Probabilistic Preemption Control using Frequency Scaling for Sporadic Real-time Tasks

Abhilash Thekkilakattil, Radu Dobrin and Sasikumar Punnekkat
Mälardalen Real-Time Research Center, Mälardalen University, Sweden
{abhilash.thekkilakattil, radu.dobrin, sasikumar.punnekkat}@mdh.se

***Abstract*—Preemption related costs are major sources of unpredictability in the task execution times in a real-time system. We examine the possibility of using CPU frequency scaling to control the preemption behavior of real-time sporadic tasks scheduled using a preemptive Fixed Priority Scheduling (FPS) policy. Our combined offline-online method provides probabilistic preemption control guarantees by making use of the release time probabilities of the sporadic tasks. The offline phase derives the probability related deviation from the minimum inter-arrival time of tasks. The online algorithm uses this information to calculate appropriate CPU frequencies that guarantees non-preemptive task executions while preserving the overall system schedulability. The online algorithm has a linear complexity and does not lead to significant implementation overheads. Our evaluations demonstrate the effectiveness of the method as well as the possibility of energy-preemption trade offs. Even though we have considered FPS, our method can easily be extended to dynamic priority scheduling schemes.**

I. Introduction

Predictable execution of real-time tasks is one of the major requirements to guarantee the temporal properties of safety- and mission critical real-time systems. These systems typically employ the preemptive fixed priority scheduling (FPS) policy, that, since the pioneering work of Liu and Leyland [1], has been widely used in industrial real-time applications mainly due to its simple run-time scheduling mechanisms and low overhead, as well as its ability to handle even task sets with incomplete attribute specifications. However, preemptions are one of the major causes for the unpredictability in, e.g., the execution times of real-time tasks in such systems, thus potentially jeopardizing the system schedulability. Preemptions incur additional costs e.g., cache related preemption delays and context switch overheads, which negatively impact the temporal behavior of the system. These costs are difficult to bound given that they vary with the point of preemption and even possibly with the state of the system at the time of preemption [2]. The preemption related costs are difficult to be accounted in the schedulability analysis, which is typically done offline. On the other hand, too pessimistic assumptions regarding the preemption related costs may lead to inefficient utilization of the resources.

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

Preemption reduction, besides reducing preemption related costs, can also be beneficial in systems with low power consumption requirements. Preemptions can increase the accesses to off-chip memory, thereby increasing the power consumption in the system. This is because, some of the cache lines are evicted during a preemption and when the preempted task resumes its execution, the evicted cache lines have to be restored, increasing the off-chip access. It has been shown that [3], an off-chip memory access is typically more expensive than an on-chip cache access in terms of energy consumption. Preemption reduction, hence, reduce these additional energy requirements that occurs due to an increased cache pollution. Preemptions can also potentially accelerate the wear and tear of the hardware that the real-time system is controlling. A non-preemptive FPS, on the other hand, may be an attractive alternative due to its lower runtime overhead. A major drawback in using non-preemptive scheduling, however, is that, due to the blocking of the higher priority tasks by the lower priority tasks, a portion of the processor time is typically *wasted*. This loss of utilization [4] cannot be bounded and hence non-preemptive scheduling can prove to be infeasible even for arbitrarily low utilizations. This loss of utilization makes non-preemptive scheduling unfavorable for most practical applications.

On the other hand, the need for energy efficient systems necessitates the use of adequate energy management techniques. One of the methods adopted to reduce the energy consumption is to utilize the possibility of Dynamic Voltage and Frequency Scaling (DVS) to reduce the CPU frequency and voltage whenever possible, without jeopardizing the temporal guarantees of the real-time system. Reducing the CPU frequency reduces the performance and increases the execution times of the real-time tasks in the system, increasing the processor utilization. The increase in CPU utilization, in its turn, increases the number of preemptions in the system [5]. The ability to scale up/down the CPU frequency to change task execution times provides the designer the possibility of using energy manipulation to influence the execution behaviour of real-time tasks, in order to achieve specified requirements. Traditionally, DVS has been used in real-time systems mainly to conserve energy, while meet-

ing the temporal requirements [6][7]. On the other hand, increasing the CPU frequency leads to shorter task execution times, and, implicitly, less preempting opportunities. We use the possibility of using frequency scaling to influence the tasks' execution times in order to control the number of preemptions.

In real-time systems, the physical events occurring in the system are mapped to a set of real-time tasks. The events are sampled at a minimum or an exact frequency that is sufficient to meet the physical requirements of the system. A sporadic task model is adopted to represent events where no two events can occur more frequently than a certain known frequency. In such systems, the task arrival rates are assumed at a maximum frequency to analyze its worst case behavior in a predictable manner. However, in many cases the probabilities of the event occurrences can be found, thus enabling the use of a probabilistic analysis. Probabilistic approaches reduce pessimism by considering the probabilistic distribution of the task attributes such as minimum inter-arrival times or WCETs [8].

In this paper, we present a method to control the preempting behavior in sporadic task systems with probabilistic release times, using CPU frequency scaling. The task release time probabilities are considered to find deviations in the inter-arrival times. This deviation is derived from the task release time probabilities such that the probability of a task release is greater than a known threshold e.g., the time instant where the probability of preemption is the highest. This information is used by an online algorithm to control the preemptions online. Considering probabilities while performing preemption control using CPU frequency scaling provides the designer the possibility of trade-off between preemption costs and energy-consumption. Simulation results clearly indicate that considering task release probabilities can provide energy preemption trade-offs. Our algorithm has a linear complexity and does not add any significant runtime overhead.

The paper is organized as follows. In section II, we discuss the related work. Section III details the system model and the various notations used throughout this paper. In Section IV, we present our methodology followed by an example in Section V. We conclude in Section VII.

II. Related Work

Preemptions are widely known to increase costs in the system and the need for reducing the preemptions in a real-time system is widely recognized in the literature [9][10][11][2]. The main preemption related costs are composed of the direct costs to perform the context-switches [11], the costs to manipulate the task queues [10][11], as well as the generally unpredictable cost of cache-related preemption delays [2]. It has been observed by Bui et. al in [9] that cache related preemption delays can increase the task execution times by 33% as the overhead due to cache related preemption delays can be as high as $655\mu S$ for a single preemption.

Since the work of Liu and Layland [1], preemptive FPS has been widely studied and several extensions were proposed for different task models. Preemptive FPS has also found widespread acceptance in the industry and is used in a large number of applications, mainly due to its flexibility and simple run-time overhead. However in reality, due to the overheads involved during a preemption, the use of preemptive FPS might not be ideal [10][12][2]. Buttazzo [5], for example, showed that the rate monotonic algorithm (RM), a widely used preemptive FPS technique, introduces a higher number of preemptions than earliest deadline first algorithm (EDF). He also observed that the number of preemption constantly increases with the task utilization.

Due to the widespread recognition for the need for preemption reduction, several methods have been proposed to reduce the number of preemptions in real-time scheduling. Preemption Threshold Scheduling (PTS) for FPS, first introduced in the ThreadX operating system [13], later formalized by Wang and Saksena [14], improves schedulability and reduces the number of preemptions and the number of threads in the system. The main disadvantage of this method is the need for a dual priority system which may not be directly suitable for, e.g., legacy systems, where scheduler modifications may not be possible. Baruah [4] studied the feasibility of limited preemption techniques and calculated the length of the longest possible non-preemptive execution of a task in a sporadic task system. Yao et. al. [15], evaluated and compared the various limited preemption methods using experiments. Earlier they had extended [4] to FPS, finding an upper bound on the length of the largest possible non-preemptive execution of tasks under FPS [16]. Bertogna et. al. [17], presented a method to place preemption points within task code assuming a fixed preemption overhead.

DVS techniques, traditionally, were used for reducing energy consumption by slowing down tasks' executions [6][7]. It was observed [6][7] that the energy consumption increases linearly with frequency and quadratically with the applied voltage. One of the disadvantages of using DVS is the increase in the number of preemptions due to an increase in task execution times. In [18], the authors observed a less than $140\mu S$ of time for a frequency switch. Another work by Lu et. al. [19] reported $2\mu S$ on an Intel StrongARM processor for the same. This cost is not significant compared to the overhead incurred due to preemptions, making CPU frequency scaling a promising approach towards controlling preemption behavior in real-time systems. Also, since this is a technology dependent cost, with the advances in technology, this overhead is expected to come down.

Dobrin and Fohler [20], proposed a method to minimize the number of preemptions by re-assigning task attributes, such as priorities, periods and offsets, without affecting the schedulability of the taskset. Later in [21], we proposed an

offline method to control the preemption behavior of periodic real-time tasks scheduled by FPS using CPU frequency scaling, by finding job level frequencies that guarantee preemption control. Later in [22] and [23], we extended this to the sporadic task model. However, the algorithm presented was a minimal algorithm and it did not make use of the task release probabilities, while determining the earliest point of preemption. In [22], for instance, only the preempting job was speed-ed up to avoid the preemption before the preemption point determined by the minimum inter-arrival time of the higher priority tasks. In [23], this was extended to speed-up the busy period before the preemption to gain more slack and thus require a lower speed-up. In this paper, we extend our previous works [22] and [23] to propose a method to control the preemption behavior of a sporadic task system scheduled by FPS by making use of task release probabilities to obtain better energy savings. We use the probabilities of task releases in order to derive probabilistic guarantees on the preemption behavior of the schedule while saving on the energy consumption. The use of probabilities to determine the earliest probable preemption point is shown to provide energy preemption trade offs.

III. System Model

In this section, we describe the system model and the notations used in this paper.

A. Processor Model

We assume a processor model that supports a set of discrete operating modes denoted by $M = \{m_1, m_2, m_3, ...m_p\}$, where each m_q is characterized by $m_q = (F_q, W_q)$. Each F_q denotes the processor frequency associated with mode m_q, and W_q is the set $W_q = \{w_q^1, w_q^2, .., w_q^r\}$, that represents the power consumption per clock cycle by the r resources used by the tasks in mode m_q. We assume that a known upper-bounded frequency-switch overhead exists. F_{max} and F_{min} respectively represents the maximum and minimum frequency supported by the processor. The task set is initially assumed to be executing at a default processor frequency $X \geq F_{sched}$, where F_{sched} denotes the minimum frequency that guarantees the system schedulability.

B. Task model

We consider a set of sporadic tasks [24] [25] denoted by $\Gamma = \{\tau_1, \tau_2, ...\tau_n\}$, where each τ_i has a minimum inter-arrival time T_i, a worst case execution requirement C_i, a unique priority P_i and a deadline D_i, relative to its release. Moreover, we assume that C_i, which is given by the largest number of clock cycles required for the execution of task τ_i, is independent of the clock frequency and is a constant [26]. Additionally, let $hp(i)$ represent the set of tasks with higher priorities than τ_i i.e., $P_j > P_i$, $\forall j \in hp(i)$ and LCM represents the Least Common Multiple of the minimum inter-arrival times of all the tasks in the taskset. We define the outstanding computations at a time instant *t* as the set of remaining clock cycles required to complete the execution of all tasks in the ready queue. The ready queue is denoted as $readyQ$. We define $C_j^r(t)$ as the time required to complete the outstanding computations of τ_j at any time t, at the default frequency, X. Moreover, we define $rel_j(t)$ as the earliest release time of the next job of τ_j at the time instant t and is easily obtained by adding T_i to the release time of its latest job that has been released before time *t*.

In addition to the above task parameters, we associate a probability mass function $f_i(t)$, $t \in \mathbb{Z}$ with every task τ_i. $f_i(t)$ gives the probability that a job of the task τ_i is actually released at time t, relative to the its earliest release time. Thus, the probability that a job of τ_i is released at a time $rel_j(t) + t_1$ is given by $f_i(t_1)$[8].

C. Energy Model

We represent the total energy consumption required by all task executions until time t by:

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

where k is given by the smallest integer satisfying:

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

and e by:

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

In the above equation, m_q is the execution mode of the processor during the b^{th} clock cycle of $\tau_{i,l}$ and $\phi_{i,d}$ is the offset in the release of the job $\tau_{i,d}$ since the end of the inter-arrival time of $\tau_{i,d-1}$. Hence, $e_{i,l}$ is the sum of the actual power consumption of all the clock cycles, which gives the total energy used for the execution of the job $\tau_{i,l}$.

D. Execution Time Model

We assume a linear relationship between frequency and execution time of a job i.e., the execution time of a job $\tau_{i,j}$, denoted by $C_{i,j}$, is inversely proportional to the processor frequency. Note that C_i represents the *execution requirement* of τ_i and $C_{i,j}$ represents the *execution time* of its job $\tau_{i,j}$ at the default processor frequency, X. Consequently, the frequency required for scaling $C_{i,j}$ to $C'_{i,j}$ is given by the equation [21]:

$$X' = \frac{C_{i,j}}{C'_{i,j}} \times X \tag{2}$$

The above equation gives the maximum frequency that guarantees a required worst case execution time for a particular job. Also note that, if $C_{i,j}$ represents the initial execution time of $\tau_{i,j}$, $\frac{C_{i,j}}{C'_{i,j}}$ also denotes the speed at which the processor must execute to complete $\tau_{i,j}$ in $C'_{i,j}$ time units.

IV. METHODOLOGY

Our solution consists of a joined offline - online approach: 1) an offline phase which calculates the deviation between the most probable inter-arrival time and the minimum inter-arrival time and 2) an online algorithm that determines the earliest possible preemption point for each task, using the values derived in the offline analysis, and calculates the optimum CPU frequencies that guarantees the preemption avoidance.

In the offline phase, the task release probabilities are considered to calculate the deviation between the most probable inter-arrival time and the minimum inter-arrival time such that the probability that a higher priority task is released does not exceed a certain pre-determined threshold. Later, in the online phase, while calculating a particular earliest preemption point we obtain a probabilistic preemption guarantee, which is less pessimistic than the probability of preemption occurrence based on the minimum inter-arrival time of the preempting task. By less pessimistic we mean that the preemption point determined by our algorithm will be farther in time than in the case when the actual inter-arrival times of the preempting tasks are considered. This is beneficial from an energy usage point of view, since the speed up required to avoid the preemption is less than the case when the actual minimum inter-arrival times are considered. In the following, we discuss the details of our method.

A. Offline Phase

Every sporadic task τ_i has a minimum inter-arrival time T_i and a task release probability $f_i(t)$ from the point of its earliest release time. For instance, if a job of τ_i was released at time t_k, the earliest release time of the next job is $t_k + T_i$ and $f_i(t)$ gives the probability of its release at time $t_k + T_i + t$, $t \in \mathbb{Z}$. Consequently, the next job of τ_i will be released at at a time

$$t_k + T_i + t_{k+1} \ s.t. \ f_i(t_{k+1}) = l_i$$

where l_i represents the threshold probability for release of a job of τ_i.

Thus the deviation between the most probable inter-arrival time and the minimum inter-arrival time can be calculated using the task release probabilities. Let R_i be the deviation from the minimum inter-arrival time to the most probable inter-arrival time of the task τ_i i.e., $f_i(R_i) = l_i$. At any time instant t, the next job of a task τ_i will be released at

$$rel_i(t) + R_i$$

The most probable time instant for a task release at a time t is:

$$rel_i(t) + R_i \ s.t. \ f_i(R_i) = max(f_i(t)) \ \ \forall \ \ t \in \mathbb{Z}$$

B. Online Preemption Control Algorithm

In a sporadic task system, since the inter-arrival times of the tasks are bounded by a lower bound, it is impossible to know the time at which a job of the task will be *actually* released. Consequently, preemptions on lower priority tasks cannot be predicted because of the indeterminism in the higher priority task releases after their minimum inter-arrival times. However, the minimum time interval during which the next job of a particular task will *not* be released can be determined during runtime. Hence, for a lower priority task, it is possible to find the maximum time for which a higher priority task will not be released i.e., it gives the maximum time for which the task can be guaranteed a non-preemption. This gives the maximum time within which the set of outstanding computations must be executed, in order to *guarantee* its non-preemptiveness considering the minimum time interval during which a higher priority job will not be released.

In our task model, every job of τ_i is released with a probability $f_i(t)$, t time units after its earliest release time. It is quite evident that higher the value of $f_i(t)$, higher the probability that the task is released. We use these task release probabilities to provide a probabilistic guarantee for removing the preemption by executing the outstanding computations before the probable point in time at which higher priority task is released.

In our previous work [23], we used the earliest release time of a job as the earliest possible preemption point. However, when considering the task release probabilities, we can relax the above assumption by exploiting the probabilities of the higher priority task releases in future, thereby deriving probabilistic preemption points. We can make use of these probabilistic preemption points to derive processor speeds such that we are able to provide probabilistic guarantees on the preemption behavior of the schedule. Algorithm 1, finds the lowest priority task (τ_u) in the ready queue whenever a job of a task τ_i starts its execution. It finds the earliest time in the future at which a job having a priority higher than τ_u can be possibly released with a known threshold probability. This gives the earliest time at which at least the lowest priority job from among the jobs in the ready queue can be preempted by the higher priority task with a probability equal to the threshold. It then computes the minimum frequency at which the processor must execute the outstanding computations to avoid a preemption at this point.

Whenever a job starts its execution, if τ_u is the lowest priority task in the ready queue, the earliest release time of a job with a higher priority than τ_u is given by:

$$t_{hp\_rel} = \min_{\forall \tau_i \in hp(u)} (rel_i(t) + R_i)$$

The maximum time for which the outstanding computations can execute non-preemptively relative to a time instant

t, is given by:

$$t_{available} = t_{hp\_rel} - t$$

Hence, in order to guarantee the non-preemptive execution of the outstanding computations at any time t, its execution time should be no greater than $t_{available}$.

Algorithm 1: The algorithm is executed at the start time of a job in order to control the number of frequency switches required and to also leverage on the potential gains due to tasks executing for less than their WCET. For instance, we consider a job of τ_i starting its execution at a time instant t. Let the lowest priority task in the ready queue at time t be τ_u. The outstanding computations must execute no greater than $t_{available}$ units of time in order to finish execution before the preemption, where,

$$t_{available} = t_{hp\_rel} - t$$

The outstanding computations require t_{out} time units to execute at the default frequency X, where:

$$t_{out} = \sum_{\forall \tau_a \in readyQ} C_a^r(t)$$

If $t_{available} \leq 0$, it means that the time instant when the release probability of a higher priority task is equal to its threshold probability has elapsed, and its job was not released. Here we could use the same reasoning by using a secondary threshold. However, to preserve the simplicity of the method, we execute the processor at the maximum speed so that the low priority computations complete as early as possible.

Algorithm 1: Find the minimum processor frequency at time t for the non-preemptive execution of the outstanding computations.

τ_u : *the lowest priority task active in readyQ*
$t_{hp\_rel} \leftarrow 9999999 (a\ large\ value)$
$i \leftarrow 1$
while $P_i > P_u$ **do**
  if $t_{hp\_rel} > rel_i(t) + R_i$ **then**
    $t_{hp\_rel} \leftarrow rel_i(t) + R_i$
  end if
  $i \leftarrow i + 1$
end while
$t_{available} \leftarrow t_{hp\_rel} - t$
if $t_{available} > 0$ **then**
  $t_{out} = \sum_{\forall \tau_a \in readyQ} C_a^r(t)$
  $X' \leftarrow \frac{t_{out}}{t_{available}} \times X$
  if $X' > F_{max}$ **then**
    $X' \leftarrow F_{max}$
  end if
  if $X' < F_{sched}$ **then**
    $X' \leftarrow F_{sched}$
  end if
else
  $X' \leftarrow F_{max}$
end if

If $t_{available} > 0$, we have three cases,

1) $t_{available} < t_{out}$
2) $t_{available} = t_{out}$
3) $t_{available} > t_{out}$

Consider case 1, $t_{available} < t_{out}$, i.e., the time required to execute the outstanding computations at time t is greater than the minimum time to the next preemption. In this case, if the outstanding computations finish their executions in $t_{available}$ time units, their non-preemptive execution can be guaranteed. Hence, the new frequency X', that guarantees their non-preemptive execution is given by:

$$X' = \frac{t_{out}}{t_{available}} \times X$$

If the calculated frequency is higher than F_{max}, i.e., the preemption avoidance cannot be guaranteed due to hardware limitations, the processor frequency is set to F_{max}. If in such a scenario, more complex algorithms are used online to calculate the probabilistic preemption points on the outstanding computations, the associated overheads increase. For example, we could use a secondary threshold probability to determine the next possible preemption point. However, to keep the method simple, we set the processor frequency to F_{max}.

In case 2, the above equation becomes $X' = X$, i.e., the processor executes at the default frequency. In case 3, i.e., $t_{available} > t_{out}$, there is a possibility to slow down the processor to conserve energy. The equation to find the new frequency is valid for this case as well. It will find a lower frequency (thus a lower voltage) that guarantees a non-preemptive execution of the outstanding computations. If the calculated frequency is lower than F_{sched}, the processor executes at F_{sched}, thus preserving the overall schedulability of the task set. Even though we have presented our method in the context of FPS, our methodology can be easily extended to dynamic priority scheduling e.g., the EDF scheduling. This can be achieved by considering the task instance priorities rather than the task priorities in the algorithm.

Computational Complexity : The algorithm 1 has a linear complexity assuming that the existence of the jobs in the ready queue is kept track of by, e.g., a simple associative array. The number of jobs in the ready queue at any time t cannot exceed the total number of tasks n. The lowest priority job is the last task in the ready queue, and finding it does not add any significant complexity to the approach. The earliest possible preemption point for the outstanding computations can be found by a simple search in $rel_i(t), \forall i \in hp(u)$. This can also be done in a time linear in the number of tasks as $rel_i(t)$ contains a maximum of

Task	C_i	T_i
X	1	5
Y	3	10
Z	3	20

Table I
EXAMPLE TASKSET

n release times. Also, finding the outstanding computations can also be done in linear time because the number of jobs in the ready queue cannot exceed n.

Implementation Considerations : The online preemption control algorithm can be easily implemented using techniques similar to the DVS algorithms. The implementation typically should occur at the operating system level, where the scheduler is modified to calculate frequencies that can enable preemption control. Whenever a new task arrives, the total outstanding computations can be updated without significant overhead, by just adding the computation requirement of the new task to the current total outstanding computations. The lowest priority task in the ready queue is the last task in the queue, thus a search through the queue can be avoided. In order to find the earliest possible preemption point on the outstanding computations, the OS has to maintain a data structure which stores the next earliest release times of each sporadic task. This is calculated by adding the inter-arrival time of the task to its latest release time. The earliest possible preemption point can be found by a simple search through this data structure.

V. EXAMPLE

We illustrate our proposed method with an example. Consider a set of sporadic tasks with execution requirements C_i and minimum inter-arrival times T_i, as given in table I. Let the time required to execute C_i computations of each task be C_i time units at speed 1. Let the probability of task releases from their respective earliest release times be given by the probability mass function in Figure 1. Note that the probabilities can be different for different tasks. In this example, for the purpose of simplicity we assume the same probability mass function for all the tasks. Due to the sporadic nature of task releases, one possible runtime scenario for the task executions is shown in Figure 2 where there are 2 preemptions when the tasks execute for their WCET's, are scheduled using FPS and the priorities are assigned according to the rate monotonic priority ordering. Let the i^{th} job of task X be represented by X_i, that of Y be represented by Y_i and of Z by Z_i.

In the offline analysis, we assume that the threshold probability is 0.20. The corresponding deviation from the minimum inter-arrival time that gives a release probability equal to the specified threshold probability is 1 time unit. Thus, the most probable time instant at which a job can be released is 1 time unit after the its earliest release time.

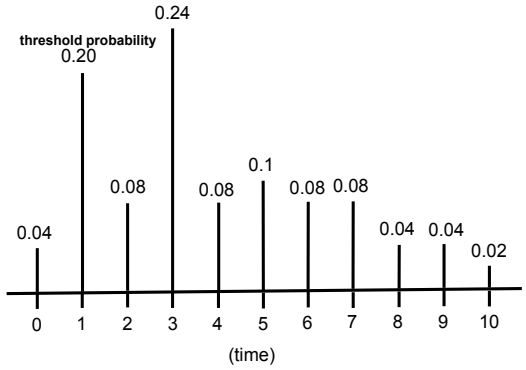

Figure 1. Example probability mass function

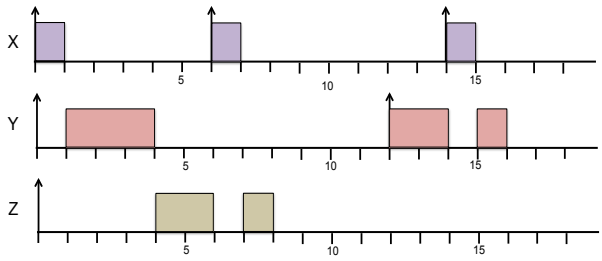

Figure 2. A part of the original FPS schedule of the sporadic task set

When X_1 starts its execution at time $t_1 = 0$, the lowest priority job in the ready queue is Z_1. The next possible release time of a job having a higher priority than Z_1 is by X_2 at time $t_2 = 5$. We add the deviation to the inter-arrival times based on the probability mass function in figure 1. Thus the next probable higher priority task release will happen at time $t_2' = 5 + 1 = 6$. Thus the processor has $t_2' - t_1 = 6$ time units available to execute the outstanding computations non-preemptively. The outstanding computations take $1+3+3 = 7$ time units to execute. The processor has to be speed-ed up by a factor of $\frac{7}{6}$ such that the outstanding computations execute non-preemptively. A part of the sporadic task schedule implementing algorithm 1 is shown in Figure 3. When Y_2 starts its execution at time 12, the earliest preemption point has already elapsed (at time instant 11). After adding the deviation of 1 time unit, we can see that the probable release time of X_3 has already elapsed. Here we could use a secondary threshold value to determine the earliest preemption point on Y_2 by a job of X. However for the purpose of simplicity, we speed up the processor such that Y_2 completes its execution as early as possible.

VI. EVALUATION

We evaluated our method on synthetic tasks by generating 1400 task sets, having 3 - 15 tasks per task set with LCM $\leq$ 2000, using the UUniFast [27] algorithm. The

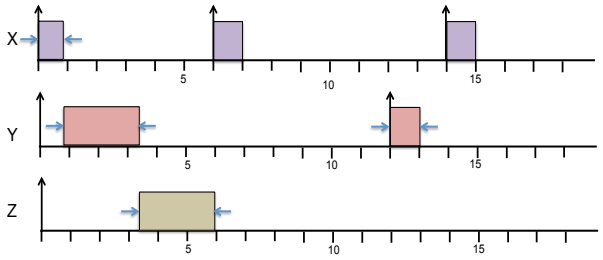

Figure 3. The sporadic task schedule after preemption control

Processor speed	0	1	2	3	4	5
Power consumption per clock cycle (mW)	0	20	50	50	200	500

Table II
PROCESSOR MODEL

processor model that we used in our evaluations, which we adapted from [28], is given in table II. The task sets were generated such that they are schedulable at speed=1. In our experiments, for each task τ_i in Γ, we generated $\frac{LCM}{T_i}$ number of instances where every task instance was released after a time t, with a probability as given in Figure 1. We assumed threshold probabilities of 0.20 and 0.24, to bound the number of preemptions. Consequently, the deviations determined in the offline phase are 1 and 3 time units. Each simulation was run until all the $\frac{LCM}{T_i}$ jobs of every τ_i were executed. We calculated the average number of preemptions that occurred for the following cases i) normal FPS ii) algorithm 1 with $R_i = 0$ iii) algorithm 1 with $R_i = 1$ and iv) algorithm 1 with $R_i = 3$. We present our results in Figures 4 and 5. Figure 4 shows the average number of preemptions for various task utilizations under the different cases described previously and figure 5 shows the average power consumption under the different cases.

Our method showed significant reduction in the number of preemptions as seen from the evaluation results presented in Figure 4. It is seen that considering probabilities while calculating the CPU frequency achieves almost an equal reduction in the number of preemptions as for the case where probabilities are not considered (i.e., $R_i = 0$) while saving energy in the system. The number of preemptions for the highest utilization range (0.8-0.9) shows a decrease for $R_i = 0$ because these task sets where found to have less number of tasks causing less preemptions. The reduction in the energy consumption achieved is particularly significant for tasks with higher utilizations where the number of preemptions is typically higher.

The task release probabilities can be used to achieve preemption-energy trade-offs in the system as can be seen from the two figures. By varying the relaxations permitted to the probable earliest release times, we observe that

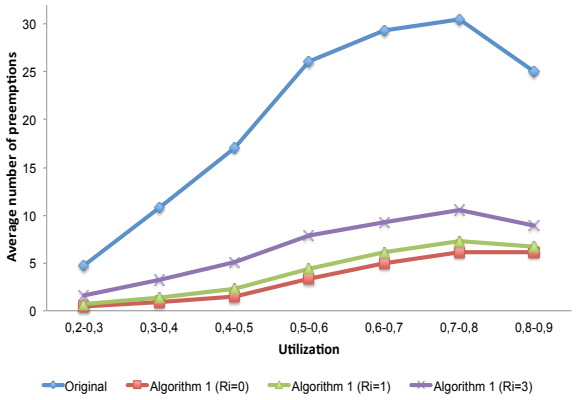

Figure 4. Average number of preemptions for various threshold probabilities

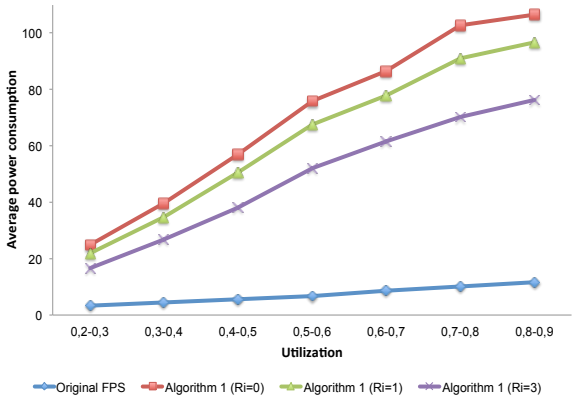

Figure 5. Average power consumption for various threshold probabilities

the preemption reduction achieved varied and so did the energy consumption, demonstrating the possibility of energy preemption trade-off.

VII. CONCLUSIONS

In this paper we presented a combined offline-online approach to control the preemptive behaviour of sporadic task systems with probabilistic inter-arrival times by using CPU frequency scaling. While an offline analysis derives the probability related deviation from the minimum inter-arrival time, an online algorithm uses this information to provide appropriate CPU frequencies that guarantees the non-preemptive task executions while preserving the overall system schedulability. We do so by finding the earliest time instant at which at least one of the jobs in the busy period can be preempted with a probability above a certain known threshold, whenever a task starts its execution. We then calculate the processor frequency such that the jobs in the busy period finishes execution before this point so that a preemption is avoided. The online algorithm has a linear

complexity and does not lead to significant implementation overheads. Evaluation results show the effectiveness of our method in reducing the number of preemptions in the schedule, as well as it also demonstrates the methods' ability to provide for trade-offs between the number of preemptions and overall energy consumption.

Ongoing efforts focus on deriving upper bounds on the speed-up required for guaranteeing a specified preemption behaviour and extensions to the multi-processor platform.

REFERENCES

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *The Journal of ACM*, 1973.

[2] H. Ramaprasad and F. Mueller, "Tightening the bounds on feasible preemptions," in *The ACM Transactions on Embedded Computing Systems*, 2008.

[3] M. Zhang and K. Asanovic, "Highly-associative caches for low-power processors," in *In Kool Chips Workshop, Micro-33*, 2000.

[4] S. Baruah, "The limited-preemption uniprocessor scheduling of sporadic task systems," in *The 17th Euromicro Conference on Real-Time Systems*, 2005.

[5] G. Buttazzo, "Rate monotonic vs. EDF: Judgment day," in *The 3rd ACM International Conference on Embedded Software*, 2003.

[6] H. Aydin, R. Melhem, D. Moss, and P. Meja-Alvarez, "Power-aware scheduling for periodic real-time tasks," *The IEEE Transactions on Computers*, 2004.

[7] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *The 18th ACM symposium on Operating systems principles*, 2001.

[8] L. Cucu and E. Tovar, "A framework for the response time analysis of fixed-priority tasks with stochastic inter-arrival times," *SIGBED Rev.*, 2006.

[9] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez, "Impact of cache partitioning on multi-tasking real time embedded systems," in *The 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2008.

[10] A. Burns, K. Tindell, and A. Wellings, "Effective analysis for engineering real-time fixed priority schedulers," *The IEEE Transactions on Software Engineering*, 1995.

[11] D. I. Katcher, H. Arakawa, and J. K. Strosnider, "Engineering and analysis of fixed priority schedulers," *The IEEE Transactions on Software Engineering*, 1993.

[12] K. Ramamritham and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," in *The Proceedings of the IEEE*, 1994.

[13] W. Lamie, "Preemption threshold," *Whitepaper*, 1997. [Online]. Available: http://rtos.com/articles/18833

[14] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *The 6th International Conference on Real-Time Computing Systems and Applications*, 1999.

[15] G. Yao, G. Buttazzo, and M. Bertogna, "Comparitive evaluation of limited preemptive methods," in *The 15th International Conference on Emerging Technologies and Factory Automation*, 2010.

[16] ——, "Bounding the maximum length of non-preemptive regions under fixed priority scheduling," in *The 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2009.

[17] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo, "Preemption points placement for sporadic task sets," in *The 22nd Euromicro Conference on Real-Time Systems*, 2010.

[18] J. Pouwelse, K. Langendoen, and H. Sips, "Dynamic voltage scaling on a low-power microprocessor," in *The 7th annual international conference on Mobile computing and networking*, 2001.

[19] Y.-H. Lu, L. Benini, and G. D. Micheli, "Dynamic frequency scaling with buffer insertion for mixed workloads," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2002.

[20] R. Dobrin and G. Fohler, "Reducing the number of preemptions in fixed priority scheduling," in *The 16th Euromicro Conference on Real-time Systems*, 2004.

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

[22] A. Thekkilakattil, R. Dobrin, and S. Punnekkat, "Preemption control using CPU frequency scaling in real-time systems," in *The 18th International Conference on Control Systsems and Computer Science*, 2011.

[23] ——, "Towards preemption control using CPU frequency scaling in sporadic task systems," in *Proceedings of the WiP of The 6th International Symposium on Industrial Embedded Systems*, 2011.

[24] A. K. L. Mok, "Fundamental design problems of distributed systems for the hard-real-time environment," *Massachusetts Institute of Technology, PhD thesis*, 1983. [Online]. Available: http://hdl.handle.net/1721.1/15670

[25] S. Baruah, A. Mok, and L. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *The 11th Real-Time Systems Symposium*, 1990.

[26] R. Melhem, D. Mosse, and E. M. Elnozahy, "The interplay of power management and fault recovery in real-time systems," *IEEE Transactions on Computers*, 2004.

[27] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, 2005.

[28] E. Bini, G. Buttazzo, and G. Lipari, "Minimizing cpu energy in real-time systems with discrete speed management," *ACM Transactions on Embedded Computer Systems*, 2009.