Real-Time Scheduling with Shared Resources Revisited

David Donari, Leo Ordinez, Rodrigo Santos, Javier Orozco
Universidad Nacional del Sur - CONICET

Abstract. Resource sharing is one of the limiting factors when dealing with real-time scheduling. In particular, synchronizing tasks and sharing resources usually impose priority inversions and the possibility of introducing deadlocks among the tasks involved. Contention policies usually bound the blocking-time that higher priority tasks may suffer, by introducing an strict order in the access to the shared resources. The way in which they do this determines the maximum time each task would have to wait for a resource. In some cases, a resource constitutes a limiting factor for the schedulability and replicating it may facilitate the scheduling. In this paper, a replicating scheme is introduced for detecting and replicating the appropriate resource so the system can become feasible within a cost constraint.

Keywords: Scheduling, shared resources, optimization, heuristics

1 Introduction

Common priority-driven real-time systems are composed of a set of preemptable tasks running concurrently on a single processor and accessing a set of reusable nonpreemptible resources. The tasks may require exclusive use of the resources imposing priority inversions and eventually deadlocks if not conveniently managed. The first problem appears when a low priority task has gained access to a resource before a higher priority one was ready to execute. When the higher priority task preempts the lower one and tries to access the resource it has to wait until the lower priority task frees the resource. Deadlocks may lead to a system starvation as the tasks involved could not finish their executions. In order to deal with these problems, usually, synchronization of shared resources is implemented by means of semaphores, noted $S$, that ensure the sequential access to the resources. Under this model only one task can be using the resource protected by $S$. In that case, it is said that the task is executing a critical section. Contention policies are necessary to synchronize the access to shared resources and the execution of critical sections. These impose constraints for the schedulability as blocking-times should be added to the execution times of the tasks that eventually will be blocked in a semaphore waiting for a lower priority task to finish.

In the literature there are two main policies for handling shared resources in priority-based real-time systems. The Priority Ceiling Protocol (PCP) proposed in [7] resolves the priority inversion and deadlock problems by introducing priority inheritance and a resource ceiling that controls the access to the protected resource. The policy blocks a task at the moment in which it tries to lock a resource already locked or that may
be used by an executing task already in a critical section. Under this policy, basically the priority of the task determines the execution order and only one priority inversion may occur. It is used in fixed priority systems. The other contention policy is the Stack Resource Policy (SRP) proposed in [1]. It introduces the concept of early blocking, preventing a task from beginning its execution until all the necessary resources are available. An executing task may run till completion without being preempted. In this sense, the system behaves more like non-preemptive. The final result in both cases is the same: a task may be blocked on a resource for at most the longest critical section of a task of lower priority. In [1] it is proved that with SRP the number of context switches is lower than with PCP and this may result in a better performance of the system.

Recently, in [5], the idea of blocking chains generating a class partition was used to analyse the system from the perspective of equivalence classes. The approach allows a clear differentiation of the tasks and resources in the system creating independent sets.

In priority based systems, the use of shared resources with contention policies reduces the schedulability margin. In one extreme, each task may have all the necessary resources for itself. This is quite expensive as many resources may have several replicas although no priority inversions or deadlocks may be present and the set of tasks can be considered as independent. On the other extreme, the set of resources is minimum in the sense that only one instance of each is present in the system and all the tasks that need it should share it. The cost in this case is minimum but the system may turn out to be not feasible because of the eventual priority inversions that should be computed. Between the two extremes, a proper amount of resources may exist that makes the system schedulable and keeps the cost of it low. The problem is not simple as it resembles the knapsack optimization problem and there is no polynomial solution for it.

Contributions: In this paper the problem previously stated is analysed as an optimization problem. An optimization function can be solved based on an Integer Linear Programming (ILP) routine by creating a set of binary variables that represent the different configurations of the system. The way in which it can be solved is presented and discussed. As the problem has many restrictions that can be used to shortcut the search in the solution’s space region, two different heuristics are presented based on the equivalence classes defined in [5].

Organization: The rest of the paper is organized in the following way. In Section 2, previous work on the subject is analysed. In Section 3 the system model is presented. In Section 4, the contention policy used is briefly described. In Section 5, the problem is presented and the ILP model is described and analysed. In Section 6, two heuristics to search in a simple way the solution’s space region are explained. Finally, in Section 7, conclusions are drawn.

2 Previous Work

The scheduling of real-time systems with shared resources has been studied in the past. In what follows a short review of the main papers is presented.

The liminar paper by Sha et al [7] presents the PCP for the handling of shared resources in fixed priority preemptive scheduling systems. It provides bounded priority inversion and the protocol is deadlock free. In this protocol each resource is assigned
a priority ceiling, which is a priority equal to the highest priority of any task which may lock the resource. A task needs to have a higher priority than the ceiling of the resource to lock it.

The SRP proposed by Baker [1] is a resource allocation policy which permits tasks with different priorities to share a single runtime stack. The protocol provides bounded priority inversion and is deadlock free. It can be used in both dynamic and fixed priority scheduling systems. The contention mechanism introduces a preemption level and a priority. A task is able to preempt another one only if its preemption level and priority are higher than the actual one in the system.

Ordinez et al extended the SRP in [5] by defining an equivalence blocking relation that partitions the set of tasks in classes. The idea of the extension is to group the tasks and resources in classes in such a way that the preemption levels of the tasks are defined by the higher preemption level in the task’s class. This modification may reduce the amount of preemptions in the system reducing the context switch costs.

Other papers dealing with shared resources but oriented to hierarchical scheduling under resource reservation mechanisms were proposed by Davis et al in [4], Benham et al in [3] and Santos et al in [6].

In none of the previous cases there is an analysis of the resources as a limiting factor in the schedulability of the system and in which way they may be replicated to turn an unfeasible system into a feasible one. In Baruah [2], this idea is introduced for the case of dynamic priority scheduling systems with SRP as contention policy. The author proposes a solution based on a simple heuristic that replicates the resource that turns out to be critical from the schedulability analysis. To do so, he uses the computation of busy periods and slack time.

3 System Model

The system is modeled as a set of periodic and preemptive tasks, $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Tasks are characterized by a worst case execution time $C_i$, a period $T_i$, an activation time $a_i$, and a relative deadline $D_i$, which is used to compute the absolute deadline $d_i = a_i + D_i$. Since tasks are periodic, they can be seen as a stream of jobs or instances $J_{ij}$, where the first subindex refers to the task and the second one to the instance.

Tasks may share a set $R = \{R_i \mid i = 1, 2, \ldots, m\}$ of non-preemptive, serially-accessed resources, which can be physical or logical. In order to maintain consistency, accesses to a shared resource must be mutually exclusive and controlled by a semaphore (mutex), $S$, with classical lock() and unlock() operations, denoted $P(S)$ and $V(S)$ respectively. A task accessing a mutually exclusive resource is said to be in a critical section. Successive accesses to critical sections must be properly nested, in the sense that it is only possible to have sequences of the type $P(S_a), \ldots, P(S_b), V(S_a), \ldots, V(S_b)$. The times of use of a critical section may be different for different tasks. They are symbolized $\xi_k(\tau_i)$ for each critical section $k$ of task $\tau_i$; $\xi_{\max}(\tau_i)$ denotes the maximum time of all the critical sections accessed by $\tau_i$. A special case of critical section is the external (or outermost) non-trivial critical section, which is a critical section not nested in any other one. Tasks are supposed to declare off-line the amount of critical sections used.
4 Contention policy

In this section the contention policy used to handle shared resources is briefly introduced. The ESRP policy extends the ideas proposed by Baker in [1] by introducing an equivalence relation named blocking relation that partitions the set of tasks in classes.

Tasks within the class inherit the highest preemption level present in the class and the scheduler only allows to start the execution of a task if its preemption level and priority are higher than the current ones operating in the system.

The set of resources used by a task, denoted $R_{\tau_i} = \{ R_j \mid \tau_i \text{ uses } R_j \}$, is defined as the resource task set. From it, it is possible to define the blocking relation denoted $\sim$ among tasks if they share directly or indirectly a resource. This is an equivalence relation as it is reflexive, commutative and transitive. Being an equivalence relation produces an important result on the set of tasks partitioning it in disjoint blocking tasks’ sets or, what is the same, classes. Formally, a blocking task set is noted, $\Upsilon = \{ \tau_i \mid \{ \tau_j \in \Gamma \mid \tau_i \sim \tau_j \} \}$. Therefore, given $\Gamma / \emptyset = \{ \Upsilon_1, \Upsilon_2, \ldots, \Upsilon_n \}$ $\Upsilon_v \cap \Upsilon_w = \emptyset$, $\forall \Upsilon \in \Gamma / \emptyset$, then $\tau_i \sim \emptyset \tau_j, \forall \tau_i, \tau_j \mid \tau_i \in \Upsilon_v \land \tau_j \in \Upsilon_w$.

The partitioned tasks classes generate a partition in the resources set too. A blocking resources set is defined as the set of resources used by the tasks in the class. It is noted $\Phi_w = \{ \bigcup R_{\tau_i} \mid \tau_i \in \Upsilon_w \}$. From the previous definitions, if $\tau_i \sim \tau_j; R_{\tau_i} \cap R_{\tau_j} \neq \emptyset$ or $\exists \{ \tau_{k1}, \tau_{k2}, \ldots, \tau_{km} \} | [R_{\tau_i} \cap R_{\tau_1}] \land (R_{\tau_i} \cap R_{\tau_2}) \land \ldots \land (R_{\tau_i} \cap R_{\tau_m}) \neq \emptyset$. Then if $[\tau_i] = [\tau_j] = \Upsilon_w; R_{\tau_i} \bigcup R_{\tau_j} \in \Phi_w$, where $\Phi_w$ is the class invariant under $\sim$.

Each task, $\tau_i$, in the system has associated a preemption level and a priority, denoted $\pi(\tau_i)$ and $p(\tau_i)$, respectively. The priority is inversely proportional to the time remaining until the absolute deadline while the preemption level is set according to the relative deadline. Based on this, the protocol is capable of providing a deadlock-free and a bounded priority inversion execution for the tasks. For a job $J_{ij}$ of any task $\tau_i$:

$$p(\tau_i(t)) = p(J_{ij}(t)) = \frac{1}{d_{ij} - t}$$

where $t$ is the actual time.

When a resource $R$ is allocated to a task, it is said that its allocation is outstanding. This is used to define an associated ceiling, $[R]$, that is an integer-valued function of the outstanding allocation of $R$. The ceiling definition can be extended to the class $\Upsilon_v$. It can be stated as $\underline{\pi}_v$, which is the maximum $[R_i]$ of all the resources in $\Phi_v$. Formally, $\underline{\pi}_v = \max [ [R_i] \mid R_i \in \Phi_v ]$. In the same way, the blocking resource set has an associated ceiling too.

The next two conditions set the basis to prevent deadlocks and multiple priority inversions.

**Condition 1 (Preventing deadlocks)** A task should not be allowed to start until all the necessary resources for its execution are available.

**Condition 2 (Multiple priority inversions)** A task should not be allowed to start until all the necessary resources for its execution and the execution of every job that may preempt it, are available.
**Condition 3** If task $\tau_i$, currently executing or allowed to preempt the currently executing task, when requesting $R_i$ may be blocked by a task with higher preemption level, then $\pi(\tau_i) \leq \lceil R_i \rceil$.

A class is **ACTIVE** if at least one task within it is executing or ready to execute. On the contrary it is **IDLE** if no task within the class is executing or in the ready queue.

**Theorem 1.** If no task $\tau_i \in \Upsilon_v$ is allowed to start until $\pi(\tau_i) > \pi_{\Phi_v}$ for every active task set $\Upsilon_u$ then: a) After it starts, no task can be blocked by any other task; b) There can be no deadlock; and c) No task can be blocked for longer than the duration of one outermost non-trivial critical section of a lower priority task (i.e., the only way that this can happen is by an early blocking before the task starts)

**Corollary 1.** A task $\tau_i \in \Upsilon_v$ can be preempted by a higher priority task $\tau_j \in \Upsilon_u$ if $\pi(\tau_j) > \pi_{\Phi_v}$.

A task $\tau_i$ is blocked before starting its execution until it is the highest priority pending request in $\Upsilon_v$ and has the higher preemption level among the tasks in the **ACTIVE** classes: $\pi(\tau_i) > \pi_{\Phi_v} | \Upsilon_v$ is **ACTIVE**

In this way, a task that starts its execution, receives immediately all the required resources and is not blocked by any other executing task. This is possible because the task must have a higher preemption level than its current set ceiling and than all the current set ceilings of every active task set. The current system ceiling can be defined as $\Pi = \max \{ \pi_{\Phi_v} | \Upsilon_v$ is **ACTIVE**$ \}$. The preemption test can finally be stated as $\pi(\tau_k) > \Pi$.

With this algorithm, when a task $\tau_k$ starts its execution, it **inherits** the maximum preemption level of the tasks in its blocking task set, or what is the same its set ceiling, and becomes the one with the highest priority among them.

The schedulability condition for EDF with ESRP semaphore locking, EDF-ESRP, is given by the following theorem:

**Theorem 2 (Schedulability).** A set of $n$ tasks is EDF-ESRP schedulable if: $\forall k \mid k = 1, \ldots, n \left( \sum_{i=1}^{k} C_i / T_i \right) + \frac{B_k}{P_k} \leq 1$

where $B_k$ denotes the duration of the longer outermost critical section, $\xi_j$, of any task $\tau_j$ that may block task $\tau_k$.

Actually, the condition stated is identical to the one proposed by Baker for the SRP.

**5 Problem statement**

As has been described in the previous sections, the existence of shared resources introduces the possibility of priority inversions that has to be considered at the moment of checking the schedulability of the system. The problem can be seen from two points of view. In the first one, all tasks are independent. That means that each one has an individual copy of all the resources it needs. Starting from this, the set of resources is reduced until the system is not schedulable anymore. The optimization problem consists in choosing the proper resources to be shared. In the second approach, the one
followed here, at the starting, the set of resources is minimum in the sense that only one instance of each resource is present in the system; later, if necessary, some of them are replicated.

The use of just one instance of each resource may make the system unfeasible because tasks in the system may incorporate blocking-times producing deadline misses. As tasks are grouped in classes, it may happen that a certain resource of the class imposes a blocking-time that leads to the loss of the feasibility. By replicating this resource and splitting the class in two, the system may become schedulable; however, choosing the proper resource to replicate is not trivial if the cost of the system is an important issue.

In the case of a resource being shared by more than two tasks, its replication may produce different combinations. Let $R$ be shared by $\tau_1$, $\tau_2$ and $\tau_3$. If $R$ is duplicated then three different possible combinations are possible: $\{\tau_1, \tau_2\}$ and $\{\tau_3\}$ and $\{\tau_2\}$ and finally $\{\tau_2, \tau_3\}$ and $\{\tau_1\}$. The problem grows exponentially with the amount of resources and tasks and the way in which the blocking relations may eventually interact at the moment of computing the blocking-times for each task. If critical sections are nested the problem is even more complex and a case by case analysis turns out to be difficult.

**Example 1** Suppose a set $\Gamma(5) = \{\tau_1, \tau_2, \ldots, \tau_5\}$, where the following blocking tasks sets are defined by the blocking relation $\Upsilon_1 = \{\tau_1, \tau_3, \tau_5\}$ and $\Upsilon_2 = \{\tau_2, \tau_4\}$, with $\pi(\tau_1) > \pi(\tau_2) > \ldots > \pi(\tau_5)$. The blocking-times for each task are computed from: $B_1 = \max\{\xi_3, \xi_5\}$, $B_2 = \max\{\xi_3, \xi_4, \xi_5\}$, $B_3 = \max\{\xi_4, \xi_5\}$ and $B_4 = \xi_5$. The original system is represented in Figure 1(a). In this case there are many possible replication schemes. For example $R_1$ can be duplicated and $\tau_1$ decoupled from $\Upsilon_1$. In that case, $B_2 = \xi_4$ because $\tau_3$ and $\tau_5$ no longer inherit the preemption level of $\tau_1$. This case is presented in Figure 1(b).

Fig. 1. Graph representation of Example 1. Nodes represent tasks and resources and the edges represent the use of the resource by the task.
Stated like this, it is an optimization problem similar to the knapsack one. Therefore it can be solved by means of an integer linear programming approach. Basically each task will have a reward that measures the value of not being blocked by lower priority tasks, and the resources will have a weight that represents the cost of replicating them. This weight not only represents an economic cost but also an opportunity cost in the sense that some resources are not replicable. In that case, the cost will be so high that the system is not feasible. In what follows a careful description of the optimization problem is presented.

5.1 Objective function formulation

Linear programming is a method to obtain the maximum profit or minimum cost out of a mathematical model. When the variables involved can have only integer values it is called Integer Linear Programming and if they only can take values from \{0, 1\} it is called Binary Integer Programming (BIP). The mathematical model for the problem previously discussed corresponds to this type. A careful description of the elements of the objective function formulation follows:

1. A set of tasks \(\tau_i, i = 1 \ldots n\),
   (a) Each task can have a different worst case blocking-time depending on the shared resources configuration of the system. This imposes exclusive alternatives at runtime.
   (b) Each critical section protected by lock/unlock operations eventually introduces a possible priority inversion taken into account in the blocking-time.
   (c) Each task \(i\) has associated a weight \(\alpha^i\) that measures the importance of the task in the system.
   (d) Each alternative at runtime \(g\) of task \(i\) has associated a weight \(\beta^g\) that measures the cost of being blocked when trying to access a resource.

2. A minimum set of \(m\) resources (disk, network interface, memory, registers, etc.) used by the tasks.
   (a) Each resource \(k\) has associated a weight \(\mu\) that represents its value.
   (b) Each resource can have \(n\) different instances where \(n\) is the number of tasks that eventually may use them. For each instance of the resource there is a weight \(\lambda\) that measures the cost of replicating it.

The optimization problem can then be stated in the following way:

\[
\min \sum_{i=1}^{n} \sum_{g=1}^{m} \alpha^i \beta^g \tau_{ig} + \sum_{k=1}^{m} \sum_{h=1}^{n} \mu_k \lambda_{kh} R_{kh}
\]
subject to the following constraints:

\[
\forall i \sum_{g=1}^{n} \tau_{ig} = 1 \\
\forall k \sum_{h=1}^{i-1} R_{kh} = 1 \\
\forall i \sum_{k=1}^{m} \frac{C_s}{T_s} + \frac{C_i + B_i}{T_i} \leq 1 \\
\sum_{k=1}^{m} \sum_{h=1}^{n} \lambda_{kh} = \text{MAX\_COST}
\]

The first constraint states that every task must be executed in one and only one shared resource configuration. In each configuration the task may be blocked at different resources. The second constraint indicates that each resource should have one and only one number of replicas, i.e. resource \( R_k \) may have one instance, two instances, \( h \) instances, but it can not have at the same time \( h = 1 \) and \( h = 3 \). For example, if after the schedulability analysis, two tasks that originally shared \( R_1 \), require two instances of the resource, then the resource is duplicated. This fact is also reflected in the tasks that, because of the duplication, stop sharing it. The third constraint states the schedulability test for each task. The blocking value depends on the shared resources configuration. Finally, the fourth constraint sets a maximum cost of resources.

With this in mind, even a simple problem requires many variables to be solved. Suppose for example a problem with two tasks and one resource that is shared by them. The resource can have one or two instances and each task may share it or not. In this simple case there are 6 different variables for the BIP formulation, two for each task and two for the resource. In general, the number of binary variables can be computed as a combinatorial number that takes into account the amount of shared resources \( m \) and tasks \( n \).

\[
\binom{m}{k} + m.n
\]

As can be seen the amount of combinations easily grows in an unbounded way. Even if it is possible to find a solution following some linear programming library routines, the computational cost of finding it is very expensive both in time and in preparation of the different data structures that the solver may need.

In the next section solutions to the problem based on guided heuristics will be presented.

### 6 Guided Heuristics

In the first heuristic only the schedulability constraint is considered and in this way a quick solution may be reached based on a simple replication mechanism. The idea behind this heuristic is to replicate the resources decoupling the task that introduce the highest preemption level. In this way, it is possible to reduce the blocking imposed by
lower priority tasks not only on the higher priority tasks but also on other tasks that may suffer indirect blocking. The steps are:

1. Order the tasks by increasing relative deadlines.
2. Determine the blocking relations and the class partitions.
3. Compute for each task $\tau_i$ the maximum time that eventually may be blocked, denoted $B_i$.
4. Perform the scheduling test for each task.
   (a) If the test is passed, advance to the next task in the list up to the end and repeat 4.
   (b) If the task is not schedulable, duplicate the resource involved decoupling the task generating the longest critical section and repeat from 2.
5. Check the cost of the system. If it is below the maximum allowed, a solution has been found.

   The second heuristic is more elaborated as it searches through the classes trying to replicate the resources that generate priority inversions among tasks in different partitions. The idea behind this heuristic is to separate the tasks both in priorities and preemption levels in such a way that the inheritance is among preemption levels in the same class and not in different ones. Basically, for the case presented in Example 1, the original partition is $T_1 = \{\tau_1, \tau_3, \tau_5\}$ and $T_2 = \{\tau_2, \tau_4\}$ because $R_1$ is shared by the tasks in $T_1$ and $R_2$ by the tasks in $T_2$. Then replicating $R_1$ in such a way that the new partition is $T_1 = \{\tau_1\}$, $T_2 = \{\tau_3, \tau_5\}$ and $T_3 = \{\tau_2, \tau_4\}$, frees $\tau_2$ from the eventual priority inversion produced by $\tau_3$ or $\tau_5$. However, it may be that instead of splitting $T_1$ is better to split $T_2$. In such case, $\tau_3$ is freed from the priority inversion of $\tau_4$. To decide which option is best, the worst case blocking time may be computed and then the appropriated one chosen. The steps are:

1. Order the tasks by increasing relative deadlines.
2. Determine the blocking relations and the class partitions.
3. Compute for each task $\tau_i$ the maximum time that eventually may be blocked, denoted $B_i$.
4. Determine the maximum blocking in each class.
5. Build a list with the classes ordered by decreasing maximum blocking.
6. Perform the scheduling test.
   (a) If schedulable, check the cost of the system. If it is below the maximum allowed, a solution has been found.
   (b) If not schedulable, then replicate the resource producing the maximum blocking in the first class in the list of step 5, decoupling the task with the longest critical section. Repeat from step 2.

In [2] an heuristic based on the computation of the Demand Bound Function (DBF) and the available slack time is used. The process replicates the resource originating the non schedulability of the system and separates the tasks based on their use of the resources involved in the blocking. The mechanism is similar to the first heuristic presented here but is more complex since the computation of the DBF and slack time is not trivial.
1. Order the tasks by increasing relative deadlines.
2. Determine the time instants, \(d_1, d_2, \ldots\) at which the slack time should be checked. Basically the instants at which tasks have deadlines or reactivate.
3. For each one of the previous instants determine the blocking time that any task with deadline before \(d_k\) may be blocked by a task with a deadline after \(d_k\), \(B(d_k)\).
4. For each instant \(d_k\) with \(k = 1, 2, \ldots\)
   (a) \(\text{SLACK}(d_k) = (d_k - \sum_{i=1}^{n} (\text{DBF}(\tau_i, d_k)))\)
   (b) If \(\text{SLACK}(d_k) < B(d_k)\) then
      i. \(\text{MINSLACK}(D_i) = \min\{\text{SLACK}(d_k) | D_i \leq d_k < D_i + 1\}\)
      ii. For each \(R_l\) such that \([R_l] \leq D_i\)
         A. If \(\exists j > i\) such that \(\xi_j > \text{MINSLACK}(D_i)\) then replicate \(R_l\)
         B. Go back to step 3.
   (c) Compute the cost of the system, if below the maximum allowed a solution has been found.

The following expression should be used to compute the DBF:

\[
\text{DBF}(d_k) = \sum_{D_i \leq d_k} \left(1 + \left[\frac{d_k - D_i}{T_i}\right]\right)C_i
\]

6.1 Analysis

Let Table 6.1 described the system under analysis. The system contains two classes, \(T_1 = \{\tau_1, \tau_3, \tau_5\}\) and \(T_2 = \{\tau_2, \tau_4\}\) because \(R_1\) is shared by the tasks in \(T_1\) and \(R_2\) by the tasks in \(T_2\). From the blocking times analysis, it comes out that task \(\tau_5\) is the one that may introduce priority inversions not only for task \(\tau_1\) but also for task \(\tau_2\). In this case the blocking time for all the tasks but the last one is seventy, \(B_1 = B_2 = B_3 = B_4 = 70\)

<table>
<thead>
<tr>
<th>Task</th>
<th>C</th>
<th>T</th>
<th>R1</th>
<th>R2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20</td>
<td>100</td>
<td>20</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>100</td>
<td>-</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>100</td>
<td>20</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>400</td>
<td>60</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>500</td>
<td>70</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1. System’s Description

Linear programming The BIP formulation to solve the problem by means of Linear Programming techniques, requires for \(n = 5\) and \(m = 2\), eighteen useful binary variables as there are some combinations that are not possible and are eliminated before starting the procedure.
First heuristic In the first heuristic, step 4 fails for task $\tau_3$. In this case, the shared resource is $R_1$ which belongs to class $\Phi_1$. The task with the longest critical section is $\tau_5$ so it is decoupled from $\Upsilon_1$ and the blocking times for each task are recomputed. The new partition is $\Upsilon_1 = \{\tau_1, \tau_3\}$, $\Upsilon_2 = \{\tau_2, \tau_4\}$ and $\Upsilon_5 = \{\tau_5\}$. The blocking times are $B_1 = 20$, $B_2 = B_3 = 60$ and $B_4 = B_5 = 0$. With this arrangement step 4 still fails for task $\tau_3$ as it can not stand the indirect blocking imposed by $\tau_4$. Thus, resource $R_2$ should be replicated. The new partition is then $\Upsilon_1 = \{\tau_1, \tau_3\}$, $\Upsilon_2 = \{\tau_2\}$, $\Upsilon_4 = \{\tau_4\}$ and $\Upsilon_5 = \{\tau_5\}$. With this configuration the system is schedulable and a solution to the problem has been found. The solution was found after three iterations of the algorithm. The computations for the blocking times are quite simple and based on the worst case scenario but the solution found is optimal.

Second heuristic In the second heuristic the analysis is quite similar. However, once the system is detected to not be schedulable in step 6, resource replication is made on the class with the highest blocking time. For the example being analysed, the ordered list of class is as follows: $\Upsilon_1$, $\Upsilon_2$. The shared resource producing the maximum blocking is $R_1$ so it is replicated decoupling from the class task $\tau_5$ that has the longest critical section. After this operation the classes are rebuilt and the lists of classes is ordered in the following way: $\Upsilon_2$, $\Upsilon_1$, $\Upsilon_5$. As in the previous case, again the scheduling test is not passed and now task $\tau_4$ from $\Upsilon_2$ is decoupled. The system becomes schedulable with the new configuration and the solution is again optimal.

Baruah heuristic First of all the set of points to check the slack and blocking times is determine. In this case it is quite simple as the periods are multiples of 100, $d = \{100, 200, 300, 400, 500\}$. The next step is to compute the slack time at each point. $SLACK(100) = 50$, $SLACK(200) = 100$, $SLACK(300) = 150$, $SLACK(400) = 100$, $SLACK(500) = 50$. The feasibility analysis is not passed as $SLACK(100) < B(100)$. The heuristic then chooses the $R_1$ to replicate decoupling task $\tau_5$. The process is repeated again and $R_2$ is replicated decoupling task $\tau_4$. It is important to notice the computational complexity involved in this heuristic that needs to compute the DBF and the slack available to determine the feasibility of the system.

The heuristics only consider the cost of the replication as the last step in the analysis. In the BIP approach instead, the cost is considered from the beginning leaving the schedulability test as the last condition to be satisfied.

7 Conclusions

In this paper a formal approach to the problem of scheduling real-time systems with shared resources has been presented. The mathematical model can be addressed by means of a Binary Integer Programming routine or by means of several heuristics. At the center of the problem is the resource contention policy. For the case here proposed, the Extended Stack Resource Policy is used. The introduction of the blocking relation as an equivalence relation allows a class partition of the set of tasks allowing a more refined analysis of the interactions of the different tasks and shared resources.
The two heuristics presented in this paper are quite simple and only required the computation of the worst case blocking times that comes out from the class partition analysis. In this sense, they can be implemented with a few steps and provide a solution to the schedulability problem.

References