Preemption Control using CPU Frequency Scaling in Real-time Systems

Abhilash Thekkilakattil, Radu Dobrin and Sasikumar Punnekkat

Mälardalen Real-Time Research Center
Mälardalen University, Västerás, Sweden
{*abhilash.thekkilakattil, radu.dobrin, sasikumar.punnekkat*}*@mdh.se*

Abstract: Controlling the preemption behavior in real-time systems can have beneficial impacts in multiple contexts as it can decrease the processor utilization, reduce the energy consumption or even enable the schedulability of the system. In this paper we study the preemption behavior of sporadic task systems scheduled using the Fixed Priority Scheduling (FPS) policy, and evaluate the feasibility of preemption control using CPU frequency scaling. We show that offline preemption control using CPU frequency scaling is difficult for sporadic task systems, and we propose an online heuristic algorithm, of linear complexity, to control the number of preemptions in a sporadic task system. Evaluation results show that online CPU frequency scaling is an attractive approach for preemption control in sporadic task systems.

Keywords: Real-time systems, Real-time preemption control, Power awareness

1. INTRODUCTION

Preemptive and non-preemptive fixed priority real-time scheduling have been widely studied during the past decades. While preemptive FPS, is generally considered to achieve higher processor utilization while guaranteeing the tasks' schedulability, it suffers from a number of preemption-related costs, e.g., undesired processor utilization, high energy consumption and, in some cases, even infeasibility. Preemption costs may also lead to unpredictable variations in task execution times. Though the task execution times can be safely determined using static timing analysis, e.g., Byhlin et al. (2005), at runtime, the preemption overheads (e.g., cache related preemption delays) can lead to variations in the Worst Case Execution Times (WCET) (Ramaprasad and Mueller (2008)). Hence, the unpredictable variations may have detrimental impacts in terms of schedulability, which is not acceptable in most of the real-time systems today. Preemptive FPS also requires the use of resource access protocols to achieve mutual exclusion, in cases where tasks communicate through shared resources. These resource access protocols, though predictable, introduce schedulability overheads to the system, as well as lead to pessimistic assumptions in the schedulability tests. Reducing the number of preemptions can also be beneficial from an energy point of view in systems with stringent requirements 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.

⋆ This work was partially supported by the Swedish Research Council project CONTESSE (2010-4276).

A non-preemptive scheduling scheme on the other hand does not allow the full utilization of the available processor time, as a low priority task cannot be interrupted by a more *urgent* higher priority task, which will result in a lower utilization bound in order to guarantee the schedulability of the tasks. Thus, one of the costs of using a non-preemptive scheduler is the loss of utilization which comes from the blocking of higher priority tasks by the lower priority tasks (Baruah (2005)). However, it should be noted that non-preemptive scheduling can be infeasible even for arbitrarily low utilizations. The advantage, however, of using a non-preemptive scheduling algorithm in real-time systems is its low runtime overhead. Also, the WCET assumptions based on the static analysis on task executions do not change, as the tasks are allowed to complete their executions once they start executing, without, e.g., context-switch or cache-related variations.

Modern processors support Dynamic Voltage and Frequency Scaling (DVS) which can be used to increase or decrease tasks' execution times by manipulating the CPU frequency at which they are executed. DVS techniques have been traditionally used for energy conservation by reducing the supplied voltage and, thus, lowering the CPU frequency. This, however, increases the task execution times, which, in its turn, potentially increases the number of preemptions in the schedule. The ability to scale up/down the CPU frequency provides us with the possibility of manipulating task execution times to achieve various goals.

Sporadic task model proposed by Mok (1983) and Baruah et al. (1990) is a widely studied task model. In this task model, each task is characterized by a minimum inter-arrival time between any two consecutive occurrences. The sporadic task model is adopted in many common real-time systems, e.g., in systems where the events occurring in the system are separated by a known minimum inter-arrival time. The periodic task model (Liu and Layland (1973)) is a special case of a sporadic task model where every occurrences of the task is separated by exactly the minimum inter-arrival time. Thus, the sporadic task

model is a more general model that is useful in the analysis of many real-time systems.

We explore the use of CPU frequency scaling for preemption control in sporadic task systems. We show that offline preemptions control using frequency scaling in a sporadic task system is difficult as it requires information which is not available until runtime. Consequently, we present an online algorithm, of linear complexity, for preemption control in sporadic task systems, and evaluate its performance through simulations using synthetic tasksets. The evaluation results show that online CPU frequency scaling is an attractive as well as affordable approach towards controlling the preemption behavior of sporadic tasks in FPS.

The paper is organized as follows. In section 2 we discuss the related work. Section 3 details the system model and the various notations used throughout this paper. In section 4, we discuss the possibility of preemption control in sporadic task systems using CPU frequency scaling, where we present an online algorithm to control the preemption behavior, followed by an example in section 5. We conclude our paper in section 7 after presenting our evaluations in section 6.

2. RELATED WORK

The need for preemption elimination is widely recognized in the literature and has been discussed in many works (Bui et al. (2008), Burns et al. (1995), Katcher et al. (1993), Ramaprasad and Mueller (2008)). The preemption cost includes the direct costs to perform the context switches (Katcher et al. (1993)) and to manipulate the task queues (Burns et al. (1995), Katcher et al. (1993)), as well as the indirect cost of cache-related preemption delays (Lee et al. (1998), Schneider (2000)). Bui et al. (2008) observed that, in an extreme case, cache related preemption delays can lead to as large as 33% increment in task execution time on a PowerPC MPC7410 with a 2 MB two way associative L2 cache. Ramaprasad and Mueller (2006) analyzed the effects of cache related preemption delays and provided an accurate analysis of the data cache behavior. They showed that the critical instant does not necessarily occur when all the tasks are released simultaneously when considering preemption delays.

Preemptive Fixed Priority Scheduling (FPS) has been extensively analyzed since the work of Liu and Layland (1973), 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 in literature (Burns et al. (1995), Ramamritham and Stankovic (1994), Ramaprasad and Mueller (2006)). Buttazzo (2003) showed that the rate monotonic algorithm (RM) introduces a higher number of preemptions than earliest deadline first algorithm (EDF). He noted that this increase in the number of preemptions can significantly increase the overheads in the system reducing the benefits of its simple runtime implementation.

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 first introduced in the ThreadX operating system (Lamie (1997)), which was later formalized by Wang and Saksena (1999), showing that this method improves schedulability and reduces the number of preemptions and the number of threads in the system. Wang and Saksena (1999) 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 by Kim et al. (2002), where the authors integrate priority inheritance protocol and priority ceiling protocol into PTS. 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. (2001) extend this scheduling model to EDF priority assignment and showed that it can reduce the memory requirements of the system. Jejurikar and Gupta (2004) presented an approach to combine PTS with DVS to enable energy efficient scheduling. Traditionally, Dynamic Voltage and Frequency Scaling (DVS) techniques have been used for reducing energy consumption by slowing down tasks' executions (Aydin et al. (2004), Bini et al. (2009), Marinoni and Buttazzo (2007), Pillai and Shin (2001)). 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. However, since there is an increase in task execution times, the number of preemptions increase significantly.

Baruah (2005) proposed an algorithm to calculate the length of the longest possible non-preemptive execution of a task in a sporadic task system, scheduled by EDF. Earlier, Baruah et al. (1990) studied the feasibility of preemptively scheduling sporadic task systems on a uniprocessor and proposed an algorithm that runs in pseudo-polynomial time for most task sets. Yao et al. (2010), evaluated the various limited preemption methods. Earlier Yao et al. (2009) presented a method to find an upper bound on the length of the largest possible non-preemptive execution of a task in fixed priority schedules and presented extensive simulation results. Bertogna et al. (2010) presented an optimal method to place preemption points under a fixed preemption overhead. The evaluations demonstrated the advantage of limited preemption models over non-preemptive systems and fully preemptive systems with preemption costs, in feasibly scheduling task sets. However, the schedulable task set ratio decreases with increase in utilization, indicating that deferred preemption models and limited preemption models could be ineffective at high utilizations levels.

In an earlier work by Dobrin and Fohler (2004), a method has been proposed to analyze offline a set of periodic tasks scheduled by FPS, and to identify 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. We (Thekkilakattil et al. (2010)) later proposed an offline method to control preemption behavior in FPS using CPU frequency scaling. This was done by finding

the minimum sufficient frequency at which a task instance must execute such that its non-preemptive execution is guaranteed.

In this paper, we extend our previous work (Thekkilakattil et al. (2010)) to sporadic task systems (Mok (1983), Baruah et al. (1990)) and study the feasibility of controlling preemption behavior in sporadic task systems, scheduled by a fixed priority scheduling scheme, using CPU frequency scaling. We provide an overview on the effectiveness of using the possibility of changing task execution frequencies to achieve preemption control. We show that pure offline preemption control using frequency scaling is not suitable for the sporadic task model, and we propose an, online algorithm of linear complexity, that takes advantage of the frequency scaling abilities provided by modern processors for preemption control in sporadic task systems scheduled by FPS.

3. SYSTEM MODEL

3.1 Task model

In this paper we build on the sporadic task model introduced by Mok (1983) and Baruah et al. (1990). We consider a set of tasks Γ= $\{\tau_1, \tau_2, ...\tau_n\}$, where each τ_i has a minimum inter-arrival time T_i, a worst case execution requirement denoted by C_i, a priority P_i and a deadline D_i, relative to the start of its period. It should be noted that T_i is also referred to as the period of τ_i. Moreover, we assume that C_i, which we define as the worst case number of clock cycles required for each task, is independent of the clock frequency and is a constant (Melhem et al. (2004)). Note that when we consider a task instance $\tau_{i,j}$, $C_{i,j}$ represent the time required to execute the C_i clock cycles at a given processor frequency. We assume that the task set is schedulable under a work conserving Fixed Priority Scheduling (FPS) algorithm and the tasks are ordered such that $P_i < P_{i+1}$. A work conserving algorithm never idles the processor when there are outstanding computations to be executed. We denote the highest priority task τ_i by the lowest P_i value. Each task is assumed to be mapped to a unique priority. LCM represents the Least Common Multiple of the time periods of all tasks in the task set.

The release time of a sporadic task instance $\tau_{j,k}$ can be represented by the following equation:

$$rel_{j,k} \geq s_j + (k-1)T_j$$

We remove the inequality by accounting for the offsets in the release time of all $k-1$ instances of τ_j,

$$rel_{j,k} = s_j + (k-1)T_j + \sum_{a=2}^{k} \phi_{j,a}$$

where $\phi_{j,a}$ is the offset in the release time of $\tau_{j,a}$. The offset in the release time of $\tau_{j,k}$ arises from two variables: s_j and $\sum_{a=2}^{k} \phi_{j,a}$. We add the initial offset s_j to $\sum_{a=2}^{k} \phi_{j,a}$ and thus obtain:

$$rel_{j,k} = (k-1)T_j + \sum_{a=1}^{k} \phi_{j,a} \tag{1}$$

Let $start_{j,k}$ and $finish_{j,k}$ represent the start time and finish time of $\tau_{j,k}$ respectively.

3.2 Energy Model

We consider a processor model which supports a set of discrete operating modes $M = \{m_1, m_2, m_3, ...m_p\}$, where each m_q is characterized by $m_q = (F_q, W_q)$. Here, F_q is the processor frequency associated to mode m_q, and W_q is the set $W_q = \{w_q^1, w_q^2, .., w_q^r\}$, representing the power consumed per clock cycle by r resources used by the tasks in mode m_q. Let the maximum frequency supported by the processor be F_{max}. We assume a known upper-bound on the frequency-switch overhead, which may occur only in conjunction with a scheduling decision.

The total energy consumption due to task executions over the period of time until time t can be represented as:

$$E_t = \sum_{i=1}^{n} \sum_{l=1}^{k} e_{i,l} \tag{2}$$

where k is given by the smallest integer satisfying:

$$(k+1)T_i + \sum_{d=1}^{k} \phi_{i,d} \geq t$$

and,

$$e_{i,l} = \sum_{b=1}^{C_i} \left\{ \sum_{a=1}^{r} w_q^a \right\}$$

Here, m_q is the execution mode of the processor during the b^{th} clock cycle of $\tau_{i,l}$. In other words, $e_{i,l}$ is the sum of the actual power consumption of all the clock cycles, which gives the total energy used for the execution of the task instance $\tau_{i,l}$. The task instances are initially assumed to execute at the minimum frequency supported by the hardware.

3.3 Execution Time Model

A linear relationship between frequency and execution time of a task instance is considered. The *execution time* of a task instance $\tau_{i,j}$, denoted by $C_{i,j}$, is inversely proportional to the frequency at which the processor executes $\tau_{i,j}$. If X is the processor frequency when $\tau_{i,j}$ executes, then its execution time $C_{i,j}$ is given by:

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

where $C_{i,j}^{max}$ is the execution time obtained at F_{max}. This implies that,

$$X = \frac{C_{i,j}^{max}}{C_{i,j}} \times F_{max} \tag{3}$$

Similarly, let, $\tau_{i,j}$ execute for a time of $C'_{i,j}$ when it execute at a frequency, X':

$$X' = \frac{C_{i,j}^{max}}{C'_{i,j}} \times F_{max} \tag{4}$$

Dividing the equation 3 by 4, we get:

$$\frac{X}{X'} = \frac{C'_{i,j}}{C_{i,j}}$$

which gives,

$$X' = \frac{C_{i,j}}{C'_{i,j}} \times X \tag{5}$$

This equation gives the frequency required for scaling $C_{i,j}$ to $C'_{i,j}$. 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 by Marinoni and Buttazzo (2007). The speed of the processor when it executes at the default frequency is assumed to be 1. Thus, if $C_{i,j}$ is the default execution time of $\tau_{i,j}$, $\frac{C_{i,j}}{C'_{i,j}}$ gives the speed at which the processor must execute to complete $\tau_{i,j}$ in $C'_{i,j}$ time units.

4. METHODOLOGY

In this section, we examine the possibility of using CPU frequency scaling to control the preemption behavior in sporadic task systems scheduled by FPS. We first show that an offline approach to control the preemption by using CPU frequency scaling cannot provide non-preemption guarantees in a sporadic task system. Then, we propose an online algorithm of linear complexity that reduces the number of preemptions as well as provides for tradeoffs between the number of preemptions and the overall energy consumption. Throughout this section, we define clairvoyance as the full knowledge about the release times of the sporadic task instances.

4.1 Offline preemption reduction

In this section, we prove that offline preemption control by using CPU frequency scaling cannot provide non-preemption guarantees in a sporadic task system.

Proposition 4.1. Clairvoyance is unachievable in a sporadic real-time task system

Similar to the traditional schedulabiltity tests, like for example discussed by Baruah et al. (1990), it is impossible to have a temporal window that can simulate the preemption behavior that will effectively reflect the preemption behavior of the entire sporadic task set. It is impractical from an implementation point of view to calculate the release behavior of the sporadic task instances. There are two main reasons for this: 1) the release times, typically, depend on several runtime events and 2) even if it was possible to predict the run time events and map them to task release times, the temporal and spatial computational complexity would render it impossible to store and process the information, for an infinite number of task instances.

Lemma 4.1. There exist no non-clairvoyant offline scheme that can determine the frequency at which a sporadic task instance must execute such that its run-time non-preemptiveness is guaranteed.

Proof. Assume that there exist a non-clairvoyant offline scheme S that determine a frequency X, at which the processor must execute $\tau_{j,k}$ to avoid a preemption.

It is known that:

$$C_{j,k} \propto \frac{1}{X}$$

The finish time of $\tau_{j,k}$ is given by,

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

Consider any $\tau_i \in hp(j)$. The release time of any task instance l of τ_i is given by

$$rel_{i,l} = (l-1)T_i + \sum_{a=1}^{l} \phi_{i,a}$$

where $\sum_{a=1}^{l} \phi_{i,a}$ is the offset in the release of $\tau_{i,l}$. Now l and $\sum_{a=1}^{l} \phi_{i,a}$ can take a value at runtime such that,

$$start_{j,k} < (l-1)T_i + \sum_{a=1}^{l} \phi_{i,a} < finish_{j,k}$$

Then, $\tau_{i,l}$ preempts $\tau_{j,k}$ contradicting our assumption. Thus there exist no non-clairvoyant offline scheme that can determine the frequency at which a task instance must execute such that its run-time non-preemptiveness is guaranteed.

Lemma 4.2. There exist no offline derived frequency for a sporadic task instance that guarantees its non-preemptive run-time execution.

Proof. Assume that there exist an offline derived frequency, X, at which the processor must execute $\tau_{j,k}$ such that its non-preemptive execution is *guaranteed*. It is known that:

$$C_{j,k} \propto \frac{1}{X}$$

Thus as,

$$X \to \delta$$
$$\Rightarrow C_{j,k} \to \varepsilon$$

where δ is a very large value and ε is a very small value. The finish time of $\tau_{j,k}$, if it executes without being preempted, is given by,

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

It is always possible that there exist a $\tau_{i,l}$ such that,

$$start_{j,k} < (l-1)T_i + \sum_{a=1}^{l} \phi_{i,a} < finish_{j,k}$$

If $\tau_{j,k}$ is executed at an infinite frequency,

$$X \to \alpha$$

the execution time,

$$C_{j,k} \to 0$$

That is, the preemption can be avoided if the task instance is executed by the processor at an infinite frequency. This however is impossible as $X \to \alpha$ is not feasible.

Theorem 4.1. There exists no offline method that guarantees the run-time non-preemptiveness of a given sporadic task instance.

Proof. From the proposition 4.1, we rule out the possibility of clairvoyance for an offline analysis in sporadic task systems.

According to lemma 4.1, there exist no non-clairvoyant offline scheme that can determine the frequency at which a sporadic task instances must execute such that its run-time non-preemptiveness is guaranteed.

It is clear from lemma 4.2 that there exist no offline derived frequency for a sporadic task instance that guarantees its non-preemptive run-time execution.

Thus there exists no offline frequency scaling method that guarantees the run-time non-preemptiveness of a given sporadic task instance.

4.2 Online preemption reduction

A key characteristic of the release time behavior in a sporadic task system is that, at any given time, it is only possible to know the minimum amount of time during which the next instance of a particular task will *not* be released. It is impossible to know the time at which an instance of the task will be *actually* released. Consequently, it is only possible to find the maximum time, if any, for which a task instance is *guaranteed* to execute non-preemptively considering the minimum time interval required for the release of the next instance among the higher priority tasks. Thus, we aim to find the maximum time for which a task instance is *guaranteed* to execute non-preemptively, whenever it starts its execution. This time is obtained from the earliest time in the future at which one of the higher priority tasks will be released. This is formally presented as algorithm 1.

Algorithm 1 Find the minimum frequency at which $\tau_{j,k}$ must execute, at a time $start_{j,k} = t$, to avoid it being preempted by the next higher priority task instance released.

$minT \leftarrow 9999999(a\ large\ value)$
$i \leftarrow 1$
while $i < j$ **do**
  if $minT > next\_rel_i$ **then**
    $minT \leftarrow next\_rel_i$
  end if
  $i \leftarrow i + 1$
end while
$C'_{j,k} \leftarrow minT - t$
if $C'_{j,k} > 0$ **then**
  $X' \leftarrow \frac{C_{j,k}}{C'_{j,k}} \times X$
  if $X' > F_{max}$ **then**
    $X' \leftarrow F_{max}$
  end if
else
  $X' \leftarrow F_{max}$
end if

This algorithm is executed at the start time of all the task instances. Consider the k^{th} instance of a task τ_j starting its execution at a time instant t i.e., $start_{j,k} = t$. The maximum amount of time for which $\tau_{j,k}$ is *guaranteed* to execute non-preemptively is the difference between the earliest among the next release times of the higher priority task instances and the current time t. If $minT$ is the earliest time in the future at which one of the higher priority task instances of $\tau_{j,k}$ can be released, the maximum time, $C'_{j,k}$ for which $\tau_{j,k}$ can execute non-preemptively is given by,

$$C'_{j,k} = minT - t$$

This information is used to calculate the frequency at which the processor must execute $\tau_{j,k}$ to ensure its non-preemptive execution (i.e., the frequency required to obtain an execution time of $C'_{j,k}$). $C'_{j,k}$ takes a negative value when the earliest release time of a high priority task instance has already passed implying that the current task can be preempted at any time instant in the future. If no available frequency can guarantee its non-preemptive execution, $\tau_{j,k}$ is executed at the maximum frequency until the release time of the higher priority task. This is because, the probability that a sporadic task instance is released increases over time. Thus, earlier the low priority task instance completes its execution, the less probable that it will be preempted by a higher priority task.

Let $next\_rel_i$ represent the end of the period of the latest instance of τ_i that has finished execution at time t, $\tau_i \in hp(j)$. This is obtained by adding T_i to the release time of its latest instance that has finished execution at time t. Thus $next\_rel_i$ is the time instant at which the next release of τ_i can occur, i.e., no instance of τ_i can occur before $next\_rel_i$ at time t. We can easily see that $minT$ represents the minimum among the $next\_rel_i$ for all $\tau_i \in hp(j)$ i.e.,

$$minT = \min_{\forall \tau_i \in hp(j)} (next\_rel_i)$$

After finding $minT$, the earliest possible release time in the future among all the higher priority task instances at time t, it is possible to find the frequency at which the processor must execute $\tau_{j,k}$, to avoid a possible preemption at $minT$.

If X is the default frequency at which the processor executes $\tau_{j,k}$, the frequency (X') at which the processor must execute $\tau_{j,k}$ to avoid a preemption is given by,

$$X' = \frac{C_{j,k}}{C'_{j,k}} \times X$$

If the calculated frequency (X') is much more than the maximum frequency, F_{max}, supported by the hardware or if there is a possibility that a higher priority task instance will be released at t or earlier, the processor executes at full speed (i.e., at the maximum frequency F_{max}) until the release time of the higher priority task instance. This is because, as mentioned earlier, the probability of sporadic task releases increases over time. Thus, if the lower priority task instance $\tau_{j,k}$ finishes its execution as early as possible, the probability of it being preempted by higher priority tasks is reduced. Running the processor at maximum frequency will consume a lot of energy. This is undesirable for embedded systems with a scarce power supply, and hence, the energy-preemption trade-offs could prolong the lifetime of the system. Our method, as opposed to running the processor at maximum frequency, provides for various trade-offs. One such trade-off, for example, is to control preemptions under a hard energy constraint. This can be easily achieved by selectively eliminating preemptions at runtime. Thus, our method can be used to effectively control the preemption behavior while keeping energy consumption under a desired level.

Algorithm 1 gives the minimum frequency at which the a task instance must be executed to avoid a preemption by the next higher priority task instance that will be released in the schedule. The above algorithm has a linear complexity as it involves only a search for the earliest of the release times among the higher priority task instances.

5. EXAMPLE

We demonstrate our method using a simple example. Consider a set of sporadic tasks with a computation requirement C_i

Task	C_i	T_i
A	1	3
B	4	10

Table 1. Example task set

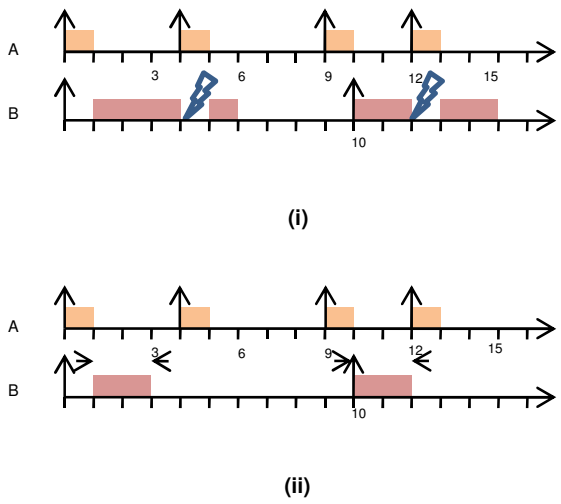

Fig. 1. (i) Schedule before preemption elimination (ii) Schedule after preemption elimination

and a minimum inter-arrival time T_i, as given in table 1. Let the default execution frequency of the processor be 1. Let the execution time of all the instances of τ_i be C_i time units corresponding to a frequency of 1. One possible runtime scenario is shown by (i) in figure 1 where there are 2 preemptions on task B when the tasks are scheduled using FPS, with the priorities assigned according to the rate monotonic priority ordering. In our method, whenever task B starts to execute, we calculate the end of the period of task A since its last release time. This time is the earliest possible release time of the next instance of A. Thus, this time gives the earliest possible preemption point of task B. If task B completes its execution before this time, we can avoid a preemption on task B by task A. We find the total time that is available between the current time and the end of the period of the latest instance of A that has completed its execution. This gives the maximum time for which B_1 can execute non-preemptively. Using this information, we calculate the speed at which the processor has to execute B such that its non-preemption is guaranteed.

Let the i^{th} instance of task A be represented by A_i and that of B be represented by B_i. When B_1 starts its execution at time $t_1 = 1$, it calculates the end of the period of the lastest instance of A (i.e.,A_1), which is $t_2 = 0+3 = 3$. If the preemption on B_1 due to a possible release of A_2 at time 3 has to be avoided, B_1 has to finish its execution by time 3. The total computations that has to be executed non-preemptively, which is the computation requirement of B_1, is 4. The available computation time, if B_1 has to execute non-preemptively, is $t_2 - t_1 = 2$. Thus, B_1 has to be speeded up $\frac{4}{2} = 2$ times to guarantee its non-preemptive execution. This is repeated at the start time of all the instances of all the tasks. Thus, task B can finish before the next possible release of task A avoiding a preemption as shown by (ii) in figure 1.

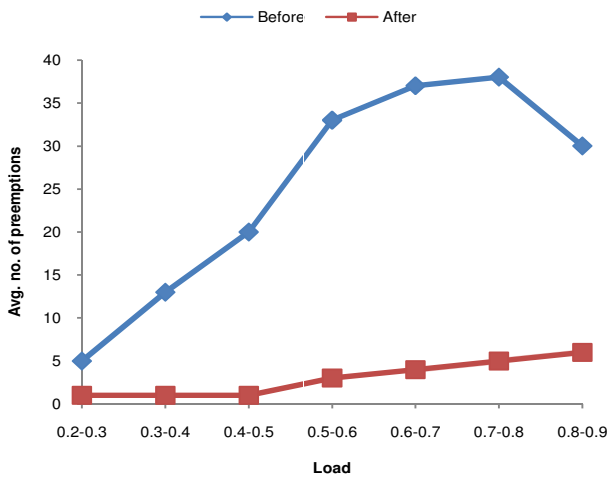

Fig. 2. Average number of preemptions eliminated using our method

6. PERFORMANCE EVALUATION

We conducted a number of experiments using synthetic tasks to evaluate our algorithm. We generated 1400 tasksets, each having 3 - 15 tasks, using the UUniFast (Bini and Buttazzo (2005)) algorithm. The LCM of the task sets generated were kept under 2000. For each task τ_i in Γ, we generated $\frac{LCM}{T_i}$ number of instances which were used in the simulations. Each task instance was released after a time t, generated randomly between 0 and 5 , since the end of the period of its previous instance to reflect the sporadic nature of the taskset. We used rate monotonic priority ordering to simulate the schedule. We assumed a processor model as shown in table 2. The speed indicate the rate of execution of the tasks. The processor, by default, is assumed to run at speed 1 and when its speed is changed to 's', the processor executes a task 's' times faster relative to the execution of the same task at speed 1. In the simulations, we assumed that the processor is the only source of power consumption and the tasks do not use any other devices. The power consumption per clock cycle of the processor for each of the speeds is used to calculate the average power consumption per unit simulation cycle. This is done by the summation of the power consumptions per clock cycle corresponding to the respective speed at which the processor runs for each task executions and dividing it by the total simulation time. We assume that when there are no tasks executing currently, the processor does not spend any energy and falls into a zero energy state.

As seen from figure 2, our method shows a very good performance in eliminating preemptions. The graph in figure 2 decreases at high utilizations $(0.8-0.9)$ because the task sets having utilization between $0.8 - 0.9$ were found to contain lesser number of tasks per taskset. The increase in power consumption can be seen from figure 3. The power consumption increases for task sets with higher utilization. This can be attributed to the fact that processor spends more time in executing tasks due to the heavy load. In other words, the processor idle time is very low. Consequently, while eliminating preemptions on tasks with large computation requirements, the processor executes longer at higher energy states. This increases the processor power consumption while eliminating the preemptions.

Mode	0	1	2	3	4	5
Speed	0	1	1-2	2-3	3-4	4-5
Power consumption per clock cycle (mW)	0	20	50	50	200	500

Table 2. Processor Model

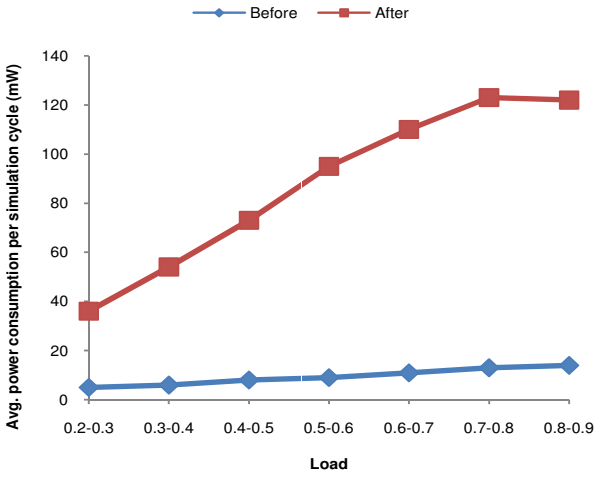

Fig. 3. Increase in power consumption

7. CONCLUSIONS

In this paper, we study the preemption behavior of sporadic task systems scheduled using FPS and the possibility of its control using CPU frequency scaling. We show that offline preemption control using CPU frequency scaling cannot provide non-preemption guarantees in a sporadic task system. We then present a linear complexity algorithm to control the preemptions online. This algorithm makes use of the minimum inter-arrival time that is required between two consecutive instances of higher priority tasks to find the largest time interval available to execute the outstanding computations of a lower priority task instance non-preemptively. Using this information, we find the frequency at which the processor has to run such that the outstanding computations are executed before the next possible release of a higher priority task instance. We also present extensive evaluations of our algorithm which indicates that CPU frequency scaling is an attractive approach for controlling the preemption behavior in sporadic task systems.

Future work will focus on using resource augmentation to evaluate the method, evaluating the possibility of using more complex algorithms to achieve preemption control and comparison of the proposed approach with other methods like Baruah (2005) and Yao et al. (2009) in feasibly scheduling tasksets with preemption overheads.

REFERENCES

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

Baruah, S. (2005). The limited-preemption uniprocessor scheduling of sporadic task systems. In *Real-Time Systems, 2005. (ECRTS 2005). Proceedings. 17th Euromicro Conference on*, 137 – 144.

Baruah, S., Mok, A., and Rosier, L. (1990). Preemptively scheduling hard-real-time sporadic tasks on one processor. 182 –190.

Bertogna, M., Buttazzo, G., Marinoni, M., Yao, G., Esposito, F., and Caccamo, M. (2010). Preemption points placement for sporadic task sets. In *ECRTS '10: Proceedings of the 2010 22nd Euromicro Conference on Real-Time Systems*, 251–260.

Bini, E. and Buttazzo, G. (2005). Measuring the performance of schedulability tests. In *Real-Time Systems*, volume 30, 129–154. Springer Netherlands.

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

Bui, B.D., Caccamo, M., Sha, L., and Martinez, J. (2008). Impact of cache partitioning on multi-tasking real time embedded systems. In *RTCSA '08: Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 101–110.

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

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

Byhlin, S., Ermedahl, A., Gustafsson, J., and Lisper, B. (2005). Applying static wcet analysis to automotive communication software. In *17th Euromicro Conference of Real-Time Systems)*.

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

Gai, P., Lipari, G., and Natale, M.D. (2001). 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*, 73–83.

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

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

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

Lamie, W. (1997). Preemption threshold. In *Whitepaper, Available at http://rtos.com/articles/18833, accessed: 15 Feb 2011*.

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

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

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

Melhem, R., Mosse, D., and Elnozahy, E.M. (2004). The interplay of power management and fault recovery in real-

time systems. *IEEE Transactions on Computers*, 53, 217–231.

Mok, A.K.L. (1983). Fundamental design problems of distributed systems for the hard-real-time environment. *Massachusetts Institute of Technology, PhD thesis*. URL `http://hdl.handle.net/1721.1/15670`.

Pillai, P. and Shin, K.G. (2001). Real-time dynamic voltage scaling for low-power embedded operating systems. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, 89–102.

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

Ramaprasad, H. and Mueller, F. (2008). Tightening the bounds on feasible preemptions. In *ACM Transactions on Embedded Computing Systems*.

Ramaprasad, H. and Mueller, F. (2006). Tightening the bounds on feasible preemption points. In *RTSS '06: Proceedings of the 27th IEEE International Real-Time Systems Symposium*, 212–224.

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

Thekkilakattil, A., Pillai, A.S., Dobrin, R., and Punnekkat, S. (2010). Reducing the number of preemptions in real-time systems scheduling by CPU frequency scaling. In *18th International Conference on Real-Time and Network Systems*.

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

Yao, G., Buttazzo, G., and Bertogna, M. (2009). Bounding the maximum length of non-preemptive regions under fixed priority scheduling. 351 –360.

Yao, G., Buttazzo, G., and Bertogna, M. (2010). Comparitive evaluation of limited preemptive methods. In *15th International Conference on Emerging Technologies and Factory Automation*.