# Efficient Performance Estimation for General Real-Time Task Systems

Hongchao (Stephanie) Liu* Xiaobo (Sharon) Hu*

Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN 46556, USA
{hliu, shu}@cse.nd.edu

## Abstract

The paper presents a novel approach to compute tight upper bounds on the processor utilization independent of the implementation for general real-time systems where tasks are composed of subtasks and precedence constraints may exist among subtasks of the same task. We formulate the problem as a set of linear programming (LP) problems. Observations are made to reduce the number of LP problem instances required to be solved, which greatly improves the computation time of the utilization bounds. Furthermore, additional constraints are allowed to be included under certain circumstances to improve the quality of the bounds.

## 1. INTRODUCTION

System-level design exploration is becoming indispensable as the time-to-market pressure ever increases and system-on-a-chip technology matures. One of the great challenges in system-level design exploration is rapid timing performance estimation since such estimation must be performed for a large number of design alternatives. Fast prediction of system timing behavior is essential to the success of any design exploration tool. For real-time systems which must respond to external events under given timing constraints , analyzing the timing performance is particularly critical because fail to respond to events on time may seriously degrade system performance or even result in a catastrophe.

One effective approach to rapidly estimating real-time system performance is to use processor utilization bounds. If a tight utilization bound exists, given a system implementation (such as a particular processor), the processor utilization can be readily computed and compared with the bound to determine whether all timing requirements can be satisfied. Utilization bounds depend on many parameters, e.g., the system architecture and scheduling policy. In this paper, we are interested in real-time systems employing a fixed-priority, preemptive scheduling policy. This scheduling policy is adopted by most real-time systems of practical interest due to its low overhead and predictability [11].

The best known utilization bound is the work by Liu and Layland in [10]. This bound is applicable to a single processor system with periodic tasks. The tasks are not allowed to have precedence constraints and a task deadline must equal its period. The authors in [4] derived an improved bound. Park, *et.al.*, presented a linear programming (LP) based method to compute the utilization bounds [12, 13], which produces much tighter bounds than the previous one. A further improvement to the results in [12, 13] is proposed in [9]. Besides being tighter, the utilization bounds derived by the LP-based methods can also be applied to the case where task deadlines are less than their periods. However, *all* the above approaches has a serious drawback. They can only deal with systems composed of *independent* tasks, i.e., tasks with no precedence constraints. This greatly hinders the use of these approaches in many real-world real-time system designs.

In this paper, we present a technique to compute the processor utilization bounds for more general real-time systems. We consider systems containing periodic tasks with precedence constraints. All tasks are assumed to be executed on a single processor according to a fix-priority preemptive scheduling scheme. This assumption, although not universal, is true for many embedded systems in practice, particularly for low-cost, high-volume consumer products [2, 6]. Note that such a model could also be used to capture those system architectural alternatives containing multiple processors and dedicated hardware components, provided that the tasks are assigned statically to the processors and the communication among the processors is achieved via shared memory.

A number of papers have been published that study the problem of performance prediction for real-time tasks with precedence constraints, e.g., [1, 7, 15, 16]. These papers all focus on providing sophisticated algorithms to check if a given implementation can satisfy the timing requirement. Unfortunately, such algorithms are generally quite time consuming especially when used repeatly during the early design exploration process.

Our approach aims at computing tight upper bounds on processor utilization. For a given system specification, the processor utilization bounds are obtained independent of the implementation (such as choices of processors). Then, the bound can be used in the design exploration process to rapidly determine if an implementation of the system satisfies the timing constraints. By carefully analyzing the relationship among tasks, we formulate the problem of finding the tightest upper bound as a set of LP problems. Furthermore, we have made a number of observations to simplify the LP formulations, which greatly reduces the time needed to solve the LPs. Our LP model allows additional constraints to be included to capture some known characteristics of the system (e.g., the relative sizes of task execution times). Adding such constraints judiciously improves the quality of the bounds. However, care must be taken when adding such constraints in order to guarantee the bounds to be valid. We present guidelines based on linear algebra principles to help the designer determine the right combination of constraints to add. We demonstrate the effectiveness of our approach through a number of experimental results. To our best knowledge, this work is the first in finding utilization bounds for periodic task systems with precedence constraints.

## 2. PRELIMINARIES

We adopt the task graph model [16] to describe a real-time system containing periodic tasks with precedence constraints. In particular, given a system with $n$ tasks, we use $n$ directed acyclic graphs (DAG) to model the system, where each DAG corresponds to a task. The vertices in a

*This research was supported in part by the National Science Foundation under Grant MIP-9701416 and MIP-9796162.

DAG represent the subtasks while the edges represent the precedence constraints among them. See Figure 1 for an example. A subtask is only ready to be executed when all of its predecessors are completed.

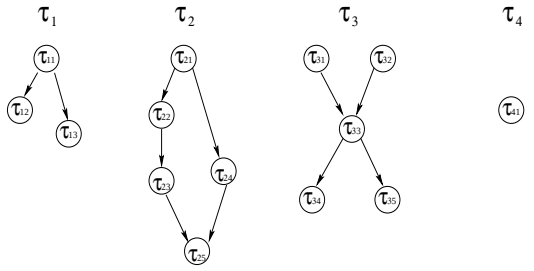



**Figure 1: The task graph representation of a periodic task system containing four tasks.**

The tasks are denoted by $\boldsymbol{\tau}_1, \boldsymbol{\tau}_2, \ldots \boldsymbol{\tau}_N$, and the subtasks of $\boldsymbol{\tau}_i$ are denoted by $\tau_{i,1}$, $\tau_{i,2}$, ..., $\tau_{i,m_i}$, where $m_i$ is the total number of subtasks in task $\boldsymbol{\tau}_i$. Let $N$ and $M$ be the total number of tasks and total number of subtasks, respectively. Given that the tasks must be executed periodically, we refer to the $k$th instance of task $\boldsymbol{\tau}_i$ as the $k$th job of $\boldsymbol{\tau}_i$. Each task and subtask is associated with the following timing parameters:

- $T_i$: the interval between the release times of two consecutive jobs of $\boldsymbol{\tau}_i$, referred to as *task period*.
- $D_i$: the maximum time allowed from the release to the completion of a $\boldsymbol{\tau}_i$'s job, referred to as *task deadline*. In general, $D_i \leq T_i$.
- $d_{i,j}$: the maximum time allowed from the release of a $\boldsymbol{\tau}_i$'s job to the completion of $\tau_{i,j}$, referred to as *subtask deadline*, and $d_{i,j} \leq D_i$.
- $c_{i,j}$: the maximum time needed to complete $\tau_{i,j}$ without any interruption, referred to as *subtask execution time*.
- $C_i$: the maximum time needed to complete $\boldsymbol{\tau}_i$ without any interruption, referred to as *task execution time*, and $C_i = \sum_{j=1}^{m_i} c_{i,j}$.
- $p_{i,j}$: the priority level statically assigned to $\tau_{i,j}$, referred to as *subtask priority*. We say that $\tau_{i,j}$ has a higher priority than $\tau_{h,k}$ if $p_{i,j} > p_{h,k}$.

The tasks are assumed to be executed on a single processor and the overheads due to preemption and context switching are assumed to be negligible.

During the design exploration process, task periods, task and subtask deadlines as well as subtask priorities are usually given while the execution times are not known and are dependent on the implementation. For a given implementation, the subtask execution times become known and the processor utilization due to executing the first $n$ tasks is computed by

$$U_n = \sum_{i=1}^{n} \frac{C_i}{T_i} = \sum_{i=1}^{n} \frac{\sum_{j=1}^{m_i} c_{i,j}}{T_i} \tag{1}$$

The problem we intend to solve is formulated as follows:

DEFINITION 1. *Given the task periods, deadlines, as well as subtask priorities of a periodic task system, find utilization bounds $B_n$ such that an implementation of the system is guaranteed to be schedulable if $U_n < B_n$ for all $1 \leq n \leq N$.*

As discussed in Introduction, several previous papers have studied the utilization bound problem in depth [10, 4, 12, 13, 9] . We briefly review the results in [12] since our method bears similarities to it. In [12], Park, *et.al.* presented an LP-based approach to compute utilization bounds for a set of independent tasks. The key idea is to find the minimum processor utilization for executing $n$ tasks under the requirement that there should not be any idle time before the first job of $\boldsymbol{\tau}_n$ is finished. To compute this minimum utilization, the authors formulate an LP problem instance as follows [12]:

$$\begin{aligned} \textbf{Minimize:} \quad & B_n = \sum_{i=1}^{n} \frac{C_i}{T_i} \\ \textbf{Subject To:} \quad & \sum_{i=1}^{n} C_i \lceil \frac{qT_k}{T_i} \rceil \geq qT_k \\ & \qquad q = 1, 2, \cdots, \lfloor T_n/T_k \rfloor \\ & \qquad k = 1, 2, \cdots, n-1. \\ & \sum_{i=1}^{n} C_i \lceil \frac{D_n}{T_i} \rceil \geq D_n \end{aligned} \tag{2}$$

The tasks are assumed to be in the decreasing order of their priorities. The constraints in the above LP are formulated as such that the processor is kept busy by executing $\boldsymbol{\tau}_n$ and tasks with higher priorities. Why the solution yields a valid bound? Let us assume that in an implementation, the processor utilization due to executing the first $n$ tasks is $U_n$. If $U_n < B_n$, according to the nature of LP problems, the given task execution times must have violated at least one of the constraints in (2). That is, the first job of $\boldsymbol{\tau}_n$ as well as higher priority tasks *is guaranteed to complete* before either one of the $qT_k$'s or $D_n$. Since the first job of $\boldsymbol{\tau}_n$ has the longest response time [10], the implementation is definitely schedulable for the first n tasks.

One serious drawback of the existing approaches for computing utilization bounds is that they only deal with systems composed of *independent* tasks. Since for independent tasks, when all tasks request at the same time, a task can be preempted by any higher priority task. Thus the total time required to finish the first job of $\boldsymbol{\tau}_n$ is the longest among all it's jobs and can be written as the left hand side of the inequalities in (2). However, for tasks with precedence constraints, this is no longer true. For example, consider a simple case in which task $\boldsymbol{\tau}_n$ contains only one subtask $\tau_{n,1}$ and task $\boldsymbol{\tau}_q$ contains two subtasks $\tau_{q,1}$ and $\tau_{q,2}$. Assume the execution of $\tau_{q,1}$ must proceed $\tau_{q,2}$, and $p_{q,1} < p_{n,1} < p_{q,2}$. Suppose a job of $\tau_{n,1}$ is released before the completion of a job of $\tau_{q,1}$. Then, $\tau_{q,2}$ cannot preempt $\tau_{n,1}$ since $\tau_{q,1}$ prevents $\tau_{q,2}$ from starting during $\tau_{n,1}$'s execution. However, if $\tau_{q,1}$ is finished immediately prior to the release of $\tau_{n,1}$, $\tau_{q,2}$ will delay the start of $\tau_{n,1}$. The existing approaches are not capable of handling such complex cases.

For the general task graph model, the effect of one subtask on the execution of another subtask need to be carefully characterized. Harbour, Klein and Lehoczky analyzed such inter-subtask effects for a special task graph model in [7], where each periodic task is assumed to contain a sequential list of subtasks with a predefined execution order within the sequence. Furthermore, the subtask execution times as well as task periods and deadlines are given. The aim is to estimate the worst-case execution time. To achieve this, a task is converted to a particular form while the other tasks are grouped with respect to this task into five different groups. *The worst-case phasing* of $\boldsymbol{\tau}_n$ due to the existence of other tasks represent the particular release times of the other tasks that create the longest response time for $\boldsymbol{\tau}_n$. The complexity of the algorithms in [7] is quite high ($O(MN^2L/(1-U))$, where $M$ is the total number of subtasks, $N$ is the number of tasks, $L = \max T_i/T_j$ and $U$ is the processor utilization). If this algorithm is used during design space exploration, it must be applied to each implementation under consideration, which can be very time consuming.

Since the technique in [7] requires all subtask execution times be known, it cannot be readily used during design space exploration. However, some useful concepts in [7] can be extended to our general task model. These will be discussed in detail in the following section.

## 3. BOUND COMPUTATION

In this section, we present our approach to determining the utilization bounds of periodic task systems with precedence constraints. Our idea is to formulate LP problem instances similar to [12]. However, we are

facing a number of challenges. Recall (from the last section) that for independent task sets, the first job of a task has the worst case phasing [10]. That is, if the first job can be finished by its deadline, all other jobs of the same task can meet their deadlines as well. For tasks with precedence constraints, what is the worst case phasing for a task or subtask? Furthermore, each constraint in (2) corresponds to one integer multiple of the period of a higher priority task. In the task graph model, the subtasks in one task does not necessarily have uniformly higher or lower priorities than the subtasks in another task. How should such constraints be formulated? We will describe our ideas to tackle these and other challenges.

To simplify our exposition, we first assume that the deadlines of the subtasks are the same as the deadlines of their corresponding tasks, i.e., $d_{n,1} = d_{n,2} = \cdots = d_{n,m_n} = D_n$. Removing this constraint only requires a minor modification to our approach, we will explain at the end of this section.

## 3.1 Analyzing worst-case phasings

We first determine the worst-case phasing between a task, say $\tau_n$, and other tasks. The importance of the worst-case phasing is that $\tau_n$ suffers the longest response time if the other tasks happen to release jobs at the worst-case phasing. In this process, we borrow some terminology introduced in [7]. However, our classification is simpler.

At first glance, it may seem that analyzing the effects of the subtasks in other tasks on the execution of $\tau_n$ require the consideration of each individual subtask in task $\tau_n$. Actually, if the deadlines of subtasks of $\tau_n$ are assumed to be the same as the deadlines of $\tau_n$, analyzing such effects only need to focus on the lowest priority subtask in $\tau_n$. This is a generalization of some observations presented in [7].

The observations in [7] are applicable to tasks each of which contains a list of subtasks to be executed in a sequential order. In our case, the subtasks in a task are modeled as a DAG rather than a one-dimensional sequential list. However, the following lemma shows that a DAG-based task model can be readily converted to a list-based model when scheduled on a single processor.

LEMMA 1. *Given a set of periodic tasks each of which consists of subtasks modeled by a DAG, the subtasks in each DAG can be converted to a one-dimensional list by Topological-Sort with priority assignment. The resulting task list has exactly the same execution schedule as the original DAG-based model when scheduled on a single processor.*

From now on, we will assume that the subtasks in each DAG are indexed by the order in the corresponding one-dimensional list specified in Lemma 1. We identify three different ways that other tasks/subtasks may impact the response time of task $\tau_n$. Let the lowest priority among all the subtask priorities in $\tau_n$ be $P_n$. Our characterizations are somewhat similar to the ones in [7] but are defined differently so as to simplify our subsequent analysis.

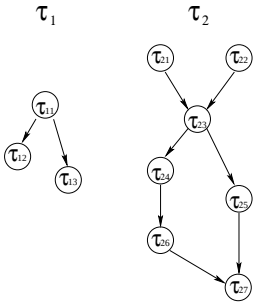
DEFINITION 2. *If the priority of every subtask in task $\tau_i$ is higher than $P_n$, $\tau_i$ is called a* multiple-preemption *task for $\tau_n$, and the set of all such tasks are denoted as $MP(n)$.*

It is easy to see that $\tau_i$ can preempt (or interrupt the execution of) $\tau_n$ more than once if $T_i < T_n$ and $T_i \in MP(n)$.

DEFINITION 3. *If the priorities of a set of subtasks, $\{\tau_{i,1}, \tau_{i,2}, \cdots, \tau_{i,k}\}$, $1 \le k < m_i$, are all higher than $P_n$ while the priority of $\tau_{i,k+1}$ is lower than $P_n$, this set of subtasks is called a* single-preemption *subtask set for $\tau_n$, and is denoted as $sp(n,i)$. The set of all such sets are denoted as $SP(n)$.*

Note that even if $T_i < T_n$, $sp(i)$ can only preempt $\tau_n$ once since the lower priority subtask $\tau_{i,k+1}$ in $\tau_i$ prevents it to be completed before $\tau_n$ is completed.

DEFINITION 4. *If the priorities of a set of subtasks $\{\tau_{i,h}, \tau_{i,h+1}, \cdots, \tau_{i,k}\}$ (where h and k are two integers such that $1 < h \le k \le m_i$) are all higher than $P_n$ while the priority of $\tau_{i,h-1}$ and $\tau_{i,k+1}$ (if $k < m_i$) are lower than $P_n$, this set of subtasks is called a* blocking *subtask set for $\tau_n$ and is denoted as $bk(n,i,j)$. The set of all such subtask sets are denoted as $BK(n)$.*




| | $p_{i1}$ | $p_{i2}$ | $p_{i3}$ | $p_{i4}$ | $p_{i5}$ | $p_{i6}$ | $p_{i7}$ | $T_i$ |
|---|---|---|---|---|---|---|---|---|
| i=1 | 6 | 8 | 5 | | | | | 70 |
| i=2 | 9 | 6 | 3 | 8 | 7 | 4 | 10 | 120 |

**Figure 2: Examples of task/subtask characterizations: $\tau_1$ is a *multiple-preemption* task for $\tau_2$, $\{\tau_{2,1}, \tau_{2,2}\}$ is a *single-preemption* subtask set for $\tau_1$, $\{\tau_{2,4}, \tau_{2,5}\}$ and $\{\tau_{2,7}\}$ are blocking subtask sets for $\tau_1$.**

In Figure 2, we provide examples of the above classifications using two tasks.

Multiple-preemption tasks for $\tau_n$ have the same effect on the execution of $\tau_n$ as that of higher priority independent tasks on a lower priority one. Single-preemption and blocking subtask sets, on the other hand, have some unique properties. They are described in the following two lemmas.

LEMMA 2. *Consider task $\tau_n$ and task $\tau_i$ which contains both a* single-preemption *subtask set, $sp(n,i)$, and multiple* blocking *subtask sets, $bk(n,i,j)$ (for $j = 1, 2, \cdots$). If $sp(n,i)$ preempts $\tau_n$, none of $bk(n,i,j)$'s can block $\tau_n$. If one of $bk(n,i,j)$'s block $\tau_n$, $sp(n,i)$ cannot preempt $\tau_n$ unless this blocking subtask set contains $\tau_{i,m_i}$.*

LEMMA 3. *Task $\tau_n$ can be blocked no more than once during the time that the processor is busy executing the subtasks of $\tau_n$ and other higher priority subtasks.*

Lemma 3 is a generalization of a lemma given in [7] where the lemma is already proved. Proofs for Lemma 2 and Lemma 3 are quite straight forward and are omitted.

If a task $\tau_i$ is not a multiple-preemption task to $\tau_n$ and contains neither a single-preemption subtask set nor blocking subtask sets, the priority of every subtask in $\tau_i$ must be lower than $P_n$. If a job of $\tau_n$ is released during the execution of $\tau_i$, $\tau_n$ can immediately preempt it. Hence, $\tau_i$ will not have any impact on the response time of $\tau_n$ (under the assumption that preemption overhead is negligible). To ensure that the above definitions capture the worst-case phasing, whenever the priority of another subtask is equal to $P_n$, we assume $P_n$ is lower . This is a pessimistic assumption but guarantees to result in valid bounds.

We are now ready to determine the worst-case phasings for $\tau_n$. The main difference between our observations and those in [7] is that in [7] the execution times of subtasks are known *a priori* and hence the worst-case phasing of another task for $\tau_n$ can be uniquely determined. In our case, the subtask execution times are unknown. To overcome this difficulty, we let the worst-case phasings be associated with either tasks or subtasks instead of with tasks alone as in [7]. We summarize the conclusions in three lemmas and omit the proofs.

LEMMA 4. *The worst-case phasing of $\tau_n$ with existence of a* multiple-preemption *task $\tau_i$ occurs when the first subtask of $\tau_i$ is released at the same time as the first subtask of $\tau_n$. The number of preemptions by $\tau_i$ is equal to $\lceil T_n/T_i \rceil$.*

LEMMA 5. *The worst-case phasing of $\tau_n$ with existence of a* single-preemption *subtask set $sp(n,i)$ occurs when the first subtask of $sp(n,i)$ is released at the same time as the first subtask of $\tau_n$.*

LEMMA 6. *The worst-case phasing of $\tau_n$ with existence of a* blocking *subtask set $bk(n,i,j)$ for $\tau_n$ occurs when the first subtask of $bk(n,i,j)$ is released at the same time as the first subtask of $\tau_n$.*

Based on these lemmas, we are ready to formulate LP instances for computing the utilization bounds.

## 3.2 Computing utilization bounds

**LP($n,i,j$)**:

**Minimize:**

$$U(n,i,j) = \sum_{\tau_k \in MP(n)} \frac{C_k}{T_k} + \sum_{\tau_{k,h} \in SP_i(n)} \frac{c_{k,h}}{T_k} + \sum_{\tau_{i,k} \in bk'(n,i,j)} \frac{c_{i,k}}{T_i} + \frac{C_n}{T_n}$$

**Subject To:**

$$\sum_{\tau_k \in MP(n)} C_k \lceil \frac{qT_l}{T_k} \rceil + \sum_{\tau_{k,h} \in SP_i(n)} c_{k,h} + \sum_{\tau_{i,k} \in bk'(n,i,j)} c_{i,k} + C_n \geq qT_l$$
$$q = 1, 2, \cdots, \lfloor T_n/T_l \rfloor,$$

where $\tau_l \in MP(n)$ and $T_l < T_n$

$$\sum_{\tau_k \in MP(n)} C_k \lceil \frac{D_n}{T_k} \rceil + \sum_{\tau_{k,h} \in SP_i(n)} c_{k,h} + \sum_{\tau_{i,k} \in bk'(n,i,j)} c_{i,k} + C_n \geq D_n$$

where $SP_i(n) = SP(n) \backslash sp(n,i)$ and

$$bk'(n,i,j) = \begin{cases} bk(n,i,j) & \text{if } \tau_{i,m_i} \notin bk(n,i,j) \\ bk(n,i,j) \cap sp(n,i) & \text{otherwise} \end{cases} \quad (3)$$

An LP-based approach such as the one in [12] has been shown to be very effective for computing utilization bounds for tasks without precedence constraints. We will present new LP formulations for computing utilization bounds of tasks with precedence constraints. Recall that a valid utilization bound must ensure that an implementation with a given combination of subtask execution times is schedulable if the corresponding processor utilization is less than or equal to the bound. The idea behind the LP-based approach is to find the minimum processor utilization given when the processor is kept busy.

The objective function of the LP is simply to minimize the processor utilization by the set of tasks/subtasks under consideration. The main challenge in constructing an LP instance is in building the constraints. In order to guarantee that the utilization bound is a valid one, the constraints in the LP must satisfy the following conditions:

1. The constraints must capture the longest possible response time for the task under consideration.

2. Satisfying the constraints means that the processor is kept busy.

The longest response time of task $\tau_n$ depends on the worst-case phasings of $\tau_n$. Since the execution times of subtasks are unknown, there are a number of possible worst-case phasings that may cause a job of $\tau_n$ to have the longest response time. These worst-case phasings are defined in Lemma 4–6. Note that during the execution of a job of $\tau_n$, the job can be preempted by many multiple-preemption tasks and single-preemption subtask sets. However, it can only be blocked once by a blocking subtask set (see Lemma 3). Thus, in the construction of the LP problem, we must formulate multiple LP instances in order to consider the effects of different blocking subtask sets.

To guarantee that the processor is kept busy at any time instant before completing $\tau_n$, we only need to make sure that the processor is busy at time instant $t$ for every $t$ satisfying (*i*) $t < D_n$ and $t = qT_i$, where $\tau_i$ is a *multiple-preemption* task for $\tau_n$ and $q$ is an integer, or (*ii*) $t = D_n$. Such time instants are referred to as the *scheduling points* of $\tau_n$. The above conclusion can be readily drawn by noticing that these time instants are the only ones when new execution requests may occur.

Based on the above observations, we formulate the LP instance, **LP($n,i,j$)**, to compute the utilization bound for task $\tau_n$ when $bk(n,i,j)$ blocks $\tau_n$. According to Lemma 3, if $bk(n,i,j)$ blocks $\tau_n$, all other blocking subtask sets have no impact on the execution of $\tau_n$. Furthermore, if $bk(n,i,j)$ does not contain the last subtask of $\tau_i$ ($\tau_{i,m_i}$), $sp(n,i)$ cannot preempt $\tau_n$. The LP formulation is given in (3).

The readers can easily verify that the constraints above indeed guarantee that the processor is kept busy during the execution of $\tau_n$.

For each task $\tau_n$, the number of LP instances needed to determine the utilization bound for satisfying $\tau_n$'s deadlines is equal to the number of blocking subtask sets in $BK(n)$. After solving these LPs, the utilization bound for $\tau_n$ is obtained by

$$B_n = \min_{i,j} \{\min U(n,i,j)\} \quad (4)$$

Given a specific implementation, i.e., a combination of subtask execution times, the following procedure can be used to determine the schedulability of the implementation.

Step 1: For task $\tau_n$, determine the effects of the other tasks/subtasks (*multiple-preemption, single-preemption,* or blocking).

Step 2: Find $\max_j \frac{\sum_{\tau_{i,k} \in bk'(n,i,j)} c_{i,k}}{T_i}$ for the given subtask execution times, where $bk'(n,i,j)$ is as defined in (3).

Step 3: Compute $U(n,i,j_{max})$ as defined in (3) for the given subtask execution times.

Step 4: If $U(n,i,j_{max}) < B_n$, then $\tau_n$ is definitely schedulable.

Repeating the above procedure for each task will determine whether the implementation is definitely schedulable. Notice that $U(n,i,j)$, instead of the total processor utilization, is compared with $B_n$. The LP instances need to be solved only once for a given system specification. A straightforward implementation of checking the schedulability of each implementation takes $O(MN^2)$ comparisons, which is much more efficient than the algorithm in [7].

## 3.3 Improving Utilization Bound Computation

Our LP-based approach discussed above can successfully produce a utilization bound for satisfying the deadline of each task. However, the number of LPs to be solved is relatively large. Though they only need to be solved once, it is still beneficial to reduce this number. Furthermore, it is important to study the quality of the bounds. In this subsection, we will present our observations related to these issues.

In the worst case, the total number of LP instances introduced in Section 3.2 is equal to $N(N-1)M/2$. These LPs are the results of a straightforward realization of the observations that we have made. The lemma and theorem introduced below can readily reduce the number of LP instances to $N$ without sacrificing any bound quality.

LEMMA 7. *To compute the utilization bound for task $\tau_n$, only one LP instance, denoted as* **LP($n,i$)**, *is needed for each task $\tau_i$ that contains one or more blocking subtask sets with respect to $\tau_n$.*

Lemma 7 immediately reduces the worst-case total number of LP instances to $N(N-1)$. The proof of it is relatively simple and is omitted due to the page limit. By employing more sophisticated observations, we can reduce the total number of LPs to $N$, which is the same as the case of independent tasks. This result is summarized in the following theorem.

THEOREM 1. *To compute the utilization bound for task $\boldsymbol{\tau}_n$, only one LP instance needs to be solved. This LP, denoted as* **LP($n$)**, *corresponds to the original LP that considers the blocking effect by the task having the longest period among all the tasks containing blocking subtask sets.*

**Proof:** Consider two LP instances, **LP($n,i$)** and **LP($n,j$)** for computing bound $B_n$. They are written as

**LP($n,i$)**:

**Minimize:**

$$U(n,i) = \sum_{\boldsymbol{\tau}_k \in MP(n)} \frac{C_k}{T_k} + \sum_{\tau_{k,h} \in SP_{ij}(n)} \frac{c_{k,h}}{T_k} + \sum_{\tau_{j,k} \in sp(n,j)} \frac{c_{j,k}}{T_j} + \sum_{\tau_{i,k} \in bk'(n,i)} \frac{c_{i,k}}{T_i} + \frac{C_n}{T_n}$$

**Subject To:**

$$\sum_{\boldsymbol{\tau}_k \in MP(n)} C_k \lceil \frac{t}{T_k} \rceil + \sum_{\tau_{k,h} \in SP_{ij}(n)} c_{k,h} + \sum_{\tau_{j,k} \in sp(n,j)} c_{j,k} + \sum_{\tau_{i,k} \in bk'(n,i)} c_{i,k} + C_n \geq t$$

for all $t$ equal to the scheduling points in $[0, D_n]$, (5)

**LP($n,j$)**:

**Minimize:**

$$U(n,j) = \sum_{\boldsymbol{\tau}_k \in MP(n)} \frac{C_k}{T_k} + \sum_{\tau_{k,h} \in SP_{ij}(n)} \frac{c_{k,h}}{T_k} + \sum_{\tau_{i,k} \in sp(n,i)} \frac{c_{i,k}}{T_i} + \sum_{\tau_{j,k} \in bk'(n,j)} \frac{c_{j,k}}{T_j} + \frac{C_n}{T_n}$$

**Subject To:**

$$\sum_{\boldsymbol{\tau}_k \in MP(n)} C_k \lceil \frac{t}{T_k} \rceil + \sum_{\tau_{k,h} \in SP_{ij}(n)} c_{k,h} + \sum_{\tau_{i,k} \in sp(n,i)} c_{i,k} + \sum_{\tau_{j,k} \in bk'(n,j)} c_{j,k} + C_n \geq t$$

for all $t$ equal to the scheduling points in $[0, D_n]$, (6)

where $SP_{ij}(n)$ is the set of the single preemption subtask sets not including $sp(n,i)$ and $sp(n,j)$. If we could show that $\min U(n,i) \leq \min U(n,j)$ for $T_i \geq T_j$, then only **LP($n,i$)** needs to be solved in the computation of $B_n$. Thus, the theorem holds.

To compare **LP($n,i$)** and **LP($n,j$)**, let us assume that $T_i \geq T_j$. From (5) and (6), one can see that the first two summation terms in both objective functions are the same, and so is the case for both constraint sets. We will focus on the remaining two summation terms. There are three cases to be considered.

**Case 1:** Neither $sp(n,i)$ nor $sp(n,j)$ are empty. If we apply the following substitutions: $C' = \sum_{\tau_{i,k} \in bk'(n,i)} c_{i,k}$ and $C'' = \sum_{\tau_{j,k} \in sp(n,j)} c_{j,k}$ in **LP($n,i$)**, and $C' = \sum_{\tau_{i,k} \in sp(n,i)} c_{i,k}$ and $C'' = \sum_{\tau_{j,k} \in bk'(n,j)} c_{j,k}$ in **LP($n,j$)**, it becomes clear that **LP($n,i$)** and **LP($n,j$)** are in fact equivalent. Thus, $\min U(n,i) = \min U(n,j)$. (Note that the above substitutions are acceptable since the two LPs are two separate problems and only the objective function values are of interest to us.)

**Case 2:** Both $sp(n,i)$ and $sp(n,j)$ are empty. Applying the following substitutions: $C' = \sum_{\tau_{i,k} \in bk'(n,i)} c_{i,k}$ in **LP($n,i$)**, and $C' = \sum_{\tau_{j,k} \in bk'(n,j)} c_{j,k}$ in **LP($n,j$)**, we conclude that **LP($n,i$)** and **LP($n,j$)** have exactly the same constraints. However, the two objective functions are different. In fact, we have

$$U(n,i) - U(n,j) = \frac{C'}{T_i} - \frac{C'}{T_j} \leq 0 \tag{7}$$

**Case 3:** One of the single-preemption subtask sets is empty. Assume $sp(n,i) = \emptyset$. (The other case can be proved similarly.) We apply the following substitutions: $C' = \sum_{\tau_{i,k} \in bk'(n,i)} c_{i,k} + \sum_{\tau_{j,k} \in sp(n,j)} c_{j,k}$ and $C'' = \sum_{\tau_{j,k} \in sp(n,j)} c_{j,k}$ in **LP($n,i$)**, and $C' = \sum_{\tau_{j,k} \in bk'(n,j)} c_{j,k}$ in **LP($n,j$)**. The constraint sets of the two LPs become the same, and the objective functions satisfy the following:

$$U(n,i) - U(n,j) = \frac{C' - C''}{T_i} + \frac{C''}{T_j} - \frac{C'}{T_j} \leq 0 \tag{8}$$

Therefore, in all three cases, we have $\min U(n,i) \leq \min U(n,j)$. Note that the cases corresponding to empty blocking subtask sets do not need to be considered as they do not introduce LP instances. □

The time needed to solve one LP is dependent on the number of variables and the number of constraints. A positive side effect of the reductions discussed above is that the substitutions decrease the number of variables. The number of constraints in the LP for task $\boldsymbol{\tau}_n$ is equal to the number of scheduling points of $\boldsymbol{\tau}_n$. The techniques to reduce the number of constraints in the LPs for the independent tasks [9] are also applicable to the LPs here. Therefore, the number of constraints can be reduced by at least one half without sacrificing any bound quality.

The quality of the utilization bounds obtained by the LP-based approach measures the tightness of the bounds. The bounds are in fact the tightest since one can always find at least one system implementation whose utilizations are equal to the bounds but which is not schedulable. However, such a system implementation may not be realistic for a particular system. The reason is that task/subtask execution times are usually non-arbitrary, non-negative numbers. In many cases, these execution times have lower and/or upper bounds, and there exist some relationships among the subtask execution times (e.g., $c_{i,j} \geq 2c_{h,k}$). Including such information as additional constraints in the LPs can further improve the quality of the bounds. We will refer to the LPs discussed previously as the *original* LPs and refer to the LPs with the additional constraints as the *extended* LPs.

It may seem tempting to add as many additional constraints on $c_{i,j}$'s or $C_i$'s as those that are known. However, care must be taken in introducing these additional constraints into the original LPs. According to the LP theory, an optimal solution to an LP must be one of the extreme points in the feasible region [14]. Thus, the solution to the original LP must have some of the constraints be satisfied as an equality. This guarantees that $\boldsymbol{\tau}_n$ is completed at or before its deadline if the utilization of the implementation under consideration is less than the bound. When adding extra constraints, one must make sure that the solution to the extended LP also has at least one of the constraints in the original LP be satisfied as an equality. Otherwise, the bound would no longer be valid.

To ensure that a bound computed by an extended LP is valid, we borrow the *rank* concept from linear algebra [8], where the rank of a matrix is equal to the number of linearly independent columns in the matrix. If we treat the coefficients in the additional constraints as a matrix, we can determine its rank easily. Only when this rank is less than the number of variables in the LP instance, these additional constraints cannot uniquely define a solution point for the LP instance. Thus, at least one of the original constraints is guaranteed to be satisfied as an equality. As we stated above, this guarantees that the bound is valid. Variables that do not appear in the objective function will not effect the bound value. Therefore, constraints containing these variables can be simplified by removing these variables through inequality manipulations.

We would like to point out that the LP reductions by Lemma 7 and Theorem 1 may no longer be valid when additional constraints are intro-

duced. Depending on the forms of the additional constraints, the objective function values of different LP instances may be increased by different amounts. How to generalize the reduction rules is left for our future work.

At the beginning of this section, we made the assumption that $d_{n,1} = d_{n,2} = \cdots = d_{n,m_n} = D_n$. We now discuss the more general case where subtask deadlines can be different from the corresponding task deadline. In this case, instead of analyzing the effects of the subtasks in other tasks on *task* $\boldsymbol{\tau}_n$, we only need to categorize such effects on a *subtask set* of $\boldsymbol{\tau}_n$. Consider a particular subtask of $\boldsymbol{\tau}_n$, say $\boldsymbol{\tau}_{ni}$, where $d_{n,i} < D_n$. It is not difficult to see that subtasks $\tau_{n,j}$ $(i < j \leq m_n)$ have no impact on the worst-case response time of $\tau_{n,i}$ (assuming that all tasks are finished by their deadlines). If we replace $\boldsymbol{\tau}_n$ by $\boldsymbol{\tau}_n^i$ where $\boldsymbol{\tau}_n^i$=$\{\ \tau_{n,1}, \tau_{n,2}, \cdots, \tau_{n,i}\}$, we can simply apply the same technique as discussed above to analyze the effects of other subtasks on $\boldsymbol{\tau}_n^i$ and to construct an LP to compute the utilization bound for satisfying the deadline of $\boldsymbol{\tau}_n^i$. Therefore, for each subtask whose deadline is less than the corresponding task deadline, we need to construct a separate LP to get the utilization bound.

# 4. EXPERIMENTAL RESULTS

In this section, we first use an example from [7] to illustrate how to perform a schedulability analysis by using the method described in the previous section. Then we present and compare some timing data collected from applying our algorithm, both before optimizing and after, to groups of randomly generated tasks.

## 4.1 A real-world example

The example is a real-time robot control system [7]. The system contains five tasks whose parameters are given in Table 1. To conduct schedulability analysis based on the utilization bound approach, one needs to consider each task. We will use task $\boldsymbol{\tau}_3$ to illustrate the procedure step by step.

| | $T_i$ | $D_i$ | $p_{i,1}$ | $p_{i,2}$ | $p_{i,3}$ |
|---|---|---|---|---|---|
| $\boldsymbol{\tau}_1$ | 40 | 40 | 10 | 7 | — |
| $\boldsymbol{\tau}_2$ | 100 | 100 | 4 | 8 | 4 |
| $\boldsymbol{\tau}_3$ | 50 | 50 | 5 | 8 | — |
| $\boldsymbol{\tau}_4$ | 200 | 200 | 9 | 2 | 3 |
| $\boldsymbol{\tau}_5$ | 400 | 400 | 3 | 1 | 6 |

**Table 1: Specification of an example task set.**

**Step 1 :** Classify the rest of the tasks according to the priority level of the lowest priority subtask in $\boldsymbol{\tau}_3$, i.e., $\tau_{3,1}$. We have $MP(3) = \{\boldsymbol{\tau}_1\}$, $SP(3) = \{sp(3,4)\} = \{\{\tau_{4,1}\}\}$, and $BK(3) = \{bk(3,2), bk(3,5)\} = \{\{\tau_{2,2}\}, \{\tau_{5,3}\}\}$. The scheduling points that need to be considered during bound computation are $t = T_1 = 40$ and $t = D_3 = 50$.

**Step 2 :** Construct the LP instance. The blocking subtask sets of $\boldsymbol{\tau}_3$ are from task $\boldsymbol{\tau}_2$ and $\boldsymbol{\tau}_5$. Given that $T_5 > T_2$, we only need to consider **LP(**$3, 5, 1$**)**, which is shown below.

**Minimize:**

$$U(3,5,1) = \frac{c_{1,1}+c_{1,2}}{T_1} + \frac{c_{4,1}}{T_4} + \frac{c_{5,3}}{T_5} + \frac{c_{3,1}+c_{3,2}}{T_3}$$

**Subject To:**

$$c_{1,1} + c_{1,2} + c_{4,1} + c_{5,3} + c_{3,1} + c_{3,2} >= 40$$
$$2c_{1,1} + 2c_{1,2} + c_{4,1} + c_{5,3} + c_{3,1} + c_{3,2} >= 50$$

Solving the above LP instance, we have
$B_n = \min U(3,5,1) = 0.125$.

If each blocking subtask set is used to construct an LP instance, the reader can easily verify that $\min U(3,2,1) > \min U(3,5,3)$.

One may notice that the bound $B_3 = 0.125$ is rather small, and may feel that it is too pessimistic. However, we should emphasize that $B_3$ is not the bound on the total processor utilization. It only bounds the processor utilization due to the *multiple-preemption* tasks, *single-preemption* and *blocking* subtask sets with respect to $\boldsymbol{\tau}_3$.

In Section 3.3, we explained that a bound can be improved if add additional constraints according to known relationships among subtask execution times. To see the effectiveness of this approach, let us assume that we have $c_{3,2} \geq c_{4,1}$ in the above example. The new utilization bound obtained from the extended LP increases to $B_3 = \min U(3,5,1) = 0.46$.

## 4.2 Randomly generated task sets

We have shown in Section 3.2 that testing the schedulability of different implementations of a system becomes very efficient once the bounds are obtained by our LP-based approach. Computing the bounds requires solving a number of LPs. However, each LP only need to be solved once while the resulting bounds can be used numerous times. Furthermore, the time which it takes to solve the LPs are not excessive at all. To demonstrate this, we have conducted a number of experiments based on randomly generated periodic task sets.

The randomly generated task sets may contain anywhere from 10 to 70 tasks and each task may contain anywhere from 1 to 20 subtasks. The subtasks are assumed to have the same deadline as their corresponding task deadlines. Task deadlines, periods, and subtask priorities are randomly generated numbers. Since the ratio of the longest task period and the shortest one can have a significant impact on the number of constraints in an LP, we limit this ratio to 100. The task graphs are generated by using the software package TGFF [5]. We applied our LP-based approach, both original and optimized, to these task graphs to compute the processor utilization bounds. The LPs are solved by a software package `lp_solve` [3]. The programs (including constructing and solving all the LPs) are executed on a Sun SPARC ULTRA 30. We summarize in Table 2 the CPU time usage by the programs. Clearly, the bound computations are quite efficient.

| Num of tasks/subtasks | CPU times | |
|---|---|---|
| | before optimizing (m) | after optimizing (s) |
| 10/70 | 1.07 | 0.65 |
| 20/127 | 20.35 | 0.664 |
| 30/195 | 58.14 | 0.9344 |
| 40/274 | - | 3.0121 |
| 50/322 | - | 4.919 |
| 60/382 | - | 7.4903 |
| 70/468 | - | 11.0252 |

**Table 2: CPU times needed to compute the bounds of randomly generated task sets before and after LP instance reductions. '-' represents a time over an hour and is already impractical.**

# 5. CONCLUSION

In this paper, we introduced an LP-based approach to determine the processor utilization bounds for periodic task sets with precedence constraints. By carefully analyzing the effects of other subtasks on the execution of a task under consideration, we have constructed LP instances to compute the bounds. Based on several observations, we have greatly reduced the number of LPs needed for computing each bound. Furthermore, we have presented guidelines for adding additional constraints to the LPs in order to obtain tighter bounds. Experimental results show that our approach is indeed effective and efficient.

To our best knowledge, this is the first attempt to analyze the schedulability of periodic task sets with precedence constraints through the utilization bounds. This approach will obtain processor utilization bounds independent of the implementation. The bound can be further used in in

the design exploration process to rapidly determine if an implementation of the system is applicable and satisfies all timing requirements. What's more, for hard-to-determine priorities of some subtasks in a particular system, our analysis will help determine a better priority assignment in order to achieve a better system performance. Such characteristic sure meets the challenge of a rapid system timing performance estimation. Our future extensions to this work will include generalizing the reduction rules when additional constraints are added, allowing task deadlines to be greater than their periods, and considering multiple processor systems.

## 6. REFERENCES


[1] N. Audsley, A. Burns, M. Richardson, K. Tindell and A.J. Wellings, "Applying new scheduling theory to static priority preemptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284-292, 1993.

[2] F. Balarin and A. Sangiovanni-Vincentelli, "Schedule validation for embedded reactive real-time systems," *Proceedings of Design Automation Conference*, pp. 52-57, 1997.

[3] M. Berkelaar, *ftp://ftp.es.ele.tue.ne/pub/ep_solve*.

[4] A. Burchard, J.Liebeherr, Y. Oh and S.H. Son, "New strategies for assigning real-time tasks to multiprocessor systems," *IEEE Transactions on Computers*, vol. 44, no. 12, pp. 1429-1442, December, 1995.

[5] R.P. Dick, D.L. Rhodes and W. Wolf, "TGFF Task Graphs for Free," *Proceedings of the Sixth International Workshop on Hardware/Software Codesign (CODES/CASHE'98)*, pp. 97-101, 1998.

[6] W.A. Halang and A.D. Stoyenko, "Next generation of real-time operating systems: industrial perspective," *Proceedings of the NATO Advanced Study Institute on Real Time Computing*, pp. 595-596, 1994.

[7] M.G. Harbour, M.H.Klein and J.P. Lehoczky, "Timing analysis for fixed-priority scheduling of hard real-time systems," *IEEE Transactions on Software Engineering*, vol. 20, no. 1, pp. 13-28, January, 1994.

[8] M.T. Heath, *Scientific Computing: An Introductory Survey*, McGraw-hill Companies, Inc., 1997.

[9] X. Hu and G. Quan, "Fast Performance Prediction for Periodic Task Systems," *Proceedings of the Eighth International Workshop on Hardware/Software Codesign (CODES'00)*, pp. 72-76, 2000.

[10] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46-61, 1973.

[11] J.W.S. Liu, *Real-Time Systems*, Prentice Hall, NJ, 2000.

[12] D. Park, S. Natarajan, A. Kanevsky and M.J. Kim, "A generalized utilization bound test for fixed-priority real-time scheduling," *Proceedings of the Second International Workshop on Real-Time Computing Systems and Applications*, pp. 73-76, Oct. 1995.

[13] D. Park, S. Natarajan and A. Kanevsky, "Fixed-priority scheduling of real-time systems using utilization bounds," *Journal of Systems Software*, vol. 33, pp. 57-63, 1996.

[14] G.V. Shenoy, *Linear Programming Methods and Applications*, John Wiley & Sons, Inc, NY, 1989.

[15] Silva-de-Oliveira-R and da-Silva-Fraga-J, "Fixed priority scheduling of tasks with arbitrary precedence constraints in distributed hard real-time systems," *Journal of Systems Architecture*, vol. 46, no. 11, pp. 991-1004, Sept. 2000.

[16] T.-Y. Yen and W. Wolf, "Performance estimation for real-time distributed embedded systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 11, pp. 1125-1136, Nov. 1998.