishes. Could then the interrupts hit $[rt_2, ft_1(2)]$ so that C_2 does not fit? That is, can:

$$C_2 + K > ft_1(2) - rt_2$$

be fulfilled for some K? K denotes the possible interrupt interference in $[rt_2, ft_1(2)]$. By combining the above equation with *(2)*, we get

$$C_2 + K > ft_1(2) - (ft_2(2) - K_2 - C_2)$$

Which rewritten looks like

$$K - K_2 > ft_1(2) - ft_2(2)$$

K_2 and K represent the maximum interrupt interference in $[rt_2, ft_2(2)]$ and $[rt_2, ft_1(2)]$ respectively. And thus $K - K_2$ represent the interrupt interference in the interval $[ft_2(2), ft_1(2)]$. The above equation can not possibly hold since there cannot be more interrupt interference in an interval than the length of that interval. Thus there are no worse finishing time for task 2 than $ft_1(2)$.

Case 2 $ft_1(2) < ft_2(2)$:

How could t_2 finish later than $ft_2(2)$? We did not consider any interference from t_1.

Can the possible interference from t_1 prolong the finishing time for t_2? That is, can

$$K + C_1 + C_2 > ft_2(2) - rt_1$$

be fulfilled for some K? K denotes the possible interrupt interference in $[rt_1, ft_2(2)]$. By combining the above equation with *(1)*, we get

$$K + C_1 + C_2 > ft_2(2) - (ft_1(2) - K_1 - C_1 - C_2)$$

Which rewritten looks like

$$K - K_1 > ft_2(2) - ft_1(2)$$

K_1 and K represent the maximum interrupt interference in $[rt_1, ft_1(2)]$ and $[rt_1, ft_2(2)]$ respectively. And thus $K - K_1$ represent the interrupt interference in the interval $[ft_1(2), ft_2(2)]$. The above equation can not possibly hold since there cannot be more interrupt interference in an interval than the length of that interval. Thus there are no worse finishing time for task 2 than $ft_2(2)$.

❑

A detailed discussion and the general proof can be found in [8].