DUAL PRIORITY ASSIGNMENT:

A Practical Method for Increasing Processor Utilisation†

A. Burns
A.J. Wellings

Real-Time Systems Research Group
Department of Computer Science
University of York, UK

ABSTRACT

Static priority schemes have the disadvantage that processor utilisations less than 100% must be tolerated if a system is to be guaranteed off-line. By comparison earliest deadline scheduling can theoretically utilise all of a processors capacity, although in practice run-time overheads are increased that undermine this advantage. In this paper a dual priority scheme is proposed: a task may execute in two phases; each phase has a static priority assigned, the transition from one phase to another is made at a fixed offset in time from the release of the task. Utilisations equal to 100% can now be obtained whilst retaining an essentially static priority model. Examples of the scheme are given and some of its properties investigated. Implementation issues are also considered.

1. INTRODUCTION

In their seminal paper of 1973 Liu and Layland[1] introduced the fixed priority scheme known as *rate monotonic priority assignment*, and the dynamic priority scheme known as *earliest deadline*. Their theoretical treatment produced the following important results for a set of n periodic process with period T equal to deadline D; and computation time, C:

- The process set is schedulable by the static rate monotonic priority assignment scheme if

$$U \leq n\left(2^{\frac{1}{n}} - 1\right) \tag{1}$$

- The process set is schedulable by the dynamic earliest deadline scheme if

$$U \leq 1 \tag{2}$$

In both of these equations the processor utilisation figure, U, is given by

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

Test (1) converges, approximately, on the value 0.69 for large n. For n equal to 2 it is close to 0.83. These values are however pessimistic in that is assumes the worst period phasings for the task set. Necessary and sufficient tests that take into account the periods of the tasks have been given more recently by Sha *et al*[2].

---

† This report represents a slightly extended version of a paper (by the same name) that appears in the proceedings of the 5th Euromicro Workshop on Real-Time Systems, Oulu, Finland, IEEE Computer Society Press, June 1993, pp48-53.

The advantage of the fixed priority scheme is that it leads to a simple and efficient run-time support system. Dynamic priority schemes must hold more information at run-time and have increased overheads. Nevertheless the greater possible utilisation makes these schemes attractive. Since 1973 there has been a continuous and rigorous debate about which scheme is the more appropriate.

In this paper a minimal dynamic scheme is introduced that allows a task to change (increase) its priority during its execution. Each task has at most two priority levels: many tasks will continue to have only one priority. At run-time standard preemptive priority based scheduling is required.

The paper is organised as follows. The remainder of this section describes a framework for the paper. The priority assignment scheme is introduced in the next section and examples of its use are given. In section 3 a method of constructing the necessary priorities is given; and some properties of the model are derived in section 4. Section 5 then considers the practical ramifications of the analysis. Some concluding remarks are contained in the final section.

1.1. Framework

The analysis presented assumes a fixed set of n periodic tasks denoted as τ_1, ..., τ_n. The subscripts are ordered according to period length, with τ_1 having the shortest period. Each task τ_i has a computation requirement of C_i, a period of T_i and a deadline D_i which is equal to its period. In addition to this usual notation each task has a intermediate deadline S_i at which time it undergoes a step change (increase) in priority. For all tasks:

$$0 < S_i \leq D_i$$

Each task has a *phase_1* priority P_i^1 and if $S_i < D_i$ a *phase_2* priority P_i^2.

The following assumptions are used throughout the analysis:

Assumption A1:

Any task, in either phase, can be preempted by any other task running at a higher priority.

Assumption A2:

Tasks do not suspend themselves other than at the end of their computations.

Assumption A3:

The time required to perform context switching, priority changes etc is ignored (i.e. assumed to be zero).

The consequence of removing the third (unrealistic) assumption is considered in section 5. The assumption enables the basic analysis to be undertaken in a straightforward way.

All analysis presented in this paper assumes a single processor implementation, or a static allocation of tasks to nodes in a distributed architecture.

2. MOTIVATION FOR THE SCHEME

If a feasible task set (i.e. utilisation not greater than 1) fails the necessary and sufficient test for rate monotonic priority assignment then a fully static scheme cannot be used. However it is not necessarily the case that a complete change of run-time behaviour is needed in order to guarantee the task set. Ideally a task set that just fails the test could be subject to a minor change that enables it to become guaranteed. If such a scheme were available then there would no longer be such a fundamental difference between the rate monotonic and earliest deadline models.

The earliest deadline scheme results in potential 100% utilisation because it allows a task to increase its priority as its deadline approaches. The scheme presented in this paper allows a task (τ_i) to have two phases for its execution. During the first phase, i.e. in the interval [0, S_i), it will execute if P_i^1 is the highest priority value for runnable tasks. If it still has computations to do after S_i then it will execute when P_i^2 is the highest priority value for runnable tasks. A simple example will illustrate the benefits of this scheme. Table 1 gives the details of a three task systems. Note that priority 1 is high and 4 is low.

	T	*C*	*P*	*U*
τ_1	6	3	2	50%
τ_2	8	2	3	25%
τ_3	12	3	4	25%
Total Utilisation	100%			
LCM of Task Set	24			

Table 1: Task Set E1

If the dual phasing is ignored (i.e. the tasks are treated as having single priorities) then the task set is not schedulable by rate monotonic priority assignment (or any other static priority assignment scheme as rate monotonic is optimal). This is illustrated in Figure 1 where a simulation run is presented using the *STRESS* facility[3].

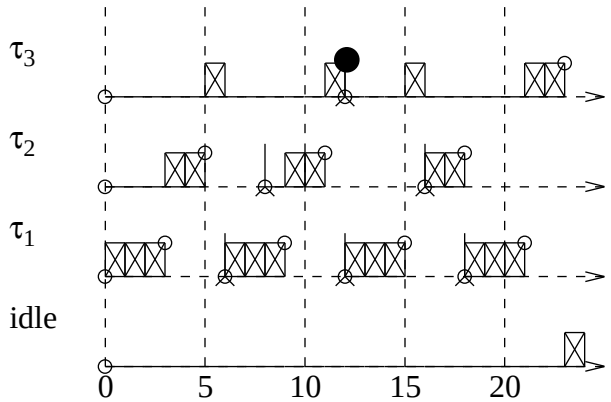

Figure 1: Task Set E1 with Rate Monotonic Priority Assignment

A black disk indicates a missed deadline; for example τ_3 misses its first deadline at time 12. It therefore loses its third computation tick. Hence, even though total utilisation is 100%, the idle task is executed prior to the task set's LCM. Task τ_3 fails because τ_2 executes at the critical time. This can be countered by allowing τ_3 to raise its priority (to above that of τ_2) two ticks before its

deadline (2 because it executes in tick 11 anyway). Figure 2 now illustrates the execution sequence — all deadlines are met.

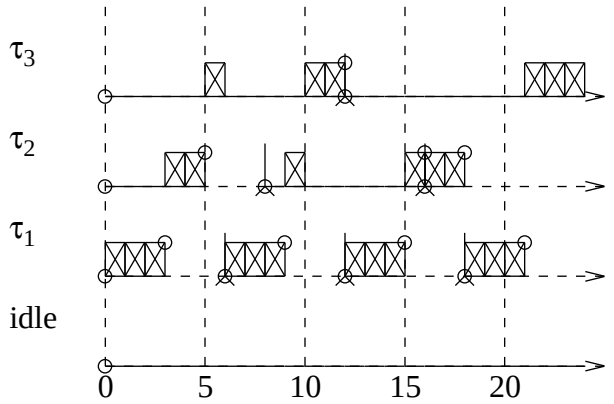

Figure 2: Task Set E1 with Dual Priority Assignment

For completeness Figure 3 shows the execution of the task set with earliest deadline scheduling — again all deadlines are met although the sequence of execution is not the same.

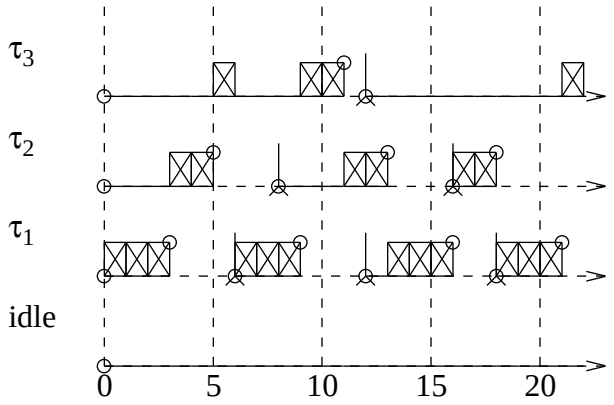

Figure 3: Task Set E1 with Earliest Deadline Scheme

The model advocated in this paper allows a task to have its priority raised after a period of (real) time following its release. Harbour *et al*[4] also consider utilisation increases by fixed priority schemes. They observe that by allowing a task to increase its priority after a period of *computation time*, task sets can be made schedulable that were otherwise not possible to guarantee. In effect they split a task into precedence constrained subtasks. The method presented here differs as the priority change occurs after a period of *elapse* time; as a result each "subtask" does not necessarily undertake the same amount of computation on each release. As a result is it possible to guarantee task sets by the dual priority methods that cannot be guaranteed by using precedence constrained subtasks — this is illustrated by an example later (E4).

3. PRIORITY ASSIGNMENT

The motivation behind the dual priority scheme is to make it minimally dynamic. Hence it has been found to be adequate for a task to have only two priority phases. Moreover only the minimum number of tasks have these two phases; in the example above τ_1 and τ_2 are "normal" single fixed priority tasks.

The proposed model has the following axioms:

- All phase_1 priorities are assigned using the rate monotonic scheme.
- If a task has a phase_2 priority then it is higher than its phase_1 priority.

Thus as you move down the task set the phase_1 priorities decrease whilst the phase_2 priorities may actually increase (if they exist). In general if task τ_i has a phase_2 priority then all tasks τ_j, $i < j$ have phase_2 priorities. The purpose of the phase_2 priority is to "steal" execution time (termed *ticks* in the following discussions) from higher priority phase_1 executions. "Steal" in the sense that "higher" priority computations are being postponed until after the deadline of the "stealing" task.

A consequence of the above rules is that τ_1 never has a phase_2 priority — there is no higher priority phase_1 task from which to steal. Hence for n tasks at most $2n-1$ priority levels are needed. Figure 4 shows an extreme 4 task case where three tasks have phase two priorities (the integers in the boxes represent the priority values).

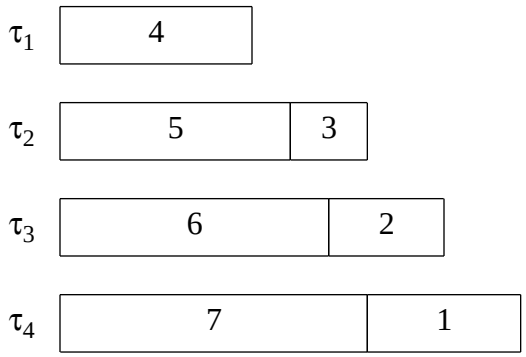

Figure 4: Example Task Priorities

There are two major questions concerning the dual priority scheme:

- Which task set can be guaranteed by the approach?
- How can the appropriate step times and phase_2 priorities be found?

The answer to the first takes the form, for now, of a conjecture:

Conjecture C1

For any task set with total utilisation less than or equal to 100% there exists a dual priority assignment that will meet all deadlines.

The second question is answered in the following subsection in which a mechanism (algorithm) is given for constructing the dual priority assignment.

3.1. Constructive Phase 2 Attributes

In this section we shall describe the construction algorithm. Proof of its properties are left until the following section.

The proposed algorithm starts with each task having a single phase_1 priority assigned by the rate monotonic scheme. Each task in then considered, in turn, starting with τ_n (and working towards τ_1). If the task in question fails to meet its deadline then the worst release time is considered (this corresponds to the task's *critical instance*). The step time is progressively moved back from T until the task meets all its deadlines; the priority of the phase_2 is sufficiently high so that it can steal from the appropriate task (or tasks). For example if τ_i only needs to steal from

τ_{i-1} then its phase_2 priority will be set to a value between the phase_1 priorities of τ_{i-1} and τ_{i-2}; alternatively if τ_i has to steal from all higher priority tasks (or even just τ_1) it must have a phase_2 priority higher than any phase_1 priority.

Having accommodated one task the next task is considered. When finally τ_2 has been analysed, τ_1 is automatically guaranteed (see discussion in section 4).

Two examples are now given which illustrate this process. In each example the total system utilisation is 100% (i.e. the task sets are potentially very difficult to schedule). Example E2 is shown in Table 2:

	T	*C*	*P*	*U*
τ_1	12	3	4	25%
τ_2	16	4	5	25%
τ_3	20	4	6	20%
τ_4	20	6	7	30%
Total Utilisation	100%			
LCM	240			

Table 2: Task Set E2

With no task having a second phase, the system is not schedulable; this is illustrated in Figure 5:

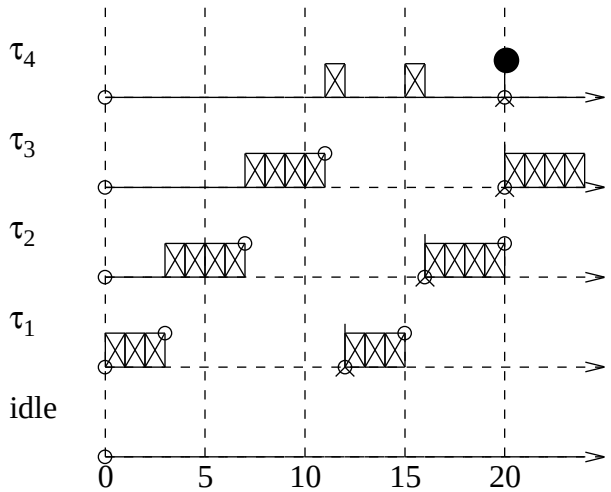

Figure 5: Task Set E2 — First Attempt

As expected τ_4 fails its first deadline. It is 4 time units short. Hence the first attempt for S_4 is 16; with P_4^2 set at a value between P_2^1 and P_1^1. These values were found by the use of a simulation facility. The new behaviour is illustrated in Figure 6:

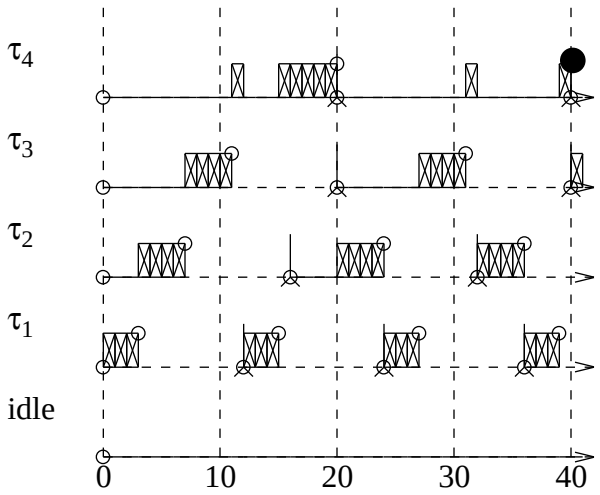

Figure 6: Task Set E2 — Second Attempt

Now τ_4 fails at time 40 (four ticks short). Its next tick must however now be stolen from τ_1 and hence its phase_2 priority is raised to be above the priority of τ_1. Unfortunately when S_4 is reduced to 15, τ_4 missed (by one tick) at time 60. However with S_4 equal to 14 τ_4 meets all deadlines; but now τ_3 fails at time 140 (see Figure 7).

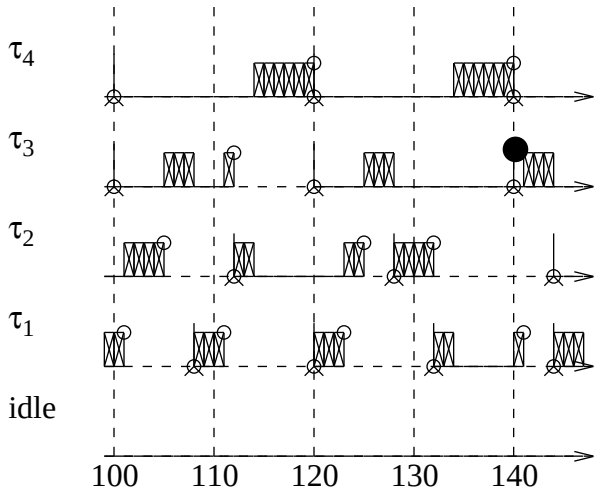

Figure 7: Task Set E2 — Third Attempt

To obtain schedulability for τ_3 it is necessary to reduce S_3 to 13, and to also increase its phase_2 priority to be above that of τ_1. All deadlines for all tasks are now met (see Figure 8).

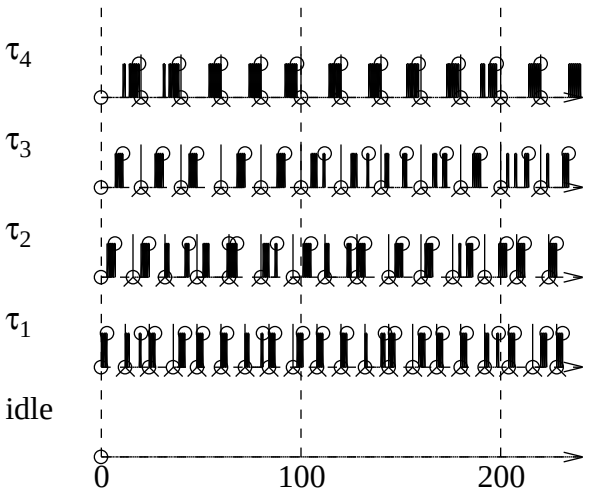

Figure 8: Task Set E2 — Fourth Attempt

Table 3 now has the complete phase_2 information for the task set:

	T	*C*	*S*	P^1	P^2	*U*
τ_1	12	3	-	4	-	25%
τ_2	16	4	-	5	-	25%
τ_3	20	4	13	6	2	20%
τ_4	20	6	14	7	1	30%
Total Utilisation	100%					
LCM	240					

Table 3: Task Set E2

The next example has two tasks that would miss their deadlines if straight rate monotonic priority assignment were used. The task set is defined in Table 4.

	T	*C*	*P*	*U*
τ_1	16	4	4	25%
τ_2	20	5	5	25%
τ_3	28	11	6	39.29%
τ_4	56	6	7	10.71%
Total Utilisation	100%			
LCM	560			

Table 4: Task Set E3

If this task set is analysed (with static priorities) it is clear that τ_4 gets no computation in its first interval [0, 56). Hence S_4 must be made equal to 50. Interestingly, the use of precedence constrained tasks is not appropriate here as τ_4 does not even get a single tick at priority 7

following its first release. Note that in subsequent releases one or two ticks are consumed at this level and hence to run τ_4 always at priority level 1 is incorrect.

Having dealt with τ_4 there remains the problem that τ_3 also missed its first deadline (see Figure 9).

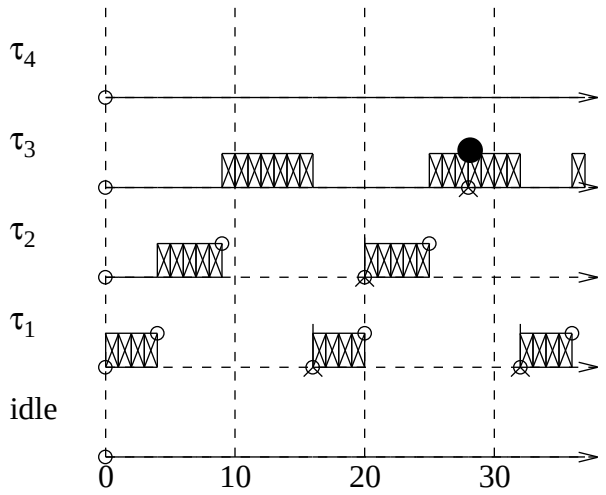

Figure 9: Task Set E3 — First Attempt

Task τ_3 needs to run in favour of τ_2 for the final period of its execution. To achieve this S_3 must be set to 19. This then results in τ_2 missing its 17th deadline (i.e. at time 340), see Figure 10.

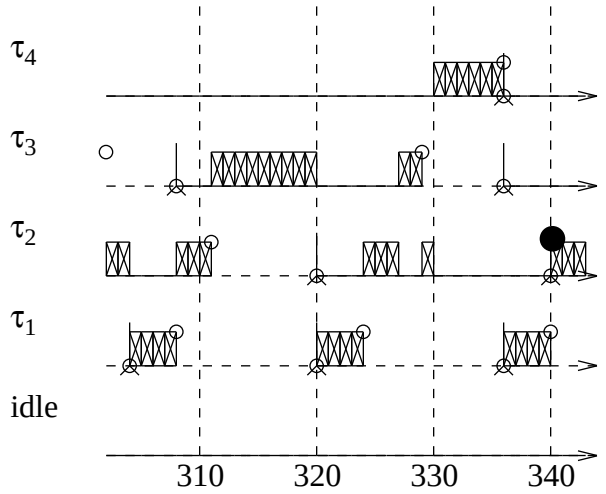

Figure 10: Task Set E3 — Second Attempt

It must preempt τ_1, and hence S_2 is set to 19. The task set is now guaranteed with the attributes described in Table 5.

	T	*C*	*S*	P^1	P^2	*U*
τ_1	16	4	-	4	-	25%
τ_2	20	5	19	5	3	25%
τ_3	28	11	19	6	2	39.29%
τ_4	56	6	50	7	1	10.71%
Total Utilisation	100%					
LCM	560					

Table 5: Task Set E2

The proposed algorithm works by allowing a "lower priority" task to preempt a "higher priority" task when its deadline is nearer and it needs the preemption. If all its computations were already undertaken then such preemption is not needed.

Even for task sets with 100% utilisation there is more than one method of assigning step times and priorities; the above approach however is adequate and systematic. For arbitrary task sets where some task's executions are long compared to others tasks periods, it is not necessary to raise the phase_2 priority of a task more than is necessary. This section is concluded by a further example of this type; details of the task set are given in Table 6.

	T	*C*	*P*	*U*
τ_1	4	1	4	25%
τ_2	6	1	5	16.67%
τ_3	12	3	6	25%
τ_4	30	5	7	16.67%
τ_5	36	6	8	16.67%
Total Utilisation	100%			
LCM	180			

Table 6: Task Set E4

The final task (τ_5) misses its first deadline by 3 ticks. However on this occasion it does not need to preempt τ_1, τ_2 or τ_3, hence it must raise its priority to (in effect) 6.5. This results is τ_4 missing just one tick at time 150. To overcome this is must steal one tick from τ_3. The task set is then schedulable — see Table 7 for necessary attributes and Figure 11 for an illustration of the execution sequence up to the first LCM.

	T	C	S	P^1	P^2	U
τ_1	4	1	-	4	-	25%
τ_2	6	1	-	5	-	16.67%
τ_3	12	3	-	7	-	25%
τ_4	30	5	29	9	6	16.67%
τ_5	36	6	23	10	8	16.67%
Total Utilisation	100%					
LCM	180					

Table 7: Task Set E4

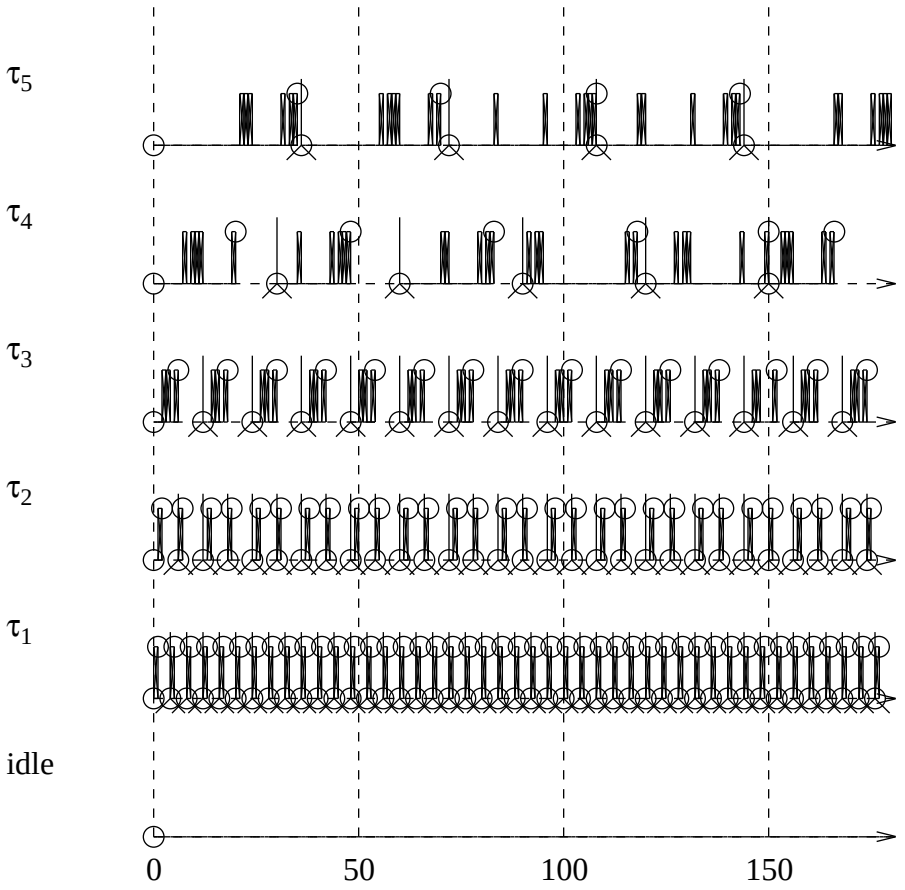

Figure 11: Task Set E4

4. PROPERTIES OF THE MODEL

In order for the above scheme to work it must be possible for a task that would otherwise miss its deadline to steal ticks from tasks with higher phase_1 priorities. A task's critical release will occur when it gets the maximum interference from higher priority phase_1 and phase_2 subtasks. However the theoretically worst critical release (all higher priority phase_1 tasks released at the same time and all lower priority task with identical completion times) may not arise in practice.

To prove that a task can always steal execution time if the utilisation of the task set is less than (or equal to) 100% we follow the spirit of the proof of the earliest deadline test given by Liu

and Layland — i.e. their Test(2)[1]. Consider task τ_i with an initial single priority P[1]. The other tasks fall into three categories:

α Tasks with higher phase_1 priorities; i.e. τ_1 .. τ_{i-1}.

β Tasks with lower phase_1 and lower phase_2 priorities.

γ Tasks with lower phase_1 but higher phase_2 priorities.

Assume that τ_i has a deadline at time Q. With Q being some time between time 0 and the LCM of the task set. From time 0 it should have done the following amount of computation:

$$\frac{Q}{T_i} \; C_i$$

It can steal from uncompleted α tasks and hence the total amount of computation done by α tasks is

$$\sum_{j=1}^{i-1} \left(\left\lfloor \frac{Q}{T_j} \right\rfloor \; C_j \right)$$

All β and γ tasks with deadline before (or at) Q will also have completed:

$$\sum_{j=i+1}^{n} \left(\left\lfloor \frac{Q}{T_j} \right\rfloor \; C_j \right)$$

The only other computations that will take place before Q come from γ tasks that have a priority step before Q and a deadline after Q

Let R be the time of the first such γ task phase_2 execution:

$$\left(Q - T_i \right) \; < \; R \; < \; Q$$

In the worst case, all computations after R up to the completion of all γ tasks will be undertaken by γ tasks — this follows from the definition of the step times to be as late as possible, commensurate with deadline satisfaction. Thus if τ_i is to meet its deadline at Q it must actually complete by R.

If one now considers the execution of tasks in the interval $[Q - T_i, Q)$ then they behave in an equivalent way to earliest deadline (i.e. all deadlines before Q are satisfied, τ_i meets its deadline at R, with $R < Q$ all computations after R are needed by tasks with deadlines after Q. As the earliest deadline scheme has a 100% utilisation bound it follows that τ_i must meet its deadline with this utilisation bound. What is true for τ_i is true for all other tasks.

5. PRACTICAL IMPLEMENTATION

The motivation for the dual priority model is to have the minimum dynamic behaviour for the utilisation demanded. This may involve only a single task having a phase_2 priority. The run-time behaviour is thus still essentially static. If the kernel (run-time support system) does not directly support phase_2 executions then it must provide a primitive that will allow a task to change the *base priority* of another task. With this primitive it would be possible to program a high priority task whose only job is to execute at priority step-up times and to furnish the set changes.

In the model presented above the cost of a context switch is ignored. To engineer a real system these times must be subsummed into the Worst Case Execution Time for the task. In addition the cost of overheads such as the real-time clock need to be taken into account. This can be done by decreasing the available processor utilisation from its idealistic 100% level. The proposed priority change task would constitute a further overhead that would reduce available utilisation. However this new overhead is unlikely to be high compared to the increase in effective

utilisation the dual priority model facilitates. If a kernel is produced that directly supports dual priorities (via a form of second delay queue) then there will again be an overhead to take into account.

6. CONCLUSION

Static priority schemes have the disadvantage that processor utilisations less than 100% must be tolerated. By comparison earliest deadline scheduling can theoretically utilise all of a processor's capacity. The dual priority scheme described in this paper allows task sets that would otherwise not be schedulable to be guaranteed.

An algorithm is presented that enables the dual priority assignments to be systematically accomplished. This algorithm requires the behaviour of each task at its critical instance to be analysed. No formula for finding the critical instance has been derived and thus the algorithm requires the task set to be simulated up to the LCM of the tasks. This may be impractical for some large tasks sets with large relative prime periods. A more efficient algorithm may be possible if it becomes easy to identify the critical release of each task. Further work is needed on this analysis.

The analysis presented has assumed that there are no blocking relationships between tasks. If there are, then ceiling priorities can be used to bound priority inversion. However one must use each task's maximum priority when defining ceilings. Finally it should be noted that the approach defined in this paper can also be applied to task sets where deadline is less than period. In this situation the deadline monotonic (static) priority assignment scheme is optimal. The use of dual priorities can again increase processor utilisation; although 100% utilisation is not feasible with such task set (using any scheme!).

Acknowledgements

The authors would like to thank Neil Audsley and Ken Tindell for helpful comments on an earlier version, and Mike Richardson for his help with the stress simulations.

References

1. C.L. Liu and J.W. Layland, ‘‘Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment’’, *JACM* **20**(1), pp. 46-61 (1973).
2. J.P. Lehoczky, L. Sha and V. Ding, ‘‘The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior’’, Tech Report, Department of Statistics, Carnegie-Mellon (1987).
3. N. C. Audsley, A. Burns, M. F. Richardson and A. J. Wellings, ‘‘STRESS: A Simulator For Hard Real-Time System’’, RTRG/91/106, Real-Time Research Group, Department of Computer Science, University of York (October 1991).
4. M.G. Harbour, M.H. Klein and J.P. Lehoczky, *Fixed Priority Scheduling of Periodic Tasks with varying Execution Priority*, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, USA (1991).