Calculating Exact Worst Case Response Times for Static Priority Scheduled Tasks with Offsets and Jitter

Ola Redell and Martin Törngren
{ola, martin}@md.kth.se
Mechatronics Lab., Dept. of Machine Design, KTH, 100 44 Stockholm, Sweden

Abstract

A method to perform exact worst case response time analysis for fixed priority tasks with offsets and release jitter is described. Available methods are either pessimistic or inefficient as they incorporate numerous time consuming schedule simulations in order to find the exact response times of tasks. The method presented here is based on the creation of the worst case conditions for the execution of each individual task instance within a hyperperiod. The worst case is built by choosing release jitter of higher priority tasks appropriately. Given these conditions, the corresponding task instance response time is calculated using partly iterative algorithms. Experiments comparing the efficiency of the proposed method to an alternative exact method based on schedule simulation show that the new method outperforms the latter. The analysis is expected to be particularly useful when analysing response times and schedulability of tasks that form transactions in distributed systems.

1. Introduction

In this paper we present a method for exact analysis of worst case response times for fixed priority scheduled task sets with offsets and release jitter. The method is based on separate analysis of each individual task instance within a hyperperiod (the LCM - least common multiple of all task periods). Even though time consuming, this is the only current way to calculate exact response times since the worst case for each instance has to be created individually. We show by experiment that the presented method performs much better than traditional exact analysis based on simulation, when release jitter is considered.

Exact response time analysis of task sets in which the task phasings are given by offsets is in general much more complex than the analysis of arbitrarily phased tasks. This is a result of the difficulty to identify the critical instant for a task. A critical instant is the instant at which a task should be released in order to experience its longest response time. In a task set with arbitrary phasings, the critical instant for a task occurs when a task is released at the same time as all the higher priority tasks, [5]. It is pessimistic to assume that this situation will occur in systems in which the task phasings are fixed by offsets. Since the critical instant for a task in an offset related task set is difficult to identify, the worst case response time is found by analysing the response times for each task instance within a hyperperiod. The maximum response time of these instances is the worst case response time of that task.

The analysis of response times of task instances can be performed through schedule simulation. This method has been discussed by e.g. Leung and Whitehead, [4], and Audsley [1]. The latter showed that the minimum interval for simulation of task sets with no release jitter is an interval as long as the hyperperiod. Schedule simulation has to our best knowledge not been applied to task sets with release jitter before. That extension is however quite straightforward: A worst case has to be built, and a schedule simulation be performed, for each task instance within the hyperperiod.

In [11], Tindell discusses how task offsets can be used to model transactions of tasks. In Tindell’s model, static offsets limited by the task periods, are a way to handle precedence relations between tasks. Even though offsets are used to define the time from the start of a transaction to the request of the tasks within that transaction, the analysis is based on the assumption that different transactions are arbitrarily
phased. I.e. the external events that trigger the transactions are arbitrarily phased. This simplifies the analysis, since critical instants can more easily be constructed for tasks, but it also makes it pessimistic if applied to a task model in which also the transactions have fixed phasings. Conditions like these are likely to appear when many transactions are triggered from the same processor, from processors with synchronised clocks, or even from the same external event.

The task model used by Tindell also allows tasks to experience release jitter limited by the task periods: In some task models, tasks are not assumed to be placed in the schedulers ready queue at the instant they arrive. In such models tasks may experience some delay called release jitter before being released (placed in the ready queue). Audsley et al. developed a method to take jitter into account in the analysis of tasks with arbitrary arrival times [2]. Release jitter can be used to model delays within the kernel due to the kernel tick period or tasks that suspend themselves at the start when waiting for e.g. network messages to arrive [7].

In [6], Palencia and Harbour developed Tindell’s method further. Their analysis handles task sets with arbitrary offsets and jitter and they also show how the method can be used to analyse tasks with pure precedence relations. Precedence relations are modelled using offsets and jitter that are dynamic in the sense that they vary during the course of the analysis. In Palencia and Harbour’s method the transactions are still assumed to be arbitrarily phased.

Offsets are commonly used in embedded systems for e.g. automatic control, to help reduce output jitter for functions performing actuation and to enforce precedence relations between tasks. Furthermore, since offsets can sometimes be used to increase the schedulability of task sets (when compared to arbitrarily phased ones), it is important to exploit this property in the analysis. The methods described above do that to some extent in that they handle offsets between tasks that belong to the same transactions. To fully exploit the advantages of offsets however, new methods need to be developed. This paper describes one such new method that promises to be useful since it outperforms the traditional schedule simulation when release jitter is included in the task model. It does not however remove the inherent complexity of analysis of offset related task sets. Analysis of large task sets with many task instances within the hyperperiod is still a problem.

2. Problem formulation

In this paper we study fixed priority scheduling on a uniprocessor where tasks arrive periodically and have fixed arrival times (offsets). Tasks are numbered in priority order with the smallest number given to the task with highest priority. Task \( i \) is denoted \( \tau_i \) while \( \tau_{ik} \) denotes the instance of \( \tau_i \) that arrives at

\[
a_{ik} = k \cdot T_i + O_i \quad k \text{ integer}; \quad k \geq 0
\]

where \( a_{ik} \) is the arrival time, \( T_i \) is the period of the task and \( O_i \) denotes the offset. We will assume, without loss of generality, that the offsets are smaller than the task periods (\( O_i < T_i \)). If larger offsets are used, these can be normalized such that \( O_i^{\text{normalized}} = O_i \mod T_i \), and the analysis applied to the normalized task set. This is possible since different instances of the same task are indistinguishable from each other.

Each task instance has an execution time within the interval \([c^\text{min}_i, c^\text{max}_i]\) and a delay from arrival to release, a release jitter, within \([0, J_i] \cdot J_i \), which is the maximum release jitter for \( \tau_i \), may be arbitrarily large. The release time of \( \tau_{ik} \) is denoted \( r_{ik} \) and must hence be in the interval \([a_{ik}, a_{ik} + J_i] \).

Each task is associated with a deadline, \( D_i \), that may be arbitrarily large. This means that many instances of the same task can be active (in the ready queue) at the same time. We assume that the scheduler handles tasks with the same priorities using a FIFO rule. Hence an early instance of a task has priority over a later, and must be completed before the later instance is allowed to start.

We further assume that the total utilisation of all tasks, \( U \), is strictly less than 1. This will later be shown to be a necessary condition for the analysis.

Tasks may use shared resources protected by some mutual exclusion mechanism, provided that the protection follows a protocol such that a worst case blocking time \( B_i \) can be derived for each task, e.g. the priority ceiling protocol [9].

The aim is to find a method to exactly calculate the worst case response times of the tasks without performing a schedule simulation. Schedulability can then be proven by comparing the calculated worst case response times to the task deadlines. The worst case response time (the time between arrival and completion), \( R_i \) of a task \( \tau_i \) will be derived by comparing the worst case response times of all instances of \( \tau_i \) during a hyperperiod as long as the LCM.
For notational convenience, and to simplify the description of the analysis, some further notations are needed. The analysed task will be called task \( i \) (\( \tau_i \)) and the currently studied instance of that task will be called \( \tau_{ik} \). Any higher priority task will be called task \( j \) (\( \tau_j \)). If this does not apply, it will be clearly stated in the text. Furthermore, the instance of \( \tau_j \) that arrives first after (or at) the release of \( \tau_{ik} \) will be denoted \( \tau_{jp} \). The following instance is \( \tau_{(p+1)} \), and the preceding is \( \tau_{(p-1)} \) etc.

The analysis will be performed by studying the level-1 busy period in which \( \tau_{ik} \) executes. It is important to notice that we adopt the definition of a busy period given by e.g. Sun and Liu [10]. In their definition a level-1 busy period starts and ends with a level-1 idle point as opposed to the idle intervals that form the limits for a busy period in the definition by Lehoczky [3].

3. Response time analysis

The analysis described here is in many parts similar to the analysis derived by Palencia and Harbour in [6]. The difference lies in our assumption that all tasks have fixed phases (in terms of arrival times) to each other. The major contribution in the derivation described here lies in the analysis of the interference from higher priority tasks that are released prior to the release of the analysed task instance. That part of the analysis is derived in 3.2.

The method for deriving the worst case response time for a task is based on the calculation of worst case response times for all the individual task instances within a full LCM. To find these, the worst case for each instance \( \tau_{ik} \) has to be created. Since the arrival times of all tasks are set, only the release jitter of the analysed task itself and all higher priority tasks can be selected to make the response time of \( \tau_{ik} \) the longest.

In order to create the worst case for \( \tau_{ik} \), we need to separate all task instances that may interfere with \( \tau_{ik} \) into three sets:

- **Set 0** (\( S_0 \)) includes all instances of higher or equal priority tasks (hence including \( \tau_j \)) that arrive before \( r_{ik} \) and that cannot be released at \( r_{ik} \) even if they experience their maximum jitter.
- **Set 1** (\( S_1 \)) is the set of instances of higher priority tasks that arrive before \( r_{ik} \) but can be delayed by release jitter to be released at \( r_{ik} \).
- **Set 2** (\( S_2 \)) is the set of instances of higher priority tasks that arrive at or after \( r_{ik} \).

**Theorem 1.** Worst Case for \( \tau_{ik} \)

The worst case response time for \( \tau_{ik} \) is achieved when it experiences maximum release jitter and when the following is satisfied:

1. All instances belonging to \( S_0 \) experience their maximum release jitter.
2. All instances belonging to \( S_1 \) experience jitter such that they are released at \( r_{ik} \).
3. All instances belonging to \( S_2 \) are released immediately on arrival, without any release jitter.

**Proof.** If \( \tau_{ik} \) does not experience its maximum jitter, the response time of \( \tau_{ik} \) will not decrease if its release is delayed by increasing the release jitter. The response time will either increase or stay the same when the jitter is increased. Hence maximum jitter will generate the longest response time for \( \tau_{ik} \).

1. If an instance \( \tau_{jm} \in S_0 \) does not experience its maximum jitter, the probability of interfering with the execution of \( \tau_{ik} \) can only increase by increasing the amount of jitter. The maximum probability for interference occurs when the release jitter of \( \tau_{jm} \) is maximum.
2. If an instance \( \tau_{jm} \in S_1 \) does not experience jitter enough to be released at \( r_{ik} \), its interference with \( \tau_{ik} \)’s execution can only be increased by increasing the jitter. If \( \tau_{jm} \) is released at \( r_{ik} \), its interference with \( \tau_{ik} \) cannot be increased by further increase of the jitter. Its interference with \( \tau_{ik} \) will stay the same or vanish completely. Therefore, to create the worst case response time for \( \tau_{ik} \), any \( \tau_{jm} \in S_1 \) should experience jitter such that it is released at \( r_{ik} \).
3. The probability for an instance \( \tau_{jm} \in S_2 \) to interfere with the execution of \( \tau_{ik} \) increases the closer to \( r_{ik} \) it is released. The maximum probability is achieved when the jitter is the smallest.

Q.E.D.

In order to find the worst case response time of \( \tau_{ik} \), the interference by higher priority tasks needs to be derived based on Theorem 1. Figure 1 shows an execution of a low priority task \( \tau_j \) together with one higher priority task \( \tau_j \) whose instances experience jitter according to Theorem 1. Vertical arrows are used to show the arrival of the instances of \( \tau_j \) while the dashed horizontal lines show how the releases are delayed by jitter. Execution of instances of \( \tau_j \) are shown as shaded rectangles while the execution of \( \tau_{ik} \) is shown using white rectangles. In Figure 1, \( \tau_{(p-3)} \) and earlier instances belong to \( S_0 \); \( \tau_{(p-1)} \) and \( \tau_{(p-2)} \) belong to \( S_1 \); and \( \tau_{(p)} \) and later instances belong to \( S_2 \).

To create an expression for the worst case response time of \( \tau_{ik} \), we define \( W_{ik} \) to be the complete amount of level-1 workload that has to be executed from \( r_{ik} \)
until \( \tau_{ik} \) finishes. Notice that this may include workload by task instances released prior to \( r_{ik} \). Hence \( W_i^C \) is the end of the level-\( i \) busy period in which \( \tau_{ik} \) executes. The busy period may start before \( r_{ik} \). \( W_i^C \) is illustrated in Figure 1. If \( W_i^C \) can be derived, the worst case response time, \( R_{ik} \) of \( \tau_{ik} \) is given by

\[
R_{ik} = W_i^C + J_i
\] (2)\

We note from the derivation of Theorem 1 that \( W_i^C \) can be expressed as follows:

\[
W_i^C = B_j + C_{ij}^{\text{max}} + I_0(\tau_{ik}) + I_1(\tau_{ik}) + I_2(\tau_{ik})
\] (3)

where \( B_j \) is the maximum blocking time of \( \tau_j \); \( C_{ij}^{\text{max}} \) is its maximum execution time; and \( I_0(\tau_{ik}) \), \( I_1(\tau_{ik}) \) and \( I_2(\tau_{ik}) \) represent the interference from the task instances belonging to \( S_0 \), \( S_1 \) and \( S_2 \) respectively. In order to find the worst case response time of \( \tau_{ik} \) we need to derive methods to compute these different parts of \( W_i^C \).

### 3.1. Interference by tasks released at \( r_{ik} \) or later

The interference represented by \( I_1(\tau_{ik}) \) and \( I_2(\tau_{ik}) \) in (3) belong to higher priority task instances that are released at \( r_{ik} \) or later. In this section these interferences will be examined. We begin by finding the number of instances of a task \( \tau_j \) that belong to \( S_k \). This leads to a simple expression for \( I_1(\tau_{ik}) \). The corresponding expression for \( I_2(\tau_{ik}) \) is easily derived and is simply stated in the end of this section.

To find a time relation between \( \tau_{ik} \) and the instances of \( \tau_j \), we define the time from \( r_{ik} \) to \( r_{jp} \) (the release time of the first instance of \( \tau_j \) to arrive at or after \( r_{ik} \)) to be the phase between \( \tau_{ik} \) and \( \tau_j \). The phase is denoted by \( \phi_{ik,j} \) and can be derived by using the facts that \( 0 \leq \phi_{ik,j} < T_j \) and that \( \tau_{jp} \) arrives and is released at \( r_{jp} = r_{jp} = p \cdot T_j + O_j \) (see Figure 2). Therefore the following relation holds:

\[
p \cdot T_j + O_j = r_{ik} + \phi_{ik,j}
\] (4)

By using the limits on \( \phi_{ik,j} \), we find that

\[
0 \leq p \cdot T_j + O_j - r_{ik} < T_j
\] (5)

which has the only integer solution:

\[
p = \left\lfloor \frac{r_{ik} - O_j}{T_j} \right\rfloor
\] (6)

Inserting this into (4) gives the phase from \( r_{ik} \) to \( r_{jp} \) as:

\[
\phi_{ik,j} = \left\lfloor \frac{r_{ik} - O_j}{T_j} \right\rfloor T_j - (r_{ik} - O_j)
\] (7)

Since the maximum jitter of a task can be larger than the task period, more than one instance of a higher priority task \( \tau_p \) may belong to \( S_1 \). The number of instances \( \tau_j \) that belong to \( S_1 \) is denoted \( N_{ik,j} \) and is given by:

\[
N_{ik,j} = \left\lfloor \frac{\phi_{ik,j} + J_j}{T_j} \right\rfloor
\] (8)

This is most easily realized by noting that if \( N_{ik,j} \) instances of \( \tau_j \) belong to \( S_1 \), then the following two inequalities hold:

\[
\phi_{ik,j} - N_{ik,j} \cdot T_j + J_j \geq 0
\] (9)

\[
\phi_{ik,j} - (N_{ik,j} + 1) \cdot T_j + J_j < 0
\] (10)

The former inequality expresses that the first instance of \( \tau_j \) belonging to \( S_1 \) is \( \tau_{jp(N_{ik,j} + 1)} \). The latter inequality states that the instance that arrives \( T_j \) earlier does not belong to \( S_1 \). It belongs to \( S_0 \). The two expressions (9) and (10) can together be solved for \( N_{ik,j} \) to give (8). In the example in Figure 1, \( N_{ik,j} = 2 \).

With (8) at hand, an expression for \( I_1(\tau_{ik}) \) can be derived:

\[
I_1(\tau_{ik}) = \sum_{j=1}^{i-1} \left\lfloor \frac{\phi_{ik,j} + J_j}{T_j} \right\rfloor C_{ij}^{\text{max}}
\] (11)
The term \( I_2(\tau_{ik}) \) in (3) representing the total interference of task instances in \( S_2 \) is easily shown to be

\[
I_2(\tau_{ik}) = \sum_{j=1}^{i-1} \left[ \frac{W_i^C - \phi_{ikd}}{T_j} \right] C_j^{max} \tag{12}
\]

Figure 2 is a magnified version of Figure 1 showing the executions of instances of \( \tau_j \) around \( r_{ik} \). The notation used in this section and the next is highlighted. Note that the figure shows a special case with instances from only one higher priority task. In a task set with more higher priority tasks the instance \( \tau_{(p-1)} \) could be delayed to interfere with \( \tau_{ik} \) as well.

![Figure 2. Visualisation of \( \phi_i, \varphi_j, \delta_i, n_j \) and \( t \).](image)

3.2. Interference by tasks released before \( r_{ik} \)

In order to find the contribution of \( I_0(\tau_{ik}) \) to \( W_i^C \), the task instances in \( S_0 \) have to be studied. These instances are released prior to \( r_{ik} \) and have the potential to interfere with the execution of \( \tau_{ik} \).

By using the fact that in the worst case, the instances in \( S_0 \) will experience their maximum jitter, their release times can be expressed in relation to \( r_{ik} \). The last instance of \( \tau_j \) that belongs to \( S_0 \), \( \tau_{j(p-N_{ik}-1)} \), is released at

\[
r_{j(p-N_{ik}-1)} = r_{ik} + \phi_{ikd} - (N_{ik} + 1) \cdot T_j + J_j \tag{13}
\]

We will now study the amount of level-i workload that is released during an interval \([r_{ik}-t, r_{ik})\). Notice that the interval is closed to the left but open to the right, meaning that it does not include the task instances released at \( r_{ik} \).

We define \( \delta_{ik,j} \) to be the time from the release of the last instance of \( \tau_j \) to belong to \( S_0 \) to \( r_{ik} \), i.e. \( \delta_{ik,j} = r_{ik} - r_{j(p-N_{ik}-1)} \) (this is shown graphically in Figure 2). By making use of expression (13) we find that

\[
\delta_{ik,j} = (N_{ik} + 1) \cdot T_j - (\phi_{ikd} + J_j) \tag{14}
\]

Substituting \( N_{ik} \) in this equation with the expression in (8) and using the fact that \( a \mod b = a - b \cdot \lfloor a/b \rfloor \) gives

\[
\delta_{ik,j} = T_j - ((\phi_{ikd} + J_j) \mod T_j) \tag{15}
\]

Now, there are two different cases that need to be considered for \( t \) in \([r_{ik}-t, r_{ik})\): \( t < \delta_{ik,j} \) and \( t \geq \delta_{ik,j} \). In the first case it is obvious that no instance of \( \tau_j \) is released within \([r_{ik}-t, r_{ik})\) and hence \( \tau_j \) does not contribute to the workload released in the interval. In the latter case however, at least one instance of \( \tau_j \) is released within \([r_{ik}-t, r_{ik})\). To find out exactly how many instances of \( \tau_j \) that are released in the interval in the second case, we define \( \varphi_j \) to be the time from \( r_{ik} \) to the first following release of an instance of \( \tau_j \), \( \varphi_j \) is visualized in Figure 2. In the worst case for \( \tau_{ik} \) (as defined in Theorem 1), \( \varphi_j \) has to satisfy \( 0 \leq \varphi_j < T_j \) since the time between the releases of all consecutive instances of \( \tau_j \) in \( S_0 \) is \( T_j \). The length of the interval \([r_{ik}-t, r_{ik})\) can be expressed as:

\[
t = \varphi_j + (n_j - 1) \cdot T_j + \delta_{ik,j} \tag{16}
\]

where \( n_j \) is the number of instances of \( \tau_j \) that are released in the interval. Since \( 0 \leq \varphi_j < T_j \), equation (16) can be solved for \( n_j \) giving

\[
n_j(\tau_{ik}, t) = \left\lceil \frac{t - \delta_{ik,j}}{T_j} \right\rceil + 1 \quad \text{for } t \geq \delta_{ik,j} \tag{17}
\]

We note that since \( \delta_{ik,j} \leq T_j \), equation (17) will give

\[
n_j(\tau_{ik}, t) = 0 \quad \text{when } t < \delta_{ik,j}
\]

and therefore (17) is correct for any \( t \geq 0 \) which includes the first case described above. It follows that the total level-i workload that is released in the interval \([r_{ik}-t, r_{ik})\) is given by

\[
W_i^P(\tau_{ik}, t) = \sum_{j=1}^{i} n_j(\tau_{ik}, t) \cdot C_j^{max} \tag{18}
\]

Note that the sum also includes early instances of the analysed task \( \tau_i \).
Given \( W^p(\tau_{ik}, t) \), the remaining execution need at \( r_{ik} \), if only workload from level-\( i \) task instances released in \([r_{ik} - t, r_{ik})\) is considered, can be written

\[
\Delta_{ik}(t) = \max(W^p(\tau_{ik}, t) - t, 0)
\]  

(19)

The maximisation takes care of the cases when all tasks released in \([r_{ik} - t, r_{ik})\) are completed before \( r_{ik} \), and hence \( W^p(\tau_{ik}, t) - t < 0 \). In order to find \( I_d(\tau_{ik}) \), we need to find the interval given by \( t \) that maximizes the above expression. Hence,

\[
I_d(\tau_{ik}) = \max_{t \geq 0}(\Delta_{ik}(t))
\]  

(20)

The maximum in (20) will be found for a \( t = \tau^* \) such that \( r_{ik} - \tau^* \) corresponds to the start of the level-\( i \) busy period within which \( \tau_{ik} \) executes. Therefore, an upper bound to \( \tau^* \) is \( L_i^* \), which is the length of the longest level-\( i \) busy period in the schedule. The existence of a finite \( L_i^* \) is guaranteed by the assumption that \( U \leq 1 \). An upper bound on \( L_i^* \), \( L_i^U \), which is easier to find, can be used to limit the interval over which the maximisation should be done. The derivation of \( L_i^U \) is further detailed in section 3.3.

It is possible to further narrow down the interval to be searched for \( \tau^* \). By applying recursion on (18) such that

\[
w^{n+1} = \sum_{j=1}^{i} n_j(\tau_{ik}, w^n) \cdot C_{j}^{max}
\]  

(21)

an even smaller upper bound on \( \tau^* \) will be reached. The recursion will converge to a stable value since (21) is monotonically decreasing with \( w \) and since the number of possible values of \( w \) is finite. The stable value achieved through iteration, \( w^* \), must be larger than \( \tau^* \) since the right hand side of (21) includes all level-\( i \) workload released within \([r_{ik} - w^*, r_{ik})\) and \( \tau^* \) corresponds to the start of a level-\( i \) busy period that finishes after \( r_{ik} \).

By using the new upper bound on \( \tau^* \), the interval to search for the maximum of \( \Delta_{ik}(t) \) in (20) is reduced. The maximisation can be performed by noting that (17) only changes values when \( t \) is equal to a release time of a task instance in \( S_0 \), and therefore (19) has local maxima at these instants. A finite set of values for \( t \) over which to perform the maximisation in equation (20) can be generated. Task instances in \( S_0 \) are released at \( r_{ik} - \delta_{ik,j} - l \cdot T_j \), \( \forall l \geq 0 \), corresponding to \( t = \delta_{ik,j} - l \cdot T_j \) in equation (20). The maximum number of release instants that needs to be checked for each task, \( \tau_j, j = 1, ..., i \), is found by noting that \( \delta_{ik,j} + l_{j}^{max} \cdot T_j \leq w^* \) is sufficient. Therefore

\[
l_{j}^{max} = \left[ \frac{w^* - \delta_{ik,j}}{T_j} \right]
\]  

(22)

and equation (20) can be reformulated as

\[
I_d(\tau_{ik}) = \max_{t \in A(\tau_{ik})(\Delta_{ik}(t))}
\]  

(23)

for \( A(\tau_{ik}) = \left\{ \delta_{ik,j} + l \cdot T_j | j = 1, ..., i; l = 0, ..., l_{j}^{max} \right\} \).

where \( \delta_{ik,j} \) is given by (15).

3.3. Deriving an upper bound \( L_i^U \) to \( L_i^* \)

We will create an upper bound, \( L_i^U \), to the length of the longest level-\( i \) busy period by assuming that all tasks can be arbitrarily phased ignoring the task offsets. This is very similar to the traditional way of calculating worst case response times for arbitrarily phased tasks. The longest level-\( i \) busy period starts at the critical instant for a task \( \tau_{i+1} \). This critical instant, \( t_c \), is created by phasing all higher priority tasks, \( \tau_1, ..., \tau_p \) such that these are released at \( t_c \) after having experienced their maximum release jitter. The following instances of these tasks are given jitter such that they are released at \( t_c \) if they arrive before \( t_c \). The instances arriving after \( t_c \) are given their minimum jitter. The corresponding scenario would be achieved for \( \tau_j \) by setting \( \phi_{ik,j} \) for each task \( \tau_j \) such that the equality in equation (9) holds. In this scenario, the number of instances of \( \tau_j \) that arrive before \( t_c \) and are released at \( t_c \) will be \( \lceil J_j/T_j \rceil \). Furthermore, the time between \( t_c \) and the first following arrival (and release) of \( \tau_j \) is \( \phi_{ik,j} = \lceil J_j/T_j \rceil \cdot T_j - J_j \). Therefore, an expression for the longest level-\( i \) busy period for arbitrarily phased tasks is

\[
L_i^U = \sum_{j=1}^{i} \left( \left\lceil \frac{J_j}{T_j} \right\rceil + \left\lceil \frac{L_i^U - \phi_{ik,j}}{T_j} \right\rceil \right) C_{j}^{max} + B_i
\]  

(24)

\( L_i^U \) can be derived by iteration over (24). We note that the total number of instances of \( \tau_j \) that are released at \( t_c \) is \( \lceil J_j/T_j \rceil + 1 \) when \( J_j/T_j \) is an integer, and \( \lceil J_j/T_j \rceil \) otherwise. Hence, the total number of instances of \( \tau_j \)
that are released at \( t_i \) is \( \left\lfloor \frac{J_i}{T_i} \right\rfloor + 1 \) for any \( J_i \). The iteration over (24) is therefore started with

\[
L^0_i = \sum_{j=1}^{i} \left( \left\lfloor \frac{J_j}{T_j} \right\rfloor + 1 \right) C^{\text{max}}_j + B_i
\]  

(25)

and terminated when \( L^m_i = L^n_i \). The result is \( L^m_i \) which is an upper bound to the longest level-\( i \) busy period for the tasks with their offsets fixed. The sum in (24) is taken over 1,...,i since instances of \( \tau_i \) are part of the level-\( i \) busy period.

### 3.4. Algorithm summary

Having derived expressions for every part of the algorithm, we are now ready to summarize the steps needed to calculate worst case response times for tasks with offsets and release jitter. Table 1 gives a summary of the algorithm with references to the used equations.

In step 3 of the algorithm, a new hyperperiod (the LCM of the periods of tasks 1,...,i) is calculated for each task \( i \). This is because lower priority tasks have no effect on the hyperperiod for task \( i \) and can therefore be omitted.

#### Table 1. Algorithm summary

```plaintext
for each \( \tau_i, i = 1,...,n \) do
    \( R_i = 0 \)
    Calculate \( L^U_i \)  
    \( \text{LCM}_i = \text{LCM}(T_1,...,T_i) \)  
    \( K = \text{LCM}_i/T_i \)

for each \( \tau_{ik}, k = 0,...,K-1 \) do
    \( r_{ik} = d_{ik} \cdot J_i \)  
    \( \phi_{ik} \), \( \delta_{ik} \), and \( N_{ik} \)  
    \( l_d(\tau_{ik}) \)  
    Calculate \( W^C_{ik} \) through iteration  
    \( R_{ik} = W^C_{ik} + J_i \)

if \( (R_{ik} > R_i) \) then \( R_i := R_{ik} \)
```

### 3.5. Alternative analysis method

The traditional method for exact analysis of task sets with offsets is to perform a schedule simulation. A schedule simulation is basically a simulation of the execution of tasks on a processor given the complete a priori knowledge about the task release times, execution needs, priorities etc. For a task model with offsets but without release jitter (arrival and release times are equal and known) a complete table of all task release times within a hyperperiod can be prepared. The simulation is carried out by stepping forward in (simulation-) time until the next scheduling instant. A scheduling instant is either the release of a new task instance or the completion of the one currently “executing”. At the scheduling instant the remaining execution need of the active task is updated and a new scheduling decision is taken based on the priority of the active task and the ones in the ready queue.

The simulation is complete after one hyperperiod after which the schedule repeats itself. The worst (and best) case response times of the tasks are achieved by comparing the logged release and completion times of all task instances. If the analysed task may be blocked by a lower priority task, the maximum blocking time is added to the task’s execution time.

As stated in the introduction, if tasks cannot experience any release jitter, it is sufficient to perform one schedule simulation over an interval as long as the LCM, [1]. When release jitter is included in the task model however, a single schedule simulation is not sufficient. Just like in the analysis described in this paper, each task instance has to be analysed separately, since a worst case must be created for each individual task instance. The worst case defined in Theorem 1 is used also for the schedule simulation. Similar to the analysis developed above, it is sufficient to perform each schedule simulation over an interval limited by the longest level-\( i \) busy period. The same arguments as in 3.2 can be used to show that the simulation for an instance \( \tau_{ik} \) can be started at a time \( L^U_i - C^{\text{max}}_{i} \), before the release of \( \tau_{ik} \) and stopped once \( \tau_{ik} \) has completed.

### 4. Results

A schedule simulator and the analysis method described in this paper were implemented and their performances compared in Matlab. The comparisons were performed on a 750 MHz Pentium III. It is important to notice that since both the proposed methods produce exact worst case response times for tasks, the actual response times produced will be identical. Hence it is not possible to compare the efficiency of the methods in terms of tightness of the produced bounds as is common in other papers similar to this. Instead we will here present results that relate the methods to each other in terms of computational complexity.
The comparison between the two analysis methods is strictly experimental and no theoretical complexity analysis has been performed. Both methods were tested with the same set of randomly generated task sets. The time for completion of each method was measured and these times compared.

In a first test, a hundred different task sets where generated and analysed. The task sets where given a random number of tasks in the interval [2, 15]. The task periods where picked randomly from a limited set: \{1, 5, 15, 30, 60, 100\}. This gives a maximum hyperperiod of 300. The execution times, offsets and maximum jitters where all randomly generated. The execution times were scaled to make the utilisation of the task set 0.9, while the offsets and jitters were limited by \(0 \leq O_i < T_i\) and \(0 \leq J_i < 3 \cdot T_i\) respectively. The choice of interval for the jitter was quite arbitrary.

Figure 3 shows the analysis execution times in seconds (y-axis) for the hundred task sets and for both analysis methods. The analysis times are plotted against the number of analysed task instances (the complete number of instances within a hyperperiod). Crosses show the execution times of the schedule simulation while dots show the execution times for the new method. We note a strong (almost linear) dependency between the analysis time and the number of task instances to analyse. Note that the y-axis has a logarithmic scale.

In a last test, the periods of the tasks were taken randomly from \{1,...,20\} making very long hyperperiods possible. The number of tasks in the task sets where increased continuously until one analysis round took more than 10 minutes to complete. In the last run, 553136 task instances of 14 tasks where analysed within a hyperperiod of 232560. The analysis took 800 seconds.

These tests show that the method developed in this paper performs much better than a method based on schedule simulation. They also show that the new method can be used to analyse reasonably large task sets. However, the analysis time grows close to linearly with the number of instances within the hyperperiod.
5. Conclusions

This paper describes a method for the calculation of exact worst case response times for fixed priority scheduled tasks with offsets and release jitter. This type of analysis is important since offsets can be used to improve the schedulability of tasks and this improvement should be exploited in the analysis methods, such that schedulable systems are not deemed unschedulable due to pessimistic assumptions. Furthermore, offsets and release jitter are useful for modeling precedence relations in transactions of tasks. Current methods for analysing the type of task sets discussed here are either pessimistic or perform badly. The traditional analysis through schedule simulation can be applied to the problem, if it is altered to incorporate a simulation for every single task instance.

The analysis method has been implemented in Matlab together with the alternative exact method based on schedule simulation. Experiments show how the new method outperforms the simulator. When this kind of comparison is done, the efficiency of the implementation does play some role. The schedule simulator could probably be implemented more efficiently, but so could also the analysis method presented in this paper.

It should be noted again that the new method is still very sensitive to task sets in which task periods are chosen unwisely. Periods that are relative primes make the LCM long compared to the task periods and hence the number of task instances that need to be analysed grows. It may seem reasonable though, that task periods usually are chosen such that the LCM does not get too long. If this is the case, the analysis described here can be successfully applied.

We should also note that the major contribution in this paper is the search and identification of the start of the busy period that contains the analysed task instance. A similar method where the beginning of a busy period is identified, has earlier been used by Redell and Sanfridson [8] to derive the best case response times of tasks.

As described in this paper, the method should be directly applicable to the type of task sets described by Tindell in [11], with the addition that different transactions can also be given fixed phasings. The applicability to the analysis of Palencia and Harbour, [6], that models tasks with dynamic offsets and jitter needs to be further investigated.

Further work will concentrate on applying this method on distributed systems with precedence related tasks. The possibilities to use the method for synthesis of real-time systems will also be examined. This includes using the method for deriving suitable offsets for tasks and for assigning priorities to tasks. Audsley describes an optimal priority assignment methodology in [1] that should be possible to use together with the method developed here.

References