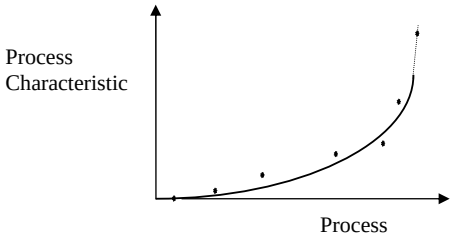
REAL-TIME DISTRIBUTED SYSTEMS PERFORMANCE MODELLING

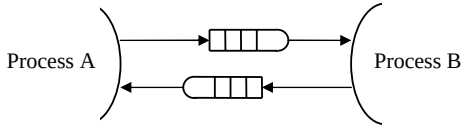
Performance analysis is of considerable importance for real-time systems because of the presence of time-constraints on various system actions and responses. It allows the detection of potential problems in advance and can be used to explore alternative software designs and hardware configurations.

Types of Performance Models

- Regression Models - an empirical technique that takes statistical data on system performance and applies a curve fitting approach to the data (e.g. least squares). The limitation of the approach is that a static model is developed that treats the system structure and behaviour as a "black-box" which gives little insight into design alternatives.



- Queuing Models - these models usually involve a number of simplifying assumptions to make the model amenable to analysis:



  a) a "memory-less" property in the probability of requests → this allows the interarrival time distribution to be modelled as exponential. Thus the smallest interarrival time (zero) has the highest probability. This situation is almost never found in real computational environments.

  b) generally only steady-state performance can be analysed.

The main application of queuing type models is as high-level abstract models to obtain an overall view of whether the system can meet performance goals → other techniques are needed for more detail.

- Simulation Models - this is a dynamic model which models both the real-world system and the software design → it uses an algorithmic abstraction of the system structure and behaviour. Such a model is a discrete event model, i.e. simulation is time based but the simulation can "skip" over time intervals between system events.

  The problem with this modelling approach is finding the right level of abstraction that captures just enough of the system behaviour to be realistic but not too much to be too costly to implement → trade-off between model "realism" and simulation time and cost.

- Petri Net Models - finite state machines are sequential models so they can only be used to indirectly model concurrent systems. Petri nets can model concurrent system directly and be augmented with time information to model real-time systems. Petri Nets also support powerful analysis capabilities and can be combined with stochastic models to analyse steady-state properties of the system.

- Real-time Scheduling & Event Sequence Models - where the system has been designed based on some systematic assignment of task priorities based on scheduling theory to give the highest possible resistance to variability in performance under different loading conditions. A simplified model of the task architecture and scheduling environment can be used to support an analysis of performance for critical events sequences through the system.

The last mentioned approach is the one that has achieved the highest impact with industry, with the other approaches being more commonly used for safety-critical real-time distributed systems.

Real-Time Scheduling Theory

Where hard deadlines are specified in a real-time system it is necessary to specify the priorities of the concurrent tasks in the system. This approach assumes that priority pre-emption scheduling is supported by the operating system (i.e. the highest priority task executes when it is ready).

The approach has been applied to progressively more complicated scheduling situations, e.g. independent periodic tasks, mixture of periodic and aperiodic tasks, scheduling with task synchronization (e.g. scheduling for Ada tasks).

Scheduling Periodic Tasks

A basic rate monotonic approach is used, i.e. where each task is assigned a fixed priority based on an inverse relationship with the period of execution of the task.

- Given a periodic task with period T and an execution time C, the CPU utilization $U = C/T$.
- A task is said to be schedulable if all deadlines can be met (i.e. execution is completed before its period expires).

Utilization Bound Theorem [Liu & Layland, 1973]:

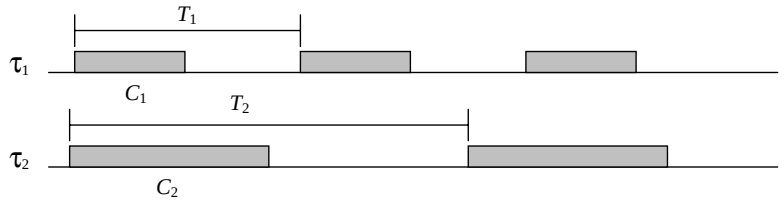
A set of n independent periodic tasks will always meet their deadlines provided the sum of the C/T ratios is below an upper bound of overall CPU utilization, i.e:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \Lambda + \frac{C_n}{T_n} \quad \leq \quad n(2^{1/n} - 1) \quad = \quad U(n)$$

This applies for all task t_i "phasings" with execution times C_i and periods T_i.

As $n \rightarrow \infty$ the upper bound $U(n) \rightarrow \ln 2 = 0.69$.

Derivation of Rate Monotonic Scheduling Bound

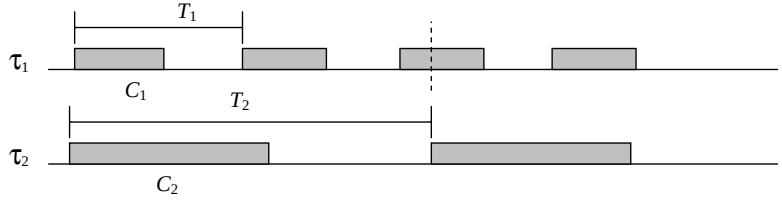


- Apply a Rate Monotonic scheduling policy for two tasks τ_1 and τ_2:
  - if $T_2 > T_1$ then task τ_1 priority is set higher than task τ_2 priority

- Consider two cases:
  1. All requests for τ_1 are completed before the second τ_2 request (above diagram)
     - The number of requests for τ_1 during period T_2 is $\lceil T_2 / T_1 \rceil$
     - τ_1 consumes $C_1 \lceil T_2 / T_1 \rceil$ of the CPU time within T_2's period
     - The largest possible C_2 remaining is $T_2 - C_1 \lceil T_2 / T_1 \rceil$ and the CPU utilisation is:

$$U = C_1/T_1 + C_2/T_2 = C_1/T_1 + 1 - C_1/T_2 \lceil T_2 / T_1 \rceil$$

  2. A request for τ_1 overlaps the second τ_2 request (see diagram)

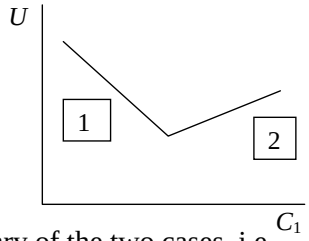


     - Now the $\lceil T_2 / T_1 \rceil$th request of τ_1 overlaps the second τ_2 request
     - After these requests have been satisfied the largest possible C_2 remaining is $(T_1 - C_1) \lfloor T_2 / T_1 \rfloor$ and the CPU utilisation is:

$$U = C_1/T_1 + C_2/T_2 = C_1/T_1 + (T_1 - C_1)/T_2 \lfloor T_2 / T_1 \rfloor$$

$\Rightarrow$ The two utilisations are:

1. $U = C_1/T_1 + 1 - C_1/T_2\lceil T_2/T_1 \rceil$
2. $U = C_1/T_1 + (T_1 - C_1)/T_2\lfloor T_2/T_1 \rfloor$



The minimum value U occurs at the boundary of the two cases, i.e. where:

$$C_1/T_1 + 1 - C_1/T_2\lceil T_2/T_1 \rceil = C_1/T_1 + (T_1 - C_1)/T_2\lfloor T_2/T_1 \rfloor$$

i.e. $C_1 = T_2 - T_1\lfloor T_2/T_1 \rfloor$

The utilisation at this point is:

$$U = 1 - T_1/T_2\,(\lceil T_2/T_1 \rceil - T_2/T_1)\,(T_2/T_1 - \lfloor T_2/T_1 \rfloor)$$

Letting $f = T_2/T_1 - \lfloor T_2/T_1 \rfloor$ we have:

$$U = 1 - f(1-f)/(\lfloor T_2/T_1 \rfloor + f)$$

Now the minimum U occurs at the smallest possible value of $\lfloor T_2/T_1 \rfloor$ and since $T_2 > T_1$, this is 1. Thus we can rewrite U:

$$U = 1 - f(1-f)/(1+f) = (1+f^2)/(1+f)$$

Minimizing U over f, gives

$$dU/df = (f^2 + 2f - 1)/(1+f)^2 \quad \text{i.e. } (1 + 2f + f^2) = 0 \Rightarrow f = 2^{1/2} - 1$$

Hence $U = 2(2^{1/2} - 1) \approx 0.83$.

Thus the least upper bound CPU utilisation for two tasks with rate monotonic scheduling is 83%:

- This occurs where $T_2/T_1 - \lfloor T_2/T_1 \rfloor = 0.41$, i.e. the period of T_2 is about 41% greater than T_1.
- Corresponds to a τ_1 execution period that is about 41% of T_1 and a τ_2 execution period that is T_1-C_1. Note with $f = 0$ then $U = 1$ as expected.

Generalize for n tasks: $U(n) = n(2^{1/n} - 1)$.

Note carefully that the Utilization Bound Theorem is a worst-case approximation and on less conservative criteria:

- $U(n) = 0.88$ for a randomly chosen group of tasks (i.e. permits some "beneficial" phasings as well) [Lehoczy, 1989].
- $U(n)$ is even higher where the tasks have harmonic periods (i.e. have periods that are multiples of each other).

The advantage of the rate monotonic algorithm is that it is very stable in the presence of transient overload $\rightarrow$ all the highest priority tasks (which by definition have the shortest period) will still meet their deadlines while lower priority tasks will occasionally miss deadlines as processor load increases.

Example:

Task t_1: $C_1 = 20$ ms; $T_1 = 100$ ms $\rightarrow$ $U_1 = 0.2$
Task t_2: $C_2 = 30$ ms; $T_2 = 150$ ms $\rightarrow$ $U_2 = 0.2$
Task t_3: $C_3 = 60$ ms; $T_3 = 200$ ms $\rightarrow$ $U_3 = 0.3$

Assume that the context switch overhead is included in the CPU times. The upper bound from the Utilization Bound Theorem is:

$$U(3) = 3(2^{1/3} - 1) = 0.78$$

which is greater than the total utilization of these tasks, i.e. $U_{total} = 0.7$ $\rightarrow$ all three tasks can meet their deadlines.

Suppose that t_3's performance changes to:

Task t_3: $C_3 = 90$ ms; $T_3 = 200$ ms $\rightarrow$ $U_3 = 0.45$

Now, $U_{total} = 0.85$ which is greater than the bound $\rightarrow$ the tasks fail to meet their deadlines. The first two tasks can be checked in the same way, e.g. $U_{total} = 0.4$ and the upper bound becomes:

$$U(2) = 2(2^{1/2} - 1) = 0.828$$

which is greater than the total utilization of these tasks $\rightarrow$ at least the first two tasks can meet their deadlines.

Completion Time Theorem [Lehoczy, 1989]:

The Utilization Bound Theorem is very pessimistic, and a more exact schedulability criterion can also be checked. The worst-case assumption is made that all periodic tasks are ready to execute at the same time.

It has been shown before [Liu, 1973] that provided a task completes execution before its first period, then it will never miss a deadline. Thus this theorem checks if all tasks can complete execution before the end of their respective first periods.

A set of *n* independent periodic tasks scheduled by a rate monotonic algorithm will always meet their deadlines, for all task phasings, iff:

$$\forall i, \quad 1 \le i \le n, \quad \min_{(k,p) \in R_i} \left\{ \sum_{j=1}^{i} C_j \frac{1}{pT_k} \left\lceil \frac{pT_k}{T_j} \right\rceil \right\} \quad \le \quad 1 \qquad \ldots (1)$$

where C_j is the execution time, and T_j is the period, of task t_j, and where $R_i = \{(k,p) \mid 1 \le k \le i, p = 1, \Lambda\ , \lfloor T_i / T_k \rfloor\}$

Note that for convenience we can rewrite eqn. (1) to:

$$\forall i, 1 \le i \le n, \forall (k,\ p)\ \in R_i, \ \sum_{j=1}^{i} C_j \left\lceil \frac{pT_k}{T_j} \right\rceil \quad \le \quad pT_k \qquad \ldots (2)$$

so that one of these inequalities must be met for each *i*.

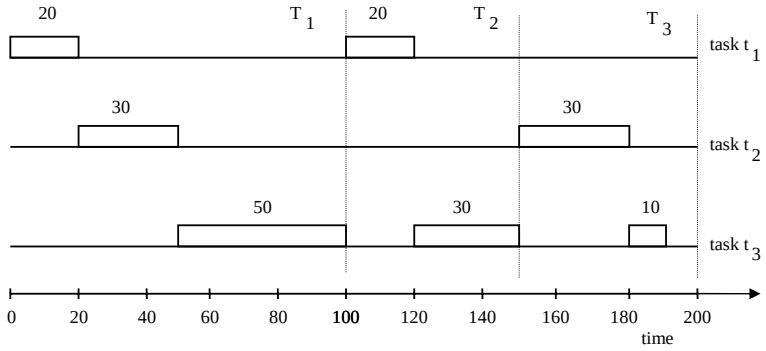
Example

Task t_1: C_1 = 20 ms; T_1 = 100 ms $\rightarrow$ U_1 = 0.2
Task t_2: C_2 = 30 ms; T_2 = 150 ms $\rightarrow$ U_2 = 0.2
Task t_3: C_3 = 90 ms; T_3 = 200 ms $\rightarrow$ U_3 = 0.45

The worst-case scenario is with all three tasks ready to execute at the same time. Using rate monotonic scheduling t_1 executes first, followed by t_2 then t_3.

Note that task t_i will execute once for a CPU time of C_i during a period T_i and higher priority tasks will execute more often and may pre-empt task t_i. Thus it is necessary to consider the CPU time used by all higher priority tasks.

The ends of the first periods of each of the tasks, T_i, are referred to as *scheduling points*.



At time T_1, task t_1 must be rescheduled, and being of shorter period (hence higher priority), it pre-empts task t_3.

At time T_2, task t_2's scheduling point, task t_2 pre-empts t_3 again. Finally t_2 completes and allows t_3 to be rescheduled to complete its 90 msec execution time before its deadline, and before t_1 needs to be rescheduled.

All three tasks complete execution before the end of their respective first periods $\rightarrow$ all can meet their deadlines.

Note that a total of 190 msec of CPU time is used up over the first 200 msec $\rightarrow$ utilization of 0.95. After an elapsed time of the least common multiple of the three periods, i.e. 600 msec, the utilization averages out to 0.85.

Applying the Completion Time Theorem, equation (2) to this example:

- t_i is the lowest priority task to be checked and t_k is the task with a scheduling point to be checked against it. Each value of p represents the scheduling points of task t_k to be checked.
- Check from task t_3 against t_1's SP's, i.e. $i = 3$ and $k = 1$
  $\rightarrow$ $p = 1, \ldots, \lfloor T_i / T_k \rfloor = 1, \ldots, \lfloor 200/100 \rfloor = 1, 2.$
  i.e. $R_i = (k,p) = (1,1), (1,2)$

for $(k, p) = (1,1)$ $\Rightarrow C_1\left\lceil \frac{(1)T_1}{T_1} \right\rceil + C_2\left\lceil \frac{(1)T_1}{T_2} \right\rceil + C_3\left\lceil \frac{(1)T_1}{T_3} \right\rceil \leq (1)T_1$

$\Rightarrow C_1\left\lceil \frac{(1)100}{100} \right\rceil + C_2\left\lceil \frac{(1)100}{150} \right\rceil + C_3\left\lceil \frac{(1)100}{200} \right\rceil \leq T_1$

$\Rightarrow C_1 + C_2 + C_3 \leq T_1 \Rightarrow 20 + 30 + 90 \leq 100?$

and this inequality is not met.

for $(k, p) = (1,2)$ $\Rightarrow C_1\left\lceil \frac{(2)T_1}{T_1} \right\rceil + C_2\left\lceil \frac{(2)T_1}{T_2} \right\rceil + C_3\left\lceil \frac{(2)T_1}{T_3} \right\rceil \leq (2)T_1$

$\Rightarrow C_1\left\lceil \frac{(2)100}{100} \right\rceil + C_2\left\lceil \frac{(2)100}{150} \right\rceil + C_3\left\lceil \frac{(2)100}{200} \right\rceil \leq 2T_1$

$\Rightarrow 2C_1 + 2C_2 + C_3 \leq 2T_1 \Rightarrow 40 + 60 + 90 \leq 200$

and this inequality is met.

- Check from task t_3 against t_2's SP's, i.e. $i = 3$ and $k = 2$
  $\rightarrow$ $p = 1, \ldots, \lfloor T_i / T_k \rfloor = 1, \ldots, \lfloor 200/150 \rfloor = 1.$
  i.e. $R_i = (k,p) = (2,1).$

for $(k, p) = (2,1)$ $\Rightarrow C_1\left\lceil \frac{(1)T_2}{T_1} \right\rceil + C_2\left\lceil \frac{(1)T_2}{T_2} \right\rceil + C_3\left\lceil \frac{(1)T_2}{T_3} \right\rceil \leq (1)T_2$

$\Rightarrow C_1\left\lceil \frac{(1)150}{100} \right\rceil + C_2\left\lceil \frac{(1)150}{150} \right\rceil + C_3\left\lceil \frac{(1)150}{200} \right\rceil \leq T_2$

$\Rightarrow 2C_1 + C_2 + C_3 \leq T_2 \Rightarrow 40 + 30 + 90 \leq 150$?

and this inequality is not met.

- Check from task t_3 against t_3's SP's, i.e. $i = 3$ and $k = 3$
  $\rightarrow$ $p = 1, \ldots, \lfloor T_i / T_k \rfloor = 1, \ldots, \lfloor 200/200 \rfloor = 1.$
  i.e. $R_i = (k,p) = (3,1).$

for $(k, p) = (3,1)$ $\Rightarrow C_1\left\lceil \frac{(1)T_3}{T_1} \right\rceil + C_2\left\lceil \frac{(1)T_3}{T_2} \right\rceil + C_3\left\lceil \frac{(1)T_3}{T_3} \right\rceil \leq (1)T_3$

$\Rightarrow C_1\left\lceil \frac{(1)200}{100} \right\rceil + C_2\left\lceil \frac{(1)200}{150} \right\rceil + C_3\left\lceil \frac{(1)200}{200} \right\rceil \leq T_3$

$\Rightarrow 2C_1 + 2C_2 + C_3 \leq T_2 \Rightarrow 40 + 60 + 90 \leq 200$?

and this inequality is also met.

Note from eqn. (2) that only one inequality must be met for task 3.

- Check from task t_2 against t_1's SP's, i.e. $i = 2$ and $k = 1$
  $\rightarrow$ $p = 1, \ldots, \lfloor T_i / T_k \rfloor = 1, \ldots, \lfloor 150/100 \rfloor = 1.$
  i.e. $R_i = (k,p) = (1,1).$

for $(k, p) = (1,1)$ $\Rightarrow C_1\left\lceil \frac{(1)T_1}{T_1} \right\rceil + C_2\left\lceil \frac{(1)T_1}{T_2} \right\rceil \leq (1)T_1$

$\Rightarrow C_1\left\lceil \frac{100}{100} \right\rceil + C_2\left\lceil \frac{100}{150} \right\rceil \leq T_1$

$\Rightarrow C_1 + C_2 \leq T_1 \Rightarrow 20 + 30 \leq 100$?

and this inequality is met.

- Check from task t_2 against t_2's SP's, i.e. $i = 2$ and $k = 2$
  $\rightarrow$ $p = 1, \ldots, \lfloor T_i / T_k \rfloor = 1, \ldots, \lfloor 150/150 \rfloor = 1.$
  i.e. $R_i = (k,p) = (1,1).$

for $(k,\ p)\ = (1,1) \Rightarrow C_1\left\lceil \frac{(1)T_2}{T_1} \right\rceil + C_2\left\lceil \frac{(1)T_2}{T_2} \right\rceil \ \leq \ (1)T_2$

$$\Rightarrow C_1\left\lceil \frac{150}{100} \right\rceil + C_2\left\lceil \frac{150}{150} \right\rceil \ \leq \ T_2$$

$$\Rightarrow 2C_1 + C_2 \leq T_2 \Rightarrow 40 + 30 \leq 150 \ ?$$

and this inequality is met.

Note from eqn. (2) that only one inequality must be met for task 2.

- Check from task t_1 against t_1's SP's, i.e. $i = 1$ and $k = 1$
  $\rightarrow\ p = 1, \ldots, \lfloor T_i / T_k \rfloor = 1, \ldots, \lfloor 100 / 100 \rfloor = 1.$
  i.e. $R_i = (k,p) = (1,1)$.

for $(k,\ p)\ = (1,1) \Rightarrow C_1\left\lceil \frac{(1)T_1}{T_1} \right\rceil \ \leq \ (1)T_1$

$$\Rightarrow C_1\left\lceil \frac{100}{100} \right\rceil \ \leq \ T_1$$

$$\Rightarrow C_1 \leq T_1 \Rightarrow \ 20 \ \leq \ 100 \ ?$$

and this inequality is met.

Note from eqn. (2) that only one inequality must be met for task 1.

Thus in this example the inequalities are satisfied for all values of $1 \leq\ i\ \leq\ n = 3$, so that as all three tasks meet at least **one** of their scheduling point deadlines, then the tasks are schedulable.

Scheduling Periodic and Aperiodic Tasks

Rate monotonic theory can be extended to accommodate aperiodic (asynchronous) tasks, i.e. tasks that "arrive" randomly within some period T_a (which is the *minimum interarrival time* of the event which activates the task).

If C_a is the CPU time consumed by the aperiodic task, then the scheduler can effectively reserve a "ticket" of value C_a with period T_a. If the task is not activated during the T_a period, then the ticket is discarded → the worst-case CPU utilization of the aperiodic task is C_a/T_a.

Thus aperiodic tasks can be accommodated in the theory by assigning appropriate priority levels based on their worst-case minimum interarrival time.

Scheduling with Task Synchronization

Task synchronization is required when a task enters a critical section and other tasks must be stopped from entering it also. The problem can arise that a lower priority task enters the critical section before a higher priority task (which is then held up while the lower priority task executes). This situation is referred to as *priority inversion*.

This problem can become an *unbounded priority inversion* where the lower priority process (while in the critical section) becomes blocked on another (or the same) higher priority process.

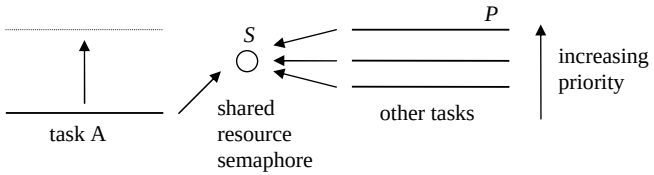
A simple solution to the problem is to prevent tasks from being pre-empted while in critical sections → this is fine where critical sections are short, but still not ideal because higher priority tasks have to wait.

The priority ceiling protocol:

One mechanism that avoids mutual deadlock and provides a bounded priority inversion is the *priority ceiling protocol*.

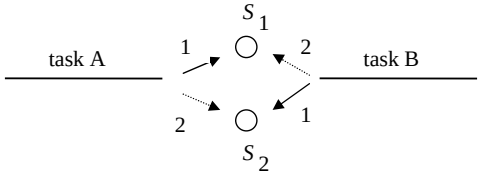
The basic concept is that only one lower priority task is able to block a higher priority task at a time - consider the simplest case with only one critical section to illustrate.

All tasks must have adjustable priority levels. Consider that a low priority task *A* is in its critical section, and all other higher priority tasks are blocked because they want to use the same resource. The priority of task *A* is increased to that of the highest priority task being blocked by it $\rightarrow$ this will speed up the execution of task *A* so that the blocking time for the higher priority tasks is minimized:



For a binary semaphore *S* that protects a shared resource, it has a *priority ceiling P* (the highest priority task that is allowed to acquire it). Thus a low priority task that acquires semaphore *S* can have its priority raised to a maximum of *P*.

Potentially deadlock can also occur here, i.e. where two tasks each need to acquire two identical resources (but in a different sequence):



The priority ceiling protocol handles this situation also, i.e. if semaphore S_1 priority is P_1 then task *A* priority is raised to P_1 which is higher than task *B* so that task *B* cannot acquire S_2 and task *A* is able to acquire S_2 next.

Generalized Real-Time Scheduling Theory

Of course, in practice, tasks may not be able to execute at their assigned rate monotonic priorities $\rightarrow$ need to extend the theory to accommodate these cases.

Where an aperiodic task occurs, the conventional rate monotonic approach is to treat it as a task with period set to the worst case interarrival time. Where the aperiodic task is interrupt driven, it may be the case that even though it occurs infrequently (so it has a high interarrival time and hence a low priority) we may want to have a fast response $\rightarrow$ it needs a high priority.

Thus we need to distinguish a task's *rate monotonic priority* (which is based entirely on the task's period) from an actual priority which may be based on the importance of the task. This leads to a form of *rate monotonic priority inversion* - e.g. we could have that task A (with period 25 msec) has a higher rate monotonic priority than an interrupt driven task B (with worst case interarrival period 50 msec), but task B's actual priority is higher because we want it to pre-empt the periodic task.

The extension to basic rate monotonic scheduling theory includes:

- the blocking effect of lower priority tasks.
- pre-emption by higher priority tasks that do not observe rate monotonic priorities.

Consider a task t_i with period T_i during which it consumes C_i units of CPU time. The extensions to the existing theorems (Utilization Bound and Completion Time) explicitly check that each task t_i can meet its first deadline:

1. **Pre-emption time by higher priority tasks with periods less than that of t_i:** Such tasks can pre-empt task t_i many times, so let this set of tasks be H_n. If C_j is the CPU time and T_j is the period for task t_j (and note that $T_j < T_i$). The utilization of a task t_j in the set H_n is given by C_j/T_j.

2. **Execution time for task t_i:** Task t_i executes once during period T_i and consumes C_i units of CPU time.

3. **Pre-emption by higher priority tasks with longer periods:** These tasks have non-rate monotonic priorities, and they can only pre-empt task t_i once, as they have longer periods than t_i. Call this set of tasks H_I and let the time used by a task in this set be C_k. The worst-case utilization of a task t_k in the H_I set is given by C_k/T_i.

4. **Blocking time by lower priority tasks:** These tasks can only execute once as they have longer periods. Blocking delays can be analysed on an individual basis for each task t_i to find the worst case blocking situation. If B_i is the worst case blocking time for a given task, then the blocking utilization for the period T_i is B_i/T_i.

For any task t_i the standard Utilization Bound and Completion Time theorems accommodate items 1 and 2 above. The *Generalized Utilization Bound Theorem* accommodates items 3 and 4 above:

$$U_i = \sum_{j \in H_n} \frac{C_j}{T_j} + \frac{1}{T_i}\left(C_i + \sum_{k \in H_I} C_k + B_i \right)$$

where U_i is the utilization bound during a period T_i for task t_i. The four terms in the utilization expression arise from the respective terms identified above.

The utilization for any task U_i can be determined and compared against the worst case bound $U(n)$. This bound must be met by all tasks since showing that a given task meets its bound does not now guarantee that all higher priority tasks will meet their bounds.

Should the Utilization Bound test fail, a more precise test can be applied, i.e. a generalized Completion Time Theorem.

Example - application of Generalized Real-Time Scheduling Theory

We have four tasks - two periodic and two aperiodic:

- periodic task t_1: $C_1 = 20$ ms, $T_1 = 100$ ms $\rightarrow$ $U_1 = 0.2$
- aperiodic task t_2: $C_2 = 15$ ms, $T_2 = 150$ ms $\rightarrow$ $U_2 = 0.1$
- interrupt driven aperiodic task t_a: $C_a = 4$ ms, $T_a = 200$ ms $\rightarrow$ $U_a = 0.02$
- periodic task t_3: $C_3 = 30$ ms, $T_3 = 300$ ms $\rightarrow$ $U_3 = 0.1$

Suppose t_1, t_2 and t_3 all access the same data object, which is protected by a semaphore *s*. Context switch time is assumed to be included in the indicated CPU times.

Using rate monotonic priority assignment, the priorities would be t_1, t_2, t_a, t_3. Because a fast response is required to interrupts the priority of t_a is raised to be the highest.

The overall CPU utilization is 0.42 which is less than the utilization bound $U(4) = 4(2^{1/4}-1) = 0.76$. Because of the non-rate monotonic priority assignment it is necessary to consider each task individually:

- Consider task t_a - highest priority with $U_a = 0.02$ $\rightarrow$ no trouble meeting its deadline.
- Consider task t_1 - apply the Generalized Utilization Bound Theorem
  a) Pre-emption by high-priority tasks with periods less than T_1 (there are none here).
  b) Execution utilization $U_1 = 0.2$.
  c) Pre-emption by high-priority tasks with longer periods. Task t_a falls into this category $\rightarrow$ utilization in the period of the task is $C_a/T_1 = 4$ ms/100 ms = 0.04.
  d) Blocking time by lower priority tasks. Both t_2 and t_3 can potentially block t_1 $\rightarrow$ assuming the priority ceiling algorithm is being used, at most only one task can block t_1, so take the worst-case of t_3 (since it has the longer execution time), i.e. blocking utilization during the period of the task is $B_3/T_1 = 30$ ms/100 ms = 0.3.

From the Generalized Utilization Bound Theorem we have:

Worst Case Utilization =
Pre-emption utilization (periods less than task period)
+ Execution utilization
+ Pre-emption utilization (periods greater than task period)
+ Blocking utilization

For task t_1, the worst case utilization = 0.2 + 0.04 + 0.3 = 0.54
which is less than the worst case utilization bound = 0.76.
→ task t_1 will meet it's deadline.

- Consider task t_2 - apply the Generalized Utilization Bound Theorem
  e) Pre-emption by high-priority tasks with periods less than T_2. Task t_1 has a period less than T_2, so its pre-emption utilization during the period is $U_1 = 0.2$.
  f) Execution utilization $U_2 = 0.1$.
  g) Pre-emption by high-priority tasks with longer periods. Task t_a falls into this category → utilization in the period of the task is C_a/T_2 = 4 ms/150 ms = 0.03.
  h) Blocking time by lower priority tasks. Task t_3 can potentially block t_2 → again assuming the priority ceiling algorithm is being used, at most only one task can block t_2, so take the worst-case of t_3, i.e. blocking utilization during the period of the task is B_3/T_2 = 30 ms/150 ms = 0.2.

For task t_2, the worst case utilization = 0.2 + 0.1+ 0.03 + 0.2 = 0.53
which is less than the worst case utilization bound = 0.76.
→ task t_2 will meet it's deadline.

- Consider task t_3 - apply the Generalized Utilization Bound Theorem
  i) Pre-emption by high-priority tasks with periods less than T_3. Tasks t_1, t_2 and t_a have periods less than T_3, so its pre-emption utilization during the period is: $U_1 + U_2 + U_a$ =0.2 +0.1 + 0.02 = 0.32.
  j) Execution utilization $U_3 = 0.1$.
  k) Pre-emption by high-priority tasks with longer periods. There are no tasks in this category.

  l) Blocking time by lower priority tasks. There are no lower priority tasks.

For task t_3, the worst case utilization = 0.32 + 0.1 = 0.42
which is less than the worst case utilization bound = 0.76.
→ task t_3 will meet it's deadline.

Thus all four tasks will meet their deadlines.

Use of Real-Time Scheduling Theory in the Design Phase

In the design phase it is usual to be more conservative and apply the worst case Utilization Bound Theorem (i.e. for an infinite number of tasks, or 0.69). If this worst case bound cannot be achieved then alternative solutions should be attempted.

It is acceptable to have utilizations above 0.69 provided all utilizations above 0.69 are to due to lower priority soft real-time tasks (or even non-real-time tasks) → missed deadlines by these tasks are not serious.

Priority assignment is based on the following guidelines:

1. Assign rate monotonic priorities based on worst case task periods.
2. Interrupt driven tasks are assigned the highest priorities.
3. Tasks having the same period (hence the same rate monotonic priority) are given slightly different priorities based on their application importance.

Performance Analysis using Event Sequence Analysis and Real-Time Scheduling Theory

Based on the requirements phase of the project, the system response times are specified. After the task structuring phase, preliminary time budgets can be allocated to tasks, and an *event sequence analysis* applied to determine the sequence of tasks that need to be executed to service any external event.

Steps in the process:

1. Select an external event.
2. Find the I/O task activated by the event and estimate the CPU time for the I/O task (any uncertainties are accommodated via a worst case upper bound).
3. Identify all tasks activated by this task and estimate their CPU time.
4. Identify all I/O tasks that generate the system's response to that event and estimate the CPU times.
5. Include CPU overhead time, i.e. context switching overhead, interrupt handling overhead, intertask communication, and synchronization overhead.
6. Sum the CPU times for all event sequence tasks and add the CPU overhead.
7. Compare the total CPU time against the specified response time to the external event.

Note that CPU utilization estimation may require the tracing of multiple paths through each task → then estimate the CPU time for each path.

The frequency of activation of the various paths in the task need to be estimated and then multiplied by the path execution times
→ sum all task CPU times as before.

To apply Real-Time Scheduling Theory it is only necessary to compare the event sequence CPU times against the deadlines rather than individual task CPU times as before:

1. Locate all tasks in the event sequence with the same priority → these tasks can then be considered as one equivalent task from a scheduling viewpoint (and has a CPU time equal to the sum of the task CPU times plus all overheads).
2. The worst case interarrival time of the external event that initiates the event sequence is then made the period of the equivalent task.

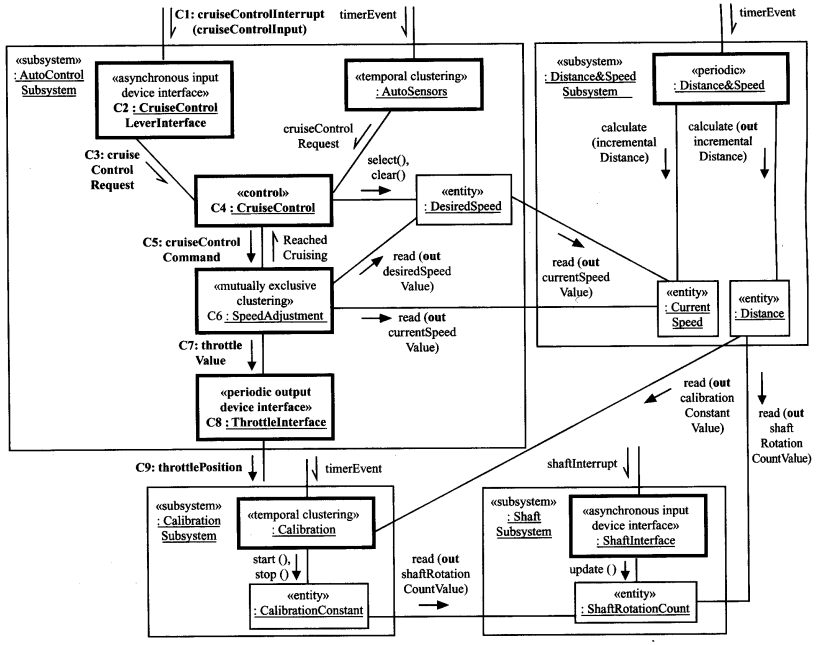
3. The Real-Time Scheduling Theorems are applied to determine if the equivalent task can meet its deadlines, i.e. consider pre-emption by higher priority tasks, blocking by lower priority tasks, and execution time of this task.

Note this approach is a simplification, i.e. in practice one task may appear in a number of different event sequences or executing the equivalent task at that priority would stop other tasks meeting deadlines.

To overcome this limitation, the tasks in the event sequence need to be analysed separately and assigned different priorities. Then the tasks within the event sequence also need to be checked against the deadlines.

Example - event sequence analysis

Consider the Cruise Control subsystem for a particular event sequence specified for a Cruise Control Interrupt:



[Gomaa, 2000]

Assumptions:

- The priority of all Monitoring subsystem tasks and the Calibration task is lower than any of the tasks in the event sequence.
- The Cruise Control subsystem is in the Initial state.

Suppose a performance requirement is that the system must respond to the driver's engaging the cruise control in the ACCEL position within 250 msec. The sequence of internal events is shown above (C1-9) and they can be listed with the CPU time C_i required for each event i:

1. Cruise control interrupt (C_1)
2. Cruise control interface reads the cruise control lever (C_2)
3. Cruise control interface sends a cruise control request message to Cruise Control (C_3)
4. Cruise Control executes its statechart with a state change from Initial to Accelerating (C_4)
5. Cruise Control sends an increase speed command to task Speed Adjustment (C_5)
6. Speed Adjustment executes the command and computes the throttle value (C_6)
7. Speed Adjustment sends a throttle value message to the Throttle Interface task (C_7)
8. Throttle Interface computes the new Throttle Position (C_8)
9. Throttle Interface outputs to the throttle.

As four tasks are involved in the event sequence $\rightarrow$ there is a minimum delay of $4*C_x$ where C_x is the task context switch time.

The Total CPU time required to support this event sequence is:

$$C_e = C_1 + C_2 + C_3 + C_4 + C_5 + C_6 + C_7 + C_8 + 4*C_x$$

and if the communication overhead C_m is the same in all cases, we have:

$$C_e = C_1 + C_2 + C_4 + C_6 + C_8 + 3*C_m + 4*C_x$$

It is necessary to include the execution of other tasks that can occur during this event sequence, i.e. assume:

- Auto Sensors (C_{10}) is periodically activated every 100 msec $\rightarrow$ it could execute 3 times in the 250 msec period.
- Shaft Interface (C_{11}) is activated on every shaft rotation, so at 6000 rpm $\rightarrow$ once every 10 msec $\rightarrow$ 25 times in this period.
- Distance and Speed (C_{12}) is periodically activated every 250 msec $\rightarrow$ executes once in the period.

Every time a task executes that is not in the event sequence there could be two context switches.

Thus the total CPU time for these additional three tasks is:

$$C_a = 3*(C_{10}+2*C_x) + 25*(C_{11} + 2*C_x) + (C_{12} + 2*C_x)$$

and the total CPU time is then:

$$C_t = C_e + C_a$$

Estimates are required for all the timing parameters, these are derived from the TBSs for all tasks involved in the event sequence and any other overhead tasks, e.g:

C_1 = 1 msec, C_2 = 4 msec, C_4 = 6 msec, C_6 = 14 msec,
C_8 = 5 msec, C_{10} = 5 msec, C_{11} = 1 msec, C_{12} = 10 msec,
C_m = 1 msec, C_x = 0.5 msec

From these values we have:

C_e = 35 msec, C_a = 79 msec $\rightarrow$ C_t = 114 msec $<$ 250 msec

Note that perturbations can now be applied to the task execution times, the context switch times and the communication times to explore how close to specification the response time is expected to be, e.g:

C_x = 0.5 msec $\rightarrow$ 1 msec so C_t = 114 msec $\rightarrow$ 145 msec

Real-Time Scheduling Theory based Performance Analysis

The approach is to consider a steady state situation involving only periodic tasks, and then consider the driver-imposed aperiodic demands on the system.

The worst-case steady-state situation is examined first, i.e. at maximum CPU demand with the car at maximum shaft revolution rate. For the periodic tasks, let the period be T_i, the CPU time C_i, and each task's CPU utilization is $U_i = C_i/T_i$. The two context switch times for each task are included in the CPU time for each task.

Consider each periodic task:

1. Shaft Interface: assume periodic (actually aperiodic) at worst-case 6000 rpm $\rightarrow$ T_1 = 10 msec, C_1 = 1 + 2*0.5 = 2 msec.
2. Auto Sensors: T_2 = 100 msec, C_2 = 5 + 2*0.5 = 6 msec.
3. Distance and Speed: T_3 = 250 msec, C_3 = 10 + 2*0.5 = 11 msec.
4. Calibration: T_4 = 500 msec, C_4 = 4 + 2*0.5 = 5 msec.
5. Speed Adjustment: once activated under automated control, T_5 = 250 msec, C_5 = 14 + 2*0.5 = 15 msec.
6. Throttle Interface: once activated under automated control, T_6 = 100 msec, C_6 = 5 + 2*0.5 = 6 msec.
7. Trip Reset Buttons Interface: T_7 = 500 msec, C_7 = 4 + 2*0.5 = 5 msec.
8. Trip Average Timer: executes relatively infrequently, T_8 = 1 sec, C_8 = 20 msec (but not time critical).
9. Maintenance Reset Button Interface: executes very infrequently, say T_9 = 1 sec, C_9 = 6 msec (but not time critical).
10. Maintenance Timer: executes very infrequently, say T_9 = 2 sec, C_9 = 15 msec (but not time critical).

The rate monotonic priority assignment is based on an inverse relationship with task period, so the priority assignment is to assign the highest priority to tasks of shortest period.

Note that there are three cases where tasks have the same period:

- Throttle Interface and Auto Sensors have a period of 100 msec $\rightarrow$ as Auto Sensors is always active, while Throttle Interface is only active under automated vehicle control and it receives an input from Auto Sensors, Auto Sensors is given the higher priority.
- Speed and Distance and Speed Adjustment have a period of 250 msec $\rightarrow$ as Speed and Distance computes current speed that is used by Speed Adjustment (if active) then it is given higher priority.
- Calibration and Trip Reset Buttons Interface have a period of 500 msec $\rightarrow$ as Calibration is associated with the Cruise Control subsystem (rather than the Maintenance Monitoring subsystem) it is given higher priority.

The utilizations $U_i = C_i/T_i$ are all computed:

- Shaft Interface - U_1 = 2 ms/10 ms = 0.2
- Auto Sensors - U_2 = 6 ms/100 ms = 0.06
- Throttle Interface - U_3 = 6 ms/100 ms = 0.06
- Distance and Speed - U_4 = 11 ms/250 ms = 0.04
- Speed Adjustment - U_5 = 15 ms/250 ms = 0.06
- Calibration U_6 = 5 ms/500 ms = 0.01
- Trip Reset Buttons Interface - U_7 = 5 ms/500 ms = 0.01
- Trip Averages Timer - U_8 = 20 ms/1000 ms = 0.02
- Maintenance Reset Buttons Interface - U_9 = 6 ms/1000 ms = 0.01
- Maintenance Timer - U_{10} = 15 ms/2000 ms = 0.01

Thus the total utilization = 0.48 < 0.69 (the worst-case upper bound), so that by the rate monotonic Utilization Bound Theorem, all tasks should meet their deadlines. It is assumed that any data access time to shared data stores is negligible.

Combined Real-Time Scheduling Theory and Event Sequence based Performance Analysis

Define an Equivalent Aperiodic Task: consider the event sequence when the driver initiates an external event, e.g. via the cruise control lever or the brake - tasks in an event sequence can be replaced by a single aperiodic task which represents the additional CPU load.

For a cruise control lever external event, the tasks required to be executed are Cruise Control Lever Interface, Cruise Control, Speed Adjustment Control, and Throttle Interface $\rightarrow$ the required CPU time is C_e. The combined four tasks can be referred to as an (*aperiodic*) *event sequence task* since for this particular event sequence, these tasks must execute sequentially with associated communications and task switch overhead.

From RT Scheduling Theory, an aperiodic task can be treated as a periodic task with period given by the minimum interarrival time of aperiodic requests, e.g. in this example the desired response time to the driver's inputs is 250 msec $\rightarrow$ also the worst-case period (in practice it will be much greater than this).

Note that two other tasks (Speed Adjustment and Distance and Speed) have the same period $\rightarrow$ give the aperiodic task the higher priority of the tasks with the same period (e.g. priority level 4). From earlier we have C_e = 35 msec and T_e = 250 msec (and e = 4 here), so reassigning the task priorities gives the utilizations:

- Shaft Interface - U_1 = 2 ms/10 ms = 0.2
- Auto Sensors - U_2 = 6 ms/100 ms = 0.06
- Throttle Interface - U_3 = 6 ms/100 ms = 0.06
- Event Sequence Task - U_4 = 35 ms/250 ms = 0.14
- Distance and Speed - U_5 = 11 ms/250 ms = 0.04
- Speed Adjustment - U_6 = 15 ms/250 ms = 0.06
- Calibration - U_7 = 5 ms/500 ms = 0.01
- Trip Reset Button Interface - U_8 = 5 ms/500 ms = 0.01
- Compute Average Mileage - U_9 = 20 ms/1000 ms = 0.02
- Maintenance Reset Button Interface - U_{10} = 6 ms/1000 ms = 0.01
- Maintenance Timer - U_{11} = 15 ms/2000 ms = 0.01

Thus the total utilization = 0.62 < 0.69, so that again by the rate monotonic Utilization Bound Theorem, all tasks should still meet their deadlines.

Activation of the brake can be treated in the same way, i.e. the event sequence is now Auto Sensors, Cruise Control, Speed Adjustment, and Throttle Interface and the estimated CPU time for the only task that is different (Auto Sensors) is 5 msec which is the same as Cruise Control Lever Interface $\rightarrow$ we have the same equivalent aperiodic task C_e.

The period for Monitor Auto Sensors was 100 msec (to ensure that brake or engine inputs are not missed, but we assume that the maximum likely rate of brake actuations is also 250 msec.

Given that both actuations are driver initiated, it is not realistic to assume (even worst-case) that both brake and cruise control actuations could both occur every 250 msec $\rightarrow$ assume either (but not both) events can occur every 250 msec, so the total utilization of 0.62 is also obtained for the brake actuation event sequence.

Incorporating Non-rate Monotonic Priorities

In the last example an assumption was made that all tasks can be allocated their rate monotonic priorities:

- Thus the event sequence task is allocated priority 4 after Shaft Interface, Auto Sensors and Throttle Interface $\rightarrow$ it could potentially miss the Cruise Control Interrupt if it has to wait for the other higher priority tasks to execute.
- Alternatively, if the event sequence task is allocated priority 1 (i.e. a non-rate monotonic priority then it may force Shaft Interface to miss its interrupt.
- A sensible choice would be to allow the Shaft Interface task to retain priority 1 (given its 10 msec period) and the event sequence task to be priority 2 (with period 250 msec) ahead of Auto Sensors and Throttle Interface (both have period 100 msec)

With this priority assignment we have:

- Shaft Interface - U_1 = 2 ms/10 ms = 0.2
- Event Sequence Task - U_2 = 35 ms/250 ms = 0.14
- Auto Sensors - U_3 = 6 ms/100 ms = 0.06
- Throttle Interface - U_4 = 6 ms/100 ms = 0.06
- Distance and Speed - U_5 = 11 ms/250 ms = 0.04
- Speed Adjustment - U_6 = 15 ms/250 ms = 0.06
- Calibration - U_7 = 5 ms/500 ms = 0.01
- Trip Reset Button Interface - U_8 = 5 ms/500 ms = 0.01
- Compute Average Mileage - U_9 = 20 ms/1000 ms = 0.02
- Maintenance Reset Button Interface - U_{10} = 6 ms/1000 ms = 0.01
- Maintenance Timer - U_{11} = 15 ms/2000 ms = 0.01

We then need to check each task explicitly again to ensure it meets its deadline. The first two tasks in priority order (hence rate monotonic) have a utilization of 0.34 $\rightarrow$ well under the bound. The next two tasks in priority order must be analysed in a worst-case situation to show that they do not miss their deadlines:

Consider a 100 msec period:

- Shaft Interface is scheduled 10 times $\rightarrow$ 20 msec used
- The four tasks in the event sequence are active once $\rightarrow$ 35 msec used
- The Auto Sensors and Throttle Interface task each consume 6 msec each

The total CPU time is 67 msec < 100 msec so all tasks meet their deadlines.

Note: the earlier rate monotonic analysis with an equivalent aperiodic task gave a CPU utilization of 0.62 and bursts of activity can produce transient loads which are much higher. For example, here we have an effective CPU utilization (in a particular 100 msec period) of 0.67 and if we include the next highest priority task, Distance and Speed, the CPU utilization increases to 0.78.

Application of the RT scheduling algorithm guarantees that all tasks can meet their deadlines regardless of sudden bursts of activity.

Analysis using Generalized Real-Time Scheduling Theory

We assume that we have rate-monotonic priority assignment policy with the exception of the first Cruise Control Lever Interface task in the event sequence which is given priority 2. The four tasks in the event sequence need to be analysed separately over the period T_e = 250 msec to determine that all four tasks meet their deadlines:

1. *Execution time for the task event sequence* - the total execution time for the four tasks is the event sequence is C_e = 35 msec and T_e = 250 msec $\rightarrow$ execution utilization = 0.14.
2. *Pre-emption time by higher priority tasks with periods less than T_e* - there are three tasks to consider:
   a) Shaft Interface - period of 10 msec can pre-empt any of the four tasks a maximum of 25 times * CPU time of 2 msec = 50 msec.
   b) Auto Sensors and Throttle Interface - both with periods of 100 msec can pre-empt any of the three lower priority tasks a maximum of 3 times * CPU time of 12 msec = 36 msec.

   Total pre-emption time = 50 + 36 = 86 msec
   Total pre-emption utilization = 0.2 + 0.06 + 0.06 = 0.32
3. *Pre-emption by higher priority tasks with longer periods* - no such tasks.
4. *Blocking time by lower priority task* - no such tasks.

Worst-case utilization = Pre-emption utilization + Execution utilization + Blocking utilization = 0.32 + 0.14 = 0.46 < 0.69 (the GUBT upper bound) $\rightarrow$ all four tasks in the event sequence will meet their deadlines.

It is then necessary to determine that the two tasks with shorter period of 100 msec will meet their deadlines:

1. *Execution time for the two tasks* - the total execution time for the two tasks (which have the same period) is 6 + 6 msec = 12 msec and the execution utilization = 0.06 + 0.06 = 0.12.

2. *Pre-emption time by higher priority tasks with periods less than 100 msec* - there is only one task to consider:
   - Shaft Interface - period of 10 msec which can pre-empt any of the two tasks a maximum of 10 times * CPU time of 2 msec = 20 msec.

   Total pre-emption time = 20 msec
   Total pre-emption utilization = 0.2

3. *Pre-emption by higher priority tasks with longer periods* - there is only one task to consider:
   - Cruise Control Lever Interface - period of 250 msec which can pre-empt only once for a CPU time of 6 msec.

4. *Blocking time by lower priority task* - no such tasks.

Worst-case utilization = Pre-emption utilizations + Execution utilization + Blocking utilization = 0.2 + 0.06 + 0.12 = 0.38 < 0.69 (the GUBT upper bound) $\rightarrow$ the two periodic tasks with shorter periods will meet their deadlines.

The same analysis can be performed for each of the lower priority periodic tasks to show that they meet their deadlines also.

Re-design Using Task Clustering Criteria

Where the design doesn't meet the performance criteria, modification of the task architecture is usually required, and this is done through re-application of the clustering criteria, i.e: temporal task inversion, multiple instance task inversion, and sequential task inversion

For example, in the Cruise Control subsystem, sequential task inversion can be applied to combine the Cruise Control task with Speed Adjustment and the Throttle Interface task to produce an Inverted Cruise Control task $\rightarrow$ reduces the communications overhead from 3*C_m to C_m and reduces the context switch overhead from 4*C_x to 2*C_x which then reduces the event sequence CPU time from 35 msec to 32 msec.