Scheduling Fixed-Priority Tasks with Preemption Threshold

Yun Wang Manas Saksena*
Department of Computer Science
Concordia University
Montreal, Quebec H3G 1M8, Canada
{y_wang,manas}@cs.concordia.ca

Abstract

In the context of fixed-priority scheduling, feasibility of a task set with non-preemptive scheduling does not imply the feasibility with preemptive scheduling and vice versa. In this paper, we use the notion of preemption threshold, first introduced by Express Logic, Inc. in their ThreadX real-time operating system, to develop a scheduling model that subsumes both preemptive and non-preemptive fixed priority scheduling. Preemption threshold allows a task to only disable preemption of tasks up to a specified threshold priority. Tasks having priorities higher than the threshold are still allowed to preempt.

With this new scheduling model, we show that schedulability is improved as compared to both preemptive and non-preemptive scheduling models. We develop the equations for computing the worst-case response times, using the concept of level-i busy period. Some useful results about the generalized model are presented and an algorithm for optimal assignment of priority and preemption threshold is designed based on these results.

1. Introduction

Since the pioneering work of Liu and Layland [10], much work has been done in the area of real-time scheduling theory, and in particular fixed priority preemptive scheduling theory (e.g., [4, 5, 8, 7, 11]). A common wisdom prevails that preemptive schedulers give better schedulability than non-preemptive schedulers. However, 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).

In this paper, we consider a generalized model of fixed-priority scheduling that subsumes both preemptive and non-preemptive schedulers. The new model is based on the notion of *preemption threshold*, introduced by Express Logic, Inc. [6]. In this model, each task has a preemption threshold, in addition to its priority. Within the kernel, there is an integer, representing the current preemption priority level (which may be higher than the priority of the currently executing task). The preemption threshold prevents the preemption of a task from other tasks, unless the preempting task's priority is higher than the current preemption threshold.

In essence, preemption threshold results in a dual priority system. Each task has a regular priority, which it uses to preempt other tasks, and a preemption threshold (priority) which is its effective priority when it is running (or when a scheduling decision is being taken when a task completes processing). Preemption threshold can be emulated by changing the priority of a task to its threshold at the beginning of its execution, and then resetting it to its regular priority at the end of its execution. For recurring tasks, such as periodic ones, this must be done for each instance of the task.

It is easy to see that both preemptive and non-preemptive scheduling are special cases of scheduling with preemption threshold. If the threshold of each task is the same as its priority, then we have preemptive scheduling. On the other hand, if the threshold of each task is the highest priority in a system, then we have non-preemptive scheduling. Therefore, preemption threshold can potentially be used to take advantage of the characteristics of both preemptive scheduling and non-preemptive scheduling, and thereby make some task sets schedulable that are not schedulable under either preemptive or non-preemptive policies.

In this paper, we consider the problem of scheduling with preemption thresholds. We first develop schedulability analysis equations for this new model, by extending the traditional fixed priority worst case response time analysis [4, 5, 8, 7, 11]. We then develop an algorithm, which, given a periodic task set, optimally chooses the preemption thresholds and priorities for tasks such that if there exists any assignment that makes the task set schedulable, then the algorithm will find it.

*Manas Saksena has moved to Department of Computer Science, University of Pittsburgh.

The rest of the paper is organized as follows. Section 2 demonstrates an example illustrating that a task set with the generalized model has better schedulability as compared to both preemptive and non-preemptive scheduling models. Section 3 explains the task model and run-time model used in this paper and gives a formal statement of the problem we are trying to solve. In Section 4, we show how response times for tasks can be determined under this new model, given their priorities and thresholds. Section 5 presents an algorithm for optimal preemption threshold assignment for a task set with predefined priorities. Then, in Section 6, we develop a search algorithm to find an optimal assignment of priorities and preemption thresholds for a given task set. We give a brief review of related work in Section 7, and finally present some concluding remarks.

2. A Motivating Example

In this section, we give a simple example to show how preemption threshold can be used to improve schedulability of a task set. Table 1 shows a task set with 3 independent periodic tasks; each task is characterized by a period, a deadline, and a computation time. Assuming fixed-priority scheduling, the optimal priority ordering for this task set is deadline monotonic ordering with both preemptive scheduling [9] and non-preemptive scheduling [3]. The worst-case response times for the tasks are also shown in Table 1. 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.

Task	C_i	T_i	D_i	π_i	**WCRT** Preemptive	**WCRT** Non-Preemptive
τ_1	20	70	50	3	20	**55**
τ_2	20	80	80	2	40	75
τ_3	35	200	100	1	**115**	75

Table 1. Worst Case Response Times without Preemption Threshold

The use of preemption threshold, however, allows us to make the task set schedulable. By setting the preemption threshold for τ_2 as 3, and τ_1 as 2, the response times of all tasks are less than their respective deadlines, as 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

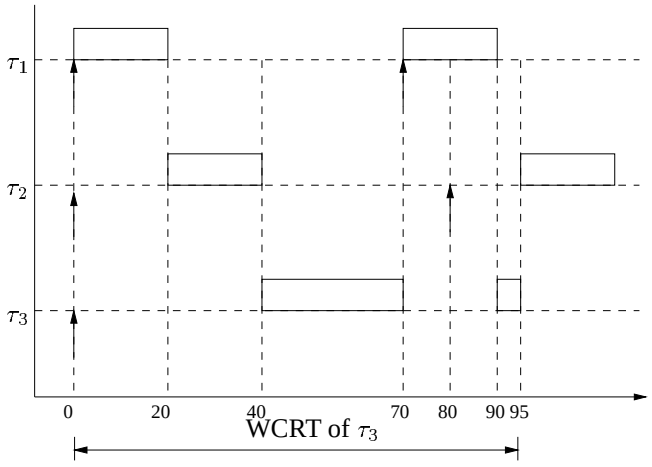

Figure 1. Run-time Behavior with Preemption Threshold

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 τ_2 arrives. It can not preempt the execution of τ_1. But after τ_1 finishes, τ_3 resumes execution since it is executing with a preemption threshold of 2. This is different from the preemptive scheduling without preemption threshold. Therefore, with preemption threshold, low priority task τ_3 improves its worst-case response time by preventing the interference from τ_2 after it starts execution. Note however that this also adds blocking time to τ_2, which increases its worst-case response time.

It is also interesting to see a little 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	**Priority**	**Preemption Threshold**	**WCRT**
τ_1	3	3	40
τ_2	2	3	75
τ_3	1	2	95

Table 2. Worst Case Response Times with Preemption Threshold

3. Problem Description

The example presented in the previous section illustrates how schedulability can be enhanced by using preemption thresholds. This raises a question of how to find the optimal assignment of task priorities and preemption thresholds. In this section, we first present the task model and run-time model used to perform worst-case response time analysis in this paper. Then, we give a formal statement of the problem being addressed in this paper, and finally give an overview of our solution approach.

3.1. Task Model

We consider a set of N independent periodic or sporadic tasks $\tau_1, \tau_2, \ldots, \tau_n$. 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 $\gamma_i \in [\pi_i, N]$. We assume that larger number denotes higher priority, and that no two tasks have the same priority.

3.2. Run-Time Model

Our run-time model is fixed-priority, preemptive scheduling with preemption threshold. It is a generalized fixed-priority scheduling model, which includes both traditional preemptive and non-preemptive scheduling as special cases. In this model, the priority and preemption threshold of task is fixed. When a task is released, it competes for the CPU at its priority. After a task τ_i has started execution, it can be preempted by another task τ_j if and only if $\pi_j > \gamma_i$. In other words, task τ_i runs as if it has its priority boosted up to its preemption threshold when it gets CPU for the first time and keeps this priority until it finishes its computation.

3.3. Problem Statement

With the task model we presented above, the problem of finding optimal priority ordering and preemption threshold assignment can be stated as following:

> *Given a set of tasks,* $\Gamma = \{\tau_i = \langle C_i, T_i, D_i \rangle | 1 \leq i \leq n\}$*, find whether there exists a set of values* $\{\langle \pi_i, \gamma_i \rangle | (1 \leq i \leq n)\}$ *s.t.* Γ *is schedulable (i.e., the worst-case response times of all tasks are no more than their respective deadlines).*

A naive approach to solving the problem requires an exhaustive search to find a feasible assignment of priorities and preemption thresholds, and has a search space of $O(n! \cdot n!)$. Clearly, an exhaustive search is unreasonable, and more effective ways of searching this space are needed.

3.4 Solution Approach

Our solution to this problem is presented in three 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 predefined priorities. We show that the search space can be reduced to $O(n^2)$ (instead of $O(n!)$), and present an algorithm to determine the optimal preemption threshold assignment.

(3) Finally, we consider the general problem where both the priorities and preemption threshold are to be determined. We develop a branch and bound algorithm that performs efficient search making use of the algorithm to determine preemption thresholds when task priorities are known.

4. Schedulability Analysis

We first begin with schedulability analysis of task sets when both the priority and the preemption threshold of each task is given. The schedulability analysis is based on the response time analysis of each task. With given priority and preemption threshold, if the worst-case response time of a task is less than the deadline, then the task is schedulable. If all the tasks in a system are schedulable, the system is schedulable. The response time analysis we are using in this paper is an extension of the well-known level-i busy period analysis [4, 5, 8, 7, 11], in which the response time is calculated by determining the length of busy period, starting from a critical instant.

The response time of a task includes two parts: (1) *interference* from other higher or equal priority tasks, including previous instances of the same task (2) *blocking* from lower priority tasks caused by the preemption threshold. Note that the terms are defined with respect to priorities of tasks, and not the preemption thresholds.

4.1. Blocking Time Analysis

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.

The following lemmas are very helpful in determining the upper bound for the blocking time from lower priority tasks due to the preemption threshold. We define the *preemption protection space* for task τ_i as $[\pi_i,\gamma_i]$.

Lemma 4.1 *There is no overlapping of preemption protection space between the tasks that have already started execution and have not finished yet.*

Proof: The proof is simple by noticing that for a task to start execution, it should have a priority higher than the preemption threshold of all the tasks that have already started execution and have not finished yet. □

Lemma 4.2 *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 τ_i can not preempt a lower priority τ_j only if π_i falls in the preemption protection space of τ_j, i.e., $\pi_j < \pi_i \leq \gamma_j$. From Lemma 4.1, we know that preemption protection spaces of started tasks will not overlap. Therefore, τ_i will only fall into at most one of these preemption protection spaces, i.e., be blocked by the owner of that preemption protection space. Furthermore, it is easy to see that any lower priority tasks, which have not started execution before the arrival of τ_i will not block τ_i. □

Therefore, we can derive the expression of *maximum blocking time* for a task τ_i as following:

$$B(\tau_i) = \max_j\{C_j :: \gamma_j \geq \pi_i > \pi_j\} \tag{1}$$

4.2. Busy Period Analysis

The busy period analysis studies the detail of multi-thread preemptive scheduling from the arrival of a task instance to the finish of the instance. The worst-case response time is calculated based on our busy period analysis. 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. Thus we can assume that they are periodic tasks whose period equals the minimum inter-arrival time.

The busy period of task τ_i in traditional fixed-priority, preemptive scheduling can be iteratively computed by using the following equation [11]:

$$w_i(q) = q \cdot C_i + \sum_{\forall j, \pi_j > \pi_i} \left\lceil \frac{w_i(q)}{T_j} \right\rceil \cdot C_j \tag{2}$$

where $w_i(q)$ is the length of the busy period for τ_i, q is the instance of τ_i in the busy period. The length of the busy period is then given by:

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

and where $W_i(q)$ is the smallest value of $w_i(q)$ that satisfies Equation 2.

We extend the original level-i busy period analysis by dividing the busy period of task τ_i into two parts: the first one starts from time 0 and stops when τ_i starts execution (i.e. when τ_i gets the cpu for the first time), and the second one starts exactly after the first one and stops when the task instance finishes. We use $\mathcal{S}_i(q)$ to denote the worst-case start time for instance q of task τ_i, and use $\mathcal{F}_i(q)$ to denote the end of busy period (i.e., the worst-case finish time). Furthermore, the arrival time of instance q of task τ_i can be presented by $(q-1) \cdot T_i$. We compute $\mathcal{S}_i(q)$ and $\mathcal{F}_i(q)$ for $q = 1, 2, 3, \ldots$, until we reach a $q = m$, such that $\mathcal{F}_i(m) \leq q \cdot T_i$. Then, the worst-case response time of task τ_i is given by:

$$\mathcal{R}_i = \max_{q \in [1,\ldots,m]} (\mathcal{F}_i(q) - (q-1) \cdot T_i) \tag{4}$$

Computing Worst-case Start Time: Before a task τ_i starts execution, there is blocking from lower priority tasks and interferences from higher priority tasks. Among all lower priority tasks, only one lower priority task can cause blocking as we proved in Lemma 4.2. When τ_i arrives, this task must have arrived and begun executing. 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 than instance q should be finished before the start time. Therefore, $\mathcal{S}_i(q)$ can be computed iteratively using the following equation.

$$\mathcal{S}_i(q) = B(\tau_i) + (q-1) \cdot C_i + \sum_{\forall j, \pi_j > \pi_i} \left(1 + \left\lfloor \frac{\mathcal{S}_i(q)}{T_j} \right\rfloor\right) \cdot C_j \tag{5}$$

Computing Worst-case Finish Time: Once the task starts execution, we have to consider the interference to compute the finish time of the busy period. From the definition of preemption threshold, we know that only tasks with higher priority than the preemption threshold of τ_i can preempt it. Furthermore, after τ_i starts execution, it runs as if it has the priority equal to its preemption threshold. Thus, only tasks with higher priority than γ_i can get the cpu before τ_i finishes. Therefore, interference only comes from tasks with

higher priority than γ_i during the execution of τ_i after it starts. Based on this, we can derive the equation for computing $\mathcal{F}_i(q)$ as following:

$$\mathcal{F}_i(q) = \mathcal{S}_i(q) + C_i + \sum_{\forall j, \pi_j > \gamma_i} \left(\left\lceil \frac{\mathcal{F}_i(q)}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{\mathcal{S}_i(q)}{T_j} \right\rfloor \right) \right) \cdot C_j \quad (6)$$

5. Preemption Threshold Assignment with Predefined Priorities

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 preemption thresholds to tasks with predefined priorities, such that the assignment ensures schedulability. In the next section, we will propose a systematical approach to find the optimal priority and preemption threshold assignment using this algorithm as a subroutine.

5.1. 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. 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 a systematical process for preemption threshold assignment. Here, we have the assumption that the task set has a given priority assignment.

Lemma 5.1 *Changing preemption threshold of 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 been seen clearly by looking at the equations used to determine the worst-case response time. As a direct derivative, we can get a useful corollary as following:

Corollary 5.1 *The worst-case response time of task τ_i will not be affected by the preemption threshold assignment of any task τ_j with $\pi_j > \pi_i$.*

The schedulability of a task set is based on the worst-case response time analysis for each task. Changing the preemption threshold of one task will affect the worst-case response time of other tasks. This complicates the search for a feasible preemption threshold assigment. However, Corollary 5.1 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 5.1, presented below, helps us to 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. If there are more than one preemption thresholds that can make the task schedulable, then the minimum of them is the optimal one with the concern of maximizing the schedulability of other tasks in the task set.

Theorem 5.1 *Suppose that a set of n tasks $\Gamma = \{\tau_i = \langle C_i, T_i, D_i \rangle | 1 \leq i \leq n\}$ is schedulable with a set of values $\{\langle \pi_i, \gamma_i \rangle | (1 \leq i \leq n)\}$. 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.*

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

A given task set may be unschedulable with any preemption threshold assignment. Therefore, we need to find out this case and stop our process of preemption threshold assignment. Lemma 5.2 gives the sufficient condition to claim a task set to be unschedulable.

Lemma 5.2 *With given priority π_i, if setting the preemption threshold γ_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 6 with Equation 2, we can see that preemption threshold reduces the worst-case response time of low priority task τ_i by preventing the interference from some higher priority tasks after τ_i starts execution, thus improving its schedulability. 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. □

Figure 2 gives the pseudo code of the preemption threshold assignment algorithm. The algorithm is designed based on the results presented above. Corollary 5.1 suggests us to start threshold assignment from the lowest priority task to highest priority since the schedulability analysis only depends on the thresholds of tasks with lower priority than the current task. While searching the optimal preemption threshold for a specific task, Theorem 5.1 implies that the search

starts from its priority to the highest priority in the system and stops at the first one that makes the task schedulable, i.e. the minimum preemption threshold that makes it schedulable. Furthermore, Lemma 5.2 tells us when we can say the system is not schedulable and stop the algorithm.

The algorithm assumes that the tasks are numbered 1,2,...,n, and the predefined priorities of these tasks are also in this order. Function WCRT(task, threshold) will calculate the worst-case response time of a task based on its characteristic (i.e. computation time, period, priority) and the parameter preemption threshold.

```
Algorithm: Assign Preemption Thresholds
// Assumes that task priorities are already known
(1)  for (i := 1 to n)
(2)      γ_i = π_i
         // Calculate worst-case response time of τ_i
(3)      R_i = WCRT(τ_i,γ_i) ;
(4)      while (R_i > D_i) do    // while not schedulable
(5)          γ_i++ ; // increase threshold
(6)          if γ_i > n then
(7)              return FAIL; // system not schedulable.
(8)          endif
(9)          R_i = WCRT(τ_i,γ_i) ;
(10)     end
(11) end
(12) return SUCCESS
```

Figure 2. Algorithm for Preemption Threshold Assignment

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.

6. 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, if one exists) priority and preemption threshold assignment for a given task set. We give a branch-and-bound search algorithm that efficiently 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] for preemptive priority scheduling of a task set. Unfortunately, the introduction of preemption thresholds adds another dimension to the search space.

```
Search(TaskSet, Priority)
     /* Terminating Condition */
(1)  if (Priority == N+1) then
         /* Call the algorithm in Figure 2 for optimal
         preemption threshold assignment */
(2)      if (AssignThresholds() == SUCCESS) then
(3)          return SUCCESS
(4)      else return FAIL
(5)      endif
(6)  endif
(7)  foreach τ_k in TaskSet do
(8)      Delay_k := WCRT(τ_k) - D_k;
(9)      if Delay_k ≤ 0 then
(10)         π_k := Priority;
(11)         if Search(TaskSet-τ_k, Priority+1) == SUCCESS then
(12)             return SUCCESS
(13)         else
(14)             return FAIL
(15)         endif
(16)     endif
(17) end
     /* Not schedulable without preemption threshold */
     /* Sort the task set by ascending order of Lateness_k */
(18) SortedList = GenerateList(TaskSet);
     /* Refine SortedList, eliminating those tasks that are
     unschedulable even with preemption threshold equal to N */
(19) Refine(SortedList);
     /* Recursively perform depth first search */
(20) foreach τ_i in SortedList do
(21)     π_i := Priority;
(22)     if Search(TaskSet-τ_i, Priority+1) == SUCCESS then
(23)         return SUCCESS
(24)     endif
(25) end
(26) return FAIL
```

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

Our search algorithm 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. Initially, the sorted part is empty and all tasks are in the unsorted part. It recursively moves one task from

the unsorted list to the sorted list, by choosing a heuristically decided candidate at each stage. When all tasks are in the sorted list, a complete priority ordering is generated, and the threshold assignment is called.

When considering the next candidate to move into the sorted list, all tasks in the unsorted list are examined in turn. If any of these tasks is schedulable under preemptive priority scheduling without thresholds, then it is chosen to move to the sorted list. If no task is schedulable then the tasks are sorted according to a heuristic function defined later. The search for a priority ordering then proceeds in a best first manner.

6.1. Pruning Infeasible Paths

From the algorithm description above, we can see that the only case where several branches are necessary is when all the tasks in the unsorted part are unschedulable (under preemptive priority scheduling without thresholds). As we have shown in the response time analysis, the worst-case response time of a task can be improved by assigning it a preemption threshold higher than its priority. Since we do not know the priority ordering of the unsorted higher priority part, we can not decide the value of the preemption threshold to make the task schedulable. Therefore, we can not know moving which task to the sorted part will make the solution optimal. Thus, the only choice left for us is try all the tasks.

However, we notice that if a task is unschedulable by assign its preemption threshold to the highest priority, N, there will be no possible solution with this task at the next priority (i.e., it must have a higher priority than some of the tasks in the unsorted list). Therefore, we can eliminate it from the choice we should try. Thus, we reduce the search space by using the worst-case response time with preemption threshold equal to N as a bounding function.

6.2. Heuristic Function

After we prune the infeasible paths, we should also decide the order of our trial. A good heuristic policy may reduce the search time in a real world implementation. Here, we propose a policy of shortest lateness first. We define *lateness* as the difference between the deadline and the worst-case response time of a task when its preemption threshold equals to its priority:

$$Lateness_i = \mathcal{R}_i - \mathcal{D}_i \tag{7}$$

where $\mathcal{R}_i$ is computed using preemptive priority policy without preemption thresholds. This policy is designed by noticing that reducing worst-case response time of a specific task by increasing its preemption threshold may be at the cost of increasing worst-case response time of other tasks between its priority and its preemption threshold (which would require their preemption thresholds to be set higher, and so on). We select the one with smallest lateness with the hope that it will be the one that needs the smallest preemption threshold leaving a larger slack for the rest of the tasks.

Figure 3 gives the pseudo code of the search algorithm for optimal combination of priority ordering and preemption threshold setting. It takes two parameters: **TaskSet**, which is the unsorted part (containing all the tasks waiting for priority assignment), and **Priority**, which is the next priority to assign. The tasks that have been assigned priorities are kept separately for preemption threshold assignment at the end of the algorithm. There are four functions used in the pseudo code: **AssignThresholds()** is the algorithm in Figure 2; **WCRT(τ_k)** will calculate the worst-case response time of a task under preemptive priority scheduling without thresholds, and under the current partial priority ordering; **GenerateList(TaskSet)** will generate a linked list of tasks in TaskSet in the ascending order of lateness as defined in Equation 7; and **Refine(SortedList)** will eliminate tasks, which are not schedulable even by setting their preemption threshold to the highest possible priority N, from the link list of tasks.

7. Related Work

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 (although, the priority can be changed from the application).

With fixed-priority preemptive scheduling, the deadline monotonic ordering has been shown to be optimal [9] 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 [10]. However, for a general task set, where deadlines are not related to the periods, Lehoczky [7] points out that deadline monotonic ordering is not optimal are optimal. Audsley [1] gives an optimal priority assignment procedure with complexity of $O(n^2)$.

Feasibility in fixed priority case is based on computing the worst-case response times and by comparing them with deadlines. The notion of *level-i busy period* is introduced by Lehoczky [7] and computation of the worst-case response

time for general task sets is shown in [5, 4, 7, 11].

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 no larger computation time [3]. Furthermore, the optimal priority assignment algorithm proposed by Audsley [1] is shown to be still valid [3]. 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 [3].

The work that comes closest to ours is the scheduling of tasks with varying execution priorities [4, 2]. In [4], 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 [2] 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 them since our focus is primarily on how to assign priorities and preemption thresholds, which is not addressed in [4].

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. We illustrate that the application of preemption threshold improves the feasibility of task sets. 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 algorithm for finding optimal preemption threshold assignment for a task set with predefined priority. Finally using this threshold assignment algorithm, we develop an efficient and optimal search algorithm to find the optimal priorities and preemption thresholds.

We believe that the results presented in this paper are not only interesting from a theoretical perspective, but also from present a viable technology. Preemption threshold can be easily emulated from the user level albeit at the cost of extra system calls to set priority. On the other hand, it is relatively trivial to add this to a real-time kernel, with minimal cost. Contrast this with the dual-priority scheme of Burns [2], where the implementation costs are non-trivial. Moreover, with the reduction in context switches, we speculate that the overheads may even be less than a purely preemptive scheme.

The practical usage of the results in this paper require an empirical investigation of the benefits of scheduling with preemption threshold. In our ongoing work, we are evaluating this by quantifying the increase in schedulability using preemption thresholds over randomly generated task sets. Also, we are evaluating the effectiveness of the search algorithm in finding an optimal assignment.

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, Dec. 1991.

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

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

[4] 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.

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

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

[7] J. 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, Dec. 1990.

[8] 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, Dec. 1989.

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

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

[11] 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, Mar. 1994.