THE END OF THE LINE FOR STATIC CYCLIC SCHEDULING?

N. Audsley, K. Tindell, A. Burns

Department of Computer Science, University of York, England

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 for each task by assuming 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, giving an optimal priority ordering algorithm.*

1. Introduction

Classic static priority sufficient and necessary schedulability is determined assuming a 'critical instant': all tasks share a simultaneous release time [1]. Under this condition, rate monotonic priority assignment (for tasks whose deadlines equal their periods) and deadline monotonic priority assignment (where task deadlines may be less than their periods) are known to be optimal[1] [2]. However, when tasks are permitted to have offset relationships, a critical instant may never occur. This has two consequences: firstly, neither rate monotonic nor deadline monotonic priority assignments are optimal; secondly, although existing schedulability tests could be used, they are pessimistic.

Previously, this has led to systems that could be scheduled using static cyclic scheduling technology [3] but that were deemed unschedulable when using fixed priority scheduling.

[1] optimal in the sense that if the algorithm is unable to find a priority ordering where all tasks are schedulable then no priority ordering exists where all tasks are schedulable

The following section describes the assumed computational model. Section 3 provides a practical motivation for the use of offsets in hard real-time system design. Section 4 describes an efficient optimal priority assignment algorithm for tasks with offsets. Section 5 gives analysis to bound worst-case response times for such tasks. Section 6 shows how the analysis enables scheduling of task sets with precedence and exclusion constraints, previously deemed schedulable using only static cyclic scheduling technology. Concluding remarks are offered in Section 7.

2. 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). Tasks are permitted to have response times greater than their periods (and hence have arbitrary deadlines [6]). Furthermore, tasks are assumed to have a *release jitter* — this occurs when the *arrival time* (*i.e.* time when a task wishes to run) and the *release time* (*i.e.* the time when the task is placed in the priority-ordered run-queue) are not

[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

the same [7] (for example, a task may be delayed by the polling of a tick scheduler, or perhaps awaiting the arrival of a message).

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.

Consider 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.

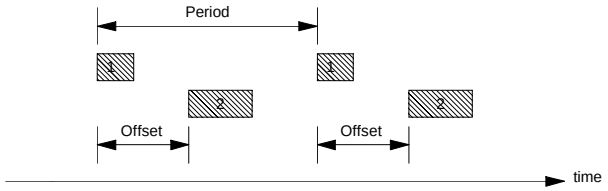

Figure 1: the shaded boxes represent computation time

3. Motivation 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).

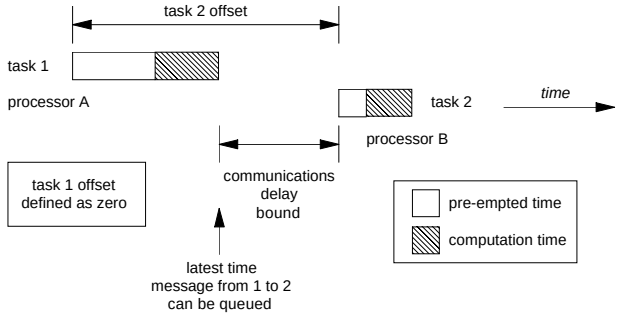

Figure 2: task 2 runs when message is guaranteed to have arrived

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 [4], [5]). 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 [9]).

Another major use for offsets is to permit tight jitter bounds on an input or output action. As described by Locke [10], 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.

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.* [10].

4. Optimal Priority Ordering

Neither the deadline monotonic nor rate monotonic priority ordering policies are optimal for tasks with arbitrary deadlines nor for tasks with offset relationships. We now describe the optimal 'bottom up' priority ordering algorithm, that is guaranteed to find a feasible priority ordering if one exists. We reproduce here the optimal priority ordering algorithm of Audsley (the derivation and proof of the algorithm is given by Audsley [11]).

The algorithm works as follows: a priority ordering is partition into two parts: a sorted part, consisting of the lower *n* priority tasks, and the remaining unsorted higher priority tasks. Initially the priority ordering is an arbitrary one, and all tasks are unsorted. All tasks in the unsorted partition are chosen in turn and placed at the top of the sorted partition and tested for schedulability. If the chosen task is schedulable then the priority of the task is left as it is, and the sorted partition extended by one position. If the task is not schedulable it is returned to its former priority. This continues until either all tasks in the unsorted paritition have been checked and found to be unschedulable (in which case there is no priority ordering resulting in a schedulable system), or else the sorted partition is extended to the whole priority map (in which case the priority ordering is a feasible one).

An arbitrary priority ordering is chosen in an array, with 0 being the highest priority, and $N-1$ the lowest (N denotes the number of tasks in the system; the algorithm assumes $N > 1$). The following pseduo-code details the algorithm:

```
ordered := N
repeat
   finished := false
   failed := true
   j := 1
   repeat
       insert j at priority ordered
       if j is schedulable then
           ordered := ordered - 1
           failed := false
           finished := true
       else
           insert j back at old priority
       end if
       j := j + 1
   until finished or j = ordered
until Ordered = 1 or failed
```

At all times the sorted partition is schedulable, since the priority ordering within the unsorted parition cannot affect the sorted tasks. The sorted partition increases in size until either all the tasks are schedulable, or none of the top *n* tasks are schedulable at priority *n*. The analysis has the property that decreasing the priority of a task cannot lead to a decrease in worst-case response time (*i.e.* a decrease in priority cannot increase schedulability). Therefore, in the case where none of the top *n* tasks is schedulable at priority *n* no priority ordering can exist where all tasks are schedulable. Therefore the algorithm must be considered optimal. Furthermore, this algorithm holds for any scheduling test where worst-case response time is monotonic with decreasing priority (*i.e.* where decreasing the priority of a task does not lead to a decrease in the worst-case response time of that task).

The algorithm has complexity $O((n^2 + n)\, E)$. This reflects that at most $(n^2 + n) / 2$ different priority orderings are examined (from a maximum of $n!$). E represents the complexity of the schedulability test.

5. The Offset Test

The analysis given in this section calculates the worst-case response time of a given task *i*, assumed by the algorithm given above. The following equations give the test:

$$r_i = \max_{\forall significant\, S} \left(w_{i,S} + J_i - S \right)$$

r_i is the worst-case response time of a task *i* (measured from the arrival of the task to the completion of all the computation of the task). J_i is the worst-case release jitter

[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

(*i.e.* the worst-case time between a task arrival time and release time). The 'significant *S*' term is defined as:

$$S \in \{\forall j \in tasks(trans(i)) \cap hp(i) \left(q + k_j\right) \bullet T_j - O_j - J_j + O_i + J_i\}$$

where q is 0, 1, 2, 3, ...; $trans(i)$ is defined as the transaction of which i is a member, $tasks(t)$ is the set of all tasks that are members of transaction t, $hp(i)$ is the set of all tasks of higher priority than i, and k_j is defined by:

$$k_j = \left\lceil \frac{J_j + O_j - O_i - J_i}{T_j} \right\rceil \qquad (1)$$

O_i is the offset of task i measured relative to the start of the transaction of which i is a member. T_j is the period of task j (note that T_i is equal to T_j, since all tasks in the same transaction share the same period)

Now, $w_{i,S}$ (which is the length of the 'busy period' [6] starting at time S before the release of task i) is given by:

$$w_{i,S} = C_i + a_i C_i + B_i + \sum_{\forall j \in tasks(trans(i)) \cap hp(i)} I_j + \sum_{\forall t \in trans - trans(i)} H_t \qquad (2)$$

Where *trans* is the set of all transactions C_i is the worst-case computation time requirement of task i, B_i is the worst-case blocking time task i can experience (derived from the priority ceiling protocol [8] for locking semaphores), and a_j is given by:

$$a_j = \left\lfloor \frac{S - O_i - J_i + O_j + J_j}{T_j} \right\rfloor \qquad (3)$$

and I_j is given by:

$$I_j = \left\lceil \frac{w_{i,S} - S + O_i - O_j + J_i + a_j T_j}{T_j} \right\rceil C_j \qquad (4)$$

and H_t is given by:

$$H_t = \max_{\forall j \in tasks(t) \cap hp(i)\, W_t = O_j + J_j} \left[\begin{array}{ll} \sum_{\forall k \in tasks(t)} \left\lceil \frac{w_{i,S} + W_t - O_k}{T_k} \right\rceil C_k & \text{if } O_k + J_k \geq W_t \\ \sum_{\forall k \in tasks(t)} \left\lceil \frac{w_{i,S} + W_t - O_k - T_k}{T_k} \right\rceil C_k & \text{otherwise} \end{array} \right]$$

The evaluation of r_i for increasing values of q continues until $w_{i,S} + J_i - S \leq T_i$ for all values of j at a particular value of q.

6. Static Priority *vs* Static Cyclic Scheduling

So far we have shown how periodic (or sporadic) transactions of tasks, with time offsets between tasks of the same transaction, can be analysed using newly derived scheduling theory. This theory is able to find good worst-case response time bounds. We now discuss some of the ramifications of the offset scheduling theory.

Unrelated strictly periodic tasks sharing the same period can be incorporated into the same transaction, with offsets between the tasks. This increases schedulability because the computation due to these tasks can be 'spread out'. Tasks with similar but different periods can be transformed into tasks sharing the same period by choosing a common period which is smaller than the original periods. This has the advantage of reducing the least common multiple of the task periods, and simplifying the transaction (at the expense of a small loss in schedulability). Tasks with periods which are exact divisors of a given transaction period can also be incorporated into the transaction by adding multiple instances of the same task with offsets between them. The offset approach is more general that this, however, since it allows any instance to have different attributes (*e.g.* priority); this generality can be used to improve schedulability (for example, if only one instance of a task were unschedulable then the priority of just that instance might be increased).

The above description of constructing transactions will be familiar: it is exactly the procedure that is adopted in finding static cyclic schedules [3]. It therefore reasonable to conclude that the Offset Test, coupled with a method for configuring tasks (choosing priorities [12], offsets, *etc.*), provides a means by which static cyclic schedules can be analysed. However, the offset approach provides a major additional benefit: other transactions are also permitted to run concurrently. Thus a number of sporadic and periodic tasks can be run alongside any cyclic schedule. Indeed, a number of cyclic schedules can be run concurrently on the same processor. Thus we can see that the Offset Test bridges the gap between static cyclic scheduling and static priority pre-emptive scheduling, and that static cyclic scheduling is merely a special case of the more general pre-emptive scheduling algorithm.

Meeting Precedence and Exclusion Constraints

This paper has already mentioned how concurrency control can be obtained using the priority ceiling protocol to guard small critical sections (we are also able to use offset and priority assignment to ensure total exclusion[5] [3]). We have yet to adequately address precedence constraints.

Constraint: *Task B is constrained to run only when a task A has finished*

This constraint can be achieved two ways: through offsets and through priority. We deal with the priority approach first. If task *B* is assigned a lower priority than task *A* and is released at the same time as (or after) task *A* then task *A* will immediately pre-empt *B* (or already be released and delay *B*). Task *B* will not be dispatched until it becomes the highest priority task; this cannot happen until task *A* has finished. Therefore we can say that the precedence constraint will be satisfied if:

$$\left(O_B \geq O_A \wedge O_B + r_B \leq O_A + T\right) \wedge \text{pri}(B) < \text{pri}(A) \quad (5)$$

Given that task *B* is of lower priority than task *A*.

The following approach can be used when task *B* has a lower *or* higher priority than task *A*: if task *B* has an offset larger than the offset of *A plus* the worst-case response time of *A* then task *B* will never be released before *A* is finished, *i.e.*:

$$\left(O_B \geq O_A + r_A\right) \wedge \left(O_B < O_A\right)$$

We have to be careful about periodic task execution where the subsequent invocation of a task may disturb the precedence relationship. For example, if task *A* were assigned a priority higher than task *B* then the subsequent re-arrival of task *A* could pre-empt a currently running invocation of *B*. If we additionally require exclusion between *B* and the subsequent invocation of *A* then we must ensure that *B* always finishes before *A* is re-released. Alternatively, we could turn task *A* into two logically different tasks, each of period twice that of the original, with an offset equal to the original period between the two tasks. We could then assign a lower priority to the second task *A* so that task *B* would not be pre-empted. In the example we take precedence to be stronger than exclusion and hence require that task *B* terminate before the subsequent invocation of task *A*.

[5] Xu and Parnas [3] use the term 'exclusion' between two tasks to mean no part of the execution of either task can overlap; where the critical section is small a total exclusion constraint is too harsh, and an alternative concurrency control protocol (such as the priority ceiling protocol) can be used more efficiently; of course, the priority ceiling protocol cannot be used in a static cyclic schedule

Example Task Set

Xu and Parnas [3] provide an example which they claim cannot be scheduled using either fixed or dynamic priority approaches. We include this example here and show how offset test analysis can be used to show how it can be feasibly scheduled with fixed priority scheduling.

Five tasks, *A*-*E*, all share the same period, have exclusion and precedence relationships:

1. *B* precedes *D*
2. *A* excludes *D*
3. *A* excludes *B*.

The following table gives the requirements, as stated by Xu and Parnas:

	d	*r*	*C*
A	161	0	30
B	51	11	30
C	90	60	30
D	100	41	10
E	140	90	50

C is the worst-case execution time; *d* is the deadline of the task, measured from time zero; r_{Xu} is the minimum release time measured from time zero.

Now consider the modified task set we can obtain:

	d	*r*	*C*	*D*	*O*	*pri*	*r*	*r*	
A	161	0	30	110	51	5	110	150	✓
B	51	11	30	40	11	1	30	30	✓
C	90	60	30	30	60	3	30	70	✓
D	100	41	10	59	41	2	10	40	✓
E	140	90	50	50	90	4	50	120	✓

The timing constraint *d* stated by Xu and Parnas is measured relative to time zero. In our offset analysis we measure the deadline from the release time of the task. Therefore we obtain a new deadline *D* measured from the release of the task and equal to *d* – *O*. Furthermore, r_{Xu} is the minimum release time of the task; *O* must be greater than this required value.

We have set the offset of task *A* such that task *A* cannot not execute before time 51. Note that in this example

there are only five tasks, all members of the same transaction — using a pre-emptive scheduler in a larger system we could have other sporadic and periodic tasks (using the static cyclic scheduling approach it is very difficult to guarantee tight bounds on servicing sporadic requests).

In the above table, the worst-case response time of the task, r, is measured relative to O, the offset (which is itself measured relative to time zero). The column headed r_{old} is the old scheduling analysis applied to the same problem; we can see how poor the performance of the old analysis is in this situation. The ✓ symbol indicates that the timing constraint ($r \leq D$) has been met.

7. Summary and Conclusions

This paper has provided extensions to static priority scheduling theory for tasks with offset relationships. This has been achieved in two ways: efficient optimal priority assignment and less-pessimistic worst-case response time analysis. This theory is shown to provide sufficient analysis for task sets previously thought schedulable using only static cyclic scheduling technology. Thus we have illustrated a form of coverage equivalance between static priority and static cyclic scheduling.

8. References

[1] C. L. Liu and J. W. Layland, "*Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment,*" Journal of the ACM 20(**1**) (1973) pp.46-61.

[2] J. Y. T. Leung and J. Whitehead, "*On The Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks,*" Performance Evaluation (Vol. 2, Part 4, Dec 1982) pp.237-250.

[3] J. Xu and D. L. Parnas, "*Scheduling Processrs with Release Times, Deadlines, Precedence, and Exclusion Relations,*" IEEE Transactions on Software Engineering 16(**3**) (March 1990) pp.360.

[4] K. W. Tindell, A. Burns and A. J. Wellings, "*Guaranteeing Hard Real Time End-to-End Communications Deadlines,*" Real Time Systems (Submitted).

[5] J. K. Strosnider, T. Marchok and J. Lehoczky, "*Advanced Real-Time Scheduling Using the IEEE 802.5 Token Ring,*" Proceedings of the IEEE Real-Time Systems Symposium (1988) pp.42-52.

[6] J. P. Lehoczky, "*Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines,*" Proceedings 11th IEEE Real-Time Systems Symposium (5-7 Decmeber 1990) pp.201-209.

[7] N. Audsley, A. Burns, M. Richardson, K. Tindell and A. Wellings, "*Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling,*" Report RTRG/92/120 Department of Computer Science, University of York (February 1992).

[8] L. Sha, R. Rajkumar and J. P. Lehoczky, "*Priority Inheritance Protocols: An Approach to Real-Time Synchronisation,*" IEEE Transactions on Computers 39(**9**) (September 1990) pp.1175-1185.

[9] H. Kopetz and W. Ochsenreiter, "Clock Synchronisation in Distributed Real-Time Systems," IEEE Transactions on Computers C-36(8) (August 1987).

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

[11] N. C. Audsley, "Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times," YCS 164, Dept. Computer Science, University of York (December 1991).

[12] K. Tindell, A. Burns and A.J. Wellings, "Allocating Real-Time Tasks (An NP-Hard Problem made Easy)," Real-Time Systems 4(2) (June 1992) pp.145-165.