Scheduling Fixed-Priority Tasks with Preemption Threshold[*]

Yun Wang
Department of Computer Science
Concordia University
Montreal, QC H3G 1M8, Canada
`y_wang@cs.concordia.ca`

Manas Saksena
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA
`manas@cs.pitt.edu`

Abstract

While it is widely believed that preemptability is a necessary requirement for developing real-time software, there are additional costs involved with preemptive scheduling, as compared to non-preemptive scheduling. Furthermore, in the context of fixed-priority scheduling, feasibility of a task set with non-preemptive scheduling does not imply feasibility with preemptive scheduling (and vice-versa). In this paper, we use the notion of preemption threshold to develop a new scheduling model, which unifies the concepts of preemptive and non-preemptive scheduling, and subsumes both as special cases.

The notion of preemption threshold was introduced by Express Logic, Inc. in their ThreadX real-time operating system. It allows a task to disable preemption from tasks up to a specified threshold priority. Tasks having priorities higher than the threshold are still allowed to preempt. In our new model, each task has a regular priority and a preemption threshold. We show how the new scheduling model can be analyzed for response times. We also develop algorithms for optimal assignment of priorities and preemption threshold.

In this paper we also provide evidence that this new scheduling model provides substantial quantitative benefits over both preemptive and non-preemptive scheduling models. We show that the new model can result in substantial improvement of schedulability by taking advantage of aspects of both preemptive and non-preemptive scheduling. Furthermore, we show that the new model provides lower run-time costs by eliminating unnecessary preemptions.

[*]This work was presented in part at RTCSA'99, Hongkong, December 1999 and a brief version of this work was published in the conference proceedings of RTCSA'99 by the IEEE Computer Society Press.

1 Introduction

Since the pioneering work of Liu and Layland [14], much work has been done in the area of real-time scheduling, and in the analysis techniques to a priori predict schedulability of task sets under a particular scheduling discipline. In particular, significant progress has been made in schedulability analysis of task sets under fixed priority preemptive scheduling (e.g., [8, 9, 11, 12, 18]). The benefits of fixed priority preemptive scheduling include relatively low run-time overheads (as compared to dynamic priority schemes, such as earliest deadline first) and ability to support tighter deadlines for urgent tasks (as compared to non-preemptive scheduling).

While preemptability is often necessary in real-time scheduling, it is fallacious to assume that it always results in higher schedulability. Indeed, it can be shown that, in the context of fixed priority scheduling, preemptive schedulers do not dominate non-preemptive schedulers, i.e., the schedulability of a task set under non-preemptive scheduling does not imply the schedulability of the task set under preemptive scheduling (and vice-versa). Moreover, preemptive schedulers have higher run-time overheads as compared to non-preemptive schedulers.

In this paper, we propose a generalized model of fixed priority scheduling that integrates and subsumes both preemptive and non-preemptive schedulers. The model uses the notion of *preemption threshold*, which was introduced by Express Logic, Inc. in their ThreadX real-time operating system to avoid unnecessary preemptions [10]. In our new scheduling model, each task has a preemption threshold, in addition to its priority. In essence, this results in a dual priority system. Each task has a regular priority, which is the priority at which it is queued when it is released. Once a task gets the CPU, its priority is raised to its preemption threshold. It keeps this priority, until the end of its execution. For recurring tasks, this process repeats each time the task is released.

The preemption threshold scheduling model can be used to get the best aspects of both preemptive and non-preemptive scheduling. By choosing the preemption threshold of a task that is higher than its priority, a task avoids getting preempted from any task that has a priority lower than its preemption threshold. By varying the preemption thresholds of tasks, the desired amount of non-preemptability may be achieved. In this way, the preemption threshold model may be viewed as introducing non-preemptability in a controllable manner. A suitable setting for the preemption thresholds can thus be used to get "just enough preemptability" needed to meet the real-time responsiveness requirements; thereby eliminating run-time overheads arising from unnecessary preemptability in the scheduling model.

It is also easy to see that both preemptive and non-preemptive scheduling are special cases of schedul-

Task	**Comp. Time** $\boldsymbol{C_i}$	**Period** $\boldsymbol{T_i}$	**Deadline** $\boldsymbol{D_i}$
τ_1	20	70	50
τ_2	20	80	80
τ_3	35	200	100

Table 1. An Example Task Set.

ing with preemption threshold. If the preemption threshold of each task is the same as its priority, then the model reduces to pure preemptive scheduling. On the other hand, if the preemption threshold of each task is set to the highest priority in the system, then no preemptions are possible, leading to fixed priority non-preemptive scheduling.

1.1 A Motivating Example

Before we delve into a theoretical treatment of the new scheduling model, it is instructive to take a look at a simple example that shows how schedulability can be improved with this new scheduling model. We consider a task set with 3 independent periodic tasks, as shown in Table 1. Each task is characterized by a period (T_i), a deadline (D_i) and a computation time (C_i).

The scheduling attributes for each task include its priority (π_i) and its preemption threshold (γ_i). Assuming fixed-priority scheduling, the optimal priority ordering for this task set is deadline monotonic ordering with both preemptive scheduling [13] and non-preemptive scheduling [5]. Under this priority ordering[1], the worst-case response times for the tasks are shown in Table 2. We can see that τ_3 misses its deadline under preemptive scheduling, while τ_1 misses its deadline under non-preemptive scheduling. Since the priority ordering is optimal, this implies that the task set is not schedulable under either fixed-priority preemptive scheduling, or fixed-priority non-preemptive scheduling.[2]

When we use preemption threshold, we can make the task set schedulable by setting the preemption threshold for τ_2 as 3, and τ_1 as 2. By setting the preemption threshold of τ_3 to 2, we allow it to be preempted by τ_1, but not by τ_2. This effectively improves the response time of τ_1 (as compared to the non-preemptive case) since it can no longer be blocked by τ_3. At the same time, it improves the response time of τ_3 (as

[1]Throughout this paper, we use higher numbers to denote higher priorities.

[2]We also note that a slight modification to this example will show that the feasibility under preemptive scheduler does not imply feasibility under non-preemptive scheduler (by changing the deadline of τ_3 to 120), and vice versa (by changing the deadline of τ_1 to be 60).

Task	π_i	**WCRT** Preemptive $\gamma_i = \pi_i$	**WCRT** Non-Preemptive $\gamma_i = 3$	γ_i	**WCRT** Preemption-Threshold
τ_1	3	20	**55**	3	40
τ_2	2	40	75	3	75
τ_3	1	**115**	75	2	95

Table 2. Response Times for Tasks under Different Schedulers.

compared to the preemptive case) since it cannot be preempted by τ_2 once τ_3 has started running. The resultant response times are also shown in Table 2.

Figure 1 illustrates the run-time behavior of the system with preemption threshold, and why it helps in increasing schedulability. In the figure, the arrows indicate arrival of task instances. The figure shows the scheduling of tasks starting from the critical instant of τ_3, which occurs when instances of all tasks arrive simultaneously (time 0). The scenario leads to the worst-case response time for τ_3. We can see that at time 70, a new instance of τ_1 arrives. Since the priority of τ_1 is higher than the preemption threshold of τ_3, τ_3 is preempted. At time 80, a new instance of τ_2 arrives. It can not preempt the execution of τ_1. However, at time 90, when τ_1 finishes, a pure preemptive scheduler would have run τ_2, delaying τ_3. In contrast, by setting the preemption threshold of τ_3 to 2, we have τ_3 scheduled at time 90 under our scheduling model. This effectively improves the worst-case response time of τ_3, making it schedulable. Note however that this also adds blocking time to τ_2 (as compared to the preemptive case), which increases its worst-case response time, but does not affect its schedulability in this example.

The use of preemption thresholds also reduces the run-time overheads associated with preemptions and the associated context-switches. This is due to the introduction of some non-preemptability into the scheduling model. We simulated the execution of tasks in this example for one LCM length (i.e., 2800 time units). When all tasks were released simultaneously at time 0, we find that preemptive scheduling results in 17 preemptions, while with preemption thresholds we get 8 preemptions. If we stagger the release times, such that τ_3 is released at time 0, τ_2 at time 1, and τ_1 at time 2, then the number of preemptions with preemptive scheduling is 30, while using preemption thresholds reduces it to 10.

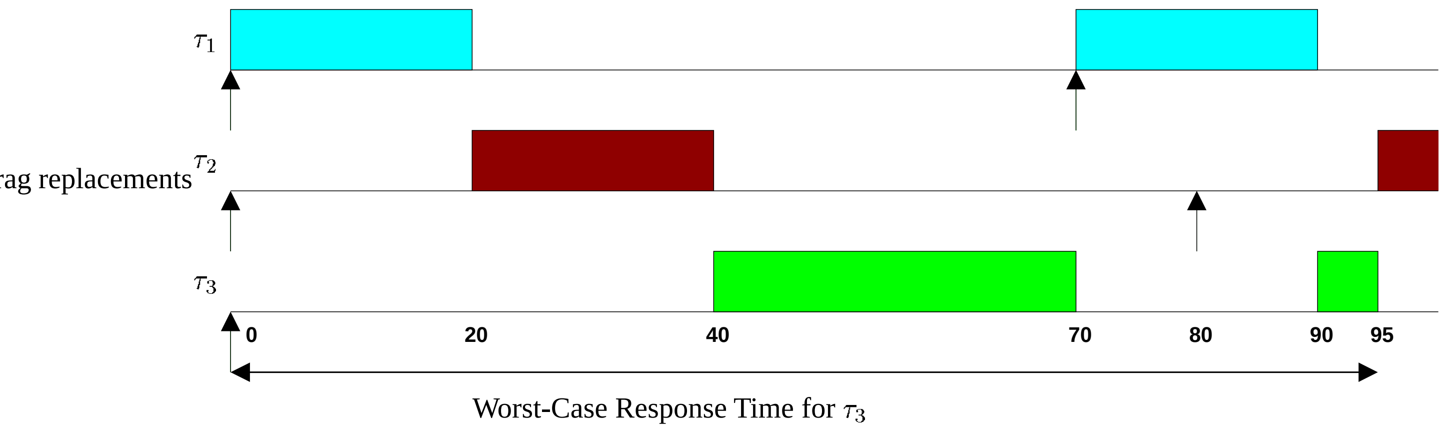


Figure 1. Run-time Behavior with Preemption Threshold

1.2 Contributions

The main contribution of this paper is in developing the proposed scheduling model using preemption threshold, showing how it improves over pure preemptive and non-preemptive scheduling, and systematically allocating feasible scheduling attributes (priority and preemption threshold) to task sets. We begin with developing the response-time analysis for the new model based on well-know technique using critical instants and busy periods [8, 9, 11, 12, 18]. We then develop an algorithm for assignment of task priorities and preemption thresholds. Our solution comes in two steps. First, given an assignment of priorities, we develop an efficient algorithm that computes an optimal set of preemption thresholds using a search space of $O(n^2)$ (instead of $O(n!)$ for an exhaustive search). The algorithm is optimal in the usual sense of finding a feasible assignment, i.e., one that makes the task set schedulable, if one exists. Using this algorithm as a "sub-routine" we then propose an optimal search algorithm to find an optimal assignment of both priorities and preemption thresholds by extending Audsley's optimal priority assignment algorithm [1, 18]. We also present an efficient (but non-optimal) greedy heuristic algorithm that is used in our simulations.

The proposed scheduling model provides some interesting features such as improved schedulability (compared to both preemptive and non-preemptive schedulers) and reduced preemption overhead (compared to preemptive schedulers). We quantify the benefits of the proposed scheduling model through simulations over randomly generated task sets. To quantify the improvement in schedulability, we use breakdown utilization [11] as a measure of schedulability. Our simulations show that depending on the task set characteristics, using preemption threshold can increase the schedulability by as much as $15-20\%$ of processor utilization as compared to preemptive scheduling, and even more as compared to non-preemptive scheduling. We also present simulation results that quantify the reduction in preemptions. For this purpose, we develop another algorithm that given a feasible assignment of priorities and preemption thresholds (as generated, for instance, by our greedy algorithm) attempts to assign larger preemption thresholds while still keeping the task set schedulable. Using this new assignment, we then simulate the execution of a task set and measure the number of preemptions, and compare that with pure preemptive scheduling. Our results show that, again depending on the task set characteristics, we can get a significant reduction (up to 30%) in the number of preemptions (averaged over multiple task sets with the same characteristics).

1.3 Paper Organization

The remainder of this paper is organized as follows. Section 2 describes the proposed scheduling model, and the problems that are being addressed in this paper. In Section 3, we develop schedulability analysis equation for our new model. Section 4 focuses on developing an optimal algorithm for assignment of priorities and preemption thresholds for a given task set. In Section 5, we quantify the improvement in schedulability using the preemption threshold scheduling model. Then, in Section 6, we show how to systematically reduce the number of preemptions. Finally, we present some concluding remarks.

2 Problem Description and Solution Overview

The example presented in Section 1.1 showed that by using preemption thresholds we can potentially increase schedulability and at the same time reduce run-time overheads. This raises the question of how to make effective use of the new scheduling model to get these benefits. In this section, we first present the task and run-time model assumed in this paper. Then, we give a statement of the problems being addressed in this paper, followed by an overview of our solution approach.

2.1 Task and Scheduling Model

We consider a set of n independent periodic or sporadic tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i is characterized by a 3-tuple $\langle C_i, T_i, D_i \rangle$, where C_i is its computation time, T_i is its period (or minimum inter-arrival time), and D_i is its relative deadline. We assume that (a) the tasks are independent (i.e., there is no blocking due to shared resources), (b) tasks do not suspend themselves, other than at the end of their computation, and (c) the overheads due to context switching, etc., are negligible (i.e., assumed to be zero).

Each task will also be given a priority $\pi_i \in [1, n]$ and a preemption threshold $\gamma_i \in [\pi_i, n]$. These values are assigned off-line and remain constant at run-time. We assume that larger numbers denote higher priority, and that no two tasks have the same priority. We assume that the run-time dispatching is preemptive, priority based, where tasks are dispatched from a notional ready queue sorted by priorities. When a task is released it is inserted into the notional ready queue at its priority π_i. When the task is dispatched, its priority is effectively raised to its preemption threshold γ_i, and it keeps this priority until it finishes execution. In other words, it can only be preempted by another task τ_j if $\pi_j > \gamma_i$. Note that since tasks are recurring, they will go through these dual priorities for each release of a task.

2.2 Problem Statement

In the rest of the paper our objective is to show how to effectively use the preemption threshold scheduling model to realize its potential benefits. Accordingly, we address the following inter-related problems.

2.2.1 Schedulability Analysis

First, we consider the problem of assessing schedulability of a task set under the preemption threshold scheduling model, assuming that the scheduling attributes are already known. For this, we develop the worst-case response time analysis for a task set by extending the well-known critical instant/busy-period analysis used in fixed-priority scheduling. Let Π be a priority assignment, and Γ be a preemption threshold assignment. Let $\mathcal{R}_i(\Pi, \Gamma)$ denote the worst-case response time of a task τ_i under the given assignments. Then, the schedulability of a task set is given by the following boolean predicate:

$$sched(\mathcal{T}, \Pi, \Gamma) \overset{\text{def}}{=} (\forall i :: 1 \leq i \leq n)\ \mathcal{R}_i(\Pi, \Gamma) \leq D_i \qquad (1)$$

2.2.2 Attribute Assignment

We then consider the problem of assigning scheduling attributes, i.e., priority and preemption thresholds, to tasks. The primary goal of finding the scheduling attributes is to determine if there exists a set of scheduling attributes that will make the task set schedulable. That is, we wish to determine whether the following predicate is true:

$$(\exists \Pi)(\exists \Gamma) \ :: \ sched(\mathcal{T}, \Pi, \Gamma) \qquad (2)$$

If it is determined that a task set is schedulable, then we additionally want to find a set of scheduling attributes that will minimize run-time overheads by eliminating any unnecessary preemptions.

2.2.3 Quantitative Assessment

Since the preemption threshold scheduling model generalizes both preemptive and non-preemptive scheduling models, it will do no worse than either of them in terms of schedulability. However, we still need to determine whether there are any significant gains in schedulability if the preemption threshold scheduling model is employed. To this end, we wish to make a quantitative assessment of the improvement in schedulability through the use of preemption thresholds.

Likewise, we are interested in quantitative assessment of run-time overheads for feasible task sets. Unlike schedulability, a non-preemptive scheduler will result in lower run-time overheads. Moreover, since non-preemptive scheduling is a special case, our scheduling model will incur the same overhead as a non-preemptive scheduler for task sets that are schedulable with non-preemptive scheduling. Therefore, our interest is in assessing run-time overheads due to preemptions and compare them with a preemptive scheduler.

2.3 Solution Overview

Our solution consists of the following parts:

(1) First, we show how given both the priorities and the preemption thresholds, we can find the worst-case response times for the tasks, and hence determine feasibility of a particular priority and threshold assignment.

(2) We then consider the problem of determining a feasible preemption threshold assignment with pre-defined priorities. We develop an optimal and efficient algorithm that has a search space of $O(n^2)$ (instead of $O(n!)$ for brute force search) to solve this problem.

(3) We use the optimal preemption threshold assignment algorithm to develop a branch and bound search algorithm to find a feasible set of scheduling attributes. Since the search algorithm may result in an exponential search space, we also develop an efficient but non-optimal greedy heuristic algorithm.

(4) We use the greedy algorithm for quantifying the improvement in schedulability (as compared to both preemptive and non-preemptive scheduling models) using randomly generated task sets. This ensures that the off-line costs of using different scheduling models are approximately similar resulting in fair comparisons.

(5) To eliminate any unnecessary preemptions, we present an algorithm that takes a feasible priority and preemption threshold assignment as input and refines it (by increasing preemption thresholds

of tasks) to eliminate unnecessary preemption while maintaining feasibility. Again, we quantify the improvements using simulations over randomly generated task sets.

3 Schedulability Analysis

We begin with schedulability analysis of task sets when the priority and the preemption threshold of each task are already known. The schedulability analysis is based on computation of the worst-case response time of each task. If the worst-case response time of a task is no more than its deadline, then the task is schedulable. If all the tasks in a system are schedulable, then the system is called schedulable. The response time analysis employed in this paper is an extension of the well-known level-i busy period analysis [8, 9, 11, 12, 18], in which the response time is calculated by determining the length of busy period, starting from a critical instant. The busy period at level-i is defined as a continuous interval of time during which the processor is continuously executing tasks of priority i or higher.

3.1 Review of Level-i Busy Period Analysis

To calculate the worst-case response time of a task, the busy period analysis essentially simulates the effect of scheduling under a worst-case scenario for the task. The busy period for task τ_i is constructed by starting from a *critical instant (time 0)*. The critical instant occurs when (1) an instance of each higher priority task comes at the same time (time 0), and (2) the task that contributes the maximum blocking time $B(\tau_i)$ has just started executing prior to time 0. Furthermore, to get the worst-case response time, all tasks are assumed to arrive at their maximum rate.

3.1.1 Preemptive Scheduling

For our task model described earlier, with arbitrary task deadlines, the busy period of task τ_i in traditional fixed-priority, preemptive scheduling can be iteratively computed by using the following equation [18]:

$$w_i(q) = q \cdot C_i + \sum_{\forall j, \pi_j > \pi_i} \left\lceil \frac{w_i(q)}{T_j} \right\rceil \cdot C_j \tag{3}$$

In this equation, $w_i(q)$ denotes the length of a busy period for task τ_i, where q instances of τ_i are included in the busy period. The length of the busy period for task τ_i is given by:

$$W_i = \min_{q \in \{1,2,3,\ldots\}} w_i(q) :: w_i(q) \leq q \cdot T_i \tag{4}$$

where $w_i(q)$ is the smallest value of $w_i(q)$ that satisfies Equation 3. Essentially the equation states that during the busy period for τ_i, all instances of τ_i and higher priority tasks that arrive within the busy period must also be executed within the busy period. The busy period length is computed by iteratively computing $w_i(q)$ for $q = 1, 2, 3, \ldots$ using Equation 3 until we reach a q, $q = m$ such that $w_i(m) \leq m \cdot T_i$

The worst-case response time of τ_i will be the longest response time of all instances that arrive and finish in the busy period. Let us denote $\mathcal{F}_i(q)$ (q-th finish time) as the smallest value of $w_i(q)$ that satisfies equation 3. Since the q-th instance of τ_i arrives at $(q-1) \cdot T_i$, the worst-case response time for τ_i is given by:

$$\mathcal{R}_i = \max_{q \in [1,\ldots,m]} (\mathcal{F}_i(q) - (q-1) \cdot T_i) \tag{5}$$

3.1.2 Non-Preemptive Scheduling

Even though the analysis described above was done for preemptive scheduling, the same technique, with minor modifications, can be used for non-preemptive scheduling as well [5]. However, there are two important differences. First, a task may be blocked by any lower priority task, if the lower priority task started executing just prior to the critical instant. Second, once a task gets the CPU, it cannot be preempted by any higher priority task until it finishes execution.

The blocking time from lower priority tasks is easy to incorporate. A task may be blocked by only one lower priority task. In the worst case, this task would have started executing just prior to the critical instant. Thus, the worst-case blocking time for task τ_i, denoted as B_i is given by:

$$B_i = \max_{\forall j, \pi_j < \pi_i} C_j \tag{6}$$

Equation 3 is modified for non-preemptive scheduling as:

$$w_i(q) = B_i + q \cdot C_i + \sum_{\forall j, \pi_j > \pi_i} \left(1 + \left\lfloor \frac{w_i(q) - C_i}{T_j} \right\rfloor \right) \cdot C_j \tag{7}$$

Note that the term for interference from higher priority tasks is modified to include task arrivals up to (and including) time $w_i(q) - C_i$, i.e., when the q-th instance task τ_i starts executing.

3.2 Response Time Analysis with Preemption Thresholds

Our response time analysis with preemption thresholds follows the same principles of the analysis presented in previous section. As in the non-preemptive case, we need to worry about blocking time from lower priority tasks. Also, as in preemptive scheduling, higher priority tasks may cause interference even after a task starts executing. However, the interference is different from pure preemptive scheduling, since the task's effective priority changes to its preemption threshold once it starts executing and only tasks with higher priority than this effective priority may cause interference. Therefore, we explicitly define and compute both q-th start time for a task τ_i (denoted by $\mathcal{S}_i(q)$) and the q-th finish time (denoted by $\mathcal{F}_i(q)$). After we get the q-th finish time by the modified busy period analysis, the worst-case response time is calculated as before, following Equation 5.

3.2.1 Computing Blocking Time

We begin with determining the blocking time for a task. The blocking is caused due to preemption thresholds, when a lower priority task has a preemption threshold higher than the priority of the task under consideration. Then, if the lower priority task is already running, it cannot be preempted (due to higher preemption threshold) by the task under consideration, leading to blocking time.

Definition 3.1 (Blocking Range) *The blocking range of a task τ_i is defined as the range of priorities given by $[\pi_i, \gamma_i]$.*

Definition 3.2 (Active Task) *A task (instance) is called active if it has started running, but is not finished yet.*

Proposition 3.1 *There is no overlapping of blocking ranges between the set of active tasks at any instant of time.*

Proof: By contradiction. Suppose τ_i and τ_j are active at some time, and their blocking ranges overlap. Without loss of generality assume that τ_i started execution first. Then, for τ_j to start running before τ_i finishes, it must be the case that $\gamma_i < \pi_j$. That is $\pi_i \leq \gamma_i < \pi_j \leq \gamma_j$. Thus, the blocking ranges do not overlap. □

Proposition 3.2 *A task τ_i can be blocked by at most one lower priority task τ_j. Furthermore, it must be the case that $\gamma_j \geq \pi_i$.*

Proof: A new arriving task instance of τ_i can not preempt a lower priority active task τ_j only if π_i falls in the blocking range of τ_j, i.e., $\pi_j < \pi_i \leq \gamma_j$. From Proposition 3.1, we know that blocking ranges of active tasks will not overlap. Therefore, τ_i will only fall into at most one of these blocking ranges, i.e., be blocked by the owner of that blocking range. Furthermore, it is easy to see that any lower priority tasks that have not started execution before the arrival of τ_i will not block τ_i. □

Proposition 3.2 shows that in computing the blocking time for a task τ_i, we need to consider blocking from only one lower priority task τ_j such that $\gamma_j \geq \pi_i$. Furthermore, this is true for the entire busy period of a task τ_i since during the busy period a lower priority task cannot start execution (due to the pending instances of τ_i). Therefore, the maximum blocking time of a task τ_i, denoted by $B(\tau_i)$, is given by:

$$B(\tau_i) = \max_{\forall j, \gamma_j \geq \pi_i > \pi_j} C_j \tag{8}$$

3.2.2 Computing q-th Start Time:

Before a task τ_i starts execution, there is blocking from lower priority tasks and interference from higher priority tasks. Among all lower priority tasks, only one lower priority task can cause blocking as we showed in Proposition 3.2. This task must have arrived and begun executing before the busy period starts. In the worst case, it just starts executing before time 0. All higher priority tasks that come before the start time $\mathcal{S}_i(q)$ and any earlier instances of task τ_i before instance q should be finished before the q-th start time. Therefore, $\mathcal{S}_i(q)$ can be computed iteratively using the following equation.

$$\mathcal{S}_i(q) = B(\tau_i) + (q-1) \cdot C_i + \sum_{\forall j, \pi_j > \pi_i} \left(1 + \left\lfloor \frac{\mathcal{S}_i(q)}{T_j} \right\rfloor \right) \cdot C_j \tag{9}$$

3.2.3 Computing q-th Finish Time:

Once the q-th instance of task starts execution, we have to consider the interference to compute its finish time. From the definition of preemption threshold, we know that only tasks with higher priority than the preemption threshold of τ_i can preempt and get the CPU before τ_i finishes. Furthermore, we only need to consider new arrivals of these tasks, i.e., arrivals after $\mathcal{S}_i(q)$. Based on this, we get the following equation for computing $\mathcal{F}_i(q)$:

$$\mathcal{F}_i(q) = \ \mathcal{S}_i(q) \ + \ C_i + \sum_{\forall j, \pi_j > \gamma_i} \left(\left\lceil \frac{\mathcal{F}_i(q)}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{\mathcal{S}_i(q)}{T_j} \right\rfloor \right) \right) \cdot C_j \tag{10}$$

4 Feasible Assignment of Priority and Preemption Threshold

In the previous section, we showed how to compute the worst-case response times when priorities and preemption thresholds of tasks are known. In this section, we develop an algorithm to systematically assign priorities and preemption thresholds to tasks, such that the assignment ensures schedulability. We first present some theoretical results about our model. Then, we present an algorithm motivated by these results for optimal preemption threshold assignment to task sets with predefined priority. After that, we propose a systematic approach to find the optimal priority and preemption threshold assignment using this algorithm as a sub-routine. Finally, we modify the optimal algorithm and present an efficient, but approximate, greedy algorithm. The greedy algorithm is used in our simulations to enable a fair comparison with preemptive and non-preemptive scheduling strategies.

4.1 Properties of the Model

With the response time analysis we have done above, we notice that this generalized fixed-priority scheduling model has some interesting properties. Assuming that the priorities of tasks are fixed, these properties help us reason about the effect of changing preemption thresholds of tasks. Furthermore, these properties will help us to reduce the search space for finding the optimal preemption threshold assignment. Therefore, we will use these properties as guidelines while designing an algorithm for preemption threshold assignment.

Lemma 4.1 *Changing the preemption threshold of a task τ_i from γ_1 to γ_2 may only affect the worst-case response time of task τ_i and those tasks whose priority is between γ_1 and γ_2.*

This can be seen by examining the equations developed for calculation of response times. The preemption threshold of a task τ_i determines which (higher priority) tasks may be blocked by τ_i. These tasks are those whose priorities fall in the range $[\pi_i, \gamma_i]$. Therefore, the response time of all these tasks may be affected when τ_i's preemption threshold is modified. It also may affect its own response time since it changes the set of tasks that can preempt it once it has started running. Note that, the preemption threshold of τ_i doesn't affect the interference from τ_i on any lower priority task τ_j (which depends on τ_j's threshold, and τ_i's priority). Therefore, changing the preemption threshold doesn't affect any lower priority task. A useful proposition directly follows from this lemma, and is presented below.

Proposition 4.1 *The worst-case response time of task τ_i will not be affected by the preemption threshold assignment of any task τ_j with $\pi_j > \pi_i$.*

Proposition 4.1 is useful in developing a strategy for optimal assignment of preemption thresholds. It shows that the schedulability of a task is independent of the preemption threshold setting of any task with higher priority. Therefore, this suggests that the threshold assignment should start from the lowest priority task to highest priority task.

Furthermore, Theorem 4.1, presented below, helps us determine the optimal preemption threshold assignment for a specific task. The preemption threshold of a task can range from its own priority to the highest priority in the task set. From the equations for worst-case response time analysis, we can see that a task may reduce its worst-case response time by increasing its preemption threshold, which restricts the set of (higher priority) tasks that can preempt it. However, this is done at the cost of a possible increase in the blocking time of higher priority tasks which may lead to increased worst-case response time of the higher priority task. Therefore, if there is a set of preemption threshold values that can make a task schedulable, choosing the minimum of them will maximize the chances of finding a feasible preemption threshold assignment.

Theorem 4.1 *Consider a set of n tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$, and a set of scheduling attributes $\Pi = \langle \pi_1, \ldots, \pi_n \rangle$ and $\Gamma = \langle \gamma_1, \ldots, \gamma_n \rangle$, such that the task set is schedulable with Π and Γ (i.e., $sched(\mathcal{T}, \Pi, \Gamma)$ is true). Then, if changing only the preemption threshold of τ_j from γ_j to γ'_j ($\gamma'_j < \gamma_j$), can still make τ_j schedulable, then the whole system is also schedulable by setting γ'_j as the preemption threshold of τ_j. That is,*

$$\mathcal{R}_j(\Pi, \Gamma(\gamma_j/\gamma'_j)) \leq \mathcal{D}_j \Rightarrow sched(\mathcal{T}, \Pi, \Gamma(\gamma_j/\gamma'_j))$$

Proof: When the preemption threshold of τ_j changes from γ_j to γ'_j ($\gamma_j > \gamma'_j$), the worst-case response time of any task τ_k with $\pi_k < \pi_j$ or $\pi_k > \gamma_j$ will not change. The worst-case response time of a task τ_k such that $\gamma'_j \geq \pi_k > \pi_j$ will also stay the same. Furthermore, any task τ_k with priority $\gamma'_j < \pi_j \leq \gamma_j$ will have no worse worst-case response time with γ'_j than with γ_j. Moreover, we already know that τ_j is schedulable with γ'_j. Therefore, if the whole system is schedulable with γ_j, it is also schedulable with γ'_j. □

A given task set may be unschedulable with any preemption threshold assignment. The following theorem gives a sufficient condition to claim a task set to be unschedulable.

Theorem 4.2 *For any given priority assignment if there exists a task τ_i, such that setting the preemption threshold of a task τ_i equal to the highest priority in the system can not make a specific task τ_i schedulable, then the task set is unschedulable.*

Proof: Comparing Equation 10 with Equation 3, we can see that preemption threshold reduces the worst-case response time of a task τ_i by preventing the interference from some higher priority tasks after τ_i starts execution. By setting the preemption threshold of τ_i to the highest priority in the system gives the maximum reduction to the worst-case response time of τ_i. If τ_i is still not schedulable, since the priority is predefined, there is no way to make it schedulable. Furthermore, the task set will also be unschedulable. □

Algorithm: AssignThresholds

```
        // Assumes that task priorities are already known
(1)  for ($i$ := 1 to $n$)
(2)      $\gamma_i = \pi_i$ // start from lowest value
        // Find worst-case response time based on the tentative assignment
(3)      $\mathcal{R}_i$ = WCRT($\tau_i$,$\gamma_i$) ;
(4)      while ($\mathcal{R}_i > D_i$) do    // while not schedulable
(5)          $\gamma_i$++ ; // increase preemption threshold
(6)          if $\gamma_i > n$ then
(7)              return FAIL; // system not schedulable.
(8)          endif
(9)          $\mathcal{R}_i$ = WCRT($\tau_i$,$\gamma_i$) ;
(10)     end
(11) end
(12) return SUCCESS
```

Figure 2. Algorithm for Preemption Threshold Assignment

4.2 Optimal Preemption Threshold Assignment with Given Priorities

Based on the results developed in the previous section, we have developed an algorithm that finds an optimal preemption threshold assignment assuming that the priorities are known and fixed. The algorithm is optimal in the sense that if there exists a feasible preemption threshold assignment then the algorithm will also find a feasible assignment.

Figure 2 gives the pseudo code of the preemption threshold assignment algorithm. The algorithm assumes that the tasks are numbered $1, 2, \ldots, n$, and that $\pi_i = i$. The algorithm considers the preemption threshold assignment of one task at a time starting from the lowest priority task. For each task considered, it finds the lowest preemption threshold assignment that will make the task schedulable. This is done by computing the response time of the task using the function **WCRT(task, threshold)**, and comparing it with its deadline. Note that, the response time calculation is possible even with a partial assignment since we consider tasks from low priority to high priority.

It is easy to see the worst-case search space for this algorithm is $O(n^2)$. The algorithm is optimal in the sense that if there exists a preemption threshold assignment that can make the system schedulable, our algorithm will always find an assignment that ensures the schedulability. The optimality of the algorithm is given in the following Theorem 4.3.

Theorem 4.3 *Given a fixed priority assignment, the algorithm* **AssignThresholds** *will find a feasible preemption threshold assignment, if there exists one.*

Proof: Assume a set of n tasks $\mathcal{T} = \{\tau_i \mid 1 \leq i \leq n\}$, with a priority assignment $\Pi = \langle \pi_i, \ldots, \pi_n \rangle$. Without loss of generality, assume that the tasks have been labeled such that $\pi_i = i$. Furthermore, let the task set be schedulable with a set of preemption threshold values $\Gamma = \{\gamma_i \mid (1 \leq i \leq n)\}$.

The algorithm assigns preemption thresholds to the tasks starting from τ_1 and going upto τ_n. Let the preemption thresholds found by the algorithm be labeled γ_1', γ_2', etc. Assume that task τ_i is the first task such that the preemption threshold found by the algorithm is different from the given feasible assignment, that is, $\gamma_i' \neq \gamma_i$. Then, it must be the case that $\gamma_i' < \gamma_i$, otherwise, our algorithm will find γ_i rather than γ_i'. Based on Theorem 4.1, we know that the task set will still be schedulable if we use γ_i' to replace γ_i in the above feasible preemption threshold assignment.

By repeatedly using the above argument, we can see that the algorithm will also find a feasible preemption threshold assignment $\{\gamma_i' \mid \gamma_i' \leq \gamma_i, 1 \leq i \leq n\}$. $\square$

4.3 Optimal Assignment of Priorities and Preemption Thresholds

In this section, we address the general problem of determining an optimal (i.e., one that ensures schedulability) priority and preemption threshold assignment for a given task set. We give a branch-and-bound search algorithm that searches for the optimal assignment. Whether more efficient algorithms can be found for this problem, remains an open question at this time. Our algorithm borrows the basic ideas from the optimal priority assignment algorithm presented in [1, 18] for preemptive priority scheduling of a task set. Unfortunately, the introduction of preemption thresholds not only adds another dimension to the search space but also brings more branches into the search, making the search space exponential in size.

Our search algorithm, presented in Figure 3, proceeds by performing a heuristic guided search on "good" priority orderings, and then when a priority ordering is complete, it uses the algorithm presented in the previous section to find a feasible threshold assignment. If a feasible threshold assignment is found then we are done. If not, the algorithm backtracks to find another priority ordering.

The algorithm works by dividing the task set into two parts: a sorted part, consisting of the lower priority tasks, and an unsorted part, containing the remaining higher priority tasks. The priorities for the tasks in the sorted list are all assigned. The priorities for the tasks in the unsorted list are unassigned, but are

Algorithm: AssignSchedAttributes($\mathcal{T}$, π)

```
        /* Terminating Condition; assign preemption thresholds */
(1)     if ($\mathcal{T}$ == $\{\}$) then
        /* Use algorithm in Figure 2 for preemption threshold assignment */
(2)         return AssignThresholds()
(3)     endif

        /* Heuristically generate a priority assignment */

        /* Assign Heuristic Value to Each Task */
(4)     L := $\{\}$ ;
(5)     foreach $\tau_k \in \mathcal{T}$ do
(6)         $\pi_k$ := $\pi$;   $\gamma_k := n$;   $\mathcal{R}_k$ := WCRT($\tau_k$);
(7)         if $\mathcal{R}_k > D_k$ then Continue ; /* prune */
(8)         $\gamma_k := \pi_k$ ;   $\mathcal{R}_k$ := WCRT($\tau_k$);
(9)         if $\mathcal{R}_k \leq D_k$ then
(10)            $H_k :=$ GetBlockingLimit($\tau_k$);   /* positive value */
(11)        else
(12)            $H_k := \mathcal{D}_k - \mathcal{R}_k$;   /* negative value */
(13)        endif
(14)        L := L + $\tau_k$;
(15)        $\pi_k$ := $n$ ;   /* reset */
(16)    end

        /* Recursively perform depth first search */
(17)    while (L != $\{\}$) do
        /* Select the task with the largest heuristic value next */
(18)        $\tau_k :=$ GetNextCandidate(L) ;
(19)        $\pi_k := \pi$;
(20)        if AssignSchedAttributes($\mathcal{T} - \tau_k$, $\pi$+1) == SUCCESS then
(21)            return SUCCESS ;
(22)        endif
(23)        L := L - $\tau_k$;
(24)    end
(25)    return FAIL
```

Figure 3. Search Algorithm for Optimal Assignment of Priority and Preemption Threshold

all assumed to be higher than the highest priority in the sorted list. Initially, the sorted part is empty and all tasks are in the unsorted part. The algorithm recursively moves one task from the unsorted list to the sorted list, by choosing a candidate task based on heuristics, as described below. When all tasks are in the sorted list, a complete priority ordering has been generated, and the threshold assignment algorithm is called.

When considering the next candidate to move into the sorted list, all tasks in the unsorted list are examined in turn. To make the search more efficient, we select the "most promising" candidate first, using a heuristic function, described below. If the algorithm fails to find a solution with that partial assignment, it will backtrack and then select the next task. Additionally, we prune infeasible paths by not considering tasks that cannot be made schedulable at the current priority level.

Figure 3 gives the pseudo code of the search algorithm, which is presented as a recursive algorithm. It takes two parameters: $\mathcal{T}$, which is the unsorted part (containing all the tasks waiting for priority assignment), and π, which is the next priority to assign. The tasks that have been assigned priorities are kept separately (and not explicitly shown in the pseudo-code) for preemption threshold assignment at the end of the algorithm. The list of candidates to search is created in L. The computation of worst-case response times assumes that all tasks with unassigned priorities have the highest priority, and that all unassigned preemption thresholds are equal to the task priority.

Pruning Infeasible Paths. First, we tentatively assign a task the current priority and compute its response time with its preemption threshold set to the highest priority in the system. If its computed response time exceeds its deadline, then the task cannot be made schedulable at this priority level (Theorem 4.2). This is because, we assume at this stage that all other tasks have preemption thresholds equal to their priorities. Therefore, we prune such a branch to make the search more efficient.

Heuristic Function. To compute the heuristic function, we compute the response time for the task by tentatively assigning it the current priority and assuming that the preemption threshold equals its priority. Let $\mathcal{R}_k$ be the computed response for a task τ_k in this manner. Then, the heuristic function is given by:

$$H_k = \begin{cases} BL_k & \textbf{if } \mathcal{R}_k \le D_k \\ \mathcal{R}_k - D_k & \textbf{else} \end{cases} \tag{11}$$

where BL_k denotes the blocking limit for τ_k. The blocking limit represents the maximum blocking that the task can get while still meeting its response time. Note that at this stage since we have assumed that priorities equal preemption thresholds, there is no blocking. However, once the priorities are fully assigned, it is possible that in the preemption threshold assignment stage a lower priority task may be assigned a threshold that is higher than this task, and can cause blocking. The blocking limit captures the maximum blocking that a task can tolerate while still meeting its deadline. This can be computed by assigning a blocking term to the task, repeating the worst-case response time computation, and checking if it still meets the deadline.

The blocking limit is meaningful if $\mathcal{R}_k \leq D_k$. Otherwise, it is still possible that τ_k may be schedulable at this priority with an appropriate preemption threshold. Thus tasks that need a smaller reduction in interference from higher priority tasks are better candidates for selection. Accordingly, we assign a heuristic value of $D_k - \mathcal{R}_k$ for each task. Note that these values are negative, while BL_k is positive. Thus, such tasks have a lower heuristic value, which is as desired.

4.4 A Greedy Algorithm

The efficiency of the optimal algorithm shown above depends heavily on the characteristics of the task set. In the worst case, it has exponential search space in term of the number of tasks. Clearly, this algorithm becomes infeasible to use, even with a modest number of tasks. Therefore, we have developed a greedy-heuristic algorithm, which we use in our simulations for schedulability comparison. The basic idea of this greedy algorithm is the same as the optimal algorithm. The only difference lies in the branching part. The optimal algorithm will try all possible branches before it find a solution. However, the heuristic algorithm will only try the one that is most promising, or in another word, the one at the head of the candidate list.

This heuristic algorithm dominates the preemptive scheduling algorithm, i.e., if a task set is schedulable with preemptive scheduling, then the algorithm will be able to find a feasible assignment as well. This is not surprising since the algorithm extends Audsley's optimal algorithm for priority assignment. On the other hand, there are cases when the algorithm is not able to find a feasible assignment, when a non-preemptive priority assignment algorithm is able to find a feasible assignment. Since a non-preemptive priority assignment is also a feasible solution for our model, the algorithm can be trivially extended to use the non-preemptive priority assignment algorithm first, and then use this algorithm. Without actually doing so, we assume that this is the case, and this extended algorithm is used in our simulations. In this way, our extended algorithm dominates both preemptive and non-preemptive scheduling algorithms.

5 Schedulability Improvement

Since our scheduling model subsumes both preemptive and non-preemptive scheduling, any task set that is schedulable with either a pure preemptive or a non-preemptive scheduler is also schedulable in our model. In section 1.1, we gave a simple example task set, which is not schedulable under either preemptive or non-preemptive scheduling, but can be scheduled with our model. Nonetheless, it may be the case that any such schedulability improvement comes only in exceptional cases, and any such improvement may only be marginal. Accordingly, in this section we want to quantify the schedulability improvement that may be achieved by using this new model. Our strategy is to use randomly generated task sets and test for their schedulability under the different scheduling models.

To compare the schedulability of a task set under different scheduling policies, we need a quantitative measurement. One possibility is to use the binary measure of schedulability, i.e., either a task set is schedulable or not. A better measure, and one that is commonly used is the notion of *breakdown utilization* [11]. The breakdown utilization of a task set is defined as the associated utilization of a task set at which a deadline is first missed when the computation time of each task in the task set is scaled by a factor. In this paper we use breakdown utilizations to measure schedulability.

5.1 Simulation Design

We use randomly generated periodic task sets for our simulations. Each task is characterized by its computation time C_i and its period T_i. To keep the number of variables small, we assume that $D_i = T_i$ for all tasks. We vary two parameters in our simulations: (1) number of tasks $nTasks$, from 5 to 50; and (2) maximum period for tasks $maxPeriod$, from 10 to 1000. For any given pair of $nTasks$ and $maxPeriod$, we randomly generate 100 task sets.

A task set is generated by randomly selecting a period and computation time for each of the $nTasks$. First, a period T_i is assigned randomly in the range $[1, maxPeriod]$ with a uniform probability distribution function. Then, we assign a utilization U_i in the range $[0.05, 0.5]$, again with uniform probability distribution function. The computation time of the task is then assigned as $C_i = T_i * U_i$.

For each randomly generated task set, we measure the breakdown utilization for each of (1) pure preemptive scheduling, (2) non-preemptive scheduling, and (3) scheduling with preemption threshold. We do this by scaling the computation times to get different task set utilization, and then testing for schedulability. Since the randomly generated task sets may have utilization greater than 1 to begin with, we initially scale the utilization to 100%. We then use binary search to find the maximum utilization at which the task set is schedulable under a particular scheduling algorithm.

5.2 Attribute Assignment

Before we can do the response time analysis with a particular scheduling algorithm, we must derive the scheduling attributes. In the case of preemptive and non-preemptive scheduling algorithms, the only scheduling attribute is task priority, while for the preemption threshold scheduling, both task priority and preemption threshold must be derived. We derive optimal priority assignments for both preemptive and non-preemptive scheduling algorithms. For the preemptive case, the optimal priority assignment is simply the rate-monotonic assignment [14]. For the general case, with arbitrary deadlines, Audsley's optimal priority assignment algorithm [1] can be used. For non-preemptive scheduling, we use optimal priority assignment algorithm presented in [5] which is basically the same as Audsley's algorithm, but adapted for non-preemptive scheduling. For scheduling with preemption thresholds, we use the greedy-heuristic algorithm presented in last section as an efficient approximation to assign priorities and preemption thresholds. The time complexity of this algorithm is the same as Audsley's algorithm. This provides a fair ground for comparison between the algorithms.

5.3 Simulation Results

In this section, we describe the results for schedulability improvement (as measured by breakdown utilization) using preemption thresholds. As mentioned earlier, we controlled two parameters: (1) number of tasks ($nTasks$), and (2) maximum period ($maxPeriod$). We did the simulations for $nTasks \in \{5, 10, 15, 20, 25, 30, 35, 40, 50\}$ and $maxPeriod \in \{10, 20, 50, 100, 500, 1000\}$. For each task set, we measured the breakdown utilization for each of the three cases: pure preemptive scheduling, non-preemptive scheduling, and preemptive scheduling with preemption threshold.

In Figures 4 and 5 we show the schedulability improvement as the number of tasks is varied. The results are shown for $maxPeriod =$ 10 and 100; the results are similar for other values. In each case, we plot the average and maximum increase in breakdown utilization when using preemption thresholds. Figure 4 shows schedulability improvement as compared to pure preemptive priority scheduling. As the plot shows, when looking at average improvement, there is a modest improvement in schedulability (3%-6%), depending on the number of tasks. As the number of tasks increases, the improvement tends to decrease. Perhaps, more interesting is the plot for maximum increase, which shows that the schedulability improvement can be as high as 18% improvement in breakdown utilization for selected task sets, although once again the improvement decreases as the number of tasks is increased.

The results showing the schedulability improvement with non-preemptive scheduling are more varied. First, for most ranges of the parameters, the schedulability improvement is much more than the preemptive case (which also means that preemptive scheduling gives higher breakdown utilization, as compared to non-preemptive scheduling). The result should not be surprising since non-preemptive scheduling performs very badly even if one task has a tight deadline, and any other task has a large computation time. In such cases, the breakdown utilization can be arbitrarily low, as can be seen in Figure 5(b).

While non-preemptive scheduling performs poorly in general, there are selected cases when it performs better than preemptive scheduling, and better than our heuristic algorithm for preemption threshold. Note, however, that we assume that our heuristic algorithm is augmented with the schedulability check using non-preemptive scheduler as well, and so in the plots those cases simply show up as zero percent improvement. These results can be seen in Figure 5(a), where non-preemptive scheduling performs the best of all three when $nTasks \geq 20$; similar results are obtained for other smaller values of $maxPeriod$, but this effect goes away when $maxPeriod = 100$ or more. The reason for this is that with large number of tasks, and a small value of $maxPeriod$, the computation times for all tasks are small. This means that any blocking caused by non-preemption has little effect on schedulability, which gives rise to higher breakdown utilization.

6 Preemption Overheads

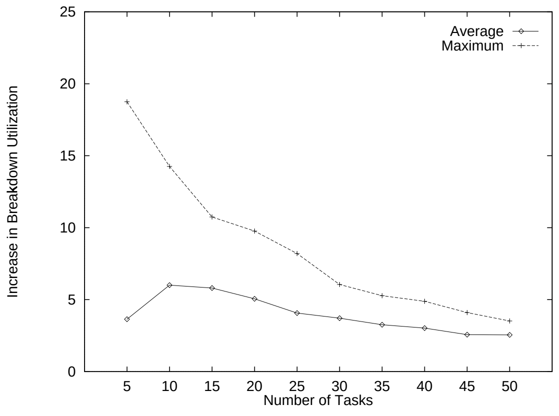
The application of our fixed priority scheduling model with preemption threshold not only provides better schedulability, but also may reduce the number of preemptions as compared to pure preemptive scheduling. Indeed, this was one of the primary motivations behind the development of preemption threshold in the ThreadX operating system [10]. Certainly, in the extreme case if a task set is schedulable using non-preemptive scheduling, then there are no preemptions. Even otherwise, we expect that the use of preemption thresholds should prevent some unnecessary preemptions.

One possible benefit brought by reduced preemptions is further improvement in schedulability when the scheduling overhead caused by preemptions is taken into account. Typically, such analysis associates two context switches per task, and that overhead is added to the computation time of a task [18]. Unfortunately, it does not seem that this can be reduced (in the worst-case) with the use of preemption thresholds. On the other hand, if we can show that there are reduced number of preemptions (on an average), then any such time savings implies that the processor is available for other background/soft-real-time jobs, which can only be beneficial.

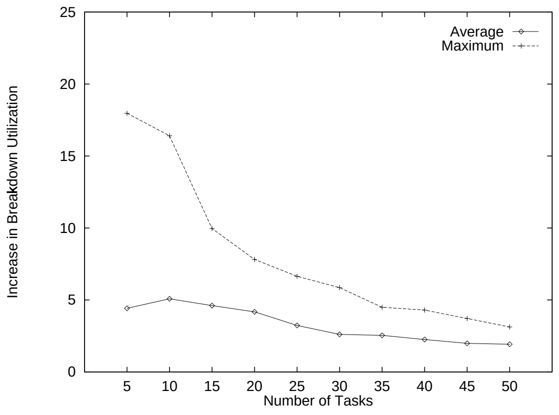
In this section, we first show how the number of preemptions in a given (schedulable) task set can be reduced in a systematic manner, while maintaining schedulability. We then use simulation results to see if there is any substantial reduction in the number of preemptions.

6.1 Preemption Threshold Assignment to Reduce Preemptions

Clearly, any feasible assignment of priorities and preemption thresholds will be no worse than pure preemptive scheduling. However, for a given task set (and assuming that priorities are given), there may



(a) Maximum Period = 10

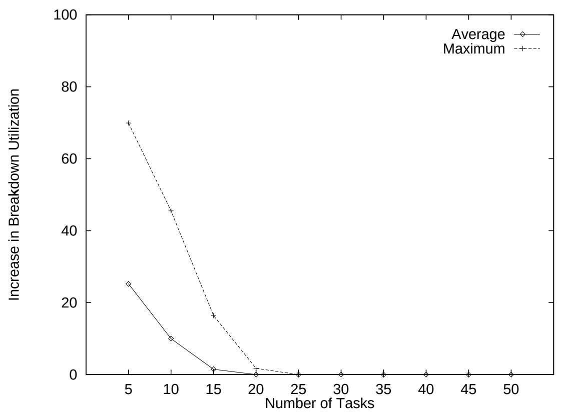


(a) Maximum Period = 100

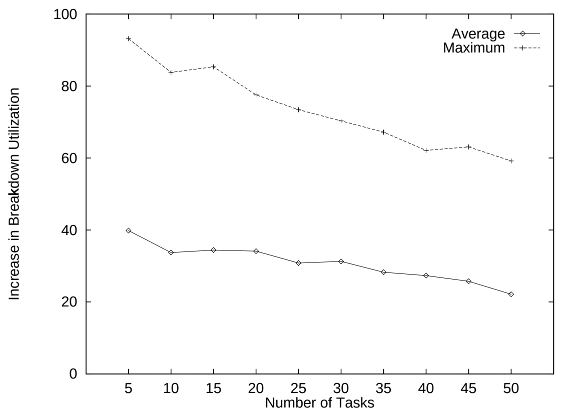
Figure 4. Schedulability Improvement with Preemption Threshold as compared to Preemptive Scheduling.

be several feasible threshold assignments, and some will reduce preemptions more than others. Earlier in this paper, we presented an optimal algorithm to assign preemption thresholds (Figure 2). The algorithm attempts to assign the smallest feasible preemption threshold values, which was necessary to ensure its optimality. However, it is possible that there are other feasible assignments with larger preemption thresholds that will incur fewer preemptions.

Our approach to reduce preemptions is to use the assignment generated by our optimal threshold assignment as a starting point. Then, we iterate over the solution and attempt to assign the largest feasible preemption threshold values to tasks such that the task set remains schedulable. Clearly, larger preemption



(a) Maximum Period = 10



(a) Maximum Period = 100

Figure 5. Schedulability Improvement with Preemption Threshold as compared to Non-Preemptive Scheduling.

threshold values reduce the chances of preemptions, and therefore, should result in less preemptions.

Figure 6 gives the algorithm that attempts to assign larger preemption threshold values to tasks. The algorithm considers one task at a time, starting from the highest priority task, and tries to assign it the largest threshold value that will maintain the feasibility of the system. We do this one step at a time, and check the response time of the affected task to ensure that the system stays schedulable. By going from highest priority task to the lowest priority task, we ensure that any change in the preemption threshold assignment in latter (lower priority) tasks cannot increase the assignment of a former (higher priority) task, and thus we only need to go through the list of tasks once.

Algorithm: Assign Maximum Preemption Thresholds

```
        /* Assumes that task priorities are fixed, and a */
        /* set of feasible preemption thresholds are assigned */
(1)     for (i := n down to 1)
(2)         while (schedulable == TRUE) && ($\gamma_i < n$)
(3)             $\gamma_i$ += 1;    /* try a larger value */
(4)             Let $\tau_j$ be the task such that $\pi_j = \gamma_i$.
                /* Calculate the worst-case response time of task j
                    and compare it with deadline */
(5)             $\mathcal{R}_j$ := WCRT($\tau_j$);
(6)             if ($\mathcal{R}_j > D_j$) then // Task j is not schedulable.
(7)                 schedulable := FALSE
(8)                 $\gamma_i$ -= 1;
(9)             endif
(10)        end
(11)        schedulable := TRUE
(12)    end
(13)    return
```

Figure 6. Algorithm for Finding Maximum Preemption Threshold

6.2 Simulation Design and Results

We use the same randomly generated task sets that we used in the previous section. We simulate the execution of a task set for 100000 time units, and track the number of preemptions. With $maxPeriod = 1000$, this gives at least 1000 instances of each task in a simulation run. We want to see the savings in preemptions when using preemption thresholds as compared to pure preemptive scheduling. Accordingly, we use percentage reduction in the number of preemptions as the metric, which is defined as:

$$\frac{NumPreemptions_p - NumPreemptions_{pt}}{NumPreemptions_{pt}} * 100$$

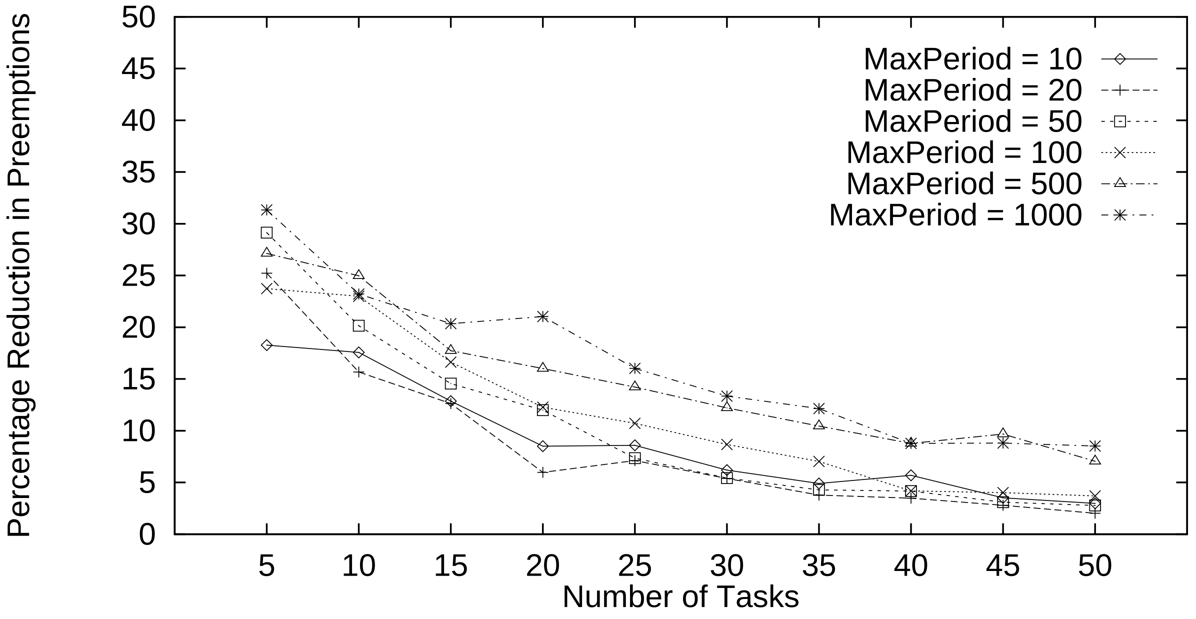


Figure 7. Average Percentage Reduction in Number of Preemptions for Preemption Threshold Scheduling as compared to Pure Preemptive Scheduling

where $NumPreemptions_p$ and $NumPreemptions_{pt}$ are the number of preemptions encountered in a particular simulation run with preemptive scheduling and preemption threshold scheduling respectively.

We did one simulation run for each task set generated in the simulations of Section 3. The computation times for the tasks were chosen by scaling them to the largest value at which the task set was schedulable under preemptive scheduling (i.e. the breakdown utilization). We assigned priorities to these task sets using the rate-monotonic (optimal) algorithm. We use the same priorities for the preemption threshold case, but additionally assigned preemption thresholds to the tasks as described above. That is, we first generated a feasible assignment of preemption thresholds. Then, we used the algorithm in Figure 6 to assign a new set of preemption thresholds. During the simulation, we randomly assigned the initial arrival time of each task τ_i in the range $[0, T_i]$.

We plot the average percentage reduction in the number of preemptions for preemption threshold scheduling as compared to preemptive scheduling. The results are shown in Figure 7. As can be seen in the figure, there is a significant reduction in preemptions for small number of tasks, but it tapers down to less than 5% as the number of tasks is increased. Also, for any given number of tasks, the number of reductions is larger for larger values of $maxPeriod$, i.e., when the period range is larger.

7 Related Work

Like many papers in real-time literature, our work has roots in real-time scheduling. In the last twenty years, scheduling theory has been widely studied to provide guidelines for developing hard real-time systems. Many results have been produced for non-idling scheduling over a single processor. These results fall into two categories: fixed-priority schedulers and dynamic priority schedulers. Alternatively, one can categorize the schedulers as preemptive or non-preemptive. While dynamic priority schedulers can achieve higher schedulability for task sets, fixed priority schedulers are commonly preferred due to lower overheads in implementation. Moreover, most contemporary real-time operating systems provide direct support for fixed priority preemptive scheduling, and almost none provide support for dynamic priority scheduling algorithms such as earliest deadline first (although, the priority can be changed from the application).

Schedulability analysis in fixed priority preemptive scheduling is often based on computing the worst-case response times and by comparing them with deadlines. The notion of *level-i busy period* was introduced by Lehoczky [12] and computation of the worst-case response time for general task sets is shown in [9, 8, 12, 18].

With fixed-priority preemptive scheduling, the deadline monotonic ordering has been shown to be optimal [13] for task sets with relative deadlines less than or equal to periods. A special case of the deadline monotonic ordering is the rate monotonic ordering when deadlines equal periods [14]. However, for a general task set, where deadlines are not related to the periods, Lehoczky [12] pointed out that deadline monotonic ordering is not optimal. Audsley [1, 18] gives an optimal priority assignment procedure with complexity of $O(n^2)$.

Deadline monotonic priority ordering is no longer optimal in the context of non-preemptive fixed-priority scheduling of general task set. However, it is optimal for a task set with deadlines less than or equal to periods if smaller deadline implies smaller or equal computation time [5]. Furthermore, the optimal priority assignment algorithm proposed by Audsley [1] is shown to be still valid [5]. The feasibility is still closely related with computation of worst-case response time. Moreover, the level-i busy period analysis is also valid in the context of non-preemptive scheduling [5].

The scheduling work that comes closest to ours is the scheduling of tasks with varying execution priorities [8, 4]. In [8], schedulability analysis for tasks with varying execution priorities is given, and in fact it is recognized that the schedulability of a task set can be improved by increasing a task's priority sometime during its execution. The same general idea is used in [4] with dual priority scheduling except that the task priority is raised after a fixed amount of (real) time. The notion of preemption threshold can be viewed as a special case of scheduling with varying execution priorities. However, our work differs from these in that we go beyond response time analysis, and focus on scheduling attribute assignment.

Our work on priority and preemption threshold assignment for tasks is related to the general problem of assigning feasible task attributes. Besides the priority assignment algorithms mentioned above, several researchers have addressed this problem in different contexts. In [6] feasible task attributes such as periods, deadlines, and offsets were derived from end-to-end timing requirements specified on system inputs and outputs. The work has been followed up by a number of other researchers, addressing different variations of feasible task attribute assignment problem [3, 15, 16, 7].

The notion of preemption threshold is used in various other contexts. Most notably, it is used to avoid unbounded priority inversions in priority ceiling protocols [17, 2] by raising the priority of a task while in a critical section.

8 Concluding Remarks

We present a generalized fixed-priority scheduling model with the notion of preemption threshold. This model bridges the gap between preemptive and non-preemptive scheduling and includes both of them as special cases. It captures the best of both models, and as a result it can feasibly schedule task sets which are not feasible with either pure preemptive scheduling, or non-preemptive scheduling. In this paper, we present simulation results over randomly generated task sets showing our model achieves substantial improvement of the schedulable utilization of task sets over both preemptive and non-preemptive cases. Furthermore, using the concept of level-i busy period, we derive the equation for computing the worst-case response time of periodic or sporadic tasks in this general model.

Using this analysis, we addressed the problem of finding an optimal priority ordering and preemption threshold assignment that ensures schedulability. Based on the theoretical results we developed under the model, we proposed an efficient algorithm for finding optimal preemption threshold assignment for a task set with predefined priority. Finally using this threshold assignment algorithm, we develop an optimal search algorithm to find the optimal priorities and preemption thresholds. A heuristic algorithm, which will dominate both preemptive and non-preemptive scheduling is also proposed to achieve better efficiency.

One interesting aspect of this new scheduling model is that it achieves better schedulability without increasing the overheads. This is in sharp contrast to dynamic priority schemes, such as earliest deadline first, which achieve higher schedulable utilization at the cost of significantly higher overheads as compared to fixed priority scheduling. The only additional cost of this new scheduling model is an extra field to be associated with tasks (for keeping their preemption threshold values). With a minor modification, the kernel can switch between the normal priority and the preemption threshold of a task, as dictated by the model. Contrast this with the dual-priority scheme of Burns [4], where the implementation costs are non-trivial. We also note that the model can be easily emulated at user level, when it is not supported by a real-time kernel. Of course, there is an additional cost in this case to change priorities.

Not only does this new scheduling model not add any additional overheads, it can reduce overheads by incorporating the best aspects of non-preemptive scheduling into the preemptive scheduling model. Thus, the model can be used to introduce only enough preemptability as is necessary to achieve feasibility. In this way, it can be used to reduce scheduling overheads by reducing the number of preemptions. Our simulation results support this conjecture, and show that in many cases there can be a marked reduction in preemptions, making the CPU bandwidth available for background and soft real-time priority tasks.

References

[1] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS 164, Department of Computer Science, University of York, England, December 1991.

[2] T. Baker. A stack-based resource allocation policy for realtime processes. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 191–200. IEEE Computer Society Press, December 1990.

[3] I. Bate and A. Burns. An approach to task attribute assignment for uniprocessor systems. In *Proceedings of the 11th Euromicro Conference on Real Time Systems*, 1999.

[4] A. Burns and A. J. Wellings. A practical method for increasing processor utilization. In *Proceedings, 5th Euromicro Workshop on Real-Time Systems*, June 1993.

[5] L. George, N. Rivierre, and M. Spuri. Preemptive and non-preemptive real-time uni-processor scheduling. Technical Report N^o 2966, INRIA, France, sep 1996.

[6] R. Gerber, S. Hong, and M. Saksena. Guaranteeing Real-Time Requirements with Resource-based Calibration of Periodic Processes. *IEEE Transactions on Software Engineering*, 21:579–592, July 1995.

[7] S. Goddard and K. Jeffay. Analyzing the real-time properties of a dataflow execution paradigm using a synthetic aperture radar application. In *Proceedings, IEEE Real-Time Technology and Applications Symposium*, June 1997.

[8] M. Harbour, M. Klein, and J. Lehoczky. Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority. In *Proceedings, IEEE Real-Time Systems Symposium*, pages 116–128, December 1991.

[9] M. Joseph and P. Pandya. Finding response times in a real-time system. *Computer Journal*, 29(5):390–395, 1986.

[10] William Lamie. Preemption-threshold. White Paper, Express Logic Inc. Available at http://www.threadx.com/preemption.html.

[11] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 166–171. IEEE Computer Society Press, December 1989.

[12] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 201–209. IEEE Computer Society Press, December 1990.

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

[14] C. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[15] M. Ryu, S. Hong, and M. Saksena. Streamlining Real-Time Controller Design - From Performance Specifications to End-to-End Timing Constraints. In *Proceedings IEEE Real-Time Technology and Applications Symposium*, June 1997.

[16] M. Saksena and S. Hong. Resource conscious design of distributed real-time systems: An end-to-end approach. In *Proceedings, IEEE International Conference on Engineering of Complex Computer Systems*, October 1996.

[17] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39:1175–1185, September 1990.

[18] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *The Journal of Real-Time Systems*, 6(2):133–152, March 1994.