Scalable Real-Time System Design using Preemption Thresholds

Manas Saksena*
manas@timesys.com

Yun Wang†
y_wang@cs.concordia.ca

Abstract

The maturity of schedulabilty analysis techniques for fixed-priority preemptive scheduling has enabled the consideration of timing issues at design time using a specification of the tasking architecture and estimates of execution times for tasks. While successful, this approach has limitations since the preemptive multi-tasking model does not scale well for a large number of tasks, and the fixed priority scheduling theory does not work well with many object-oriented design methods. In this paper we present an approach that scales well even when the design consists of a large number of concurrent jobs. The approach avoids any unnecessary preemptability in the system, thereby resulting in reduced run-time overheads from preemptions and associated context-switches. It also allows significant memory savings by grouping jobs into non-preemptive groups and then sharing the stack space between them. Our approach is based on our earlier work on scheduling using preemption thresholds that allows parametric control over preemptability in a priority based system. We show that our approach provides significant advantages over one using fixed-priority preemptive scheduling architecture. The benefits include higher schedulability for small number of tasks, and lower run-time overheads, and hence better scalability. We develop algorithms that allow design time consideration of schedulability, and automatic synthesis of an implementation model to minimize run-time overheads.

1. Introduction

Early consideration of timing issues is considered an essential aspect of real-time system design, but has not been possible until recently due to the lack of a sound mathematical basis that could support it. The maturity of schedulability analysis techniques, and in particular those based on fixed priority scheduling theory [7, 11, 12, 19] has facilitated the introduction of timing analysis in the early stages of real-time system design [2, 6, 20]. One prominent and representative example is the HRT-HOOD design method for hard real-time systems [2]. HRT-HOOD provides design abstractions that are motivated by the tasking models of fixed-priority scheduling theory, thereby providing direct support for schedulability analysis of HRT-HOOD designs. Such abstractions include the cyclic and sporadic objects, representing periodic and sporadic tasks respectively.

The approach followed in HRT-HOOD works well when the system can be decomposed into a relatively small number of periodic and sporadic tasks, and when each task is of a relatively coarse granularity. Both these limitations relate to the nature of the underlying scheduling model, i.e., preemptive multi-tasking. Preemptive multi-tasking incurs relatively high costs in context switching, and these costs become significant when the task granularity is small (since the context switching overhead is amortized over a smaller execution time) and when there is a large number of tasks (more tasks would typically result in increased context switching). In addition, there is a per-task memory cost, largely due to the need to maintain a separate stack for each task. While these run-time costs may be irrelevant in most non embedded environments, they play a significant role in many embedded real-time systems since such systems tend to be resource constrained.

Another limitation of this approach is that it doesn't work well with some of the object-oriented design methods; for example ROOM [17] and OCTOPUS [1]. Such design methods view the system as a collection of concurrent (or active) objects that cooperate in implementing system functionality. Thus, each concurrent object participates in multiple system functions, and is subject to multiple timing constraints. Moreover, to maintain internal consistency of the object, the requests on an object are processed in a "run-to-completion" manner, i.e., there is no internal concurrency within an object. In order to reduce multi-tasking costs, it is common to put an entire object (or even multiple such objects) into a sin-

*TimeSys Corporation, 4516 Henry Street, Pittsburgh, PA 15213, USA

†Dept. of Computer Science, Concordia University, Montreal, PQ H3G 1M8, Canada

0-7695-0900-2/00 $10.00 © 2000 IEEE

gle task. Such a task would receive processing requests for the object(s) in a queue and process them one by one [1, 4, 17, 18] by invoking the appropriate method to handle the request. Unfortunately, this implementation model is, in general, not analyzable by the fixed priority scheduling theory since each task in such an implementation is subject to possibly multiple timing constraints – making it difficult to find a suitable scheduling priority for the task and avoiding unbounded priority inversions [13, 16]. Methods such as HRT-HOOD circumvent this difficulty by restricting parts of the design model that have timing constraints [2].

1.1. Approach and Contributions

In this paper, we present an approach that addresses these problems. A unique aspect of our design approach is that it exploits non-preemptability, as much as possible, to reduce run-time overheads. The theoretical basis for this work is grounded in our earlier work on dual-priority scheduling that integrates and subsumes both preemptive and non-preemptive scheduling models and provides parametric control over the degree of non-preemptability in the system [22].

In our design approach, we assume that a real-time system design is given as a set of concurrent jobs with defined inter-arrival times, execution time requirements and deadlines. The scheduling model for the design requires that each job be assigned two priorities: a regular priority, and a preemption threshold (which is no less than its preemption threshold). When a job is released, its effective priority is its regular priority. Once the job is scheduled for the first time, its effective priority is raised to its preemption threshold.

It is easy to see that both preemptive and non-preemptive scheduling are special cases of this scheduling model. If the preemption threshold of each job is the same as its priority, then the model reduces to regular preemptive scheduling. On the other hand, if the preemption threshold of each job is set to the maximum priority in the system then we get non-preemptive scheduling. By choosing an appropriate preemption threshold value, we are potentially able to restrict preemptability in the system to the desired level, and thus reduce the multi-tasking costs. Moreover, by doing so, we can also potentially make task sets schedulable that are not schedulable under both preemptive and non-preemptive schedulers.

The preemption threshold for each job limits the number of (higher priority) jobs that can preempt it. Clearly, a higher value of preemption threshold for jobs will limit the number of preemptions (and thus context switches) in the system, resulting in lower run-time costs. Additionally, jobs may be grouped together to form "non-preemptive groups," i.e., a set of jobs in which no job can preempt another one in the set. Since the jobs in a non-preemptive group do not preempt each other, it is possible to achieve memory savings by running all jobs in a non-preemptive group from the same stack.

Our design approach allows us to separate out "functional" considerations from "non-functional" (in this paper, timeliness and run-time costs) considerations. When identifying jobs in the design, the designer does not need to worry about scheduling parameters, and their effect on timeliness and run-time costs. We present algorithms that automatically synthesize scheduling parameters (priorities) for jobs to meet the timeliness requirements. For a system that is deemed to be schedulable, we present algorithms to optimize the values of the scheduling parameters such that the run-time costs are minimized by merging jobs into as few non-preemptive groups as necessary to meet the timing requirements. Finally, we also show how such a design may be implemented.

Our design approach provides many benefits:

(1) We provide a seamless and automatic way to transform a design model to an implementation model, based on the timing requirements of the system. In conjunction with automatic code-generation support [9, 15], this allows for automatic translation of design models into executable implementations, avoiding error-prone and time-consuming translation, and costly fine-tuning of implementations.

(2) The approach is flexible, and applicable with a wider range of design methods and design level architectures, as compared to other approaches like HRT-HOOD [2] and MetaH [20] that are based on fixed priority preemptive multi-tasking architecture.

(3) The approach simultaneously provides higher schedulability and lower overheads. Our simulation results over randomly generated sets of jobs show that for small number of jobs, we can get significantly higher schedulability. On the other hand, with a large number of jobs, our approach scales much better and results in much lower run-time overheads.

1.2. Related Work

Preemptability is considered a necessary pre-requisite to meet timing requirements in real-time system design. Consequently, little attention has been given to non-preemptive scheduling models, except for very small static systems where some form of non-preemptive, cyclic executive scheduling can be suitably employed. While our design approach also requires a preemptive scheduling model, it exploits the fact that in most cases a preemptive priority system results in unnecessary preemptability.

Interestingly enough this observation was made several years ago by Jeffay [8], where he conjectured that "if preemption is required for feasibility, it will be limited to a few tasks." Unfortunately, this observation has not been exploited until now, and our work provides a systematic method that makes use of this observation for designing real-time systems. Perhaps more interesting is the fact that, independent of our efforts, very similar ideas have been recently proposed in [3]. Their work, as ours, uses a dual-priority scheduling model to limit preemption, and grouping of jobs into non-preemptive groups. The difference between our work and theirs is in the algorithmic details used to synthesize implementations.

2. Problem Description

In this section, we present a formal description of the problem addressed in this paper. In order to keep the focus on the essentials of our approach, we use a simple abstract design model to build our solution. We note that our approach can be extended and used in conjunction with industrial strength design methods — interested readers may refer to [21] for details.

We assume that a real-time system design is specified using a set of independent periodic or sporadic jobs, $\mathcal{T} = \{\tau_1, \tau_2, \ldots \tau_N\}$, where each job τ_i is characterized by: (1) a period (or minimum inter-arrival time) T_i, (2) an execution time budget (or worst-case execution time) C_i, and (3) a deadline D_i. We assume that these attributes are known, and given as constants.

For implementation and scheduling purposes, each job is also characterized by its scheduling attributes, which include a (nominal) priority $\pi_i \in [1, \ldots, N]$ and a preemption threshold $\gamma_i \in [\pi_i, \ldots, N]$. These attributes are not known to begin with, and must be derived to meet the timing requirements. We assume that these scheduling attributes are determined offline, and are fixed at run-time. Finally, each job is assigned to a non-preemptive group ψ_i. Again, this assignment is not known to begin with, but is determined offline, and remains fixed during run-time.

Definition 2.1 (Implementation Model) *An implementation model, denoted as $\mathcal{I}$, for a given system $\mathcal{T} = \{\tau_1, \tau_2, \ldots \tau_N\}$ is defined by a 4-tuple: $\langle M, \Psi, \Pi, \Gamma \rangle$, where, $M \in [1, \ldots, N]$ is the number of non-preemptive groups, and:*

- $\Pi : \mathcal{T} \longrightarrow [1, \ldots, N]$ *is a priority assignment for the jobs,*
- $\Gamma : \mathcal{T} \longrightarrow [\pi_i, \ldots, N]$ *is a preemption threshold assignment for the jobs, and*
- $\Psi : \mathcal{T} \longrightarrow [1, \ldots, M]$ *is a partitioning of jobs into non-preemptive groups.*

Throughout the paper, we use the abbreviated notation $\pi_i = \Pi(\tau_i)$, $\gamma_i = \Gamma(\tau_i)$, and $\psi_i = \Psi(\tau_i)$ for convenience. Also, we assume that higher values indicate higher priorities, and that priorities and preemption thresholds are in the integer set $[1, \ldots, N]$, where N is the number of jobs. Also, we assume that the preemption threshold of a job is no less than its priority, i.e., $\gamma_i \geq \pi_i$.

The partitioning of jobs into non-preemptive groups must ensure that within any non-preemptive group, the set of jobs are pairwise mutually non-preemptive based on their scheduling attributes. The following proposition gives the condition for two jobs to be mutually non-preemptive.

Proposition 2.1 *Two jobs, τ_i and τ_j, are mutually non-preemptive if $(\pi_i \leq \gamma_j) \wedge (\pi_j \leq \gamma_i)$.*

Proof: Since $\pi_i \leq \gamma_j$, τ_i cannot preempt τ_j. Likewise, τ_j cannot preempt τ_i since $\pi_j \leq \gamma_i$. □

Based on the above, we can define a valid job partitioning that precludes two jobs from being assigned to the same non-preemptive group whenever their scheduling attributes allow one of them to preempt the other.

Definition 2.2 (Valid Job Partitioning) *A job partitioning Ψ is valid if, whenever two jobs are assigned to the same non-preemptive group, then the two jobs are mutually non-preemptive. That is, in any valid job partitioning the following holds true:*

$$(\forall \tau_i)\ (\forall \tau_j) \quad (\ \psi_i = \psi_j \Rightarrow (\pi_i \leq \gamma_j) \wedge (\pi_j \leq \gamma_i)) \quad (1)$$

Likewise, we will say that an implementation model is valid, if the job partitioning in the model is valid. In the rest of the paper we will restrict our attention to valid implementation models only. Given, a valid implementation model $\mathcal{I} = \langle M, \Psi, \Pi, \Gamma \rangle$, we can compute the worst-case response time for each job. Let, $\mathcal{R}_i(\mathcal{I})$ denote the worst-case response time for τ_i under the implementation model $\mathcal{I}$.

Definition 2.3 (Feasible Implementation Model) *An implementation model $\mathcal{I}$ for a system $\mathcal{T}$ is called feasible, if it is valid, and if the worst-case response times for all jobs under the implementation model are no more than their respective deadlines. The feasibility of an implementation model is given by a predicate $feasible(\mathcal{I}, \mathcal{T})$, which is defined below:*

$$feasible(\mathcal{I}, \mathcal{T}) \stackrel{\text{def}}{=} (\forall i \in [1, \ldots, N])\ \mathcal{R}_i(\mathcal{I}) \leq D_i \quad (2)$$

Then, the schedulability of a system $\mathcal{T}$ is given by a predicate $sched(\mathcal{T})$, which is defined below:

Definition 2.4 (Schedulability) *A system is said to be schedulable if there exists a feasible implementation model for it. We use the following predicate to define schedulability of a system*

$$sched(\mathcal{T}) \stackrel{\text{def}}{=} (\exists \mathcal{I}) :: feasible(\mathcal{I}, \mathcal{T})$$

We address the synthesis problem of determining a valid and feasible implementation model that, in addition, results in minimum number of non-preemptive groups over all feasible implementation models. By minimizing the number of non-preemptive groups we minimize the stack space requirements. Additionally, we reduce the number of preemptions and thus save on context switch overheads. We address this problem by studying two inter-related problems: a feasibility problem and an optimization problem.

Feasibility Problem. *Given a set of jobs, $\mathcal{T} = \{\tau_i = \langle C_i, T_i, D_i \rangle \mid 1 \leq i \leq N\}$, find a feasible implementation model $\mathcal{I} = \langle M, \Psi, \Pi, \Gamma \rangle$, if one exists.*

Optimization Problem. *Given a schedulable set of jobs, $\mathcal{T} = \{\tau_i = \langle C_i, T_i, D_i \rangle \mid 1 \leq i \leq N\}$, find an optimal feasible implementation model $\mathcal{I} = \langle M, \Psi, \Pi, \Gamma \rangle$, such that there exists no other feasible implementation model $\mathcal{I}' = \langle M', \Psi', \Pi', \Gamma' \rangle$ with $M' < M$.*

While the two problems can be combined into a single optimization problem, we take the approach of considering them separately. The motivation for this is that, in solving the feasibility problem, we ignore the job partitioning, making it easier to solve. Moreover, we are then able to use a solution for the feasibility problem to find a solution for the optimization problem.

Additionally, we take a pragmatic approach and focus more on cost effectiveness of solutions than on their optimality. In other words, we want to find heuristic solutions that give quick, if sub-optimal (albeit close to optimal) answers. The rationale behind this is simple – the idea behind this approach is to use it early in the design process. In an iterative development process, these problems will be addressed repeatedly. In such a development scenario, it is more useful to get quick, and sufficiently close to optimal estimates, than to spend enormous amount of computation resources to get an exact optimal answer.

In solving these problems, we will ignore the overheads of implementation. Again, this is a reflection of our desire to find efficient solutions. Clearly this approach may result in optimistic feasibility assessment. An alternative, pessimistic approach is to add some overhead to the computation time of each job. Essentially, we want to ignore any difference in overheads among different implementation models for the assessment of feasibility. This assumption allows us to ignore the job-partitioning in addressing the feasibility problem.

3. Feasible Scheduling Attribute Assignment

In this section we study the feasibility problem. Since we ignore the job partitioning for this problem, the problem reduces to finding a feasible assignment for priorities and preemption thresholds for jobs.This problem was addressed in [22], where we introduced the preemption threshold scheduling model. In [22], we proposed a simple branch-and-bound search algorithm to find a feasible solution. However, our simulation results show that the algorithm is not very efficient in finding solutions, which is not surprising since the search space is large. In this paper, we revisit the problem and develop a simple and effective strategy, which, while not optimal (i.e., it may fail to find a feasible solution, even if one exists), is quite effective in practice. In this paper, we present three different algorithms to solve the feasibility problem. The algorithms increase in computational complexity, and therefore our approach is to try them in sequence. The algorithms are described in the next three sub-sections.

3.1. Algorithm A: Pre-Assigned Preemption Thresholds

Our first algorithm concentrates solely on priority assignment, and uses pre-assigned preemption threshold assignment. The rationale for this algorithm comes from the following theorem.

Theorem 3.1 *The schedulability of a job set under a fixed priority preemptive or non-preemptive scheduling model implies the schedulability of the same job set under a preemption threshold scheduling model.*

Proof: Trivial. □

The theorem implies that if we can determine feasibility of a job set under either a preemptive priority assignment, or under a non-preemptive priority assignment, then we are done. Fortunately, the problem of feasible priority assignment has been studied earlier, and an optimal priority ordering algorithm with a search space of $O(n^2)$ was presented in [19] for preemptive scheduling, which is also applicable to non-preemptive scheduling [5]. Note that we need to try both preemptive and non-preemptive priorities, as neither dominates the other [5].

3.2. Algorithm B: A Two-Stage Greedy Algorithm

Our second algorithm follows a two-stage process. In the first stage, we assign priorities using a greedy strategy. Then, in the second stage we use the priority assignment to find a preemption threshold assignment that will make the job set feasible. This algorithm takes advantage of an efficient optimal preemption threshold assignment algorithm that was presented in [22]. The algorithm takes a priority assignment as input and produces a feasible preemption threshold assignment, if there exists one for the given priority assignment. The preemption threshold assignment has a search space of $O(n^2)$. Thus, we only have to worry about the greedy priority assignment strategy, which we present next.

3.2.1. Greedy Priority Assignment

Our greedy priority assignment algorithm is based on the strategy for optimal priority assignment presented in [19]. Our algorithm uses a greedy heuristic to assign priorities to jobs, with the hope that the selected priority assignment will be feasible through an appropriate preemption threshold assignment. The greedy priority assignment, presented in Figure 1, works by dividing the job set into two parts: a sorted part (SL), consisting of the lower priority jobs, and an unsorted part (UL), containing the remaining higher priority jobs. The priorities for the jobs in the sorted list are all assigned. The priorities for the jobs in the unsorted list are unassigned, but are all assumed to be higher than the jobs in the sorted list. Initially, the sorted part is empty and all jobs are in the unsorted part. The algorithm repeatedly moves one job from the unsorted list to the sorted list, by assigning it the next higher priority. In this way, jobs are assigned priorities from lowest priority to highest priority. When considering the next candidate to move into the sorted list, each job in the unsorted list is examined in turn, and a heuristic value assigned to it. The job with the largest heuristic value is selected to move to the sorted list, and is assigned the next priority. When all jobs are in the sorted list, a complete priority ordering has been generated, and the threshold assignment algorithm can be called to assign thresholds.

We choose a simple heuristic function to select the next job for priority assignment. First, we tentatively assign the next higher priority to the job, 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 jobs have higher priority, and not their actual priorities [22]. Also, at this stage, we have not assigned preemption thresholds – so we assume that the preemption threshold of a job is the same as its priority (i.e., the pure preemptive priority case). The computed

```
Algorithm: GreedyPriorityAssignment
(1)  UL = T ; SL = {} ;
(2)  for pri = 1 to N do
(3)      foreach τ_k ∈ UL do
(4)          Π(τ_k) = pri ;   /* tentative assignment */
(5)          R_k = WCRT(τ_k);   /* compute response time */
(6)          if R_k ≥ D_k then h_k = D_k − R_k
(7)          else h_k = GetBlockingLimit(τ_k) ;
(8)          Π(τ_k) = N ;   /* reset */
(8)      end
         /* Select the job with the largest heuristic value */
(9)      Select τ_k ∈ UL s.t. (∀ℓ :: τ_ℓ ∈ UL) h_k ≥ h_ℓ
(10)     Π(τ_k) = pri ; SL = SL + {τ_k} ; UL = UL − τ_k ;
(11) end
```

Figure 1. Greedy Priority Assignment Algorithm

worst case response time is then compared with the job deadline. Now, there are two cases:

If the computed response time is less than the deadline, then this job is a "good" candidate. However, this does not guarantee that the job will be schedulable with the final assignment. This is because, in the preemption threshold assignment stage, a lower priority job may be assigned a threshold that is higher than this job, and can cause blocking. Therefore, we assign a heuristic value that is the maximum blocking that a job can tolerate while still meeting its deadline. This can be done by assigning a blocking term to the job, repeating the worst-case response time computation, and checking if it still meets the deadline.

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

3.3. Algorithm C: Simulated Annealing

Our final approach is the use of simulated annealing to find feasible scheduling attributes. Simulated annealing is a global optimization technique that attempts to find the lowest point in an energy landscape [10]. In developing this algorithm, we again make use of the optimal preemption threshold assignment algorithm. Thus,

instead of searching over all possible priority and preemption threshold assignments, we only search over the space of priority assignments. The algorithm is presented in Figure 2, and described below.

We use the deadline monotonic priority assignment as an initial starting point for the search. Simulated annealing uses the notion of "energy" of a solution, and the objective is to find a minimum energy solution. For any given priority assignment we calculate the energy of a solution by using a modified form of the optimal preemption threshold assignment algorithm. In this modified algorithm, if no preemption threshold value makes a job feasible, then its preemption threshold is set to the maximum value. The energy of a job τ_i is calculated as $Max(0, \mathcal{R}_i - D_i)$, and the energy of a solution is simply the sum of all job energies. Thus, if the energy of a job is 0, then the job is schedulable, and if the energy of a solution is 0, then the solution is feasible. Larger energy values indicate poorer solutions.

The algorithm moves from one priority assignment to the next using a randomized scheme. First a new neighbour is generated by swapping the priorities of two randomly selected jobs. If the new solution has a lower energy then it is selected as the next candidate. If not, then the neighbour is selected as a candidate probabilistically. The probability of such upward energy jumps reduces with a control parameter (C) – the temperature – which is slowly reduced. At each setting of the control parameter, the solution space is explored until a so-called thermal equilibrium is reached. In our implementation, a thermal equilibrium is reached when either the number of downward jumps exceeds $log(2 * N)$ or when the number of solutions explored exceeds N^2. At any time if we find a solution with zero energy then we stop. Otherwise, the algorithm stops when the temperature is reduced to a point where there are virtually no upward or downward jumps – indicating that no feasible solution could be found.

3.4. Performance Evaluation

To assess the suitability of these algorithms, we have evaluated their performance on randomly generated job sets. We varied two parameters in the generation of the job sets: (1) the number of jobs $nJobs$ and (2) maximum period for the jobs $maxPeriod$. For any given pair of $nJobs$ and $maxPeriod$, the job sets were generated as follows: For each job τ_i, we first randomly selected a period in the range $[1, maxPeriod]$ with a uniform probability distribution. Then, we assigned a utilization U_i in the range $[0.05, 0.5]$, again with a uniform probability distribution. The computation time of the job was then assigned as $C_i = T_i * U_i$, and the deadline was set to T_i.

We use breakdown utilization as a measure of schedulability [11]. The breakdown utilization represents the minimum job set utilization at which the system is unschedulable. The utilization of a job set is varied by scaling the computation times of all jobs, and the breakdown utilization is found using binary search over job set utilization. Since we try the three algorithms in sequence, we only see if we can improve the breakdown utilization with algorithms B and C (in some cases, they may perform worse).

Due to space limitations, we do not present detailed results of our performance evaluation. Instead we present some of our observations from the results that are of interest.

(1) Our approach can never perform worse than either pure preemptive scheduling or pure non-preemptive scheduling; this follows directly from Theorem 3.1, and the use of Algorithm A.

(2) When the number of jobs is relatively small, e.g., 5-15, we observe that in many cases significant improvement in schedulability is possible with our approach. For example with $nJobs = 10$, we are able to improve breakdown utilization by as much as 15%. To illustrate the improvement in breakdown utilization, we look at the percentage (we used 100 randomly generated job sets) of job sets for which the schedulability improvement was significant (say

```
(1)  P_old = Deadline monotonic priority ordering
(2)  C = 2 * log(Number of Jobs) * Maximum Period
       // Starting Temperature
(3)  E_old = Energy of P_old
(4)  while ((C_0 > 0.01* Minimum Period) )
(5)      while (Thermal equilibrium is not reached)
(6)          Generate P_new, a neighbour of P_old by randomly
             swapping priorities of two jobs.
(7)          E_new = Energy of P_new
(8)          if (E_new == 0) stop // We are done.
(9)          else if E_new < E_old then
(10)             P_old = P_new ; E_old = E_new ;
                 // Always take downward energy jumps
(11)         else
(12)             x = (E_old - E_new) / C_n ;
                 // Upward energy jump; take it sometimes
(13)             if (e^x <= random(0,1)) then
(14)                 P_old = P_new ; E_old = E_new ;
(15)             endif
(16)         endif
(17)     end
(18)     C = C * 0.96 ; // Temperature Cooling
(19) end
```

Figure 2. Simulated Annealing Algorithm

	Algorithm B		Algorithm B+C	
$maxPeriod$	$> 5\%$	$> 10\%$	$> 5\%$	$> 10\%$
10	17	1	45	16
100	28	3	47	14

Table 1. Percentage of job sets showing significant schedulability improvements $nJobs = 10$ **and** $maxPeriod = 10$ **and** 100.

more than 5%). In Table 1, we present the results for $nJobs = 10$, with $maxPeriod = 10$ and 100 for illustration purposes. We show the number (percentage) of jobs for which Algorithm B and C showed an improvement of more than 5 and 10% of breakdown utilization as compared to algorithm A. We can see that with our greedy algorithm, we can get a modest schedulability improvement (5-10%) in a significant percentage of job sets. With simulated annealing we get modest schedulability improvement in almost half the job sets, and get significant schedulability improvement in a modest percentage of job sets.

(3) The schedulability improvement tends to decrease as the number of jobs is increased, such that with about 50 jobs, the schedulability improvement is marginal in most cases (less than 2%). Since our algorithms are not optimal, and with large $nJobs$, running an exhaustive search is not possible, it is hard to say whether this lack of improvement is a limitation of our algorithms or whether we have reached the limitation of the scheduling model. In any case, this may be a moot point since in most cases (with larger number of jobs) we observe the breakdown utilization to be quite high – 90% or more.

4. Optimization of Implementation Model

In this section we address the optimality problem of finding an implementation model such that the jobs can be partitioned into the minimum number of non-preemptive groups over all feasible implementation models. This problem may be viewed as a search over the space of feasible implementation models. Clearly, the problem is a non-trivial combinatorial optimization problem. One difficulty in tackling this problem is how to search through the space of feasible implementation models. We use a decomposition approach to tackle this problem. We first present an optimal algorithm to find a valid job partitioning that minimizes the number of non-preemptive groups when the scheduling attributes are already known.

The optimal partitioning algorithm can be used in conjunction with the results of the feasibility problem in a straight-forward way. First, find a feasible set of scheduling attributes using the approach given in the previous section. Then, use the optimal partitioning algorithm to minimize the number of non-preemptive groups. Of course, this does not solve the original problem optimally, and indeed may be much worse than the optimal solution, since the synthesized feasible scheduling attributes were not aimed at minimizing preemption. Therefore, we use a more intelligent approach. We first synthesize a feasible set of scheduling attributes as before. Then, we refine the scheduling attributes so as to eliminate any unnecessary preemptability, while maintaining feasibility. Finally, we use the refined feasible scheduling attributes for optimal job partitioning.

4.1. Optimal Job Partitioning

We first begin with the situation when the scheduling attributes are already determined. In this case, we can partition jobs into non-preemptive groups such that the partitioning is valid (Definition 2.2). Since many valid partitionings are possible, we try to find one that assigns jobs to a minimum number of non-preemptive groups. Recall that in a valid partitioning, if two jobs are assigned to the same group then they must be mutually non-preemptive.

Definition 4.1 (Non-Preemptive Group) *A set of jobs* $G = \{\tau_1, \tau_2, \ldots, \tau_m\}$ *forms a non-preemptive group if for every pair of jobs* $\tau_j \in G$ *and* $\tau_k \in G$, τ_j *and* τ_k *are mutually non-preemptive.*

In Figure 3 we present Algorithm **OPT-Partition** that creates an optimal partitioning of jobs into non-preemptive groups. The algorithm begins by sorting the jobs in the non-decreasing order of their preemption thresholds, with ties broken arbitrarily. Let the sorted list be denoted as L. We then remove the first job (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 job and add any job 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 jobs have been examined, we have formed one non-preemptive group, with the remaining jobs in the list L. We reiterate this process of forming groups until no jobs remain in the list L. We now formally prove that the algorithm is correct (i.e., it produces a valid partitioning) and optimal (i.e., it creates the minimum number of groups).

Theorem 4.1 *Algorithm* **OPT-Partition** *produces valid partitioning of the job set into non-preemptive groups.*

```
Algorithm: OPT-Partition
(1)  m = 0 ;   /* number of groups */
     /* Sort the jobs by γi, in non-decreasing order */
(2)  L = SortJobsbyPreemptionThreshold(JobSet) ;
(3)  while (L != NULL) do
     /* Find the job with the smallest value of γi */
(4)      τk = Head(L); G[m] = {τk}; L = L - τk ;
(5)      foreach τj ∈ L do
(6)          if (πj ≤ γk) then
(7)              G[m] = G[m] + {τj} ; L = L - τj ;
(8)          endif
(9)      end
(10)     m = m + 1;   /* one more group */
(11) end
```

Figure 3. An Optimal Algorithm for Partitioning a Job Set into Minimum Number of Non-Preemptive Groups

Proof: Clearly, the algorithm creates a partitioning, i.e., each job is placed into exactly one group. Therefore, we need to show that each group formed by the algorithm is a non-preemptive group. By definition, two jobs τ_i and τ_j are mutually non-preemptive if $\pi_i \leq \gamma_j$ and $\pi_j \leq \gamma_i$. Let us look at the representative member τ_k of a group G. Since the list of jobs 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, we have $\pi_k \leq \gamma_k \leq \gamma_j$. Also, if τ_j is added to G then $\pi_j \leq \gamma_k$. Thus, for any job $\tau_j \in G$, τ_j and τ_k are mutually non-preemptive. Now, consider any two jobs τ_i and τ_j in G. We know that $\pi_i \leq \gamma_k \leq \gamma_i$ and $\pi_j \leq \gamma_k \leq \gamma_j$. It follows that $\pi_i \leq \gamma_j$ and $\pi_j \leq \gamma_i$.

Theorem 4.2 *Algorithm* **OPT-Partition** *is optimal.*

Proof: We need to show that no other partitioning into non-preemptive groups can be done with a smaller number of groups. 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, it must be the case that τ_j and τ_k are not mutually non-preemptive. Therefore, they must be in separate non-preemptive groups in any partitioning of the job 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. □

4.2. Reducing Preemptability

Now that we know how to partition a job set with feasible scheduling attributes into minimum number of non-preemptive groups, let us look at how to refine feasible scheduling attributes so as to reduce the number of groups created by Algorithm **OPT-Partition**. For this purpose, we use a simple heuristic strategy – we attempt to reduce any unnecessary preemptability that is introduced by the scheduling attributes.

Note that the set of higher priority jobs that can preempt a job τ_i is determined by the job's preemption threshold γ_i. By increasing the value of γ_i, we can reduce the number of jobs that can preempt τ_i. Suppose we increase the preemption threshold of τ_i from a to b $(a < b)$, then this change may result in increased response times for any job τ_k, if $a < \pi_k \leq b$, since any such job may now incur a blocking from τ_i. We can safely increase the preemption threshold if the recomputed worst-case response times of these affected jobs are still no more than their deadlines.

Using the idea given above, we try to increase the preemption threshold of each job to the maximum value that will still keep the job set schedulable. Figure 4 gives the algorithm that attempts to assign larger preemption threshold values to jobs. The algorithm considers one job at a time, starting from the highest priority job, 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 job to ensure that the system stays schedulable. By going from highest priority job to the lowest priority job, we ensure that any change in the preemption threshold assignment in latter (lower priority) jobs cannot increase the assignment of a former (higher priority) job, and thus we only need to go through the list of jobs once.

4.3. Performance Evaluation

To evaluate the performance of our approach, we again used randomly generated job sets, as described in Section 3, and then used our algorithms to generate a feasible implementation model, minimizing the number of non-preemptive groups. Our results show that in many cases, we can reduce the number of non-preemptive groups significantly as compared to the preemptive scheduling case, where each job forms its own non-preemptive group. In Figure 5, we show the number of groups generated by our approach when the number of jobs is varied from 10 to 100, with $maxPeriod = 100$. To do this, we first found the breakdown utilization with preemptive scheduling. We plot both the average and the maximum number of groups produced by our approach; for comparison the straight line shows the number of groups with a purely preemptive approach. Similar results are achieved with other parameters, but are not shown here due to lack of space.

As the plots show, the number of groups increase much

Algorithm: Assign Maximum Preemption Thresholds

```
/* Assumes that job priorities are fixed,
and a set of feasible preemption thresholds are assigned */
(1)  for (i = n down to 1)
(2)      while (schedulable == TRUE) && (γ_i < n)
(3)          γ_i += 1;   /* try a larger value */
(4)          Let τ_j be the job such that π_j = γ_i.
             /* Calculate the worst-case response time of job j
             and compare it with deadline */
(5)          R_j = WCRT(τ_j);
(6)          if (R_j > D_j) then
(7)              schedulable = FALSE ; γ_i -= 1;
(8)          endif
(9)      end
(10)     schedulable = TRUE
(11) end
```

Figure 4. Algorithm for Finding Maximum Preemption Threshold

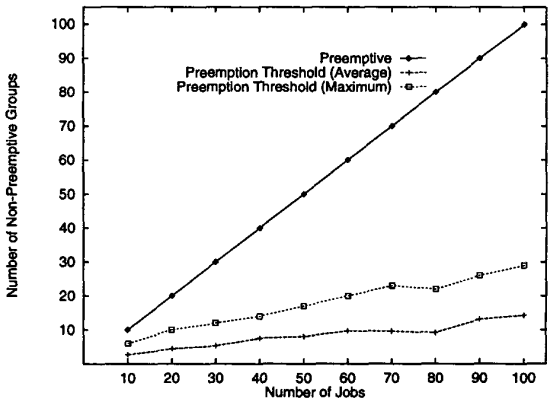

Figure 5. Number of Non-Preemptive Groups as a function of number of jobs, with $maxPeriod = 100$

more slowly than the number of jobs (which would be the case for preemptive scheduling), indicating that as the number of jobs increase, there can be substantial reduction in run-time overheads. For example, with $nJobs = 100$, we have less than 30 groups in all cases, and on an average only 14.3 groups.

5. Implementation Architecture

A successful use of our design approach requires that it be possible to efficiently implement a system constructed using this approach. This implies that the implementation must be consistent with the scheduling model and that the implementation must allow for jobs in a non-preemptive group to share the same run-time stack. The preemption threshold scheduling model is straightforward to implement in a real-time kernel, and is in fact available in the ThreadX kernel[1]. Sharing of run-time stack from multiple jobs is perhaps trickier and requires that jobs do not have any state on the stack between multiple instances. The SSX kernel[2] uses a single-shot execution model and thus naturally allows the sharing of stack between jobs in a non-preemptive group.

We present here an alternate implementation architecture that can be used with most real-time kernels that support fixed priority preemptive scheduling in a relatively straightforward manner. In this implementation architecture, job arrivals are viewed as events. Each non-preemptive group is implemented as an event handling thread. A thread maintains an event queue where arriving events are queued. The queued events are processed in a *run-to-completion* manner, that is, processing of an event is not preempted by the arrival of another event on the thread's event queue. If there are no queued events, the thread blocks itself awaiting new event arrivals. In this way, jobs in a non-preemptive group share the thread's stack and are scheduled by the thread's event handling loop in a non-preemptive manner.

The event queue for each thread is maintained as a priority queue, using event priorities. Therefore, whenever a thread selects the next event to process, it is always the highest priority queued event on the thread's event queue. To ensure that this implementation is consistent with the scheduling model assumed in our approach, the thread priorities are dynamically managed as follows:

- When an event is queued at a thread, then the thread's priority is set to the maximum of its current priority, and the priority of the event being queued,
- When a thread removes an event from its event queue to process, the thread priority is set to the preemption threshold of the event,
- When a thread finishes processing an event, it changes its priority to the highest priority pending event in its event queue.

When thread priorities are dynamically managed as above, it is easy to show that at all times, the event (job) with the highest effective priority is executing on the CPU – as assumed in our scheduling model. We omit a formal proof due to lack of space.

[1] http://www.threadx.com
[2] http://www.realogy.com

6. Discussion and Concluding Remarks

We have presented a design approach that allows us to automatically synthesize an efficient and feasible implementation for a system design specified as a set of periodic and sporadic jobs. The design approach is based on our earlier work on preemption threshold scheduling model that allows parametric control over the degree of preemptability in priority based systems. Using this model, we are able to synthesize implementations that have only as much preemptability as necessary to meet the timing requirements.

An interesting benefit of integrating preemptive and non-preemptive scheduling is the resultant higher schedulability in many cases. More importantly, by controlling and eliminating any unnecessary preemptability, we can generate implementations with lower run-time overheads from preemptions and associated context switches. It also allows us to group jobs into non-preemptive groups that can then be run using a single shared stack. We show how such a system can be implemented in traditional real-time kernels that employ fixed priority scheduling.

While we used a very simple design model in this paper – using independent periodic and sporadic jobs – our approach is motivated by our earlier work in integrating commercial strength object-oriented design methods and schedulability analysis techniques [14, 15] and is a logical continuation of that work. The simplified design model enabled us to focus on the essentials of our approach, rather than getting entangled in the many details that need to be accommodated with a more complex design model. An extension of this work for more general models (such as those used in industrial strength object-oriented methods) can be found in [21].

References

[1] M. Awad, J. kuusela, and J. Ziegler. *Object-Oriented Technology for Real-Time Systems: A Practical Approach using OMT and Fusion.* Prentice Hall, 1996.

[2] A. Burns and A. J. Wellings. HRT-HOOD: A Design Method for Hard Real-Time. *Real-Time Systems*, 6(1):73–114, 1994.

[3] R. David, N. Merriam, and N. Tracey. How Embedded Applications using an RTOS can stay within On-chip Memory Limits. In *Proceedings, Euromicro Conference on Real-Time Systems, Work-In-Progress Session*, June 2000.

[4] B. P. Douglass. *Doing Hard Time: Developing Real-Time Systems with Objects, frameworks, and Patterns.* Addison-Wesley, 1999.

[5] L. George, N. Rivierre, and M. Spuri. Preemptive and Non-Preemptive Real-Time Uni-Processor Scheduling. Technical Report N° 2966, INRIA, France, sep 1996.

[6] H. Gomaa. *Software Design Methods for Concurrent and Real-Time Systems.* Addison-Wesley Publishing Company, 1993.

[7] 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.

[8] K. Jeffay. Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems. In *Proceedings, IEEE Real-Time Systems Symposium*, December 1992.

[9] P. Karvelas. Schedulability Analysis and Automated Implementation of Real-Time Object-Oriented Design Models. Master's thesis, Concordia University, May 2000.

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[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, Dec. 1989.

[12] 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.

[13] M. Saksena, P. Freedman, and P. Rodziewicz. Guidelines for Automated Implementation of Executable Object Oriented Models for Real-Time Embedded Contol Systems. In *Proceedings, IEEE Real-Time Systems Symposium*, pages 240–251, December 1997.

[14] M. Saksena and P. Karvelas. Designing for Schedulability: Integrating Schedulability Analysis with Object-Oriented Design. In *Proceedings, Euromicro Conference on Real-Time Systems*, June 2000.

[15] M. Saksena, P. Karvelas, and Y. Wang. Automatic Synthesis of Multi-Tasking Implementations from Real -Time Object-Oriented Models. In *Proceedings, IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, March 2000.

[16] M. Saksena, A. Ptak, P. Freedman, and P. Rodziewicz. Schedulability Analysis for Automated Implementations of Real-Time Object-Oriented Models. In *Proceedings, IEEE Real-Time Systems Symposium*, December 1998.

[17] B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling.* John Wiley and Sons, 1994.

[18] S. A. Stolper. Streamlined Design Approach Lands Mars Pathfinder. *IEEE Software*, September 1999.

[19] 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.

[20] S. Vestal and P. Binns. Scheduling and Communication in MetaH. In *Proceedings, IEEE Real-Time Systems Symposium*, 1993.

[21] Y. Wang. *Real-Time System Design using Preemption Thresholds.* PhD thesis, Concordia University, Montreal, 2000.

[22] Y. Wang and M. Saksena. Fixed Priority Scheduling with Preemption Threshold. In *Proceedings, IEEE International Conference on Real-Time Computing Systems and Applications*, December 1999.