Reducing the Number of Preemptions in Real-Time Systems Scheduling by CPU Frequency Scaling

Abhilash Thekkilakattil[1], Anju S Pillai[2], Radu Dobrin[1], Sasikumar Punnekkat[1]
[1]Mälardalen Real-Time Research Center
Mälardalen University, Sweden
firstname.lastname@mdh.se
[2]Amrita School of Engineering
Amrita Vishwa Vidyapeetham, India
s_anju@cb.amrita.edu

Abstract

Controlling the number of preemptions in real-time systems is highly desirable in order to achieve an efficient system design in multiple contexts. For example, the delays due to context switches account for high preemption overheads which detrimentally impact the system schedulability. Preemption avoidance can also be potentially used for the efficient control of critical section behaviors in multi-threaded applications. At the same time, modern processor architectures provide for the ability to selectively choose operating frequencies, primarily targeting energy efficiency as well as system performance. In this paper, we propose the use of CPU Frequency Scaling for controlling the preemptive behavior of real-time tasks. We present a framework for selectively eliminating preemptions, that does not require modifications to the task attributes or to the underlying scheduler. We evaluate the proposed approach by four different heuristics through extensive simulation studies.

1 Introduction

Preemptions in real-time scheduling may cause undesired high processor utilization, high energy consumption and, in some cases, even infeasibility. The preemption cost includes the direct costs to perform the context switches [10] and to manipulate the task queues [4, 10], as well as the indirect cost of cache-related preemption delays [13, 20]. The pessimism in many schedulability analysis methods could be reduced if an efficient control of the critical section behaviors can be established and preemption eliminations are ideally suited for achieving this.

Preemptive Fixed Priority Scheduling (FPS) has been extensively analyzed since the work of Liu and Layland [14], and is used in a large number of applications, mostly due to its flexibility and simple run-time overhead. In practice, however, preemptive FPS may imply large preemption related overheads and the need for preemption control is well recognized [18, 4, 17]. Buttazzo [5] showed that the rate monotonic algorithm (RM) introduces a higher number of preemptions than earliest deadline first algorithm (EDF).

Many techniques towards eliminating/minimizing preemptions have been proposed in literature [7, 23, 21, 19, 22, 12]. Most of the work focuses on reassigning task attributes like release times, deadlines, priorities etc and cannot be applied to those real-time tasks for which the task attributes such as priority, release times and deadlines reflect strict timing constraints. Alternative choices have to be developed for such systems where task attributes cannot be modified due to the inherent nature of the application involved. Butazzo has identified Quality of Service (QoS) [6] as one of the important research areas in future real time computer systems. Current methods for preemption elimination may reduce the quality of service as they change task attributes, which can result in an increased jitter or reduced levels of service due to late execution of tasks. Thus, removing preemptions without affecting QoS and without any modifications to the task attributes would be the ideal one from a system designer's perspective. At the same time, reducing the number of preemptions can also be beneficial from an energy point of view in systems with demands on low power consumption. When a task is preempted, there is a great probability that its contents in the cache will be lost. When the execution of the task is again resumed it will cause a lot of energy consuming accesses to off-chip memory. An access to off-chip memory is typically 10-100 times more expensive than an on-chip cache access in terms of energy consumption. Reducing the number of preemptions will reduce these additional expensive memory accesses due to reduced cache pollution.

Traditionally, Dynamic Voltage and Frequency Scaling techniques have been used for reducing energy consumption by slowing down tasks' executions [1, 3, 15, 16]. This is effective in reducing the energy consumption according to the relation $P = CV^2F$, where P is the power consumed by the processor, V is the applied voltage, C is the effective

capacitance and F is the operating frequency. This means that the power dissipation increases/decreases linearly with frequency and quadratically with the applied voltage.

In this paper we apply CPU Frequency Scaling theory to control the preemption behavior in real-time system scheduling. We propose an offline method that identifies the maximum number of preemptions in a given schedule, and provides specific frequencies at which task instances need to be executed such that the preemptions are avioded. Our method is capable of guaranteeing a significantly lower number of preemptions without altering the original task attributes or modifying the underlaying scheduler. While executing tasks at a higher processor frequency may result in an increased energy demand, our method is capable of providing trade-offs between the number of preemptions and the overall energy consumption. While the methodology can be easily applied to any existing scheduling policy, in this paper we present an instantiation to FPS.

The main contributions of this paper consist of a) a formal analytical model to detect and eliminate preemptions using CPU Frequency Scaling by first detecting a preemption and then finding the minimum sufficient frequency that guarantees the completion of the preempted task before the release of the higher priority tasks, as well as b) a framework to study the effect of change in frequency at which task instances execute, over the rest of the schedule.

The rest of the paper is organized as follows: Section 2 describes the related work and in section 3 we give an overview of our system model. In Section 4 we describe our methodology illustrated by a simple example in Section 5. In Section 6 we present the experimental evaluation results and in Section 7 we discuss some important issues related to this paper. Finally, in Section 8 we present the conclusions and future work.

2 Related Work

Several methods have been proposed in the past to reduce the number of preemptions in real-time scheduling. Preemption Threshold Scheduling (PTS) for FPS was proposed by Wang and Saksena [21, 19], showing that this method improves schedulability as well as reduces the number of preemptions and the number of threads in the system. In [21] the authors describe an optimal algorithm to assign preemption threshold by iterating over the solution and attempting to assign the largest feasible preemption threshold values to tasks such that the task set remains schedulable. The results show that large threshold values reduce the probability of preemptions and therefore should result in less preemptions. However, this approach results in a dual priority system which may not be directly suitable for, e.g., legacy systems, where scheduler modifications may not be possible.

The integration of real time synchronization schemes into PTS was proposed [11], where the authors integrate priority inheritance protocol and priority ceiling protocol into PTS. The authors have proposed two integrated schemes- in the first approach, instead of priority, the preemption threshold of a blocked task is inherited when blocking occurs; in the second approach, the priority ceiling is used instead of preemption threshold. The results show that the integrated schemes can minimize worst-case context switches and are appropriate for the implementation of real-time object-oriented design models.

Gai et al. [8] extend this scheduling model to EDF priority assignment and show that it can reduce the memory requirements of the system. In [9], the authors have presented an approach to combine PTS with Dynamic Voltage Scaling (DVS) to enable energy efficient scheduling. PTS decreases the number of context switches among tasks as well as the memory requirement in the system. Furthermore, the authors describe a dynamic slack reclamation technique, in conjunction with PTS, that yields energy gains depending on the available slack.

A method to integrate preemption threshold to FPS under DVS scheduling algorithms, was proposed in [22], where two preemption-aware algorithms, ccFPPT and FPPT-WD, are studied. ccFPPT is a cycle conserving fixed priority preemption scheduling, which slows down every task instance in its cycle or working range by the same amount. All the slack times are used to slowdown the processor speed. FPPT-WDA is the FPPT- Work Demand Analysis which is more complex compared to ccFPPT. The key feature of an online WDA DVS method is to postpone the release of the tasks as much as possible. Here, most of the slack time will be used for the first several tasks that discover these times leaving very tight, or even no scale down at all, for other tasks that arrive later.

In [12], the authors present two techniques that can reduce the increased number of preemptions introduced by using DVS algorithms. The first method is an accelerated completion based technique, where the main idea is to shorten the completion time of a low priority task before the arrival of a high priority task by accelerating its execution. The second approach is a delayed preemption based control technique, in which the activation point of a high priority task is delayed so that a scheduled low priority task can complete its execution without the preemption.

In an earlier work [7], we have proposed a method that analyzes offline a set of periodic tasks scheduled by FPS, and identifies the maximum number of preemptions that can occur at run time. It then reassigns task attributes, such as the task priority, period and offsets, without affecting the schedulability of the task set, while attaining a significantly lower number of preemptions. This is achieved at the cost of increased number of tasks and/or reduced task execution

flexibility.

While the existing approaches have substantially advanced the state of the art in the field, all have either introduced potential infeasible costs or have focused on energy conservation when applying DVS. In this paper we propose the use of CPU frequency scaling to control the preemptive behavior in real-time scheduling without requiring modifications of the existing task attributes or to the underlaying scheduler.

3 System Model

3.1 Task Model

We assume a uniprocessor system implementing a preemptive fixed priority scheduling policy. We consider a periodic task set $\Gamma = \{\tau_1, \tau_2, ...\tau_n\}$ where task τ_i has a period T_i, a priority P_i, and a relative deadline D_i. The tasks are sorted in decreasing priority order, i.e., P_1 is the highest priority and P_n is the lowest. The hyperperiod of the tasks is defined by LCM representing the least common multiple of the task periods. Each task instance $\tau_{i,l}$ is characterized by a worst case execution requirement $C_{i,l}$, $i \in [1, n]$ and $l \in [1, \frac{LCM}{T_i}]$, at a discrete CPU frequency $F_p \in [F_{min}, F_{max}]$, where F_{min} and F_{max} are the minimum and maximum frequency respectively, as imposed by the hardware constraints. We assume that the tasks are initially executed at a default frequency supported by the hardware.

Additionally, we denote the release time of the l^{th} instance of task τ_i by $rel_{i,l}$, its corresponding actual start time by $start_{i,l}$, and its finishing time by $finish_{i,l}$. In the description of our method, we assume that the offsets are zero and the deadlines are equal to the periods. However, this restriction can be easily extended for non-zero offsets and deadlines shorter than periods. Finally, we assume that the tasks do not suspend themselves.

3.2 Energy Model

We consider a power-aware processor which can operate in a set of discrete operating modes identified by $M = \{m_1, m_2, m_3, ...m_p\}$, where each m_i is characterized by $m_q = (F_q, w_q)$, where F_q is processor frequency and w_q is the power (in watts) consumed by the processor in mode m_q [3]. We assume a negligible frequency-switch overhead, which may occur only in conjunction with a scheduling decision.

The total energy consumed by the system over the period of LCM can be represented as:

$$E_{LCM} = \sum_{i=1}^{n} \sum_{l=1}^{\frac{LCM}{T_i}} C_{i,l} \times w_{i,l}^{q} \quad (1)$$

where $w_{i,l}^{q}$ is the power consumed by the processor while executing the task instance $\tau_{i,l}$ in mode m_q at frequency F_q.

3.3 Execution Time Model

The execution time of a task instance is inversely proportional to the clock frequency at which the instance is executed, and can be represented as:

$$C_{i,j}^{1} = \frac{C_{i,j}^{max}}{F_1} \times F_{max}$$

where F_1 is the frequency which gives an execution time of $C_{i,j}^{1}$ and $C_{i,j}^{max}$ is the execution time obtained at F_{max}. This implies that,

$$F_1 = \frac{C_{i,j}^{max}}{C_{i,j}^{1}} \times F_{max} \quad (2)$$

Similarly to obtain an execution time of $C_{i,j}^{2}$ we require a frequency of:

$$F_2 = \frac{C_{i,j}^{max}}{C_{i,j}^{2}} \times F_{max} \quad (3)$$

Dividing the equation 2 by 3, we get:

$$\frac{F_1}{F_2} = \frac{C_{i,j}^{2}}{C_{i,j}^{1}}$$

which gives,

$$F_2 = \frac{C_{i,j}^{1}}{C_{i,j}^{2}} \times F_1 \quad (4)$$

This equation gives the frequency required for scaling $C_{i,j}^{1}$ to $C_{i,j}^{2}$. We have used this equation to derive the maximum frequency necessary to ensure a required worst case execution time for a particular task instance. This model is derived from the model presented in [15].

4 Methodology

In this paper we apply CPU Frequency Scaling theory to control the preemption behavior in fixed priority schedules. We propose an offline method that identifies the maximum number of preemptions in a given schedule, and provides specific frequencies at which task instances need to be executed such that the number of preemptions is reduced.

A preemption typically occurs when a higher priority task instance is released during the execution of a lower priority task instance. One way to avoid the preemption is to make sure that the preempted task instance completes its execution before the release of the higher priority one. As CPU Frequency Scaling can be used to speed up or

slow down task execution times within a specified range, our method attempts to provide the minimum sufficient frequencies per task instance that guarantees preemption elimination.

In our offline preemption analysis we assume that tasks execute for their WCET. However, at run-time, tasks will most likely execute for less than WCET, implying a different number of preemptions compared to the ones detected by our off-line method. Hence, we divide the preemptions in two major categories:

Initial preemptions – are detected in the off-line analysis assuming task executions equal to their WCET, i.e., a high priority task instance is *initially preempting* a low priority task instance (Figure 1).

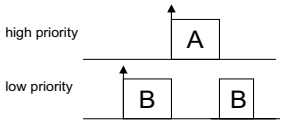

Figure 1. An offline detected *initial preemption*

Potential preemptions – that occur at run-time due to task executions less than WCET. In Figure 2 a) we can see that if tasks execute for WCET, no preemption will occur. However, in this situation we consider task A *potentially* preempting task B since, if task C, that delays the execution of B, is executing for less than its WCET, then B can start executing earlier, i.e., before the release time of A, and will actually be preempted by A (Figure 2 b)).

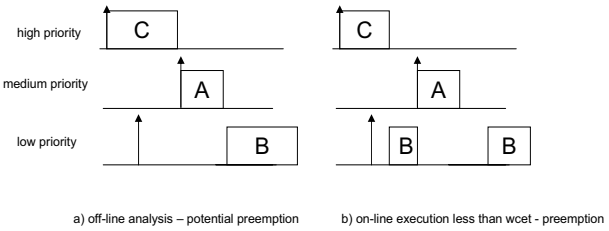

Figure 2. An off-line detected *potential preemption*

In this paper we focus on the offline part of the methodology and, thus, we do not explicitly address the elimination of potential preemptions that would mostly benefit of the use of online mechanisms, and is the aim for future work. However, as later illustrated by the evaluation results, a large number of potential preemptions are automatically eliminated in the process of eliminating initial preemptions. At the same time, future work will address the use of online mechanisms for slowing down tasks at runtime, to ensure that the remaining potential preemptions are not converted to actual preemptions, as well as to compensate for the increase in energy for removing initial preemptions.

Our approach to eliminate a particular preemption is performed in two steps: preemption identification followed by the calculation of the minimum sufficient frequency at which the preempted task instance needs to execute in order to guarantee the preemption elimination. Obviously, the such frequency needs to be available, i.e., if the required frequency may not exceed the maximum available one, i.e., F_{max}, otherwise the preemption cannot be eliminated.

As the problem of finding the set of individual task instance frequencies to minimize the number of preemption for a given set of tasks is NP-hard, the significance of offline analysis lies in the fact that complex algorithms can be used to remove preemptions, which can complement the efforts to remove initial preemptions online.

In this paper we investigate and compare four different heuristics with respect to the order in which the preemptions are eliminated. We have examined the following four possibilities:

1. HPF – highest priority preempted task first
2. LPF – lowest priority preempted task first
3. FOPF – first occurring preemption first (under LCM)
4. LOPF – last occurring preemption first (under LCM)

In each of the approaches we attempt to eliminate the preemptions recursively until all preemptions are eliminated, or no feasible frequencies can be found for the remaining ones. Note that a preemption that cannot be eliminated at a particular stage, may be eliminated at a later iteration point in the algorithm, due to earlier completion of interfering tasks in the schedule.

4.1 Preemption Identification

We say that a task instance $\tau_{i,l}$ *initially* preempts another task instance $\tau_{j,k}$ if four conditions hold simultaneously [7]:

1. $\tau_{i,l}$ has a higher priority than $\tau_{j,k}$,
2. $\tau_{i,l}$ is released after $\tau_{j,k}$,
3. $\tau_{i,l}$ starts executing after the start time of $\tau_{j,k}$
4. $\tau_{j,k}$ finishes its execution after the release time of $\tau_{i,l}$

In case of nested preemptions we consider only the cases where a context switch occurs.

4.2 Preemption elimination

To eliminate a single preemption, e.g., $\tau_{i,l}$ preempts $\tau_{j,k}$, we identify the new frequency at which the preempted instance must execute, by calculating the execution reduction that guarantees its completion before the release of the higher priority instance, i.e,:

$$finish_{j,k}^{new} \leq rel_{i,l}$$

Where, $finish_{j,k}^{new}$ is the new finishing time of the preempted instance after preemption elimination.

Theorem 4.1. *Given a preemption where $\tau_{i,l}$ preempts $\tau_{j,k}$, the worst case execution time of $\tau_{j,k}$ that guarantees the preemption avoidance is given by the relation:*

$$C_{j,k}^{new} = finish_{j,k}^{new} - start_{j,k} - I_{j,k} \quad (5)$$

The interference $I_{j,k}$ is given by:

$$I_{j,k} = \sum_{\forall l \in hp(j)} \sum_{x=1}^{\lceil \frac{LCM}{T_l} \rceil} \Psi(j,k,l,x) \times C_{l,x}$$

where

$$\Psi(j,k,l,x) = \begin{cases} 1, & if\ start_{j,k} < start_{l,x} < finish_{j,k}^{new} \\ 0, & otherwise \end{cases} \quad (6)$$

Proof. The $finish_{j,k}$ for a task instance $\tau_{j,k}$ is obtained by adding its execution time and the interference caused due to preemptions by higher priority tasks to its start time:

$$finish_{j,k} = start_{j,k} + C_{j,k} + \sum_{\forall \tau_{l,x} \in \Gamma_2'} C_{l,x}$$

Where Γ_2' is the set of all higher priority task instances released between the start time and finish time of $\tau_{j,k}$. Γ_2' can be found by a recursively checking whether any higher priority task instances start between the start time and the latest computed finish time of $\tau_{j,k}$ with the $finish_{j,k}$ initially set to $(start_{j,k} + C_{j,k})$ .Thus, we rewrite the above equation as:

$$finish_{j,k} = start_{j,k} + C_{j,k} + \sum_{\forall l \in hp(j)} \sum_{x=1}^{\lceil \frac{LCM}{T_l} \rceil} \Psi(j,k,l,x) \times C_{l,x} \quad (7)$$

where

$$\Psi(j,k,l,x) = \begin{cases} 1, & if\ start_{j,k} < start_{l,x} < finish_{j,k} \\ 0, & otherwise \end{cases}$$

with $finish_{j,k}$ set to $start_{j,k} + C_{j,k}$ initially.

We rewrite (7) as:

$$finish_{j,k} = start_{j,k} + C_{j,k} + I_{j,k}$$

Where $I_{j,k}$ is given by:

$$I_{j,k} = \sum_{\forall l \in hp(j)} \sum_{x=1}^{\lceil \frac{LCM}{T_l} \rceil} \Psi(j,k,l,x) \times C_{l,x}$$

and $\Psi(j,k,l,x)$ is given by the earlier equation. Now rearranging the terms we get:

$$C_{j,k} = finish_{j,k} - start_{j,k} - I_{j,k}$$

Here we substitute the new required finish time $finish_{j,k}^{new}$ such that the preemption on $\tau_{j,k}$ by $\tau_{i,l}$ is eliminated:

$$C_{j,k} = finish_{j,k}^{new} - start_{j,k} - I_{j,k}$$

□

After calculating the new execution time required to eliminate a single preemption, we check whether it is possible to speed up the execution of the task instance to guarantee this execution time by checking whether the corresponding frequency range is within the CPU permitted range. This is done by first calculating the required CPU frequency, denoted by F_r, using the formula:

$$F_r = \frac{C_{j,k}^{cur}}{C_{j,k}^{new}} \times F_q$$

where $C_{j,k}^{new}$ is the execution time of k^{th} instance of task τ_j to finish before it is preempted by a higher priority task, and $C_{j,k}^{cur}$ is its execution time before removing the preemption, when executing at a frequency F_q. The calculated F_r is approximated to the nearest discrete value among the values which the processor can attain, and the old frequency is retained if $F_r \notin [F_{min}, F_{max}]$.

Finally, we need to investigate the impact of the preemption elimination on the rest of the schedule by recalculating the start times and finish times of all lower priority task instances, according to the equations 8 and 11, when

$$F_r \in [F_{min}, F_{max}]$$

Theorem 4.2. *The start time of any task instance $\tau_{j,k}$ is given by,*

$$start_{j,k} = max(f_{hp}(j,k), rel_{j,k}) \quad (8)$$

where,

$$f_{hp}(j,k) = max_{\forall l \in hp(j)}(finish_{l,\lceil \frac{f_{hp}(j,k)+1}{T_l} \rceil}) \quad (9)$$

and, initially,

$$f_{hp}(j,k) = rel_{j,k} \quad (10)$$

Proof. According to our assumption, a task instance $\tau_{j,k}$ starts its execution if it is released, and after all high priority tasks in the ready queue have finished execution. This has two cases,

Case 1 : The ready queue is empty at $rel_{j,k}$ and no higher priority tasks are released simultaneously or are currently executing

Case 2 : There exists at least one high priority task instance the is released, or is currently executing at $rel_{j,k}$

The value computed by $\frac{f_{hp}(j,k)+1}{T_l}$ will give the latest instance number of all higher priority tasks τ_l. Using this instance number, equation 9 will return the maximum of the finish times of the corresponding high priority task instances. This is done recursively until a single value is obtained.

Consider Case 1, where no high priority tasks are executing/released or in the ready queue at $rel_{j,k}$. The value computed by 9 will be less than $rel_{j,k}$, since the latest of the higher priority task instances would have already completed. So equation 8 will return $rel_{j,k}$ as the start time of $\tau_{j,k}$. Hence, the equation 8 holds for Case 1.

Consider Case 2, where there exists at least one higher priority task that is currently executing at the time when $\tau_{j,k}$ is released. The value computed by 9 will be greater than $rel_{j,k}$, since 9 computes the latest of the finish times of all high priority tasks that are released in the busy period before $\tau_{j,k}$ starts executing. This finish time is the start time of $\tau_{j,k}$ as we have assumed that no task can suspend itself. Hence the equation 8 also holds for Case 2. □

Finally, we calculate the finish time of $\tau_{j,k}$.

Theorem 4.3. *The finish time for a task instance $\tau_{j,k}$ is given by the equation:*

$$finish_{j,k} = start_{j,k} + C_{j,k} + I_{j,k} \tag{11}$$

$I_{j,k}$ is given by:

$$I_{j,k} = \sum_{\forall l \in hp(j)} \sum_{x=1}^{\lceil \frac{LCM}{T_l} \rceil} \Psi(j,k,l,x) \times C_{l,x} \tag{12}$$

where $\Psi(j,k,l,x)$ is given by the equation:

$$\Psi(j,k,l,x) = \begin{cases} 1, & if\ start_{j,k} < start_{l,x} < finish_{j,k} \\ 0, & otherwise \end{cases} \tag{13}$$

Proof. The proof is similar to the one of theorem 4.1. □

Recalculation of the start times and finish times aims to investigate the impact of one preemption elimination on the rest of the schedule, i.e., whether any new preemptions have been introduced or any additional "old" preemptions have been removed. Additionally, a schedulability test is performed in order to ensure the task completions before their deadlines.

$$\forall i \in [1,n], j \in [1, \frac{LCM}{T_i}], finish_{i,j} \leq (j-1) \times T_i + D_i$$

In this paper we use 4 different heuristics, i.e., HPF, LPF, LOPF, FOPF, to recursively eliminate the preemptions in a given set of tasks, schedulable by preemptive FPS, until all preemptions are eliminated or no feasible solutions can be found for the remaining ones.

5 Example

We illustrate the proposed preemption reduction method with a simple example. We assume a set of tasks as described in the Table 1 scheduled according to the rate monotonic scheduling policy, using a default frequency of 40 MHz provided by the hardware. The time used in the example is expressed in milliseconds (ms). In this example, our method identifies 7 initial preemptions that may occur at run time (Figure 3) when the tasks execute for their worst case execution times.

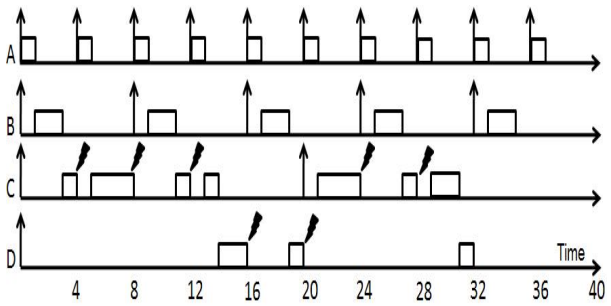

Figure 3. Original RM schedule

For explaining how a single preemption is detected, eliminated, and its effects over the rest of the schedule, we describe the removal of the preemption of C_1 by A_4. The preemption is detected as it satisfies the following condition:

$$\{P_A > P_C\} \wedge \{rel_{A,4} > rel_{C,1}\} \wedge \{start_{A,4} > start_{C,1}\}$$
$$\wedge \{finish_{C,1} > rel_{A,4}\}$$

To eliminate the preemption, C_1 needs to complete before the release of A_4.

$$finish^{new}_{C,1} \leq rel_{A,4} = 12$$

Task	Time period	Execution Time	Priority
A	4	1	1(highest)
B	8	2	2
C	20	6	3
D	40	4	4

Table 1. Example: task set

Mode	0	1	2	3	4	5
Frequency(MHz)	0	5	30	40	50	80
Power Consumption(mW)	0	20	50	50	200	500

Table 2. Example: CPU operating modes

Consequently, the new execution time for C_1 is calculated using the equation 5:

$$C_{C,1} = 12 - 3 - (1 + 1 + 2) = 5.$$

At this point, we need to check the possibility of eliminating this preemption by ensuring that the corresponding frequency is within the permissible range. For the analysis we take a variable frequency processor having different operating modes as described in Table 2 [3].

We find the frequency at which the task instance C_1 must execute to eliminate it being preempted by A_4 using equation 4.

$$F_2 = \frac{6}{5} \times 40 = 48$$

This is approximated to 50 MHz which is the next highest frequency supported, which can guarantee this execution time. C_1 will execute for 4.8 ms when it is run at 50 MHz.

Eliminating this particular preemption will affect the lower priority task's start times and finish times. Hence, we re-calculate the start times and finish times of all the lower priority task instances based on the newly calculated execution and finish time of C_1, according to the equation 11:

$$finish_{C,1} = 3 + 4.8 + (1 + 1 + 2) = 11.8$$

We calculate the start time of D_1 using equation 8:

$$\begin{aligned} f_{hp}(D,1) &= rel_{D,1} = 0, initially \\ f_{hp}(D,1) &= max(finish_{A,\lceil \frac{0+1}{4} \rceil}, finish_{B,\lceil \frac{0+1}{8} \rceil}, \\ &\quad finish_{C,\lceil \frac{0+1}{20} \rceil}) \\ &= max(finish_{A,1}, finish_{B,1}, finish_{C,1}) \\ &= max(1, 3, 11.8) = 11.8 \end{aligned}$$

Since $0 \neq 11.8$, we recursively calculate the new value for the start time for D_1 until we reach a fixed point. Here the start time of D_1 is 11.8. The newly computed value of the finish time of D_1 is:

$$finish_{D,1} = 11.8 + 4 + (1 + 1 + 2) = 19.8$$

It is now possible to do a schedulability analysis on the finish times of the task instances or find the total number of preemptions and take a decision on whether or not to eliminate this preemption. After the removal of the preemption of C_1, the total number of preemptions in the schedule is reduced to 6 (figure 4). We use this process to eliminate

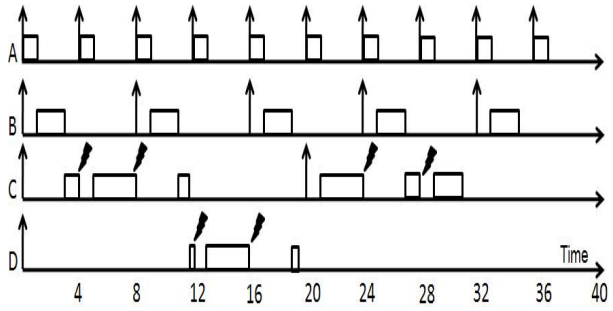

Figure 4. RM schedule after eliminating one preemption

preemptions according to the last occurring preemption first strategy (LOPF) as described in section 4, until no more preemptions can be eliminated. Figure 5 shows the resulting schedule eliminating preemptions in the reversed order of their occurrences in the schedule. The number of initial preemptions is reduced from 7 to 2, with a cost of 2.7 times increase in energy, according to equation 1. Finally,

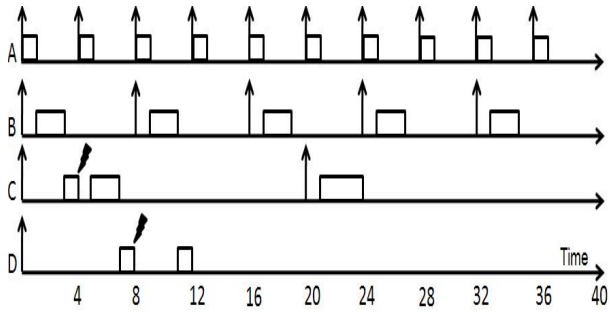

Figure 5. RM schedule after reducing preemptions using LOPF

the frequencies for all the task instances are computed and illustrated in table 3.

Instance	1	2	3	4	5	6	7	8	9	10
τ_A	40	40	40	40	40	40	40	40	40	40
τ_B	40	40	40	40	40	-	-	-	-	-
τ_C	80	80	-	-	-	-	-	-	-	-
τ_D	80	-	-	-	-	-	-	-	-	-

Table 3. Derived frequencies for each task instances

6 Performance evaluation

We performed a number of experiments to evaluate the efficiency of our proposed method. We used synthetic tasks with randomly generated attributes, schedulable by FPS. We studied the effect of the removal of preemptions in different orders. We generated task sets with utilizations ranging from 0.6 to 1.0 using the UUniFast [2] algorithm that were used to compare the efficiency of the different approaches. The tasks priorities were assigned according to the RM policy. Each set consisted of 5 to 15 tasks respectively, with time periods ranging from 5 to 1500. For the purpose of obtaining integer values of execution times, we assumed that the calculated CPU frequency is supported by the processor. However we assumed that the tasks cannot be scaled to a value less than 60% of their actual execution times i.e., any value above 60% of the original execution time was deemed acceptable, and those below unacceptable.

6.1 Experiment 1

In this scenario, we experimented the preemption elimination *based on the task priority order*. We performed rwo different runs for each taskset. In the first run we eliminated the preemptions starting with the one incurred by the first instance of the highest priority task to the last instance of the lowest priority task i.e., highest priority first (HPF), in the order $\{\tau_{1,1}, \tau_{1,2}, ...\tau_{1,\frac{LCM}{T_1}}\}$, $\{\tau_{2,1}, \tau_{2,2}, ...\tau_{2,\frac{LCM}{T_2}}\}$, $....\{\tau_{n,1}, \tau_{n,2}, ...\tau_{n,\frac{LCM}{T_n}}\}$. In the second run, we eliminated the preemptions starting from the first instance of the lowest priority task to the last instance of the highest priority task i.e, lowest priority first (LPF), in the order $\{\tau_{n,1}, \tau_{n,2}, ..\tau_{n,\frac{LCM}{T_n}}\}$, $..\{\tau_{2,1}, \tau_{2,2}, ..\tau_{2,\frac{LCM}{T_2}}\}$, $\{\tau_{1,1}, \tau_{1,2}, ..\tau_{1,\frac{LCM}{T_1}}\}$.

6.2 Experiment 2

In this experiment we eliminated the preemptions in the *order of their occurrence in the schedule*. We conducted two runs, where in the first run we removed preemptions from the first occurring preemption to the last (FOPF) and in the second run from the last occurring preemption to the first (LOPF). Our simulations results for the four heuristics used in the 2 experiments are illustrated in Figure 6.

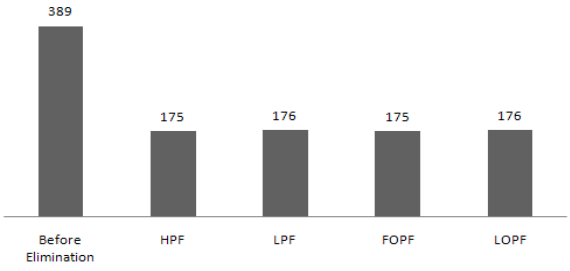

Figure 6. Average number of *initial preemptions* after preemption elimination

We also observed that a significant number of potential preemptions are also eliminated automatically in the process of removing initial preemptions (Figure 7). In this case, LPF and LOPF performed slightly better with respect to reducing the number of potential preemptions.

Figure 7. Average number of *potential preemptions* after preemption elimination

In some cases, some preemptions of medium priority tasks are automatically eliminated which will not result in a reduction of execution times of those medium priority tasks. However while eliminating preemptions in the LOPF, the preemptions which are removed automatically in the other three cases are detected and eliminated first. This result in a reduction in execution times of these medium priority task instances. As a result of this the preemptions (both initial and potential) of the lower priority tasks are reduced since they complete earlier due to this reduction in execution times of medium priority tasks.

LOPF fares slightly better than LPF in our simulations. This is because in LPF, the preemption that is removed first need not be the last preemption in the timeline. Removal of

such preemptions might result in automatic removal of preemptions occurring later in time without reducing execution times of task instances. This can result in low priority tasks completing later than as observed in LOPF.

6.3 Energy Consumption

The elimination of a preemption caused a 4.2 times increase in energy consumption when using LOPF. We found that for tasksets with high utilizations, the increase in energy was more prominent. Naturally, tasksets with a large number of tasks also showed a high increase in energy as this can be attributed to the high number of preemptions in these task sets. However, our proposed approach provides for trade-off between the number of preemptions and the energy consumption, as the user can selectively choose which preemptions are desirable to eliminate.

7 Discussion

So far, in our methodology we have not addressed two issues:

1. Speeding up tasks in the busy period before the start of the preempted task to eliminate the preemption.
2. Explicit removal of potential preemptions (although, as shown in the experiments, many of them are eliminated automatically when eliminating initial preemptions).

Consider a preemption where $\tau_{i,l}$ preempts $\tau_{j,k}$. It can be either a potential preemption or an initial preemption. In order to address both the cases described above, we must find the set Υ where,

$$\Upsilon = \tau_{j,k} \cup \{\sum_{p=1}^{n} \sum_{q=1}^{LCM/T_p} busy\_period(i,l,p,q) \times \tau_{p,q}\}$$

where, $busy\_period(i,m,p,q)$ returns 1 if $\tau_{p,q}$ is in the busy period just before $start_{i,l}$. Now we need to find $C_{a,b}$ for each $\tau_{a,b}$ in Υ such that $\forall \tau_{a,b} \in \Upsilon$:

$$\begin{aligned} finish_{a,b} &< (b-1) \times T_a + D_a, \; and \\ finish_{a,b} &< start_{i,m} \end{aligned}$$

Speeding up task executions in the busy period raises two issues:

1. One issue is finding the best execution times for each $\tau_{a,b}$ such that all of them finish before $start_{i,m}$ while meeting their individual deadlines. This is an optimization problem and has to be performed for each preemption elimination, as speeding up tasks may not be the best option to remove a preemption. It may also be that slowing down tasks in the busy period such that the preempted task starts after the preempting task can be a valid alternative. We plan to address this question in the future work by incorporating energy reduction and minimization of preemptions into the goal function of an optimization problem.
2. Eliminating potential preemptions by scaling up individual task execution times can result in a drastic increase in energy consumption. Hence, an attractive solution may be to remove potential preemptions at run time by slowing down tasks to ensure that the potential preemptions are not converted to actual preemptions. This approach again has the additional advantage of compensating the increased energy consumption due to the removal of preemptions by speeding up tasks.

8 Conclusions and Future Work

In this paper, we have proposed a methodology to reduce the number of preemptions in real-time scheduling by using CPU frequency scaling. We have provided an instantiation to FPS by analyzing a schedulable task set and calculating individual frequencies at which task instances need to execute such that the preemptions are eliminated, by taking into account the effect of preemption elimination on the rest of the schedule. The proposed approach does not imply modifications to the task attributes or the underlaying scheduler.

As the main element of cost introduced by our method is the energy consumption, the proposed framework provides for tradeoff between the number of preemptions and the cost by keeping track of the increase of energy required for each preemption elimination. Though runtime variations in the execution time of task instances can introduce (or remove) additional preemptions, the offline method can be complemented with online approaches by enabling the use of efficient algorithms to remove/minimize preemptions.

Future work will focus on optimizing the approach such that the number of preemptions is minimized while minimizing the energy consumption, as well as online extensions to cope with execution variations between best and worst case, including cache related preemption delay cost. At the same time, the method will be extended to the sporadic task model.

9 Acknowledgment

The authors wishes to thank the anonymous reviewers for their useful comments on the paper. This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research center PROGRESS

and the Erasmus Mundus External Co-operation Window programme EURECA.

References

[1] H. Aydin, R. Melhem, D. Moss, and P. Meja-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 53(5):584–600, 2004.

[2] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems Journal*, 30(1-2):129–154, 2005.

[3] E. Bini, G. C. Buttazzo, and G. Lipari. Minimizing CPU energy in real-time systems with discrete speed management. *ACM Transaction on Embedded Computer Systems*, 8(4), 2009.

[4] A. Burns, K. Tindell, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulers. *IEEE Transactions on Software Engineering*, 21(5):475–480, 1995.

[5] G. Buttazzo. Rate monotonic vs. EDF: Judgment day. In *Proc. 3rd ACM International Conference on Embedded Software*, Philadephia, USA, Oct 2003.

[6] G. Buttazzo. Research trends in real-time computing for embedded systems. *SIGBED Rev.*, 3(3):1–10, 2006.

[7] R. Dobrin and G. Fohler. Reducing the number of preemptions in fixed priority scheduling. In *16th Euromicro Conference on Real-time Systems (ECRTS 04)*, July 2004.

[8] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, pages 73–83, 2001.

[9] R. Jejurikar and R. K. Gupta. Integrating processor slowdown and preemption threshold scheduling for energy efficiency in real time embedded systems. In *Proceedings of the IEEE RTCSA*, 2004.

[10] D. I. Katcher, H. Arakawa, and J. K. Strosnider. Engineering and analysis of fixed priority schedulers. *IEEE Trans. Softw. Eng.*, 19(9):920–934, 1993.

[11] S. Kim, S. Hong, and T.-H. Kim. Integrating real-time synchronization schemes into preemption threshold scheduling. *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, page 0145, 2002.

[12] W. Kim, J. Kim, and S. L. Min. Preemption-aware dynamic voltage scaling in hard real-time systems. In *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, pages 393–398, New York, NY, USA, 2004. ACM.

[13] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Transactions on Computers*, 47(6):700–713, 1998.

[14] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[15] M. Marinoni and G. Buttazzo. Elastic DVS management in processors with discrete voltage/frequency modes. *IEEE Transactions on Industrial Informatics*, 3(1):51–62, 2007.

[16] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 89–102, New York, NY, USA, 2001. ACM.

[17] K. Ramamritham and J. A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. In *Proceedings of the IEEE*, pages 55–67, 1994.

[18] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemption points. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 212–224, Washington, DC, USA, 2006. IEEE Computer Society.

[19] M. Saksena and Y. Wang. Scalable multi-tasking using preemption thresholds. In *In Digest of Short Papers For Work In Progress Session, The 6th IEEE Real-Time Technology and Application Symposium*, 2000.

[20] J. Schneider. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. *Real-Time Systems Symposium, IEEE International*, page 195, 2000.

[21] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Sixth International Conference on Real-Time Computing Systems and Applications, 1999. RTCSA '99.*, pages 328–335, 1999.

[22] L. Yang, M. Lin, and L. T. Yang. Integrating preemption threshold to fixed priority DVS scheduling algorithms. *International Workshop on Real-Time Computing Systems and Applications*, pages 165–171, 2009.

[23] G. Yao, G. Buttazzo, and M. Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. *International Workshop on Real-Time Computing Systems and Applications*, pages 351–360, 2009.