USING OFFSET INFORMATION TO ANALYSE STATIC PRIORITY PRE-EMPTIVELY SCHEDULED TASK SETS

K. W . Tindell[1]
Department of Computer Science,
University of York, England

E-mail: ken@minster.york.ac.uk

ABSTRACT

One common way of constructing hard real-time systems is to use a number of periodic and sporadic tasks assigned static priorities and dispatched at run-time according to the pre-emptive priority scheduling algorithm. Most analysis for such systems attempts to find the worst-case response time of a task (the time between the task arriving and the task completing all necessary computation). Furthermore, the analysis also assumes that the worst-case scheduling point is when all tasks in the system are released simultaneously. Often, however, a given set of hard real-time tasks will have *offset* constraints: a number of tasks sharing the same periodic behaviour will be constrained to execute at fixed offsets in time relative to each other. In this situation the assumption of a simultaneous release of all tasks can lead to pessimistic scheduling results. In this paper we derive good response time bounds for tasks with offset information. We also demonstrate how the analysis becomes able to schedule all schedulable static cyclic scheduling problems, including those that have precedence and exclusion constraints.

1. INTRODUCTION

The problem of guaranteeing *a priori* the schedulability of a set of hard real-time tasks has received much attention in recent years. A promising approach has been to derive a set of tasks that each require a bounded amount of computation time for each invocation. To bound the total computational load, a lower bound on the time between successive invocations for each task is assumed. Most analyses require that each task is assigned a unique static priority, and is dispatched at run-time pre-emptively (*i.e.* the highest priority runnable task is dispatched). The earliest attempt at analysing such systems was made by Liu and Layland with the rate monotonic approach [11]. With this, static priorities are assigned monotonically (and uniquely) with period: a short period task has a high priority. The deadline of any such task is assumed to be at the end of the period of the task, and tasks are assumed not to block (*e.g.* by attempting to lock semaphores). Tasks are also assumed to be strictly periodic: a task must arrive

[1]This work was supported in part by British Aerospace Airbus Ltd

successively at exact intervals. The original analysis is *inexact* — certain task sets could be deemed unschedulable, when in reality all deadlines would be met.

Worst-case Response Times

Many of the restrictions described above have been lifted by subsequent work. Sha *et al* [13] have extended the analysis to permit tasks to lock semaphores. Lehoczky *et al* [8] provide exact analysis of tasks with priorities assigned according to the rate monotonic policy. Work by Joseph and Pandya [6] and Audsley *et al* [1, 5] have extended the original rate monotonic priority assignment analysis to work for any given priority ordering and to compute *worst-case response times* — instead of asking the question "will task *i* meet its deadline?" we ask a slightly different question: "how long will task *i* take to complete?" This latter question is more general; once we know how long task *i* will take to complete (*i.e.* the worst-case response time of task *i*), determining the schedulability of that task is then trivial. This approach becomes particularly useful when considering distributed hard real-time systems. For example, consider a task *a* that reads a sensor, performs some computation, and delivers its result to some later task *b* (on another processor) via a message sent across a communications network. Task *b* then performs some further computation before undertaking some action. The timing constraint derived from system requirements is that the time from the sampling of the sensor to the final action must take no longer than *D*. An implementation could meet *D* a number of ways: task *a* could be given a high priority (and hence will have a short worst-case response time), with both the message and task *b* given a long time to complete (a low priority for the message and *b* would then suffice). Alternatively, task *a* could have a low priority and task *b* a short time to complete. Using the notation r_i to denote the worst-case response time of a task *i* and *M* to denote the worst-case message delivery time, the timing constraint becomes:

$$r_a + M + r_b \leq D$$

It makes no sense to refer to a deadline *requirement* for either task *a* or *b*, since these would play no part in the overall timing constraint, other than, perhaps, to determine the priorities of tasks *a* and *b*. Throughout the rest of this paper we will therefore assume a given priority ordering and will seek to determine worst-case response times.

Offsets Between Tasks

Previous analysis has assumed the worst-case scheduling point (*i.e.* the 'critical instant') to be when all tasks are released simultaneously. Some tasks may, in reality, have *offset* relationships — tasks sharing the same period may preserve an offset in time between their arrivals (Figure 1).

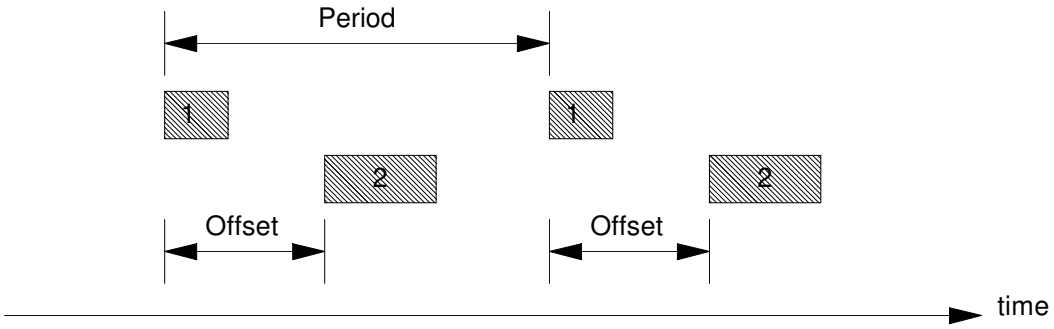

Figure 1: the shaded boxes represent computation time

In figure 1, tasks 1 and 2 share the same period. They also have an offset relationship: task 2 is released a fixed interval after the release of task 1. Current analysis would assume that the worst-case scheduling point would occur when tasks 1 and 2 are released together. Clearly these tasks can never be released together, and analysis that took account of this would be less pessimistic than current analysis.

In the rest of this paper we will derive exact analysis to bound response times of tasks that have offset relationships. We will also show how evaluation of a test based on such analysis is unfortunately computationally infeasible, and then derive a good feasible (but inexact) test. We present results which compare the performance of both the exact and inexact tests, and the original analysis of Audsley *et al* [5].

Before proceeding with the analysis we first describe the basic computational model that will be assumed for the rest of this paper, and then briefly describe how hard real-time systems could make use of the ability to specify offsets within this computational model.

A fixed number of *transactions* are assigned to a processor[2]. Each transaction is composed of a fixed number of *tasks*. Each task in a transaction requires a bounded amount of computation time for each invocation. A transaction may *arrive* periodically or sporadically, but with a minimum time between subsequent arrivals[3] — this minimum time is denoted the *period*. For each transaction arrival, each task is *released* (*i.e.* placed in a notional priority-ordered run queue) at a fixed *offset* in time, measured relative to the arrival time of the transaction; we assume that this offset is less than the period of the transaction. Necessarily all tasks in a given transaction must share the same period. Each task is assigned a unique static priority; tasks are dispatched pre-emptively based on this priority. For the moment we assume that tasks cannot lock semaphores and hence cannot be blocked (we will lift this restriction later). We also assume that there is no delay between the arrival of a task and when it is released (for example, delay due to a tick-driven scheduler being used [5]).

[2]Transactions can span processors, but time must be allowed for any message delay between transaction members on different processors; any distributed transaction can be transformed into a single transaction per processor that the transaction spans, and analysis can proceed on the basis of a single processor model

[3]A periodic task can be regarded as a special case of a sporadic task — one that is merely released at regular intervals by a timing event; the analysis throughout this paper does not distinguish between sporadic and periodic transactions

Table 1 gives the notation used:

C_i	The worst-case computational requirement of task *i*
r_i	The worst-case response time of task *i*
T_t	The minimum delay between successive invocations of transaction *t*, measured from arrival to arrival
O_i	The offset of task *i*, measured from the start of the transaction
D_i	The deadline requirement of task *i*, measured from the arrival of *i*
hp(i)	The set of all tasks of higher priority than *i*
trans	The set of all transactions
trans(i)	The transaction of which *i* is a member
tasks(t)	The set of all tasks that are members of transaction *t*

Table 1: notation

Note that we can determine if a transaction deadline (measured relative to the release and termination of different tasks within the transaction) is met by summing task offsets and response times appropriately.

Some Uses For Offsets

A precedence constrained set of tasks allocated across a number of processors can be modelled by assigning offsets to later tasks such that earlier tasks on other processors are guaranteed to have finished before the later task starts. Furthermore, a later task can be given an offset which can also allow a bounded time for a message to be sent from an earlier task to a later task on a different processor (Figure 2).

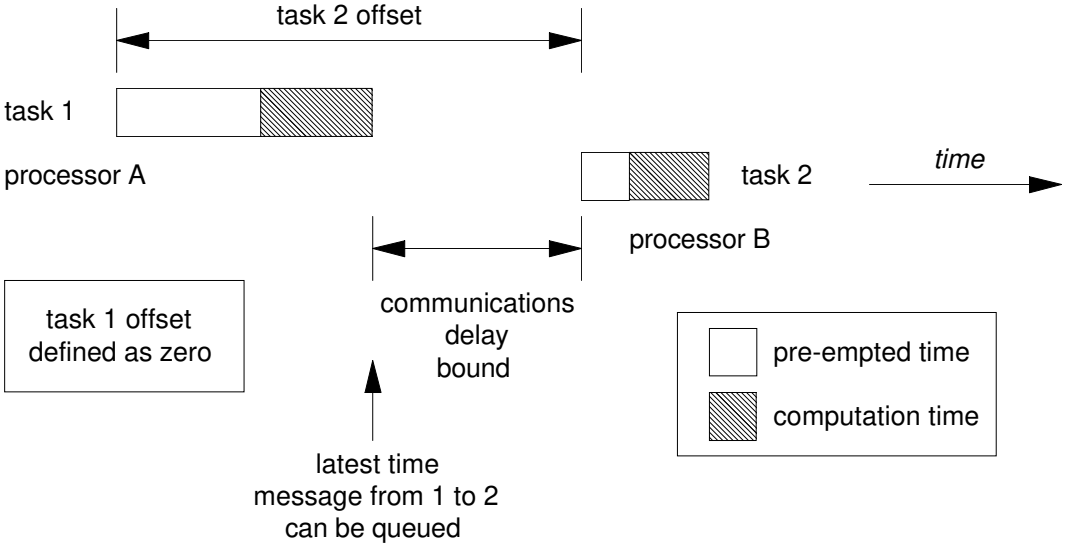

Figure 2: task 2 runs when message is guaranteed to have arrived

Task 1 could queue a message at the very last moment before terminating. The message then takes a bounded amount of time to travel from processor *A* to processor *B* (assuming the communications sub-system can provide such guarantees [17, 15]).

Task 2 is released after the worst-case message arrival time. Note that task 2 is not synchronised with the arrival of the message — task 2 has a fixed release time relative to the release of task 1; if the message arrives early task 2 is not released early (for this approach to work the clocks on each processor must be synchronised by an appropriate mechanism [7]).

Another major use for offsets is to permit tight jitter bounds on an input or output action. As described by Locke [12], jitter can occur when the computation in a periodic task is completed at irregular times (although still at a bounded rate). Task 2 in the above example can be assigned a high priority to give a shorter worst-case response time. This will reduce the variability within each period that any output from task 2 is made. If task 2 has a large computation time (*i.e.* the smallest worst-case response time, equal to the worst-case computation time, is large, and hence the smallest worst-case jitter is also large) then the task could be further split into a computation phase task and an output phase task, with the output phase task requiring little computation time and assigned a high priority.

Offsets can be used to avoid the need for a dynamic concurrency control protocol controlling access to a mutual resource (such as the priority ceiling protocol). For example, two tasks sharing a resource can be released at fixed offsets relative to each other such that neither task executes concurrently, removing the need to guard the resource. Section 5 will discuss this in more detail.

Offsets can also be used to express complex timing patterns between tasks. For example, consider a sporadic transaction controlling a disk drive (or indeed, any other non-trivial device) in real-time; the transaction is initiated by the arrival of a "get disk block" message. The disk drive is controlled by requesting a disk block, waiting for the block to be retrieved, and then fetching the data from a buffer. The drive is guaranteed to take no more than 12ms to read the block. The controlling software can be implemented with a two-task sporadic transaction: the first task (task *a*) sends the request to the disk drive. The second task (task *b*) has an offset of at least 12ms from the end of task *a*; knowing the worst-case response time of task *a*, an absolute offset can be found. Task *b* reads the data from the buffer and replies to the transaction initiator. In this manner complex timing patterns can be expressed. Note, however, that the above example requires configuration decisions to be made: the priorities and offsets of tasks *a* and *b* must be chosen such that the response time of the transaction (denoted the 'end-to-end' response time, measured from the start of the transaction to the worst-case finish time of task *b*) is less than some deadline[4], *i.e.* $O_b + r_b \leq D$ (where D is some deadline) However, the choice of offsets and priorities is further constrained by requiring that $O_b - O_a + r_a \geq 12$. These choices will almost certainly not be independent of other configuration choices elsewhere in the system; we anticipate a system-wide configuration approach being used to choose offsets, priorities, *etc.* [16]

The rest of this paper is structured as follows: Section 2 provides some essential background scheduling theory, forming the basis of analysis to take account of offset

[4]There may be no immediate deadline that can be sensibly chosen, in which case the term $r_a + 12 + r_b$ merely forms part of some larger 'end to end' response time; this larger response time will be eventually compared to a deadline

information. Section 3 derives from the original analysis an exact schedulability test for tasks with offsets. The derived test has the disadvantage that it is of combinatorial computational complexity; section 4 derives a simpler test, termed the Tractable Offset Test. Section 5 discusses the issues surrounding access to shared resources under offset constraints, and produces extensions to the Tractable Offset Test which permit tasks to lock and unlock semaphores according to the priority ceiling protocol. Section 6 discusses some interesting ramifications of the Tractable Offset Test. Section 7 summarises the analysis thus derived and gives conclusions. Appendices are included: Appendix A proves the convergence properties of original scheduling theory; Appendix B shows how the tractable offset test reduces to original analysis for two pieces of theory; Appendix C contains an extended example showing how offsets can be used to schedule a problem which would otherwise be unschedulable.

2. EXISTING SCHEDULING THEORY

This section describes some of the current scheduling theory for static priority pre-emptively dispatched independent tasks with no offset information.

We wish to determine the worst-case response time of a task i — the longest time i can take to execute a given worst-case amount of computation. To do this we consider a *time window*, starting at the release time of task i, and of width w. All higher priority tasks are assumed to arrive at the same time as i because this is the worst-case scheduling point — the point when the longest response time for a task i occurs. The window represents the worst-case response time of task i. We wish to find the smallest value of w such that all the computation of i can be completed within w, taking account of the delay due to the pre-emption and execution of higher priority tasks. We term this delay *interference*. Before proceeding further, it is worth discussing why the worst-case scheduling point occurs when all tasks are released together (the argument will be used again later in this paper).

The Worst-case: Tasks Released Together

Consider a low priority task i, and a single higher priority task j. Assume that task j is released a time Δ before the release of task i (task j released at time t *after* task i is, if course, equivalent to task j being released at time $T_j - t$ before).

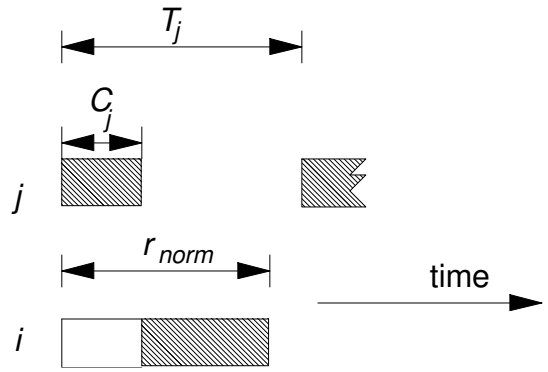

Figure 3: Task j is released at the same time as task i

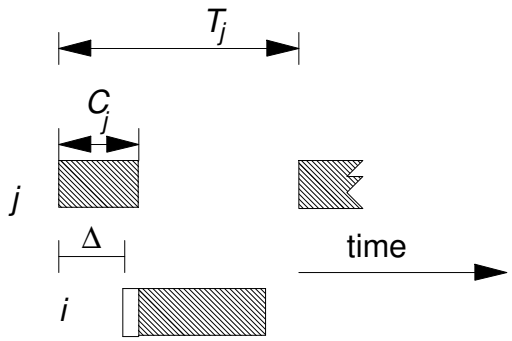

Figure 4: Task *j* is released Δ before task *i*

Let I_{norm} be the computational interference task *j* would normally impose on task *i* if it were released at the same time as *i*. Let r_{norm} be the response time resulting from this interference (Figure 3). The interference from the first invocation of *j* on *i* is C_j. When *j* is released Δ earlier this interference reduces by Δ, since the first Δ ticks of computation occur before *i* is released and hence cannot delay *i* (Figure 4). The corresponding response time of *i* must therefore reduce by at least Δ ('at least' because a reduction by Δ may result in the interference from a given higher priority task *a* reducing by C_a). All subsequent invocations of *j* can occur Δ sooner, but since the response time of task *i* reduces by at least Δ, a non-interfering subsequent invocation still cannot interfere. If Δ is large enough such that the interference from the first invocation of *j* is then zero, a subsequent invocation *can* interfere; if this is the case then interference from the first invocation of *j* can be ignored, and the next invocation of *j* can then be considered the first invocation.

Hence the worst case interference on a task *i* from a higher priority task *j* occurs when *j* is released at the same time as *i*.

Calculating the Worst-case Response Time

The interference from a task *j* on a window of width *w* can be calculated as follows:

Assume T_j is the period (*i.e.* minimum time between arrivals) of a task *j*. Suppose *w* is less than T_j. Within this window, task *j* can pre-empt *i* once — *j* cannot re-arrive sooner than T_j later. If *w* is slightly larger than T_j then task *j* can pre-empt *i* twice: once at the start, and again T_j later. If *w* is slightly larger than $2T_j$ then task *j* can pre-empt *i* three times. Hence the interference from task *j* over the window of width *w* amounts to:

$$\left\lceil \frac{w}{T_j} \right\rceil C_j$$

The total interference over the window *w* due to pre-emption by higher priority tasks is therefore given by:

$$\sum_{j \in hp(i)} \left\lceil \frac{w}{T_j} \right\rceil C_j$$

Where *hp(i)* is the set of tasks of higher priority than *i*.

The smallest value of *w* (*i.e.* r_i, the worst-case response time of task *i*) will occur when *w* is exactly equal to the total interference from higher priority tasks over *w*, plus the computation due to task *i*. Therefore the following equation must hold for the smallest value of *w*:

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

Notice how *w* occurs on both the left and right hand sides of the above equation. A value for *w* can be found by iteration:

$$w^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w^n}{T_j} \right\rceil C_j \qquad (2)$$

The iteration proceeds until $w^{n+1}=w^n$. If w^n exceeds D_i then the iteration can be terminated early; task *i* is then deemed unschedulable. It can be shown (see Appendix A) that the iteration is guaranteed to converge if the processor utilisation due to task *i* and higher priority tasks is ≤ 1, *i.e.*:

$$\sum_{k \in \{hp(i) \cup i\}} \frac{C_k}{T_k} \leq 1 \qquad (3)$$

An alternative view of the behaviour of equation 2 is to consider the window as starting with zero width, and gradually widened until all the computation from task *i* can be 'fitted into' the spare time between the computation of higher priority tasks.

The above analysis assumes tasks do not lock semaphores, but can be extended to allow tasks to lock semaphores according to the priority ceiling protocol [13] (or equivalent). With this protocol, a task *i* following a strict set of rules may lock a semaphore. Each semaphore *s* is assigned a *ceiling* value, equal to the highest priority of any task that could lock *s*. Each task *i* is then assigned a *blocking factor* B_i equal to the longest critical section of any task of lower priority than *i* which locks a semaphore with a ceiling greater than or equal to the priority of *i*. This blocking time represents the longest time a task *i* can be deferred due to the execution of a critical section of a lower priority task. This can be incorporated into our analysis by adding B_i to the computation which occurs in the window:

$$r_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i^n}{T_j} \right\rceil C_j \qquad (4)$$

For tasks which have priorities and deadlines assigned according to the rate monotonic policy (*i.e.* priorities assigned monotonically with period, and for any task i $D_i = T_i$) the above equation is equivalent to the exact analysis developed by Lehoczky *et al* [8]. For each task i the worst-case response time r_i is computed, and then compared to the period (*i.e.* deadline) T_i.

As discussed earlier, the worst-case scheduling point for a task i is when all higher priority tasks are released at the same time as i. However, it may be possible to obtain task offset information which could indicate that such a release pattern could never occur. The actual worst-case response time r_i could therefore be lower than the one found by equation 4. The next section will describe how the window approach used above can be extended to take account of offset information, and obtain better (*i.e.* lower) bounds for r_i.

3. THE OFFSET TEST: EXACT ANALYSIS

In this section we formulate an exact schedulability test termed the Exact Offset Test. Tasks within the same transaction preserve the same offset between them, since they are always released at a fixed offset in time from the first task in the transaction. As discussed in the previous section, a higher priority task released time Δ before a given lower priority task i typically will result in lower interference. The values of Δ between tasks in the same transaction are constant and known *a priori.* If, for a given task, Δ is sufficiently large it may be possible to remove entirely the interference due to a given higher priority task.

Similarly, the worst-case computation from a set of higher priority tasks in a different transaction could be reduced: if there is a sufficiently large offset between two members of a different transaction (say tasks *a* and *b),* then it will be impossible for both tasks to interfere over a certain window of width w — at most only one task could delay i (Figure 5).

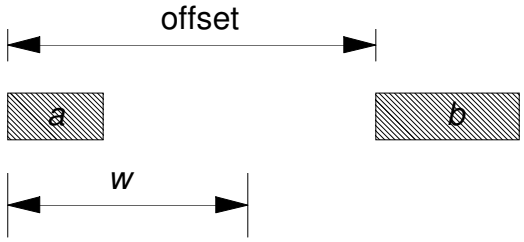

Figure 5: either task *a* or task *b,* but not both, could interfere over the window of width *w*.

Interference From Tasks In The Same Transaction

We first address the problem of finding the worst-case interference on a task i from higher priority tasks in the *same* transaction. Intuition indicates, perhaps, that the interference from higher priority tasks in the same transaction might be zero if task i is released after the earlier task finishes. However, this is not the case: if we discount the interference on a low priority task l from an earlier high priority task h in the same transaction, we are effectively saying that task h has no effect on the later task l.

However, consider a medium priority task *m* that is not a member of the transaction: task *m* is released at the same time as task *h*. Task *h* will pre-empt task *m* and defer its execution. When task *m* is eventually able to run task *l* is released. From the viewpoint of task *l* the medium priority task *m* has been deferred from executing; task *l* experiences a 'back-to-back hit' (Figure 6).

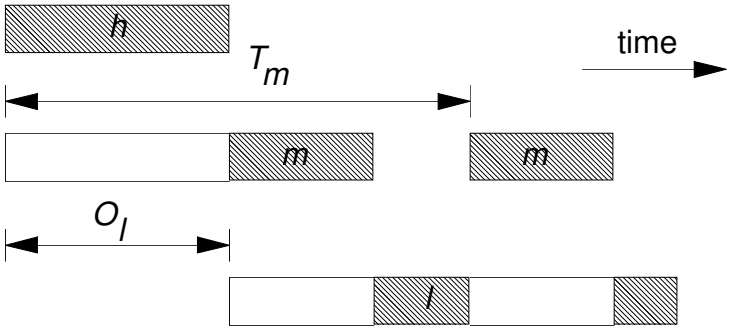

Figure 6: 'back-to-back hit' for task *l*; tasks *h* and *l* are in the same transaction; task *m* is of another transaction; task *l* is pre-empted twice by task *m*

In Figure 6, task *l* receives interference from two invocations of task *m*, where ordinarily it might be thought that only one 'hit' would occur.

Recall we said earlier that if a higher priority task *j* were to be released a time Δ later than *i* then this would be equivalent to the same task *j* being released time $\Delta - T_j$ earlier than *i*. We adopt this approach for tasks within the same transaction, and when determining the worst-case response time for task *i* we re-define the transaction of which *i* is a member such that all other members of the same transaction are released before *i*:

$$\hat{O}_j = \begin{cases} O_j & \text{if } O_j < O_i \vee j \notin tasks(trans(i)) \\ O_j - T_j & \text{otherwise} \end{cases} \qquad (5)$$

Thus if, in reality, a later task *j* has $O_j \geq O_i$ we assume that the value of $\hat{O}_j$ is $O_j - T_j$ (this will result in $\hat{O}_j < 0$; the transaction could be normalised by adding a constant to all $\hat{O}$ such that no offset is negative)[5]. We assume throughout this paper that the offset of a task is always less than or equal to the period of a task; a task must be released before the transaction starts again (*i.e.* $O \leq T$). Furthermore, we also assume that the worst-case response time of a task *i* must be less than the period of *i* (this is to ensure that different invocations of task *i* cannot run simultaneously — which complicates the run-time dispatcher implementation — and to simplify the analysis slightly).

When we earlier discussed the interference from a higher priority task *j* released Δ before task *i* we made only a qualitative statement of the interference from *j*. We now have to determine the exact interference given the release offset. We do this by considering a window of width *w* defined to start at the same time as a task *k* (*i.e.* the

[5]Note that in the case where $O_j = O_i$ both $\hat{O}_j = O_j$ and $\hat{O}_j = O_j - T_j$ are equivalent

window has an offset W — see Figure 7), where task k is of higher priority than task i. Computation starting in this window then leads to the window being widened until the window is of the same size as the computation which takes place in the window. Any invocation of a higher priority task that is released before the start of the window (task j in Figure 7) is *deemed* to have no interference over that window (we will defer for the moment the explanation of why this is assumption is valid).

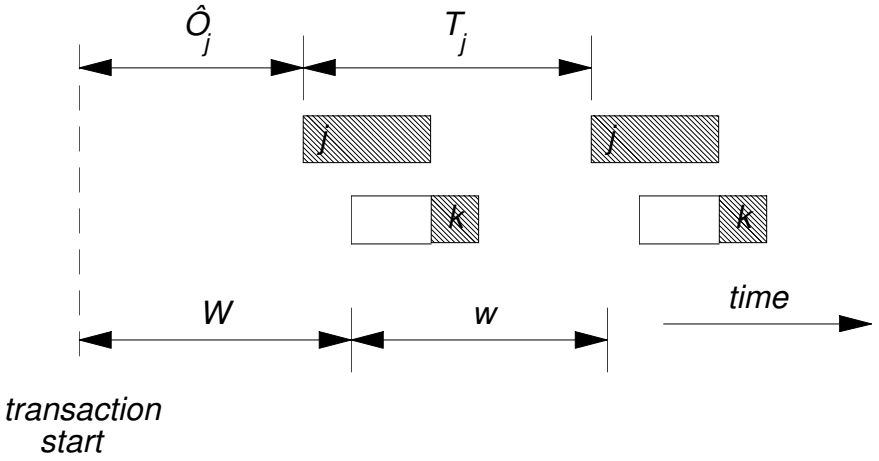

Figure 7: interference from a higher priority task j over a window of width w starting at time W after the transaction

In the above diagram task j gives a total interference of C_j: the first invocation of j is *deemed* not to have any interference over the window (W is the 'window offset' — the time at which the window starts, measured relative to the start of the transaction); the second invocation falls within w. If, however $\hat{O}_j \geq W$ (and the second invocation of j were still within w) then the interference would be $2C_j$ since the first invocation of j would start after the window.

Thus, for a window of width w that starts at time W after the release of the transaction, with $\hat{O}_j < W$, the deemed interference on a task i in the transaction from a higher priority task j of the same transaction, is given by:

$$\left\lceil \frac{w + W - \hat{O}_j - T_j}{T_j} \right\rceil_0 C_j \tag{6}$$

Where $\lceil x \rceil_0$ is a modified ceiling function, returning the ceiling of x if $x > 0$, and zero otherwise.

Similarly, the deemed interference with $\hat{O}_j \geq W$ is given by:

$$\left\lceil \frac{w + W - \hat{O}_j}{T_j} \right\rceil_0 C_j \tag{7}$$

Combining equations 6 and 7, the deemed interference from a higher priority task j over the window is therefore given by:

$$\left\lceil \frac{w + W - \hat{O}_j}{T_j} \right\rceil_0 C_j - \left[W - \hat{O}_j \right]_0^1 C_j \tag{8}$$

Where the notation $[x]_0^1$ denotes a function returning zero if x ≤ 0, and 1 otherwise.

The total deemed interference from higher priority tasks of the same transaction as *i* for a given *w* and *W* is given by:

$$\sum_{j \in task(trans(i))} \left(\left\lceil \frac{w + W - \hat{O}_j}{T_j} \right\rceil_0 - \left[W - \hat{O}_j \right]_0^1 \right) C_j \tag{9}$$

Recall that *W* is equal to the offset of a particular higher priority task in the same transaction as *i*. A view of the above equations is to consider the window width *w* as a *level i busy period*[6] [9] and that the first $O_i - W$ computation ticks of this busy period do not interfere with task *i*. If O_i is greater than $w + W$ then the 'busy period' starting at time *W* finishes before task *i* is released.

Now, the response time of a task *i* is related to the window size *w*. Task *i* is always released at the same time as, or later than, *W*. Hence, the first $O_i - W$ computation time within the window occurs before the release of task *i*. The response time r_i corresponding to the value of *w* derived from the window starting at time *W* is therefore:

$$r_i(W) = w - (O_i - W) \tag{10}$$

Where *w* is the width of the window that starts at time *W* after the arrival of the transaction.

The worst-case scheduling point (a *critical instant*) will occur when the window is released at the same time as one of the higher priority tasks in the transaction. To obtain the largest value of $r_i(W)$ (*i.e.* the worst-case response time) we must consider a number of windows in turn, each one starting at the release of a different higher priority task in the same transaction as *i*. All these offsets need to be examined because the worst-case scheduling point might occur at any one of them.

Interference From Tasks In Other Transactions

We now return to the problem of finding the worst-case interference from tasks in *other* transactions, on a window of width *w*. We have already shown that the worst-case scheduling point must occur when the start of the window coincides with the release of a higher priority task. The problem of finding the actual worst-case interference is one of finding which higher priority task of another transaction coincides with the window start at the worst-case point. In the general case there is no alternative to examining each higher priority task. We use W_t to denote the offset of

[6]Lehoczky defines a level *i* busy period as "an interval of time during which the processor is executing tasks with priorities greater than or equal to that of task *i*"

the current window relative to the transaction t (the term W used earlier can therefore be regarded as notational shorthand for $W_{trans(i)}$).

As described previously, a task released before the start of the window is *deemed* not to cause any interference in that window. We continue this approach here, and say that for a given task i, the interference on a window of width w, starting at time W_t after the release of a transaction t, from higher priority tasks in t, is given by:

$$I_t(W_t) = \sum_{k \in hp(i) \cap tasks(t)} \left(\left\lceil \frac{W_t + w - \hat{O}_k}{T_t} \right\rceil_0 - \left\lceil W_t - \hat{O}_k \right\rceil_0^1 \right) C_k \tag{11}$$

Note in the above equation we have used the notation $\hat{O}$ instead of O (however, for any task j which is not a member of the same transaction as task i $\hat{O}_j$ is defined to be equal to O_j). Now, to find the computational interference from all transactions we have:

$$\sum_{t \in trans-trans(i)} I_t(W_t) + I_{trans(i)}(W) \tag{12}$$

The above equation simply sums equation 11 over all transactions. Recall that W is notational shorthand for $W_{trans(i)}$. The computational requirements of the task we are interested in – task i – must also be accounted for, and hence the total computational load occurring in the window is given by:

$$C_i + \sum_{t \in trans-trans(i)} I_t(W_t) + I_{trans(i)}(W) \tag{13}$$

We define the width of the window, w, as being equal to the total computation that takes place in the window, such that all the computation of task i is complete. Therefore:

$$w = C_i + \sum_{t \in trans-trans(i)} I_t(W_t) + I_{trans(i)}(W) \tag{14}$$

Now, w occurs on both the left and right hand sides of the equation. A value for w can be found by iteration:

$$w^{n+1} = C_i + \sum_{t \in trans-trans(i)} I_t(W_t) + I_{trans(i)}(W) \tag{15}$$

$$I_t(W_t) = \sum_{k \in hp(i) \cap tasks(t)} \left(\left\lceil \frac{W_t + w^n - \hat{O}_k}{T_t} \right\rceil_0 - \left\lceil W_t - \hat{O}_k \right\rceil_0^1 \right) C_k$$

With $w^0 = C_i$. The iteration will have converged when $w^{n+1} = w^n$. The iteration is terminated if w^n exceeds D_i. Since $w^{n+1} \geq w^n$, the iteration will either converge or terminate.

Now, equation 15 gives the window size resulting from a window starting with an offset of W_t relative to transaction t. The response time corresponding to this considered window is given by (equation 10):

$$r_i(W) = w - \left(\hat{O}_i - W\right)$$

To find the worst-case response time r_i we simply calculate w for *all combinations of* W_t *over all transactions*:

$$r_i = \max_{\forall t \in trans \forall j \in tasks(t) | W_t = \hat{O}_j} \left(w - \left(\hat{O}_i - W\right)\right) \qquad (16)$$

Where $W = W_{trans(i)}$. Obviously this is computationally infeasible for large problems: searching all combinations is at least $O(n_1 n_2 n_3 ... n_t)$ (where n_x is the number of tasks in transaction x and t is the number of transactions); later we will show how an alternative approach provides a tractable test that finds a good (*i.e.* not too pessimistic) upper bound for r_i.

Assumptions About "Deemed" Interference

We are now at last able to return to the issue of the equations *deeming* that a task released before the window causes no interference on that window, and can now offer an explanation of the validity of the equations. There are two cases to consider: firstly, a task j has *in reality* no effect on the window; this case is trivial since the deemed interference is the same as the actual interference.

The second case is the interesting case: assume that task j in reality *can* interfere within the window. Assume also that task j is released χ before the window. If the window were to be started nearer to task j (say $\chi - \Delta$ after j) then the total interference must increase by Δ. The window width w must therefore increase by at least Δ (interference from other tasks of the same transaction as j must either stay the same or increase, possibly further increasing w). Hence the window width w cannot be smaller when the window is released sooner. Eventually as Δ increases the interference from j will not increase any more. This point must occur when j is released at the same time as the window starts (this follows directly from the argument of the worst-case scheduling point occurring at a simultaneous task release).

Now, if j is released before the window starts then we will later examine another window start time that co-incides with the start of j. If, in reality, j *does* cause interference then we will adopt the latter window, since it will be bigger than the former. We can see, therefore, that by deeming interference as zero we merely postpone the full accounting of the interference — we will always examine the interference eventually, because we find the worst-case response time.

A Small Example

We have seen so far that the worst-case response time for a given task i is found by examining a number of windows within the same transaction as i. For each window we consider all the possible release combinations: for all transactions, a window W_t starting at the release of a task is considered. For small numbers of tasks this approach is acceptable.

To illustrate this, consider a two-transaction task set, with the following attributes:

trans	*T*	*task*	*pri*	*C*	*O*	$\hat{O}$
A	14	1	1	4	0	0
		2	2	4	7	7
B	100	3	4	4	4	4
		4	3	3	0	0

In this example we determine the worst-case response time of task 3 (the lowest priority task).

For task 3 we need to consider two windows, one starting at the same time as task 3, and one starting at the same time as task 4. In transaction *A* there are two tasks, and therefore two release combinations: task 1 or task 2 released at the same time as each of the window offsets. Hence a total of 2×2 combinations need to be examined.

First, consider the window corresponding to the combination where task 2 starts at the same time as the window (*i.e.* W_A=7), and where W (*i.e.* W_B) = 4 and $w^0 = 4$.

$$
\begin{aligned}
I_A(W_A) &= \sum_{k\in\{1,2,4\}\cap\{1,2\}} \left(\left\lceil \frac{W_A + w^1 - \hat{O}_k}{T_A} \right\rceil_0 - \left[W_A - \hat{O}_k \right]_0^1 \right) C_k \\
&= \left(\left\lceil \frac{W_A + w^1 - \hat{O}_1}{14} \right\rceil_0 - \left[W_A - \hat{O}_1 \right]_0^1 \right) C_1 + \left(\left\lceil \frac{W_A + w^1 - \hat{O}_2}{14} \right\rceil_0 - \left[W_A - \hat{O}_2 \right]_0^1 \right) C_2 \\
&= \left(\left\lceil \frac{7+4-0}{14} \right\rceil_0 - \left[7-0 \right]_0^1 \right) 4 + \left(\left\lceil \frac{7+4-7}{14} \right\rceil_0 - \left[7-7 \right]_0^1 \right) 4 \\
&= 4
\end{aligned}
$$

$$
\begin{aligned}
I_B(W_A) &= \sum_{k\in\{1,2,4\}\cap\{3,4\}} \left(\left\lceil \frac{W_B + w^1 - \hat{O}_k}{T_B} \right\rceil_0 - \left[W_B - \hat{O}_k \right]_0^1 \right) C_k \\
&= \left(\left\lceil \frac{W_B + w^1 - \hat{O}_4}{100} \right\rceil_0 - \left[W_B - \hat{O}_4 \right]_0^1 \right) C_4 \\
&= \left(\left\lceil \frac{4+4-0}{100} \right\rceil_0 - \left[4-0 \right]_0^1 \right) 3 \\
&= 0
\end{aligned}
$$

$$
\begin{aligned}
w^2 &= C_i + \sum_{t \in trans-trans(t)} I_t(W_t) + I_{trans(i)}(W) \\
&= C_3 + I_A(W_A) + I_B(W_B) \\
&= 8
\end{aligned}
$$

We evaluate the equations again with w^2=8, and obtain w^3 of 12; the iteration converges to at this value. The corresponding worst-case response time r_3 is given by:

$$r_i(W) = w - \left(\hat{O}_i - W\right)$$

which is equal to 12.

For the second window, we need to re-evaluate w with W_A set to $\hat{O}_1$ (*i.e.* $W_A = 0$):

$$
\begin{aligned}
I_A(W_A) &= \sum_{k \in \{1,2,4\} \cap \{1,2\}} \left(\left\lceil \frac{W_A + w^1 - \hat{O}_k}{T_A} \right\rceil_0 - \left[W_A - \hat{O}_k \right]_0^1 \right) C_k \\
&= \left(\left\lceil \frac{w^1 - \hat{O}_1}{10} \right\rceil_0 - \left\lceil \frac{\hat{O}_1}{10} \right\rceil_0^1 \right) C_1 + \left(\left\lceil \frac{w^1 - \hat{O}_2}{10} \right\rceil_0 - \left[-\hat{O}_2 \right]_0^1 \right) C_2 \\
&= \left(\left\lceil \frac{7}{14} \right\rceil_0 \right) 4 + \left(\left\lceil \frac{4-7}{14} \right\rceil_0 \right) 4 \\
&= 4
\end{aligned}
$$

As before, $I_B(W_B)$ is zero; w^2 is therefore 8. The iteration continues and converges to a value for w of 12. The corresponding value of r_3 is 12.

For the third and fourth windows we need to set $W_B = 0$ and try both values of W_A. For the third window $W_B = O_1 = 0$ and $w^0 = C_3$; we obtain:

$$
\begin{aligned}
I_A(W_A) &= \sum_{k \in \{1,2,4\} \cap \{1,2\}} \left(\left\lceil \frac{W_A + w^1 - \hat{O}_k}{T_A} \right\rceil_0 - \left[W_A - \hat{O}_k \right]_0^1 \right) C_k \\
&= \left(\left\lceil \frac{w^1 - \hat{O}_1}{14} \right\rceil_0 - \left\lceil \frac{\hat{O}_1}{14} \right\rceil_0^1 \right) C_1 + \left(\left\lceil \frac{w^1 - \hat{O}_2}{14} \right\rceil_0 - \left[-\hat{O}_2 \right]_0^1 \right) C_2 \\
&= \left(\left\lceil \frac{4}{14} \right\rceil_0 \right) 4 + \left(\left\lceil \frac{4-7}{14} \right\rceil_0 \right) 4 \\
&= 4
\end{aligned}
$$

$$I_B(W_A) = \sum_{k\in\{1,2,4\}\cap\{3,4\}} \left(\left\lceil \frac{W_B + w^1 - \hat{O}_k}{T_B} \right\rceil_0 - \left[W_B - \hat{O}_k \right]_0^1 \right) C_k$$

$$= \left(\left\lceil \frac{w^1}{100} \right\rceil_0 - [0]_0^1 \right) 3$$

$$= 3$$

Hence $w^1 = 11$. We continue to calculate values of w until convergence to a value of 19. The corresponding value of r_3 is:

$$r_i(W) = w - \left(\hat{O}_i - W \right)$$

which is equal to 19 – (4 – 0) = 15.

Finally we need to calculate w for $W_A = \hat{O}_2$ and $W_B = 6$:

$$I_A(W_A) = \sum_{k\in\{1,2,4\}\cap\{1,2\}} \left(\left\lceil \frac{W_A + w^1 - \hat{O}_k}{T_A} \right\rceil_0 - \left[W_A - \hat{O}_k \right]_0^1 \right) C_k$$

$$= \left(\left\lceil \frac{7 + w^1 - \hat{O}_1}{14} \right\rceil_0 - \left[7 - \hat{O}_1 \right]_0^1 \right) C_1 + \left(\left\lceil \frac{7 + w^1 - \hat{O}_2}{14} \right\rceil_0 - \left[7 - \hat{O}_2 \right]_0^1 \right) C_2$$

$$= \left(\left\lceil \frac{7+4}{14} \right\rceil_0 - 1 \right) 4 + \left(\left\lceil \frac{4}{14} \right\rceil_0 \right) 4$$

$$= 4$$

$$I_B(W_A) = 3$$

$$w^1 = 11$$

We converge eventually to a value of $w = 19$, with the corresponding value of $r_3 = 15$.

We have calculated four values of r_3 (12, 12, 15, 15) and take the largest: 15. Hence the worst-case response time of task 3 is 15. Compare this to the original analysis (equation 4), assuming that tasks 1, 2, and 4 are always released at the same time as task 3 — using this analysis r_3 has a value of 23. Clearly, if D_3 were 20 then the offset analysis would be able to guarantee the task whereas the original analysis would not. We present below the STRESS[7] diagram for the worst-case scenario determined above (the notation used in STRESS diagrams is described in Appendix D).

[7] *STRESS* is a graphical simulation tool developed at York for studying scheduling theory and simulating system behaviour; a description of the capabilities of the tool can be found in the report by Audsley *et al* [3]