Scheduling Fixed-Priority Tasks with Preemption Threshold
An Attractive Technology?

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 preemptibility 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 an earlier paper, we had developed the model of scheduling fixed priority tasks with preemption threshold, which unifies the concepts of preemptive and non-preemptive scheduling, and subsumes both as special cases. In this paper we 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 (in CPU overheads due to preemptions, and stack space for tasks) as compared to preemptive scheduling, even when a task set is schedulable under the preemptive scheduling model. We believe that the new scheduling model provides a compelling technology alternative, by providing high levels of schedulable utilization at lower costs.

1. Introduction

Since the pioneering work of Liu and Layland [8], much work has been done in the area of real-time scheduling theory, and in particular fixed priority preemptive scheduling theory (e.g., [3, 4, 6, 7,

[*]Yun Wang is a phd candidate at Concordia University.

9]). 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). Moreover, preemptive schedulers have higher overheads as compared to non-preemptive schedulers.

In an earlier paper [10], we proposed a generalized model of fixed-priority scheduling that subsumes both preemptive and non-preemptive schedulers. The model used the notion of *preemption threshold* (introduced by Express Logic, Inc. [5], in their ThreadX real-time operating system). In this 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.

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 the case of preemptive scheduling. On the other hand, if the threshold of each task is the highest priority in a system, then we get non-preemptive scheduling. By choosing an appropriate preemption threshold value, we can potentially take advantage of the characteristics of both preemptive scheduling and non-preemptive scheduling.

In our previous work, which is presented in [10], we derived some necessary theoretical results for the application of preemption threshold. We first showed how to compute worst-case response times of tasks under this scheme using the well known level-i busy period analysis [3, 4, 6, 7, 9]. We then addressed the problem of determining a feasible assignment of task priorities and preemption thresholds. Our solution came in two steps. First, given an assignment of priorities, we developed an efficient algorithm that computed 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 proposed a search algorithm to find an optimal assignment of both priorities and preemption thresholds by extending Audsley’s optimal priority assignment algorithm [1, 9].

1.1. Contributions

The work presented in this paper is a continuation of our previous work. As we mentioned in our previous paper [10], our new model provides enhanced schedulability and reduces preemptions (which reduces overheads such as context switching). In this paper, we further study the benefit of our model in a quantitative way, and provide simulation results (over randomly generated task sets) to support our previous (and some new) claims. The paper makes several contributions.

We first develop a greedy heuristic algorithm to find a feasible assignment of priorities and preemption thresholds. We compare the performance of this algorithm with optimal priority assignments for both preemptive and non-preemptive scheduling. We quantify the improvement in schedulability by using simulations over randomly generated task sets, and using breakdown utilization [6] 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 (upto 30%) in the number of preemptions (averaged over multiple task sets with the same characteristics).

We also show that with preemption thresholds we can reduce memory space requirements of a task set. In preemptive scheduling, typically each task is assigned a private stack space, while in non-preemptive scheduling all tasks can share a single task. Using preemption threshold, we can identify pairs of tasks that are mutually non-preemptive. By grouping the tasks into sets that are mutually non-preemptive, we need to assign only one stack for such a set of tasks. We present an efficient optimal algorithm that minimizes the number of non-preemptive sets. Furthermore, we quantify the reduction in stack space over randomly generated task sets, and show that as we increase the number of tasks, the number of non-preemptive groups grows much more slowly, leading to substantial reduction in stack space requirements.

1.2. Paper Organization

The rest part of this paper is organized as follows. Section 2 briefly reviews the theoretical results developed in our previous work. In Section 3, we quantify the improvement in schedulability using the preemption threshold scheduling model. Then, in Section 4, we show look at the reductions in preemptions. In Section 5, we show the preemption threshold scheduling model can be used to reduce stack space. We end the paper with some concluding remarks.

2. Background: Scheduling Fixed-Priority Tasks with Preemption Threshold

In this section, we will briefly review the theoretical results for our generalized fixed priority scheduling model with preemption threshold. Under a traditional fixed priority scheduler, each task is assigned a static priority, which the scheduler uses to select the task to be executed. In our new model, each task is assigned a preemption threshold in addition to its priority. Tasks are scheduled on their priorities before they started (i.e., get CPU for the first time). However, when a task starts running, only those tasks with priorities higher than the preemption threshold of the current task can preempt the execution of current task. Thus, preemption threshold essentially results in dual priority tasks. In the rest of this section, we briefly review the main results that were presented in [10].

2.1. Worst-case Response Time Analysis with Preemption Threshold

In our analysis, we assume a task set consists of independent periodic or sporadic tasks $\tau_1, \tau_2, \ldots, \tau_n$. Each task 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 also assume that overheads (i.e. context switching, etc.) are negligible and therefore set to zero. Each task will be given a priority $\pi_i \in [1, n]$ and a preemption threshold $\gamma_i \in [\pi_i, n]$. By convention, we assume larger numbers denote higher priority and that tasks are assigned unique priorities.

The worst-case response time analysis is done by incorporating the notion of preemption threshold into the well-know level-i busy period analysis [7, 9], in which the response time is calculated by determining the length of busy period, starting from a critical instant. Since the effective priority of a task changes when the task starts executing[1], the busy-period analysis must be done by considering both worst case start times and worst case finish times. Worst-case start time denotes the time between when a task is activated and when the task starts execution for the first time. In this stage, interference includes: (1) *blocking* from lower priority tasks caused by the preemption threshold; and (2) *interference* from other higher priority tasks, including previous instances of the same task. The second stage is after the task starts, and before it finishes. In this stage, interference only comes from other tasks with priority higher than the preemption threshold of the current task. Below, we demonstrate the calculation for each stage.

Bounded Blocking Time: Blocking denotes the effect of lower priority tasks on the response time of a higher priority task. The following theorem shows that the blocking time in our model is bounded, and its proof is given in [10].

Theorem 2.1 *A task τ_i can be blocked by at most one lower priority task τ_j. Furthermore, it must be the case that $\gamma_j \geq \pi_i$.*

With Theorem 2.1, we can give the upper bound of blocking time for task τ_i as following:

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

[1]Note that this is true for non-preemptive scheduling as well.

Computing Worst-case Start Time: As we mentioned above, before the qth instance of task τ_i starts execution, there is blocking from lower priority tasks and interference from higher priority tasks and earlier unfinished instances of task τ_i. All higher priority task instances 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 of qth instance of τ_i. Therefore, $\mathcal{S}_i(q)$ can be computed iteratively using the following equation.

$$\mathcal{S}_i(q) = B(\tau_i) + (q-1)\cdot C_i + \sum_{\forall j, \pi_j > \pi_i} \left(1 + \left\lfloor \frac{\mathcal{S}_i(q)}{T_j} \right\rfloor\right)\cdot C_j \tag{2}$$

Computing Worst-case Finish Time: As we already mentioned, once the qth instance of task τ_i starts execution, only tasks with higher priority than the γ_i of τ_i can preempt τ_i. Therefore, only these tasks can cause interference in this interval. Moreover, these tasks must have arrived after $\mathcal{S}_i(q)$, otherwise they would have executed before τ_i began execution. Based on this, we can derive the equation for computing the worst-case finish time $\mathcal{F}_i(q)$ as following:

$$\begin{aligned}\mathcal{F}_i(q) = {} & \mathcal{S}_i(q) \; + \; C_i \\ & + \sum_{\forall j, \pi_j > \gamma_i} \left(\left\lceil \frac{\mathcal{F}_i(q)}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{\mathcal{S}_i(q)}{T_j} \right\rfloor\right)\right)\cdot C_j\end{aligned} \tag{3}$$

Worst-case Response Time: According to the level-i busy period analysis, the calculation terminates when the qth instance of task τ_i finishes before the next instance comes. In our model, it means $\mathcal{F}_i(q) \leq qT_i$. Assume that the calculation of busy period finishes at instance m, then the worst-case response time can be calculated by the following equation.

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

2.2. Algorithms for optimal preemption threshold assignment with given priorities

As we can see from the above discussion, the range of possible value for preemption threshold for τ_i is [π_i, n] when the priority of each task is given. If each task has a preemption threshold equal to its priority, then our model degrades to preemptive scheduler. If each task has a preemption threshold equal to the highest priority in the system, then our model degrades to non-preemptive scheduler. To test whether a task set is schedulable in our model, the intuitive way is through an exhaustive search of all possible preemption threshold assignments. However, the size of the search space will be $O(n!)$. An efficient algorithms with a search complexity of $O(n^2)$ was developed in [10]. The pseudo-code of the algorithm is reproduced below.

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

```
Algorithm: Assign Preemption Thresholds
// Assumes that task priorities are already known
(1)   for (i := 1 to n)
(2)       $\gamma_i$ := $\pi_i$
      // Calculate worst-case response time of $\tau_i$
(3)       $\mathcal{R}_i$ := WCRT($\tau_i$,$\gamma_i$) ;
(4)       while $R_i > D_i$ do    // is it schedulable?
(5)           $\gamma_i$++ ;
(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 1. Algorithm for Minimum Preemption Threshold Assignment

of a task based on its characteristic (i.e. computation time, period, priority) and the parameter preemption threshold using the methodology we have just shown. We note that response time caculation is possible in the algorithm, even with a partial assignment, due to the order in which the assignment is done, i.e., from lower to higher priorities.

3. Schedulability Improvement

As we mentioned in previous section, our model subsumes both preemptive and non-preemptive scheduling. Therefore, any task set that is schedulable with a preemptive or non-preemptive scheduler is also schedulable in our model. In [10], 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 basic strategy is to use randomly generated task sets and test for their schedulability under the different scheduling models.

To compare the schedulability of task set under different scheduling policies, we need a quantitative measurement. In [6], *breakdown utilization* is used to measure the stochastic characteristics of the schedulability of task sets. The breakdown utilization is defined as the associated utilization of a task set at which a deadline is first missed when the computation time of the task set is scaled by a factor. In this paper we use the same idea to measure the schedulability.

3.1. 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.

Each of these 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) 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 utilizations, and then testing for schedulability. Since the randomly generated task sets may have utilizations greater than 1 to begin with, we initially scale the utilization to 100%. We then do a binary search to find the maximum utilization at which the task set is schedulable under a particular scheduling algorithm.

3.2. Scheduling Algorithms

To ensure that the comparisons are fair, we use optimal priority assignments for both preemptive and non-preemptive scheduling. For the preemptive case, the optimal priority assignment is simply the rate-monotonic assignment [8]. 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 [2] which is basically the same as Audsley's algorithm, but adapted to non-preemptive scheduling.

For scheduling with preemption thresholds, we do not have any efficient optimal algorithm to assign priorities and preemption thresholds. The algorithm presented in [10] has an exponential search space in the number of tasks. Clearly, this algorithm becomes infeasible to use, even with a modest number of tasks. Therefore, we developed a greedy-heuristic algorithm which was then used for schedulability comparisons. We present this algorithm next.

```
Algorithm: GreedyAssignment(RemainingTasks, nextPriority)
         /* Terminating Condition */
(1)      if (RemainingTasks == NULL) then
                /* Call the algorithm in Figure 1 for optimal preemption threshold assignment */
(2)             if (AssignThresholds() == SUCCESS) then return SUCCESS
(3)             else return FAIL
(4)             endif
(5)      endif
         /* Assign Heuristic Value to Each Task */
(6)      foreach $\tau_k$ in RemainingTasks do
(7)             $\pi_k$ := nextPriority ;    /* tentative assignment */
(8)             $\mathcal{R}_k$ := WCRT($\tau_k$);    /* compute response time */
(9)             if $\mathcal{R}_k \geq D_k$ then $h\_val_k := \mathcal{D}_k - \mathcal{R}_k$
(10)            else $h\_val_k$ := GetBlockingLimit($\tau_k$) ;
(11)            endif
(12)            $\pi_k$ := $n$ ;    /* reset, to allow computing heuristic value for other tasks */
(13)     end
(14)     /* Select the task with the largest heuristic value next */
(15)     $\tau_k$ := max_heuristic_val(RemainingTasks) ;
(16)     $\pi_k$ := nextPriority ;    /* final priority assignment */
(17)     /* Recursively Assign Priorities to Remaining Tasks */
(18)     if GreedyAssignment(RemainingTasks $-\tau_k$,nextPriority+1) == SUCCESS then
(19)            return SUCCESS ;
(20)     return FAIL ;
```

Figure 2. Search Algorithm for Optimal Assignment of Priority and Preemption Threshold

3.3. Assignment of Priorities and Preemption Threshold: A Greedy Algorithm

The greedy algorithm works in two phases. In the first phase, it generates a priority assignment using heuristics. Then, in the second phase, it generates the preemption thresholds for the selected priority assignment using the algorithm presented earlier in Figure 1. Note that the threshold assignment algorithm is optimal for a given priority assignment. Therefore, the effectiveness of this algorithm depends on the priority assignment part.

The priority assignment phase of the algorithm is based on Audsley's priority assignment algorithm for preemptive priority scheduling. 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 all assumed to be higher than the tasks in the sorted list.

Initially, the sorted part is empty and all tasks are in the unsorted part. The algorithm repeatedly moves one task from the unsorted list to the sorted list, by assigning it the next higher priority. In this ways, tasks are assigned priorities from lowest priority to highest priority. When considering the next candidate to move into the sorted list, each task in the unsorted list is examined in turn, and a heuristic value assigned to it. The task with the largest heuristic value is selected to move to the sorted list, and is assigned the next priority. When all tasks are in the sorted list, a complete priority ordering has been generated, and the threshold assignment algorithm is called to assign thresholds.

We choose a simple heuristic function to select the next task for priority assignment. First, we tentatively assign the next higher priority to the task, and then compute its worst case response time based on this priority. The worst case response time computation is possible since we only need to know which tasks are higher priority, and not their actual priorities. Also, at this stage, we have not assigned preemption thresholds – so we assume that the preemption threshold of a task is the same as its priority (i.e., the pure preemptive priority case). The computed worst case response time is then compared with the task deadline. Now, there are two cases:

(1) If the computed response time is less than the deadline, then this task is a "good" candidate. However, unlike Audsley's algorithm, this does not guarantee that the task will be schedulable with the final assignment. This is because, 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. Therefore, we assign a heuristic value that is the maximum blocking that a task can tolerate while still meeting its deadline. This can be done by assigning a blocking term to the task, repeating the worst-case response time computation, and checking if it still meets the deadline.

(2) If, however, the computed response time is larger than the task deadline, then it is an indication that this priority is too low for the task. Note, however, that it is still possible for a task to be schedulable since it can be given a higher preemption threshold. So, in the case, that there are no tasks in the first category, we want to choose the task that needs the smallest reduction in interference from higher priority tasks. Accordingly, we assign a heuristic value of $D_k - \mathcal{R}_k$. Note that these values are negative, and therefore, no such task will be selected if there is a task in the first category.

Figure 2 gives the pseudo code of the greedy algorithm described above for assigning priorities and preemption thresholds. It takes two parameters: **RemainingTasks**, which is the unsorted part (containing all the tasks waiting for priority assignment), and **nextPriority**, 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. The first few lines correspond to the second phase of the algorithm, where all priorities have been assigned. Then we simply call the algorithm **AssignThresholds()**, the algorithm in Figure 1, to assign preemption thresholds.

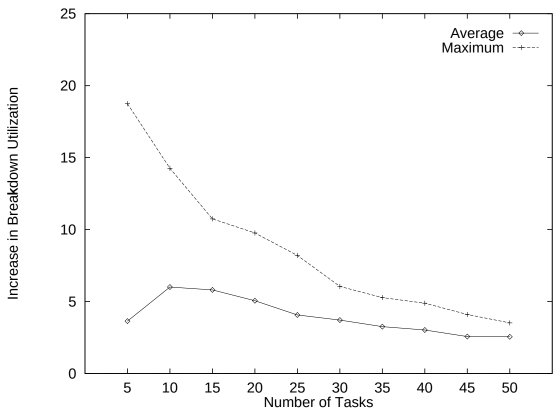
When there are tasks still left in the unsorted list, then steps 6-13 show the assignment of heuristic value for each candidate task in the **RemainingTasks** list. The rest of the code selects the task with the highest heuristic value, assigns it the next priority, and recursively calls the algorithm to assign priorities to the remaining tasks. Note that the way in which the algorithm is presented, it can be easily generalized to a search algorithm by considering additional tasks if the recursive call in step 19 returns FAIL.

The heuristic algorithm presented above 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.

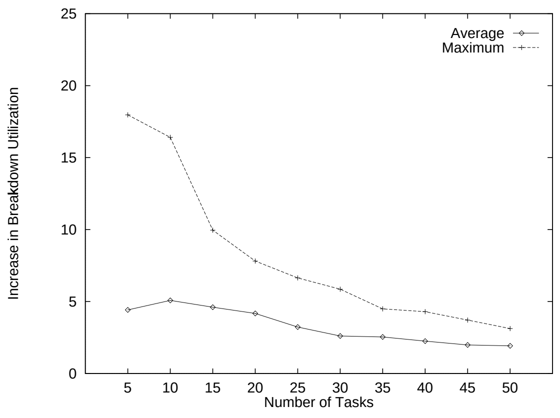
3.4. Simulation Results

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, and (2) maximum period. 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 3 and 4 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 3 shows schedulability improvement as compared to the 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



(a) Maximum Period = 10



(b) Maximum Period = 100

Figure 3. Schedulability Improvement with Preemption Threshold as compared to Preemptive Scheduling.

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 utilizations, 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 4(b).

While non-preemptive scheduling shows poor in general, there are selected cases when it performs better than preemptive scheduling, and better than our heuristic algorithm for preemption threshold. These results can be seen in Figure 4(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 utilizations.

4. Preemption Overheads

The application of our fixed priority scheduling model with preemption threshold not only provides better schedulability, but may also may reduce the number of preemptions as compared to pure preemptive

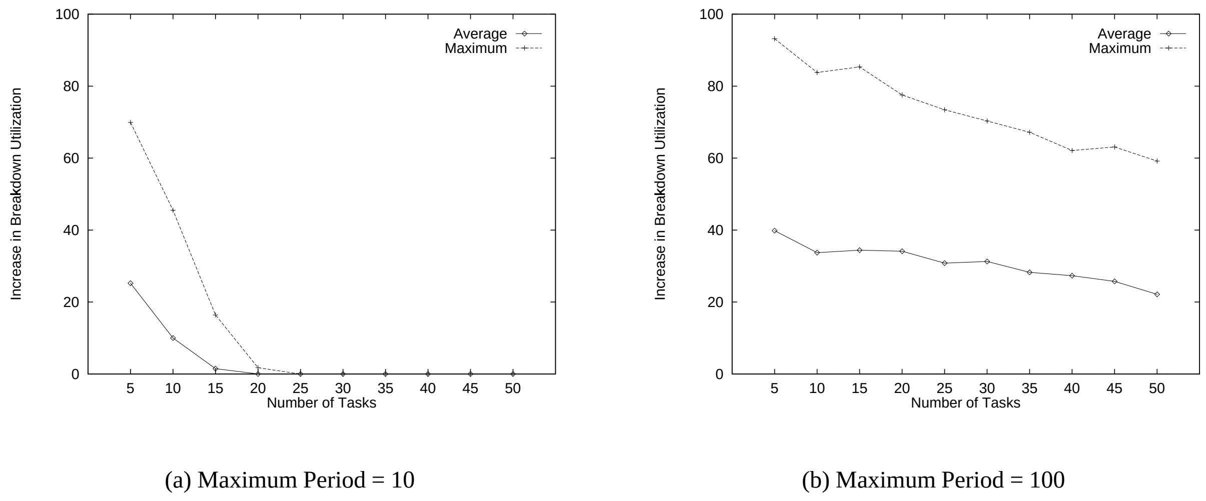


Figure 4. Schedulability Improvement with Preemption Threshold as compared to Non-Preemptive Scheduling.

scheduling. 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 preventions.

One possible benefit of 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 [9]. 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. We use simulation studies to see if there is any substantial reduction in the number of preemptions.

4.1. Simulation Design

We use the same randomly generated task sets that we had 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 a 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:

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

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. 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 assign preemption thresholds to the tasks. For this purpose, we could again use the algorithm in Figure 1, which is optimal (in terms of ensuring schedulability) for a given priority ordering. However, the algorithm tries to assign the smallest preemption threshold values to tasks such that they will be schedulable. This is necessary to ensure its optimality. However, we notice that once a task set has been deemed schedulable, then we can iterate over it again and try to assign the largest preemption threshold values such that the task set remains schedulable. Clearly, larger preemption threshold values reduce the chances of preemptions, and therefore, should result in lower preemptions.

Figure 5 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 still keep the system schedulable. 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) task cannot increase the assignment of a former (higher priority) tasks, and thus we only need to go through the list of tasks once.

4.2. Simulation Results

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 6. 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.

5. Reducing the Memory Requirements

In the previous sections, we have already seen that scheduling with preemption thresholds can improve schedulability, and can reduce some of the CPU scheduling overheads due to preemptions. In this section, we will show that it can also reduce memory overheads for tasks. Memory is a critical resource in many embedded systems, due to its high power consumption. Therefore, reduction in memory overheads for embedded systems can be very useful.

```
Algorithm: Assign Maximum Preemption Thresholds
// Assumes that task priorities and minimum 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 5. Algorithm for Finding Maximum Preemption Threshold

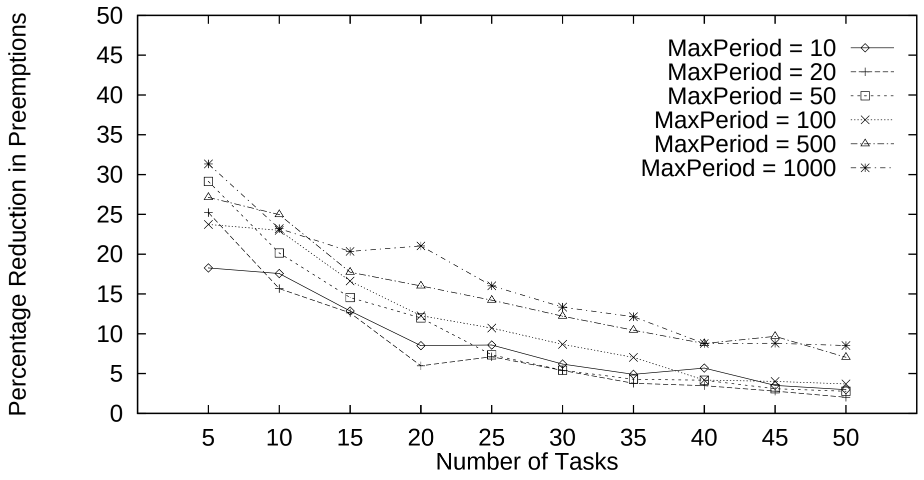


Figure 6. Average Percentage Reduction in Number of Preemptions for Preemption Threshold Scheduling as compared to Pure Preemptive Scheduling

In pure preemptive priority scheduling, each task must have its own stack space. However, consider the case when two tasks cannot preempt each other. In this case, the tasks can share their stacks, since it is never possible that the two tasks are in the preempted state at the same time. The idea can be generalized to non-preemptive groups of tasks, where no task in the group can preempt another task in the group. Then, such a group of tasks can use the same stack space. In the extreme case, for non-preemptive scheduling, only a single stack is needed for the entire task set.

The basic idea of preemption thresholds is to eliminate any unnecessary preemptibility. Therefore, in-effect it forms non-preemptive pairs of tasks. By grouping these pairs into non-preemptive groups, we can use the idea outlined above to reduce stack usage. In the remainder of this section we show how to exploit this idea. We also produce simulation results to characterize the effectiveness of this scheme in reducing stack space usage.

5.1. Partitioning Tasks into Non-Preemptive Groups

We say that two tasks τ_i and τ_j are mutually non-preemptive if τ_i cannot preempt τ_j, and τ_j cannot preempt τ_i. Let us define the *blocking range* of a task τ_i as $BR(\tau_i) = [\pi_i, \gamma_i]$. Using the concept of blocking range, it is easy to prove the following proposition.

Proposition 5.1 *Consider two tasks τ_i and τ_j, with blocking ranges $BR(\tau_i)$ and $BR(\tau_j)$. If $BR(\tau_i) \cap BR(\tau_j) \neq \phi$, then τ_i and τ_j are mutually non-preemptive.*

Proof: Since $BR(\tau_i) \cap BR(\tau_j) \neq \phi$, therefore $(\pi_i \leq \gamma_j) \wedge (\pi_j \leq \gamma_i)$. By the definition of preemption threshold they can not preempt each other. $\square$

Now, we can formally define the concept of a non-preemptive group as follows:

Definition 5.1 (Non-Preemptive Group) *A set of tasks $\mathcal{T} = \{\tau_{i_1}, \tau_{i_2}, \ldots, \tau_{i_m}\}$ forms a non-preemptive group if, for each pair of tasks $\tau_j \in \mathcal{T}$ and $\tau_k \in \mathcal{T}$, τ_j and τ_k are mutually non-preemptive.*

We now consider the problem of merging tasks into non-preemptive groups. We will assume that the priority and preemption thresholds of tasks are already assigned, such that the task set is schedulable. Given that, our objective is to partition the tasks into the minimum number of non-preemptive groups. We have developed an efficient and optimal algorithm to solve this problem, which is given in Figure 7, and is described next. The algorithm is optimal in the sense that it forms the minimum number of non-preemptive groups.

The algorithm begins by sorting the tasks in the non-decreasing order of their thresholds, with ties broken arbitrarily. Let the sorted list be denoted as L. We then remove the first task (τ_k) from this list and form a new group G. We will call τ_k as the representative of the group. Then, we look at every other task and add any task τ_j into G if $(\pi_j \leq \gamma_k)$, i.e., it is mutually non-preemptive with τ_k. Also τ_j is removed from L. Note that, since L was already sorted by preemption threshold, it must be the case that $(\pi_k \leq \gamma_j)$. Once all tasks have been examined, we have formed one non-preemptive group, with the remaining tasks in the list L. We reiterate this process of forming groups until no tasks remain the list L.

The correctness of the algorithm requires us to show two things: (1) that the groups formed are non-preemptive groups as defined using Definition 5.1, and (2) that no other partitioning into non-preemptive groups can be done with a smaller number of groups. To prove that the groups formed are non-preemptive groups, let us look at the representative member τ_k of a group G. Since the list of tasks is kept sorted by the threshold, and τ_k is the head of the list, it must be the case that $(\gamma_k \leq \gamma_j)$ for any $\tau_j \in G$. Therefore, $\gamma_k \in BR(\tau_j)$ for any $\tau_j \in G$. Hence, for any pair of tasks τ_i and τ_j in G, both their blocking ranges include the value γ_k, and therefore their intersection is non-null, which implies that they mutually non-preemptive by Proposition 5.1. Then, by Definition 5.1, G is a non-preemptive group.

Now, let us consider the optimality of the algorithm. Consider any two groups formed by the algorithm, and consider their representative members – say τ_j and τ_k. Then, due to the nature of the algorithm, the blocking ranges of the τ_j and τ_k do not overlap. Therefore, they must be in separate non-preemptive groups in any partitioning of the task set into non-preemptive groups. Since this is true for each pair of representative group members, it is not possible to have a solution with fewer groups.

5.2. Simulation Design and Results

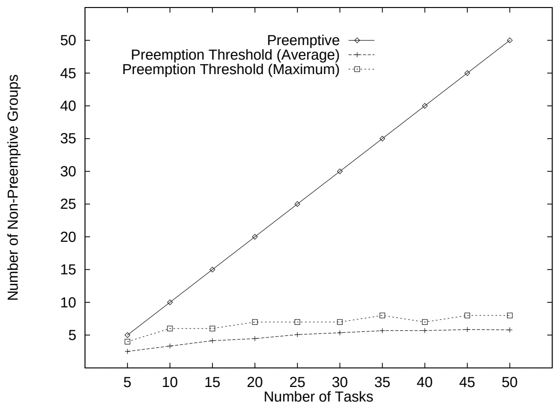
Once again, we use the randomly generated task sets used in Section 3 to study the quantitative reduction in memory requirements through the use of non-preemptive groups. For each pair of parameter values $nTasks$ and $maxPeriod$, we first choose the computation times of tasks as in Section 4, i.e., by scaling them to their breakdown utilization for preemptive scheduling. Also, we then use the algorithm in Figure 5 to assign preemption thresholds to the tasks. We then use the algorithm in Figure 7 to compute the number of non-preemptive groups, which we use as a metric.

In Figure 8 we plot the average and maximum number of non-preemptive groups obtained through our algorithm for any pair of parameter values. Figure 8(a) shows the number of non-preemptive groups as the number of tasks is varied for $maxPeriod = 10$. A similar plot is shown in Figure 8(b) for $maxPeriod = 10$. As the plots show, the number of non-preemptive groups increase much more slowly than the number of tasks (which would be the case for preemptive scheduling), indicating that as the number of tasks increase, there can be a substantial savings in memory space. For example, with $nTasks = 50$ and $maxPeriod = 10$, the number of non-preemptive group was less than 10 for all the task sets, indicating a stack space saving of more than 80% over the pure preemptive case. The plots also indicate that the number of non-preemptive groups required increases as $maxPeriod$ is raised to 100, i.e, for larger variability in task periods. Even

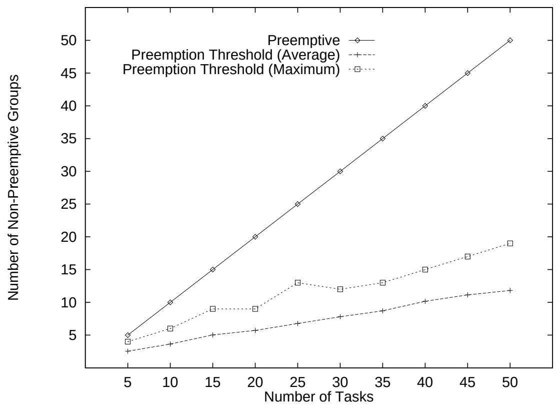
Algorithm: Partitioning Tasks into Minimum Number of Non-Preemptive Groups

```
(1)   $ngroups := 0$ ;
      /* Sort the tasks by $\gamma_i$, in non-decreasing order */
(2)   L := SortTasksbyPreemptionThreshold(TaskSet) ;
(3)   while (L != NULL) do
      /* Find the task with the smallest value of $\gamma_i$ */
(4)       $\tau_k$ := Head(L);
(5)       $G[ngroups] := \{\tau_k\}$ ;
(6)       L := L - $\tau_k$ ;
(7)       foreach $\tau_j \in$ L do
(8)           if $(\pi_j \leq \gamma_k)$ then
(9)               $G[ngroups] = G[ngroups] + \{\tau_j\}$ ;
(10)              L := L - $\tau_j$ ;
(11)          endif
(12)      end
(13)      ngroups := ngroups + 1 ;
(14)  end
```

Figure 7. An Optimal Algorithm for Partitioning a Task Set into Minimum Number of Non-Preemptive Groups



(a) Maximum Period = 10



(b) Maximum Period = 100

Figure 8. Number of Non-Preemptive Groups when using Preemption Threshold.

then, for $nTasks = 50$, the number of non-preemptive groups is less than 20 in all cases, and about 12 on the average; still a significant savings in stack space. While we do not show the plots here, the curves for larger values of $maxPeriod$ (i.e., 500 and 1000) are almost identical to that for $maxPeriod = 100$.

6. Concluding Remarks

We have studied the properties of a generalized fixed priority scheduling model with the notion of preemption threshold. The new scheduling model bridges the gap between preemptive and non-preemptive schedules and includes both of them as special cases. In doing so, 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. We present simulation results over randomly generated task sets. The simulation results show that with the new model we can substantially improve the schedulable utilization of task sets over both preemptive and non-preemptive cases.

One interesting aspect of this new scheduling model is that it achieves this better schedulability without increasing the overheads. This is in sharp contrast to dynamic priority schemes, such as earliest deadline first, which achieve higher schedulable utilizations 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. 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 introduce only enough preemptibility 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.

The use of "just-enough" preemptibility is also beneficial in reducing memory requirements. This is particularly beneficial for embedded systems where memory is a critical resource. When using preemption thresholds, tasks can be grouped into non-preemptive groups. Since there is no preemption between tasks in a group, the kernel can be modified to share the stack space between all such tasks. We show, through simulations, that the number of non-preemptive grows much slowly when the number of tasks is increased, and can lead to $60 - 80\%$ reduction in stack space requirements.

In conjunction with our earlier paper, we have developed the scheduling model using fixed priority tasks and preemption thresholds. We have developed algorithms to compute response times, check for schedulability, to assign priorities and thresholds to tasks, and to group tasks into non-preemptive groups. We have also presented quantitative evidence of significant benefits of this new scheduling model, as outlined above. Based on these results, we believe that there is compelling evidence for using this new scheduling model in real-life settings.

As a final remark, we note some of the weaknesses in this work. The main weakness of this paper is with respect to the simulation studies, for which we have not done any significance testing as yet. Nonetheless, we believe that the results shown are significant and compelling, even in the absence of rigorous statistical significance testing. Another possibility of improvement is in the algorithm to assign priorities and preemption thresholds. We presented a simple heuristic algorithm for that purpose, but believe that it should be possible to develop an algorithm that performs better.

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] 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.

[3] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] Y. Wang and M. Saksena. Scheduling fixed priority tasks with preemption threshold. In *Proceedings, IEEE International Conference on Real-Time Computing Systems and Applications*, December 1999.