# Enhanced Fixed-Priority Scheduling with (m,k)-Firm Guarantee

Gang Quan Xiaobo (Sharon) Hu
Department of Computer Science & Engineering
University of Notre Dame
Notre Dame, IN 46556
{gquan,shu}@cse.nd.edu

## Abstract

*In this paper, we study the problem of scheduling task sets with (m,k) constraints. In our approach, jobs of each task are partitioned into two sets: mandatory and optional. Mandatory jobs are scheduled according to their pre-defined priorities, while optional jobs are assigned to the lowest priority. We show that finding the optimal partition as well as determining the schedulability of the resultant task set are both NP-hard problems. A new technique, based on the* General Chinese Remainder Theorem*, is proposed to quantify the interference among tasks, which is then used to derive two partitioning approaches. Furthermore, a sufficient condition is presented to predict in polynomial time the schedulability of mandatory jobs. We prove that our partitions are never worse than those obtained in previous work. Experimental results also show significant improvement achieved by our approaches.*

## 1. Introduction

Much work has been conducted in scheduling analysis of hard real-time systems, where violating task deadlines must be avoided at all cost. However, in many real-time embedded systems, e.g., a video decoder, it is acceptable to miss task deadlines occasionally. Several models have been proposed to study such systems, e.g., the imprecise computation model [4], the "skip-over" model [11], and the (m,k) model [9]. In the (m,k) model, a dynamic failure occurs if fewer than $m$ out of any $k$ consecutive jobs of some task meet their deadlines. If $m = k$, the system becomes a hard-deadline system. For the special case of $m = k - 1$, the (m,k) model reduces to the "skip-over" model [11]. The (m,k) model can be readily incorporated into system Quality of Service (QoS) requirements, and is applicable to many real-time systems such as those in multimedia and automotive control. In this paper, we use the (m,k) model to study the scheduling problem of overloaded systems.

Some approaches [1, 2, 3, 5, 7, 9, 11, 19] apply dynamic scheduling techniques to handle overloaded real-time systems. However, in many applications, a fixed-priority scheduling algorithm is usually more attractive than a dynamic-priority one because $(i)$ it incurs lower overhead; $(ii)$ the implementation is relatively simple; $(iii)$ it gives a designer control over task priorities. Hence, we focus on fixed-priority scheduling for the (m,k) model. A few papers have been published that study the (m,k) model under fixed-priority scheduling. In [1], jobs are "promoted" to higher priorities according to some off-line patterns in order to meet the (m,k) constraints and reduce the response time of soft deadline tasks. However, further work needs to be done to search for the *effective* patterns. In [11], the "skip-over" model is used and the task set schedulability is analyzed in that context, but the results cannot be readily applied to the (m,k) model. In [16], a scheduling technique is proposed for the general (m,k) model. The beauty of the technique is that it uses a very simple algorithm to partition the jobs of each task into two sets: mandatory and optional. All mandatory jobs are scheduled according to their fixed priorities, while all optional jobs are assigned the lowest priority. It follows that if all mandatory jobs meet their deadlines, no dynamic failure will happen.

Though the technique proposed in [16] is simple and elegant, it does have some potential problems. First, the *first* job of every task is always designated as mandatory, which forces the worst case response time of every task to be that of the first job. Secondly, the job partition algorithm implicitly distributes the mandatory jobs evenly among $k$ consecutive jobs of a task. Such even distribution may not be advantageous in certain situations. Furthermore, the partition algorithm depends solely on the ratio of $m$ over $k$ of each task. That is, regardless of task periods and execution times, the mandatory jobs of two tasks having the same $m$ over $k$ ratio are always distributed in the same way among the $k$ consecutive jobs. In Section 2, we provide some examples to illustrate the consequence of the above problems. In summary, all the above problems can significantly im-

pact task set schedulability, which may then lead to overly pessimistic designs.

We believe that judicious selection of mandatory *v.s.* optional jobs plays a critical role in scheduling systems with (m,k) constraints. In this paper, we first prove that finding the *optimal* partition between mandatory and optional jobs for each task is NP-hard in the strong sense. Then, we present a heuristic algorithm to modify the partitions given in [16]. Through analyzing the effects of preemption and blocking on a mandatory jobs by higher priority ones, we design an algorithm to carefully select mandatory jobs and reduce such effects. Our experimental results show that our algorithm produces significantly better partitions than that in [16] in terms of system schedulability. We also prove that our solutions form a super set of that obtained by [16], i.e., any task set with (m,k) constraints schedulable by [16] is always schedulable with our algorithm.

The schedulability of (m,k) systems can be further improved if one can tolerate spending some more time on finding better mandatory/optional partitions off-line. A probabilistic optimization algorithm (e.g., a genetic or simulated annealing algorithm) can be very effective in this regards. One challenge in applying such algorithms is to formulate an appropriate objective function. We propose a metric used as the objective function, and demonstrate its effectiveness by implementing a genetic algorithm based on this metric. The experimental results are extremely encouraging.

Another difficulty is to determine the schedulability of tasks with (m,k) constraints for a mandatory/optional job partition, which we prove to be NP-hard. One way to solve this problem is to perform the exact analysis for a large number of possible cases as suggested in [1, 18], which is computationally intractable for large task sets. We present a sufficient condition which can be used to determine in polynomial time if a given set of mandatory jobs is schedulable. The condition was derived based on an extension to the algorithm presented in [10].

The paper is organized as follows. In Section 2, we define our problem and analyze some related work. In Section 3, we prove several theorems to demonstrate some characteristics of the problem and then introduce an important concept, *execution interference*, to capture the preemption and blocking effects among tasks. Section 4 contains a detailed discussion of our approaches on partitioning the jobs and checking the schedulability for task sets with (m,k) constraints. Experimental results are given in Section 5 and our work is summarized in Section 6.

## 2. Preliminaries and Related Work

Consider a system with $n$ independent periodic tasks, $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_n\}$, arranged in the decreasing order of their priorities. Each instance of a task is called a *job*. The $j$th job of $\tau_i$ is denoted as $\tau_{ij}$. The following timing parameters are defined for task $\tau_i$:

- *initial time* ($O_i$): the release time of the first job of $\tau_i$,
- *period*($T_i$): the interval between two consecutive job release times of $\tau_i$,
- *deadline*($D_i$): the maximum time allowed from the release to the completion of a $\tau_i$'s job,
- *execution time*($C_i$): the maximum time needed to complete $\tau_i$ without any interruption,
- $m_i$ and $k_i$: the (m,k) constraint for $\tau_i$, which mandates that at least $m$ out of any $k$ consecutive jobs of $\tau_i$ must meet their deadlines to avoid any dynamic failure.

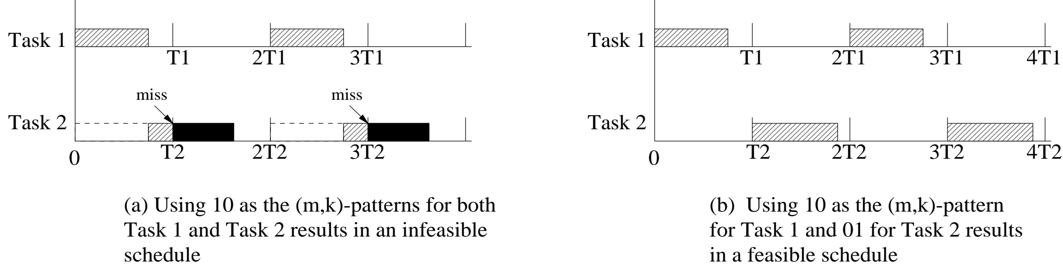
When scheduling a task set with (m,k) constraints according to a fixed-priority assignment, one critical step is to determine for each task whether its execution is mandatory or optional. This may be envisioned as each job being associated with a binary variable $\pi$. If $\pi = 1$, the corresponding job is mandatory. Otherwise, it is optional. The collection of all these binary variables forms a binary string, which we refer to as the *mandatory job pattern*. Apparently, the selection of such mandatory job pattern for each task may greatly impact the schedulability of the task set. To ease our effort in searching for the mandatory job patterns which can satisfy the (m,k) constraints while making the task set as schedulable as possible, we first introduce the following definition.

**Definition 1** *The* ***(m,k)-pattern*** *of task $\tau_i$, denoted by $\Pi_i$, is a binary string $\Pi_i = \{\pi_{i1}\pi_{i2}...\pi_{ik_i}\}$ which satisfies the following: (i) $\tau_{ij}$ is a mandatory job if $\pi_{ij} = 1$ and optional if $\pi_{ij} = 0$, and (ii) $\sum_{j=1}^{k_i} \pi_{ij} = m_i$.*

By repeating the (m,k)-pattern $\Pi_i$, we get a mandatory job pattern for $\tau_i$. It is not difficult to see that the (m,k) constraints for $\tau_i$ can be satisfied if the mandatory jobs of $\tau_i$ are selected accordingly.

Note that the length of the (m,k)-pattern for task $\tau_i$ is $k_i$. Although we may increase the length of the pattern to $2k_i, 3k_i, ...$, to improve the flexibility of selecting mandatory job patterns, this may greatly increase the complexity of scheduling analysis and complicate system implementation at the same time. For example, if the length of a pattern is chosen to be $2k_i$, then $\sum_{j=1}^{2k_i} \pi_{ij} = 2m_i$ does not necessarily guarantee the (m,k) constraint. In this case, $2k_i$ windows with size $k_i$ each need to be checked (wrap around the pattern if necessary) in order to guarantee that the (m,k) constraint is never violated.

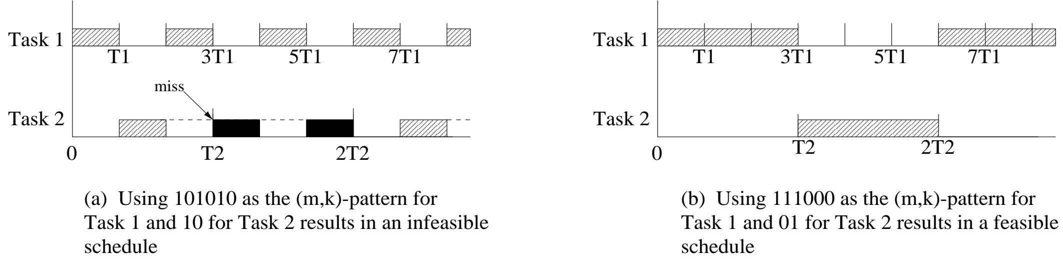
With the definition of the (m,k)-pattern, we formulate the fixed-priority (m,k) scheduling problem as follows.




(a) Using 10 as the (m,k)-patterns for both Task 1 and Task 2 results in an infeasible schedule

(b) Using 10 as the (m,k)-pattern for Task 1 and 01 for Task 2 results in a feasible schedule

**Figure 1. Different (m,k)-patterns for the same task set lead to different scheduling results.**




(a) Using 101010 as the (m,k)-pattern for Task 1 and 10 for Task 2 results in an infeasible schedule

(b) Using 111000 as the (m,k)-pattern for Task 1 and 01 for Task 2 results in a feasible schedule

**Figure 2. Evenly distributed mandatory jobs may not always improve the schedulability.**

**Definition 2** *Given a task set $\mathcal{T}$, let the mandatory jobs defined by a set of (m,k)-patterns be assigned fixed priorities and the optional jobs have the lowest priority. Find the optimal (m,k)-pattern $\Pi_i$ for each $\tau_i \in \mathcal{T}$ such that no other (m,k)-patterns can satisfy the (m,k) constraints if the optimal pattern cannot satisfy the (m,k) constraints.*

Solving the above problem consists of two challenges: $(i)$ given a task set with (m,k) constraints, how to determine if one set of (m,k)-patterns is better or easier to be scheduled than another; $(ii)$ given a set of (m,k)-patterns, how to predict if the mandatory jobs are all schedulable.

In [11], the authors consider the "skip-over" model, a special case of the above fixed-priority (m,k) scheduling problem where $m = k - 1$. They prove that determining whether a set of periodic, occasionally skippable tasks is schedulable is NP-hard in the weak sense. We will extend their proof and show that the problem of finding the optimal (m,k)-patterns is NP-hard in the strong sense. When applying the fixed-priority scheduling algorithm in the "skip-over" model, the authors in [11] implicitly adopt the so-called *deeply-red* task set to be the mandatory job set. This corresponds to the following (m,k)-pattern:

$$\pi_{ij} = \begin{cases} 1 & 1 \leq j < k_i - 1 \\ 0 & j = k_i \end{cases} \tag{1}$$

For the above (m,k)-pattern, a sufficient and necessary condition is presented in [11] to determine the schedulability. It is claimed in [11] that the worst case occurs in the deeply-red task set in the "skip-over" model. However, no further work is done on choosing different (m,k)-patterns to improve the schedulability of a task set.

In [16], the general (m,k) model is used and an algorithm is proposed for determining the (m,k)-patterns for a given task set, which leads to the following (m,k)-pattern,

$$\pi_{ij} = \begin{cases} 1 & \text{if } j = \lfloor \lceil \frac{(j-1) \times m_i}{k_i} \rceil \times \frac{k_i}{m_i} \rfloor + 1 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where $j = 1, 2, \cdots, k_i$. For the (m,k)-patterns above, one can see that the (m,k)-pattern for a task is fixed once its (m,k) constraint is defined, and the first job of every task is always labeled to be mandatory. Moreover, it is proved in [16] that the algorithm gives the most mandatory jobs from $[0, t]$ compared with those in any other interval of the same length $t$. One attractive consequence of the approach in [16] is that the schedulability analysis can be conducted by simply extending that proposed in [12], since the first job of each task always has the worst case response time. However, this advantage becomes less desirable in terms of meeting (m,k) constraints.

Consider the example in Figure 1. Here, the task set contains two tasks with the same periods and the same (m,k)-firm constraint, i.e., (1,2). It is shown in Figure 1(a) that the mandatory jobs cannot be scheduled if the (m,k)-patterns are assigned according to (2), while some different (m,k)-patterns can satisfy the (m,k) constraints (see Figure 1(b)).

In addition to forcing the worst case response time of every task to be that of the first job, the technique in [16] implicitly distributes the mandatory jobs evenly among $k_i$ consecutive jobs of $\tau_i$. Such even distribution may not be

desirable in certain situations as seen in the example given in Figure 2, where the (m,k) constraint of $\tau_1$ is $(3, 6)$ and that of $\tau_2$ is $(1, 2)$.

In the following, we present our contributions on solving the (m,k) scheduling problem.

## 3. Several Observations

In this section, we first present several observations related to the complexity issues of the (m,k) scheduling problem. Then, we discuss an important concept for estimating preemption and blocking effects among tasks with (m,k) constraints.

### 3.1. Complexity issues

We first show that selecting the "optimal" (m,k)-pattern for each task can be "very difficult".

**Theorem 1** *Given a task set $\mathcal{T}$ the problem of deciding if there exists an (m,k)-pattern for each task in $\mathcal{T}$ such that $\mathcal{T}$ is schedulable is NP-hard in the strong sense.*

**Proof:** We prove the theorem by reducing the *3-Partition problem* to our problem. The *3-Partition problem* is defined as follows: Given a set $A = \{a_1, a_2, \cdots, a_{3m}\}$ of $3m$ positive integers and a positive integer $B$ such that $\frac{1}{4}B < a_i < \frac{1}{2}B$ and $\sum_{i=1}^{3m} a_i = mB$, can $A$ be partitioned into $m$ disjointed sets, $A_1, A_2, ..., A_m$, such that $\sum_{a_i \in A_j} a_i = B$ for each $1 \leq j \leq m$? The *3-Partition problem* is proved to be NP-hard in the strong sense [6].

Given a *3-Partition problem*, we construct a task set $\mathcal{T} = \{\tau_1, \tau_2, \cdots, \tau_{3m}\}$ such that $O_i = 0, C_i = a_i, D_i = T_i = B, m_i = 1, k_i = m$. Assume we have found an (m,k)-pattern for each $\tau_i$ such that $\mathcal{T}$ is schedulable. Then, after clustering tasks with the same (m,k)-pattern to form $\mathcal{T}_i'$ and let the corresponding $a_j$ form $A_i$, we have

$$\mathcal{T}_i' \text{ is schedulable } \Longleftrightarrow \sum_{a_j \in A_i} a_j = B, i = 1, \cdots, m$$

Since the above reduction is linear, we prove the theorem. □

Another challenge in solving the (m,k) scheduling problem is to decide if the mandatory jobs given by a set of (m,k)-patterns are schedulable. Unfortunately, the problem is also NP-hard. Refer to [15] for the proof.

**Theorem 2** *Given a task set $\mathcal{T}$ and an (m,k)-pattern for each task in $\mathcal{T}$, the problem of determining whether $\mathcal{T}$ is schedulable is NP-hard.*

In Section 2, we reviewed the deeply-red task set used by the "skip-over" model in [11] and showed its (m,k)-pattern in (1). Here, we extend the deeply-red task set definition to the general (m,k)-firm guarantee model.

**Definition 3** *Given a task set $\mathcal{T}$ with (m,k) constraints, the* ***deeply-red (m,k)-pattern*** *for task $\tau_i$, $\Pi_i^r = \{\pi_{i1}^r \pi_{i2}^r ... \pi_{ik_i}^r\}$, satisfies*

$$\pi_{ij}^r = \begin{cases} 1 & 1 \leq j \leq m_i \\ 0 & m_i < j \leq k_i \end{cases}$$

For the deeply-red (m,k)-pattern, we make the following observation, (see [15] for the proof).

**Theorem 3** *For task set $\mathcal{T}$ with $O_i = 0, 1 \leq i \leq n$, if the mandatory jobs defined by the deeply-red (m,k)-patterns are schedulable, the mandatory jobs derived from any other (m,k)-patterns are also schedulable.*

The proof of the theorem can be easily obtained by recognizing that the first job of every task in the deeply-red (m,k)-pattern leads to the worst case response time among all possible (m,k)-patterns (since it occurs at the critical instant and overlaps with the most mandatory jobs from other higher-priority tasks). The theorem plays an important role in comparing the performance of different (m,k)-patterns, and is used in the presentation of our experimental results.

### 3.2. Execution interference among tasks

As discussed in the previous sections, determining the schedulability of a task set with (m,k) constraints is a challenging problem, since exact timing analysis [1, 18] for a large number of possible cases is very time consuming and is in fact intractable for large task sets. To reduce the computational cost, we propose an effective way to help characterize and quantify the preemption and blocking effects on lower priority mandatory jobs by higher priority ones.

Given two tasks $\tau_h$ and $\tau_i$ ($h < i$), we say that a $\tau_h$'s job *interferes* a $\tau_i$'s job if the execution time interval of the $\tau_h$'s job either partially or entirely overlaps with the period of the $\tau_i$'s job. We use the term *execution interference* of $\tau_h$ with

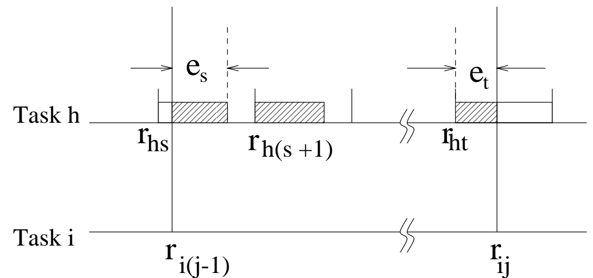



**Figure 3. Execution interference of $\tau_h$ with $\tau_{ij}$, where $r_{pq}$ is the release time of job $\tau_{pq}$.**

job $\tau_{ij}$ to capture the amount of *potential* preemption and/or blocking effect caused by $\tau_h$ during $[(j-1)T_i + O_i, jT_i + O_i]$. In Figure 3, $\tau_{hs}$,$\tau_{h(s+1)}$, and $\tau_{ht}$ all interfere with $\tau_{ij}$,

and the execution interference of $\tau_h$ with $\tau_{ij}$ is shown by the shaded regions. Formally, we define execution interference as follows.

**Definition 4** *Given two tasks $\tau_h$ and $\tau_i$ ($h < i$) and the (m,k)-pattern for each task, the* ***execution interference*** *of $\tau_h$ with job $\tau_{ij}$, denoted by $F_{ij}^h$, equals total portions of the execution times of all $\tau_h$'s mandatory jobs that fall inside $[(j-1)T_i + O_i, jT_i + O_i]$.*

(Note that in Figure 3, $e_s$ and $e_t$ become zero if the corresponding jobs are not mandatory). Mathematically, $F_{ij}^h$ can be calculated as follows,

$$F_{ij}^h = e_s + l_{ij}^h \times C_i + e_t, \tag{3}$$

where $l_{ij}^h$ is the number of mandatory jobs of $\tau_h$ that fall entirely in the interval $[(j-1)T_i+O_i, jT_i+O_i]$, $e_s = \pi_{hs} \cdot \min\{C_h + r_{hs} - r_{i(j-1)}, 0\}$, and $e_t = \pi_{ht} \cdot \min\{C_h, r_{ij} - r_{ht}\}$.

Each mandatory job of $\tau_i$ may suffer different amount of interference by $\tau_h$, and the job of $\tau_i$ that suffers the most execution interference from higher priority tasks tends to dominate the schedulability of $\tau_i$. We refer to this maximum execution interference as the execution interference of task $\tau_h$ with task $\tau_i$, and denote it by $\mathcal{F}_i^h$, i.e.,

$$\mathcal{F}_i^h = \max_j\{F_{ij}^h\}, j = 1, 2, \cdots. \tag{4}$$

Since there exists an infinite number of mandatory jobs for task $\tau_i$, it might seem daunting to determine $\mathcal{F}_i^h$. To tackle this problem, we borrow an existing theorem, **Generalized Chinese Remainder Theorem** (GCRT)[14], which is restated below.

**Theorem 4 (GCRT)** *Let $v_1, v_2, \cdots, v_r$ be positive integers, $v$ be the least common multiple of $v_1, v_2, \cdots, v_r$, and $a, u_1, \cdots, u_r$ be any integers. There exists exactly one integer $u$ which satisfies $a \le u < a + v$ and $u = u_j \pmod{v_j}$ for all $1 \le j \le r$ if and only if $u_i = u_j \pmod{\gcd(v_i, v_j)}$ for all $1 \le i < j \le r$, where $\gcd(x, y)$ denotes the greatest common divisor (GCD) of $x$ and $y$.*

(Note that $a = b \pmod c$ is equivalent to $a \bmod c = b \bmod c$.) Based on GCRT, we proof two lemmas to be used for analyzing the execution interference between tasks. For generality, we use "events" rather than "tasks" in the lemmas.

**Lemma 1** *Given two periodic events $E_1$ and $E_2$ with period $T_1$ and $T_2$, respectively, let the initial times of the two events be the same, i.e., $O_1 = O_2$. Denote the release time of any instance of $E_1$ (resp., $E_2$) by $r_1$ (resp., $r_2$). Then, $r_1 - r_2 = q * \gcd(T_1, T_2)$, $q \in Z$ ($Z$ is the set of integers).*

Lemma 1 states that the interval between the release times of any two instances of two periodic events always equals the integer multiple of the GCD of their periods, if these two periodic events have the same initial time. Similarly, for periodic events having different initial times, we have the following lemma.

**Lemma 2** *Suppose that two periodic events $E_1$ and $E_2$ have periods $T_1$ and $T_2$, and different initial times $O_1$ and $O_2$, respectively. Denote the release time of any instance of $E_1$ (resp., $E_2$) by $r_1$ (resp., $r_2$). Then, $r_1 - r_2 = p * g + (O_1 - O_2) \bmod g$, where $g = \gcd(T_1, T_2)$, $p \in Z$. Furthermore, let $\min|r_1 - r_2|$ be the* minimum distance *between the release times of any $E_1$'s instance and any $E_2$'s instance, then $\min|r_1 - r_2| \le \frac{g}{2}$.*

(The proof of these two lemmas are omitted due to page limit. Interested readers can refer to [15] for the details.)

Observe that $\tau_i$'s mandatory jobs corresponding to bit $\pi_{ij} = 1$ can be viewed as a periodic event $E_i$ with period $k_iT_i$ and initial time $O_i + (j-1)T_i$, and the mandatory jobs of $\tau_h$ can also be viewed as a periodic event $E_h$ with period $k_hT_h$ and initial time $O_h$. Let the release time of an instance of $E_i$ (resp., $E_h$) by $r_i$ (resp., $r_h$). According to Lemma 2, $r_i - r_h = \{p * g + ((j-1)T_i + O_i - O_h) \bmod g\}$, where $g = \gcd(k_hT_h, k_iT_i)$, $p \in Z$. Note that each unique value of $0 \le (r_i - r_h) < k_hT_h$ may result in a different execution interference of $\tau_h$ for the corresponding $\tau_i$'s job. However, for $(r_i - r_h) < 0$ or $(r_i - r_h) \ge k_hT_h$, the interferences simply repeat the cases for $0 \le r_i - r_h < k_hT_h$. Therefore, the computation of the execution interference between two tasks can be greatly simplified. Algorithm 1 describes a procedure to conduct this computation. The term *execution*

**Algorithm 1** Calculate the execution interference between two tasks

1: **Input:** $\tau_i = \{O_i, T_i, D_i, C_i, m_i, k_i\}$, $\tau_h = \{O_h, T_h, D_h, C_h, m_h, k_h\}, \Pi_i, \Pi_h, h < i$
2: **Output:** $\mathcal{F}_i^h$ //execution interference of $\tau_h$ with $\tau_i$
3: $\mathcal{F}_i^h = 0$;
4: $g = \gcd(k_iT_i, k_hT_h)$;
5: **for** j from 1 to $k_i$ **do**
6:   **if** $\pi_{ij} = 1$ **then**
7:     $x = (O_i + (j-1)T_i - O_h)\ mod\ g$;
8:     **while** $x < k_hT_h$ **do**
9:       $F_{ij}^h$ is calculated according to (3);
10:       **if** $\mathcal{F}_i^h < F_{ij}^h$ **then**
11:         $\mathcal{F}_i^h = F_{ij}^h$;
12:       **end if**
13:       $x = x + g$;
14:     **end while**
15:   **end if**
16: **end for**

*interference* between tasks forms the basis of our proposed approaches to be discussed in the next section.

# 4. Our Approaches

In this section, we first present a heuristic technique to improve the (m,k)-patterns obtained by [16]. We then propose a metric that can be used as an objective function in any probabilistic optimization algorithm. Finally, we derive a sufficient condition to predict the schedulability of a task set with given (m,k)-patterns.

## 4.1. Improving evenly distributed (m,k)-patterns

In Section 2, we point out that the algorithm in [16] results in (m,k)-patterns that are not always desirable. We hereby present a heuristic technique to obtain better (m,k)-patterns by judiciously "rotating" the (m,k)-patterns computed by (2). The key idea is to reduce the execution interference between tasks.

As mentioned before, execution interference between tasks can have a significant impact on the schedulability of a task set. It would be very helpful if we know at what instants the maximum execution interference for a given set of (m,k)-patterns may occur. We introduce the term *worst-case interference point* to capture this concept.

**Definition 5** *A **worst-case interference point** (WCIP) of task $\tau_i$ is the beginning instant of a time interval such that the number of mandatory jobs of $\tau_i$ is the largest among all time intervals of the same length.*

Based on the above definition, for the (m,k)-patterns defined in (2), time 0 is a *worst-case interference point* since it is proved in [16] that interval $[0, t]$ contains the largest number of mandatory jobs compared with any other interval with the same length. Note that any task,$\tau_i$, has an infinite number of WCIPs for a given (m,k)-pattern and they occur periodically with period $k_iT_i$. If a mandatory job from a lower priority task is released at the same time as one of the WCIPs of some higher priority tasks, the job will apparently suffer the largest execution interference from the higher priority tasks. Intuitively, given a set of (m,k)-patterns, if a WCIP of a lower priority task and those of higher priority tasks concur, it will be more difficult to meet the (m,k) constraints, which is the case for the (m,k)-patterns by [16].

If (m,k)-patterns can be defined such that the WCIPs between tasks are as far apart as possible, the schedulability of the task set would be improved. One way to achieve this is to modify (2) as follows.

$$\pi_{ij} = \begin{cases} 1 & \text{if } j = \lfloor \lceil \frac{((j-1)+s_i)\times m_i}{k_i} \rceil \times \frac{k_i}{m_i} \rfloor + 1 \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

where $s_i \geq 0$ and $s_i \in Z$. Note that the new (m,k)-pattern can be viewed as rotating the (m,k)-pattern in (2) right by $s_i$ bits. The new (m,k)-pattern certainly satisfies the (m,k) constraints. Furthermore, we have the following lemma.

**Lemma 3** *For $\tau_i$ with the (m,k)-pattern defined in (5), the number of mandatory jobs of $\tau_i$ is the largest in $[s_i \times T_i, s_i \times T_i + t]$ compared with those within any other interval of the same length $t$.*

The proof can be readily obtained by applying Lemma 4 in [16] and is thus omitted. According to Lemma 3, by rotating the original (m,k)-pattern defined in (2), we essentially move the first WCIP of task $\tau_i$ from 0 to $s_iT_i$. Hence, through careful selection of $s_i$ ($0 \leq s_i < k_i$) values, we can alter the separation between WCIPs of different tasks.

Our problem now becomes determining the value for $s_i$ to separate WCIPs among tasks as far as possible. Since the WCIPs of a task occur periodically, we resort to Lemma 2 in our search for better $s_i$ values. Given task $\tau_i$ and the (m,k)-pattern defined in (5), the WCIPs for $\tau_i$ is a periodic event with period $k_iT_i$ and initial time $O_i + s_iT_i$. According

**Algorithm 2** Algorithm for finding rotation values for (m,k)-patterns

```
1:  Input: Task set 𝒯 = {τ1, τ2, ..., τn}, where τi =
    {Oi, Ti, Di, Ci, mi, ki}
2:  Output: s1, ···, sn //rotation values for each tasks
3:  Ψ = ∅; // Ψ contains the tasks whose si values have
    been determined
4:  while 𝒯 is not empty do
5:    τi = task in 𝒯 with the smallest ki;
6:    if Ψ ≠ ∅ then
7:      Ω = Ψ;
8:      while Ω ≠ ∅ do
9:        τj = task in Ω such that F_i^j is maximum, where
          F_i^j is defined in (4);
10:       g = gcd(ki × Ti, kj × Tj);
11:       if g = 1 then
12:         remove τj from Ω;
13:       else
14:         break;
15:       end if
16:     end while
17:     O'j = Oj + sj × Tj;
18:     si = l such that 0 ≤ l < ki and | l × Ti + Oi − O'j |
        is nearest to one of (2q + 1) × g/2, q ∈ Z;
19:   else
20:     si = 0;
21:   end if
22:   Add τi to Ψ;
23:   Remove τi from 𝒯;
24: end while
```

to Lemma 2, the distance between the closest WCIPs of two tasks, $\tau_i$ and $\tau_j$, is never bigger than $\gcd(k_iT_i, k_jT_j)/2$. Hence, we can select $s_i$ and $s_j$ such that the distance is as close to $\gcd(k_iT_i, k_jT_j)/2$ as possible to reduce the execution interference between the two tasks. For task sets containing three or more tasks, we design a greedy algorithm shown in Algorithm 2.

The basic idea of Algorithm 2 is to reduce the worst case response time of mandatory jobs by reducing the execution interference between tasks. Observe that the larger the value $k_i$ is, the more choices task $\tau_i$ has for the position of its first WCIP. Hence, among the remaining tasks whose $s_i$ values need to be determined, the algorithm always pick the one having the smallest $k_i$ in its (m,k) constraint. Then, the algorithm selects task $\tau_j$ from the tasks whose $s_j$ values have been determined such that the execution interference between $\tau_i$ and $\tau_j$ is the largest. The $s_i$ value is then set so that the distance between the WCIPs are maximized. Note that in the case when $\gcd(k_iT_i, k_jT_j) = 1$, no matter what the initial positions of WCIPs are, they will eventually meet at some time instant in the future. If this happens, we simply go on to the next task.

Algorithm 2 is quite effective in improving the schedulability of task sets with (m,k) constraints. We will give experimental results later to illustrate this. Furthermore, we have the following theorem. Refer to [15] for the proof.

**Theorem 5** *If a task set can be scheduled with the (m,k)-patterns defined by (2), it can always be scheduled with the (m,k)-patterns defined in (5) with $s_i$ determined by Algorithm 2.*

### 4.2. A metric for evaluating (m,k)-patterns

Though the algorithm proposed in the previous section is capable of improving the schedulability of task sets employing the (m,k)-patterns derived in [16], there exist cases where no rotating (m,k)-patterns can improve the schedulability. This was demonstrated by the example in Figure 2 in Section 2. In such cases, evenly distributed (m,k)-patterns are not appropriate. We need to find other (m,k)-patterns. Since determining the optimal (m,k)-patterns is NP-hard, a natural contender for solving the problem is a probabilistic optimization approach based on such as genetic algorithms (GA) or simulated annealing (SA), both of which have been shown to be effective in solving many NP-hard problems [8, 17]. GA and SA differ in their mechanisms for escaping local minima, but both need an effective objective function to help direct the search process. A major factor to the success of such an approach is the choice of the objective function. We borrow the term *fitness* from GA to refer to the objective function, where a higher fitness value indicates a better solution. In this subsection, we present a fitness function which is quite effective for finding superior (m,k)-patterns.

An ideal fitness function should be able to reflect the fact that using one set of (m,k)-patterns may make the system "easier" to be scheduled than another set. The challenge is how to describe this "easiness". Intuitively, a set of (m,k)-patterns leading to shorter worst case response times for tasks is better. Yet, we have shown in Section 3 that, given arbitrary (m,k)-patterns, finding the worst case response time of a task is NP-hard. As discussed before, the execution interference suffered by a task directly impacts the schedulability of the task. We hereby propose to use the *execution interference* between tasks to formulate the fitness function. Specifically, let the fitness of $\tau_i$ be $f(\tau_i)$. Then, we have

$$f(\tau_i) = \frac{T_i}{C_i + \sum_{h=1}^{i-1} \mathcal{F}_i^h}, \tag{6}$$

where $\mathcal{F}_i^h$ is defined in (4). The denominator in (6) is an estimated worst case work load for $\tau_i$ and all the higher priority tasks during any time interval of length $T_i$. To define the overall fitness value for a task set with some known (m,k)-patterns, we notice that a task set is considered to be unschedulable if any one of its tasks misses the deadline. Hence, the task-set fitness, denoted by $f(\mathcal{T})$, is the minimum among the fitness values of all tasks, i.e.,

$$f(\mathcal{T}) = \min_{1 \le i \le n} f(\tau_i), \tag{7}$$

Given a task set with known (m,k)-patterns, evaluating $f(\tau_i)$ hinges on computing the execution interferences between pairs of tasks, which can be obtained by Algorithm 1. We should point out that the fitness function defined above is only an estimated metric for the task set schedulability. That is, we cannot guarantee that for any given task sets $\mathcal{T}_1$ and $\mathcal{T}_2$, $\mathcal{T}_2$ must be schedulable if $f(\mathcal{T}_1) < f(\mathcal{T}_2)$ and $\mathcal{T}_1$ is schedulable. Though more accurate execution interference may be obtained by exact schedulability analysis, the computational cost would be too large since fitness function needs to be evaluated many times.

After the fitness function is obtained, we can apply either a GA or SA approach to search for better (m,k)-patterns. Figure 4 illustrates the flowchart of a GA implementation. In our GA implementation, the population consists of $\lambda$ individuals, each of which contains $n$ tuples, i.e., ($\tau_i$, $\Pi_i$),$i = 1, ..., n$. Individuals with higher fitness values are selected as survivors. During reproduction, either mutation or crossover is used according to a predefined probability. The mutation operation randomly chooses a task and changes one bit in its (m,k)-pattern from 1 to 0 and another bit from 0 to 1. The crossover operation randomly chooses a cut point for two individuals and swaps their contents to construct two new individuals. More detailed discussion on

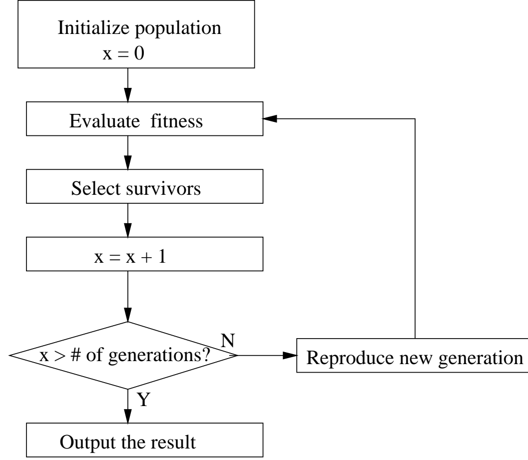



**Figure 4. A genetic algorithm procedure**

GA can be found in [8]. Our experimental results are extremely encouraging as shown later. While the effectiveness of our approach is demonstrated in the experiments, how to construct a better fitness function remains an open problem.

### 4.3. Determining the schedulability

We have proposed two methods to find better (m,k)-patterns for the (m,k) scheduling problem. Yet, we still face the challenge of determining if a task set is schedulable for some given (m,k)-patterns. Answering this question becomes critical when the first job of every task no longer has the worst case response time. In section 3, we know that this is an NP-hard problem. Note that a task $\tau_i$ with certain (m,k)-patterns can be viewed as $m_i$ periodic tasks with period $k_iT_i$, deadline $D_i$, and initial times $(a_i - 1)T_i + O_i$ ($a_i$ is the index of the mandatory job in $\tau_i$'s (m,k)-pattern). One way to deal with this problem is to apply an exact timing analysis technique[1, 18] for all the jobs possibly having the worst case response time, which is computationally prohibitive for large task sets. In the following, we construct a sufficient condition to test if a task set with known (m,k)-patterns is schedulable. Our goal is to be able to efficiently evaluate such a condition, and hence quickly decide if a task set with some (m,k)-patterns can meet the (m,k) constraints. To simplify the problem, we assume that the deadline of a task equals its period.

Our sufficient condition is an extension to the work by Han and Tyan [10]. In [10], a polynomial-time algorithm is proposed to test the schedulability of a fixed-priority hard real-time system. The basic idea is to map each task in the task set to a new task such that the new task period is less than or equal to the original period and the computation time remains the same. An additional requirement is that the new task set must be *harmonic*, i.e., any shorter task period must divide any longer task period. It is proved in [10] that if the harmonic task set is schedulable, so is the original task set. However, this is no longer true for a task set with (m,k) constraints. Figure 5 illustrates such an example, where $\mathcal{T} = \{\tau_1, \tau_2\}, T_1 = C_1 = 6, T_2 = 7, C_2 = 6$, and $(m_1, k_1) = (m_2, k_2) = (1, 2)$. The corresponding harmonic task set $\mathcal{T}' = \{\tau_1', \tau_2'\}$ with $T_1' = C_1 = 6, T_2' = C_2 = 6$, and $(m_1', k_1') = (m_2', k_2') = (1, 2)$. As shown in Figure 5(a), $\mathcal{T}'$ can be easily scheduled by executing the mandatory jobs alternatively, but $\mathcal{T}$ cannot be scheduled with the same (m,k)-patterns as shown in Figure 5(b).

We derive a sufficient condition that can be applied to tasks with (m,k) constraints. Consider $\tau_i$ in a *harmonic* task set $\mathcal{T}$. Let $\tau_j$ has a higher priority than $\tau_i$. Then for any mandatory job of $\tau_i$ released at $t_0$, at most $\lceil \frac{T_i}{T_j} \rceil$ mandatory jobs of $\tau_j$ occur in $[t_0, t_0 + T_i]$. Since $\mathcal{T}$ is a harmonic task set, so

$$\lceil \frac{T_i}{T_j} \rceil = \begin{cases} \frac{T_i}{T_j} & T_i > T_j \\ 1 & \text{otherwise} \end{cases}$$

Suppose $l_{ij}$ is the maximum number of mandatory jobs from $\tau_j$ during any time interval of length $T_i$. Let

$$W_i = \sum_{j \le i} (l_{ij} \times C_j) \qquad (8)$$

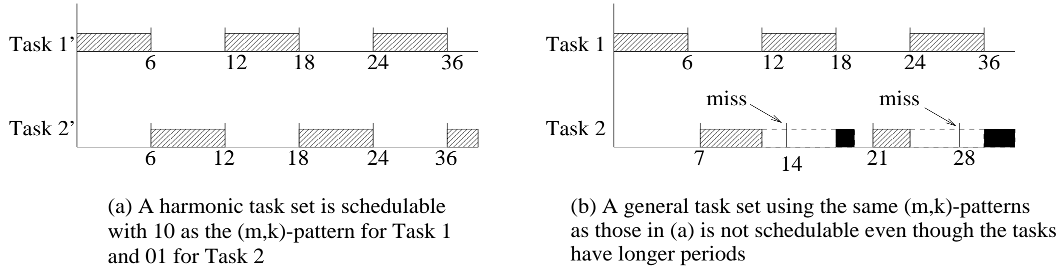
Then, if $\frac{W_i}{T_i} \le 1$, the total work load including the $\tau_i$'s job and all other higher priority mandatory jobs under consideration can be completed in one $\tau_i$'s period. Hence, task $\tau_i$ is certainly schedulable. For general task sets, we have the following theorem. (See [15] for the proof.)

**Theorem 6** *Given two task sets $\mathcal{T}$ and $\mathcal{T}'$ with $T_i' \le T_i, C_i' = C_i, m_i' = m_i, k_i' = k_i$, and $T_j'$ divides $T_i'$ if $T_j' \le T_i'$. With the given (m,k)-patterns, if $\sum_{j \le i} (l_{ij} \times C_j)/T_i' \le 1$, where $l_{ij}$ is the* maximum *number of mandatory jobs during any time interval of length $T_i'$, then $\mathcal{T}$ is schedulable.*

Given a task set with (m,k) constraints, we can apply the algorithm in [10] to find the corresponding harmonic task set, and determine $l_{ij}$ from the given (m,k)-patterns. Then, by Theorem 6, the schedulability of the task set can be tested. A straightforward implementation of our sufficient condition takes $O(n^3 klogn)$ time, where $k = \max_i k_i$ and $n$ is the number of tasks. Though our analysis above is based on the assumption that $D_i = T_i, i = 1, ..., n$, the result can be extended to the case when $D_i < T_i, i = 1, ..., n$ with the similar approach shown in [10]. Finding tighter sufficient conditions without greatly increasing the computational cost is an open problem.

## 5. Experimental Results

In this section, we present some experimental results to compare our approaches with that in [16]. For ease of explanation, we use **Alg_Orig** for the algorithm in [16], **Alg_RT**




(a) A harmonic task set is schedulable with 10 as the (m,k)-pattern for Task 1 and 01 for Task 2

(b) A general task set using the same (m,k)-patterns as those in (a) is not schedulable even though the tasks have longer periods

**Figure 5. Harmonic task set and its original task set**

| | No. of Schedulable Task Sets | | | Improvement(%) | |
|---|---|---|---|---|---|
| **Utilization** | **Alg_Orig** | **Alg_RT** | **Alg_GA** | **Alg_RT** | **Alg_GA** |
| 0.8 - 1.0 | 28.3 | 31.1 | 31.2 | 9.89 | 12.24 |
| 1.0 - 1.2 | 133.7 | 153.7 | 151.1 | 15.96 | 13.01 |
| 1.2 - 1.4 | 105.6 | 123.9 | 127.8 | 17.32 | 21.02 |
| 1.4 - 1.6 | 20.1 | 26.6 | 36.3 | 32.34 | 80.60 |
| 1.6 - 1.8 | 1.6 | 3.0 | 6.1 | 87.50 | 281.25 |
| 1.8 - 2.0 | 0.0 | 0.1 | 0.6 | NaN | NaN |

**Table 1. Experimental results comparing the three approaches**

for Algorithm 2 in Section 4.1, and **Alg_GA** for our approach discussed in Section 4.2.

Recall that the goal of our approaches is to select a set of (m,k)-patterns such that they will make the given task set easier to be scheduled. According to Theorem 3, a task set can be scheduled with any set of (m,k)-patterns if it is schedulable with the *deeply-red* (m,k)-patterns. In this case, there would be no benefit to apply the (m,k)-patterns obtained by either [16] or our approaches. Hence, we discard such task sets during our experiments. Moreover, since utilization factor values greatly impact task set schedulability, a fair comparison needs to study a large spectrum of utilization factor values.

In one set of our experiments, we consider task sets with 5 tasks starting from the same time. The period of each task is randomly selected and uniform distributed between 10 to 50. The deadline of each task is assumed to equal its period. The $m_i$ and $k_i$ values are also randomly selected. $k_i$ is uniformly distributed between 2 to 10, and $m_i$ is uniformly distributed between 1 and $k_i$. We partition the total utilization factor values into intervals of length 0.2. The execution time of each task is randomly selected such that the utilization values of the task sets are uniformly distributed within each interval. To reduce statistical errors, the number of task sets schedulable by at least one of the approaches is no less than 50 within each interval, or at least 5000 different task sets have been generated for the interval. In the genetic algorithm implementation (**Alg_GA**), both population size and the number of generations are set to 30. To precisely assess the performance of the approaches, we resort to simulation to check the schedulabilities of the task sets.

The program is run for 10 times and the average results are collected in Table 1. In our experiments, task sets with utilization values less than 0.8 are all schedulable using the deeply-red (m,k)-patterns, and none of the task set with utilization greater than 2.0 is schedulable with any of the approaches. Hence, we omit these data in Table 1. In Table 1, columns 2-4 list the average numbers of schedulable task sets by each approach across 10 runs. The columns labeled "Improvement" represent the improvements of our two approaches over the approach in [16].

From Table 1, one can conclude that both **Alg_RT** and **Alg_GA** improve the performance of **Alg_Orig**, and the improvements become more significant as the task-set utilization factor values increase. In the experiments, as we expect, a task set is schedulable with **Alg_RT** as long as it is schedulable with **Alg_Orig**. We would like to point out that there exist few cases when a task set is schedulable with **Alg_Orig** but cannot be scheduled with **Alg_GA**. The solution quality of **Alg_GA** can be improved if we increase the population size of the number of generations. Of course, this will increase the computation time.

We constructed another set of experiments to test the timing performance for each of the approaches. All the parameters are selected as above except the number of tasks in each task set is set to 30. The experiments were conducted on a SUN Ultra-1 workstation. The results show that for each task set, excluding the time for simulation,

the CPU time for **Alg_Orig** is negligible ($\approx$ 0), 0.06s for **Alg_RT**, and 203.7s for **Alg_GA**. Obviously, **Alg_GA** does take much longer CPU time than **Alg_RT** and **Alg_Orig**. Nevertheless, as shown in Table 1, for a large number of task sets, much more task sets can be scheduled with **Alg_GA**, and in most cases, **Alg_GA** has the best performance among the three approaches in terms of the number of task sets satisfying the (m,k) constraints.

## 6. Conclusions

In this paper, we address the problem of scheduling task sets with (m,k) constraints using the fixed-priority scheme. Similar to [16], our scheduling approach partitions the jobs of each task into mandatory or optional jobs. All the jobs are scheduled according to their static priorities with the optional jobs assigned the lowest priority. We prove that finding the optimal partition as well as determining the schedulability of the resultant task set are both NP-hard problems. Since traditional hard real-time analysis techniques can be very time consuming for analyzing the behavior of such a task set, we propose a new technique, based on the General Chinese Remainder Theorem, to quantify the interference between tasks. We then propose two approaches to improve the partitions proposed in [16]. Compared with the approach in [16], our approaches produce better partitions in terms of reducing the interference among mandatory jobs and thus better exploit the (m,k) constraints in overloaded systems. We prove that our solution space is a super set of that in [16]. Furthermore, we propose a sufficient condition which takes only polynomial time to predict the schedulability for a task set with arbitrary (m,k)-patterns. Experimental results show that the improvements with our approaches are quite significant.

Our future work includes constructing a more precise fitness function for a task set with given (m,k)-patterns and searching for tighter sufficient conditions to predict the schedulability of such a task set.

## 7. Acknowledgments

We would like to thank the reviewers for their valuable comments. The research is supported in part by NSF under grant number MIP-9796162 and MIP-9701416.

## References

[1] G. Bernat and A. Burns. Combining (n,m)-hard deadlines and dual priority scheduling. *Proceedings of Real-Time Systems Symposium*, pages 46–57, Dec 1997.

[2] G. Buttazzo. Value vs. deadline scheduling in overload conditions. *Proceedings of Real-Time Systems Symposium*, pages 90–99, Dec 1995.

[3] M. Caccamo and G. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. *Proceedings fo Real-Time Systems Symposium*, pages 330–339, Dec 1997.

[4] J.-Y. Chung, J. W. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 39(9):1156–1175, Sep 1990.

[5] M. K. Gardner and J. W.S.Liu. Performance of algorithms for scheduling real-time systems with overrun and overload. *Proceedings of the eleventh euromicro conference on real-time systems*, pages 287–296, Jun 1999.

[6] M. Garey and D. Johnson. *Computers and Intractability: A Guid to the Theory of NP-Completeness*. FreeMan, San Francisco, CA, 1979.

[7] K. Gilad and S. Dennis. Dover: an optimal on-line scheduling algorithm for overloaded uniprocessor real-time systems. *Electronics Letters*, 33(15):1301–1302, July 1997.

[8] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, MA, 1989.

[9] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Transactions on Computes*, 44:1443–1451, Dec 1995.

[10] C.-C. Han and H.-Y. Tyan. A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms. *Proceedings of the Real-Time Systems Symposium*, pages 36–45, 1997.

[11] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. *Proceedings of Real-Time Systems Symposium*, pages 110–117, Dec 1995.

[12] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. *Proceedings of the 1989 IEEE Real-time System Symposium*, pages 166–171, 1989.

[13] J. Y.-T. Leung and M.L.Merrill. A note on preemptive scheduling of periodic, real-time tasks. *Information Processing Letters*, 11(3):115–118, Nov 1980.

[14] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic,real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[15] G. Quan and X. Hu. *Enhanced Fixed-priority Scheduling with (m,k)-firm Guarantee*. Technical Report TR 00-09, Dept. of Computer Science & Engineering, University of Notre Dame, 2000.

[16] P. Ramanathan. Overload management in real-time control applications using (m,k)-firm guarantee. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):549–559, Jun 1999.

[17] F. Remeo. *Simulated Annealing: Theory and Applications to Layout Problems*. PhD thesis, Dept. Of Elec. Eng. & Comp. Sci., University of California, Berkeley, Mar. 1989.

[18] K. Tindell. *Adding time-offsets to schedulability analysis*. Technical Report YCS 221, Dept. of Computer Science, University of York, England, 1994.

[19] R. West and K. Schwan. Dynamic window-constrained scheduling for multimedia applications. *The 6th International Conference on Multimedia Computing and Systems*, Jun 1999.