Final Report

Neha Gholkar Xiaoqing Luo

Abstract
We intend to design and simulate a power-aware real time scheduling scheme. The scheduling schemes discussed so far in class ensure timely completion of jobs but they are oblivious to power consumption. Today with the advent of small-sized computing systems and SoCs, power management is of high importance. In this work we intend to make the existing scheduling schemes power-aware so that we can minimize power consumption while meeting the deadlines.

Motivation
Today we have an array of portable battery operated devices that need real time guarantees. Due to their dependence on batteries for power supply, there is a constraint on the possible peak power consumption. Minimizing power consumption helps lengthening the battery life. High power consumption is accompanied by high heat dissipation which in turn translates into high cooling costs. Every penny spent on cooling is every penny not spent on doing work (computing). Hence, powerawareness is critical.

Previous work suggests that for most applications the processor consumes most of the energy that is consumed by the system. Under peak load it consumes nearly 60% of the total energy consumed by the system. Hence, appropriate scheduling of this power hungry resource is critical.

Target System
We intend to focus on a combination of soft RT jobs with the goal of providing QoS guarantees. With the help of DVFS, we plan to achieve power savings as we try to operate at reduced frequencies throughout the execution. We are improvising on RM scheduling.

Problem Statement
Given a set of soft RT tasks, generate a schedule of the tasks on a uniprocessor that has reduced peak power consumption while providing the best-effort QoS.

Basic Idea
Soft RT jobs are real time jobs for which deadline misses are allowed and they do not cause any critical losses. When scheduling soft RT jobs we can take advantage of this property to minimize the peak power consumption of the system. A naïve approach would be to run all the tasks at the lowest possible frequency using **DVFS** but that would lead to infinite number of deadline misses. This approach gives a bad performance of the system. To avoid this a soft RT system is often associated with **QoS** requirements that can be specified in terms of **deadline misses**. It puts a limitation on the number of jobs that can be

allowed to miss its deadline within a **time window of relevance**. To stay within this limit, we use DVFS to speed up the tasks as the number of deadline misses reaches this limit. Also when we see that the deadline misses is well below the maximum we choose to switch to lower frequency step.
If at any time the number of deadline misses exceeds its limit, our approach starts **dropping jobs** that are in the ready queue to buy some times so that the jobs further in the queue do not miss their deadlines. However, problem with this approach is that we might end up dropping too many ready jobs that are expected to be serviced rendering the system non-functional. Hence, we limit the number of jobs that can be dropped.
After dropping the maximum number of jobs, if the deadline misses still continue to grow beyond the limit we do best-effort.

QoS
We have implemented a QoS metric based on the number of deadline misses and the number of jobs dropped. We associate a penalty of alpha “a” and a penalty of beta “b” with a deadline miss and a job drop respectively. We can have a deadline miss sensitive system or a job drop sensitive system by using different combinations of (a,b).

Time Window
Every tick is associated with a single index in the time window. We set a window size (eg. 1000 ticks) and we define maximum number of deadline misses and job drops that we allow in a window (eg. 10 misses). This window is a moving window and is implemented as a cyclic array. As the window moves forward we exclude the past misses and the drops that are out of the window.

Each job has a flag to record whether it has missed its deadline. When we detect that a job has missed its deadline during the current tick we increment DEADLINES_MISSED window's corresponding index. We check the running job and jobs in ready queue for deadline misses. During any tick when we decide to drop a job we increment the corresponding JOBS_DROPPED index.

The summation of the DEADLINES_MISSED and JOBS_DROPPED over the entire window and the respective penalties give the current QoS of the system.

QoS_m = A * deadline_misses_in_window
QoS_d = B * jobs_dropped_in_window

System_QoS = QoS_m + QoS_d

The lower the value of System_QoS the better is the quality.

Basic framework of the simulator

while(1)
{

1. For all tasks check if a job has been released at this time instance. Put the job in the ready queue
2. If a task is running
   i. if execution is not complete
      a) sys_time++;
      b) **UpdateQos**
      c) Continue;

3. If you are here : Job has just finished its execution.

Call **DVFS Algorithm to determine the frequency**

4. Schedule next job from the ready queue
   i. If the usefulness servicing the job is not 0
5. sys_time++
6. **UpdateQoS**

}

DVFS Algorithms

DVFS() sets the cFrequency to the desired frequency value for the next job executions.

Worst case_System_QoS (Maximum)
*= alpha * max_deadline_misses_in_time_window + beta * max_jobs_dropped*

We check if the deadlines_missed count for the window has crossed the defined limits. If yes then we drop a job and set the frequency to the highest frequency and return.

If we are still within the maximum number of deadline misses, we consider the job at the head of the ready queue and check whether it is useful to service this job. If its not then we drop this job. We keep dropping all the ready jobs until we find a job whose usefulness function = true.

We implemented two approaches to set the appropriate frequency:

1. Experimental Algorithm : (Neha)
I preferred to take an experimental approach. My algorithm assumes that I don't have any knowledge about a job's actual execution time at a particular frequency. This algorithm doesn't look ahead while making any dvfs decisions instead it bases its decisions

1. current Qos of the system
2. whether a job has finished early / on time (at the deadline) / past the deadline (deadline missed)

Algorithm:

```
if(runtask has missed its deadline)
{
        if(system_QoS of the system is close to the Worst case_System_QoS )
                increase frequency heavily ( by 3 steps )
        else
                increase frequency lightly ( by 1 step )
}
else
{
        if(system_QoS of the system is close to the Worst case_System_QoS )
                increase frequency lightly ( by 1 step )
        else
                decrease the frequency heavily ( by 3 steps )
}
```

The basic idea is to divide the QoS by the number of different frequency decisions that we intend to take. if max deadline misses = 10 you consider 2 distinct cases 0->5 and 5->10
In addition to this also consider whether a job of finished early contributing to the system slack or late after having missed its deadline.

2. Look-ahead Algorithm: (Xiaoqing's)

Our algorithm is based on dynamically observing the current state of Qos, and the state of current running job. We make our decision to change frequency based on whether the job finished before deadline, and whether the CalculateQos() is larger than some threshold. We also consider about the next job in the ready queue and make the next job finish on time by selecting the appropriate frequency.

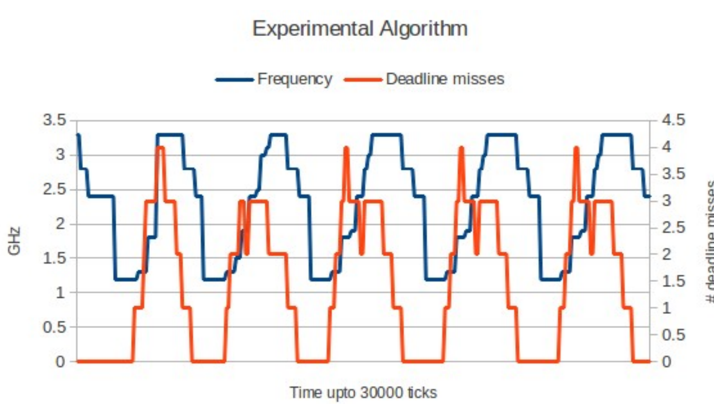
The Pseudo code of this DVFS algorithm is :

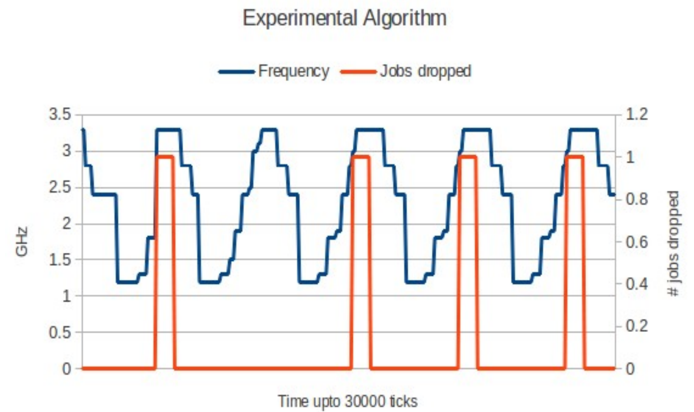
```
int DVFS( )
        int qos = CalculateQos();
        int num_miss = WINDOW*max_missing_ratio;

        if no task in the ready queue  //if its idle time
                do nothing and return

        if runtask finished later than its deadline
                if qos > num_miss*P_of_QOS
                        int need_exec = next_job_execution_time – (runtask_finish_time
– runtask->deadline)
                        search in the frequency<->execution_time array to set
appropriate frequency   //increase the frequency
                        return the new frequency index
                else
              do nothing  //if the Qos is very good, don’t need to increase the
         frequency

        if runtask finished earlier than its deadline
                if (qos > num_miss*P_of_QOS)  //means the Qos of current system is
already bad, so don’t decrease frequency
                        do nothing and return
                else
                        int need_exec = next_job_execution_time – (runtask_finish_time
– runtask->deadline)
                        search in the frequency<->execution_time array to set
appropriate frequency   //decrease the frequency
                        return the new frequency index
```

Results

Experimental Algorithm

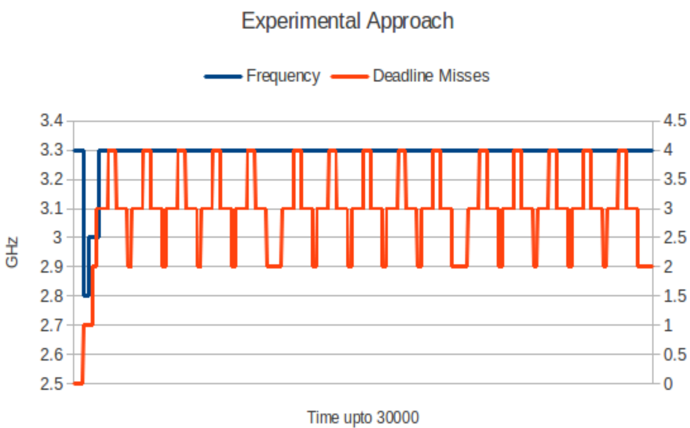
CPU intensive Tasks, Maximum deadline misses allowed = 4 Maximum job drops = 1 alpha=3 beta =1

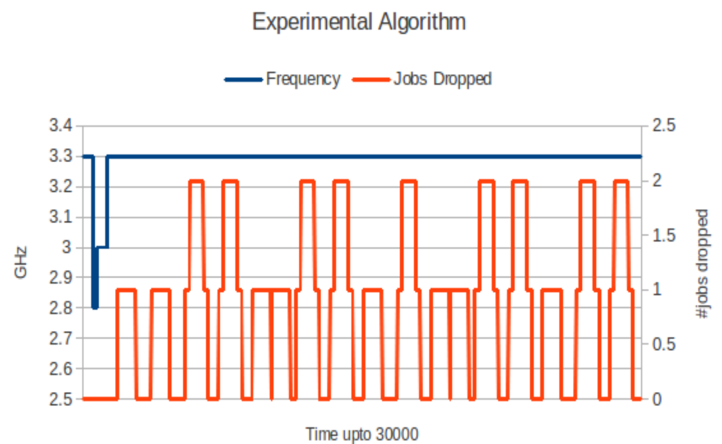




In the above graphs we see that as the number of deadline misses shoots up frequency recovery takes place. As a result, jobs start finishing faster. Following this peak the frequency starts dropping with the number of deadline misses. Corresponding to every peak in deadline misses graph is a job drop peak.
The QoS guarantees are met.

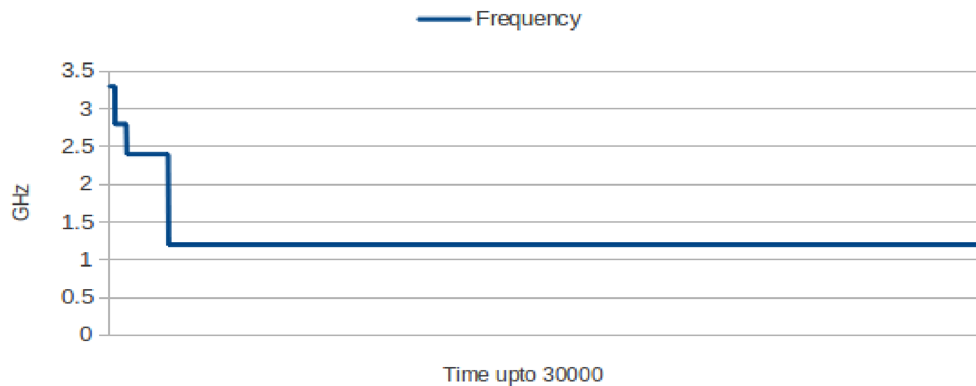
Memory Intensive heavy tasks, Maximum deadline misses allowed=4 Maximum job drops = 1 alpha=3 beta=1





Initially the frequency drops and then stabilizes to the maximum as expected. We do see that the # jobs dropped is larger than the maximum allowed drops however that is because some of the jobs due to their usefulness function become worthless after a large delay.

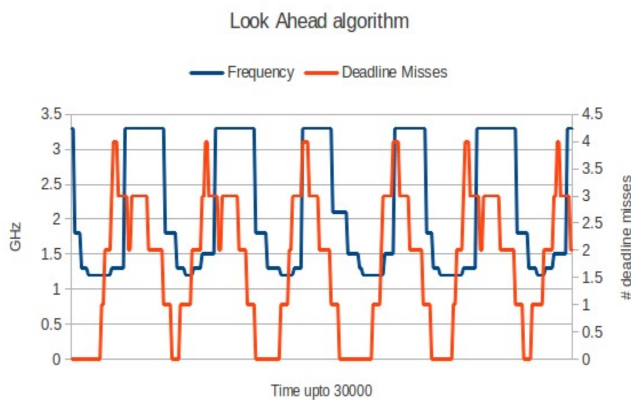
Memory Intensive light weight tasks, Maximum deadline misses allowed = 4 Maximum job drops = 1 alpha=3 beta=1

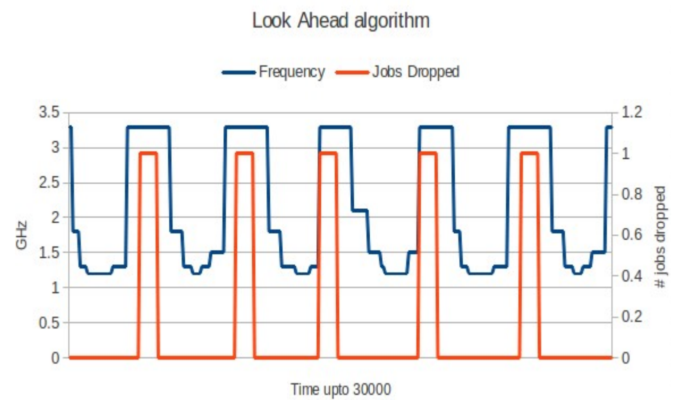


The frequency stabilizes at the lowest step.

Look-Ahead Algorithm

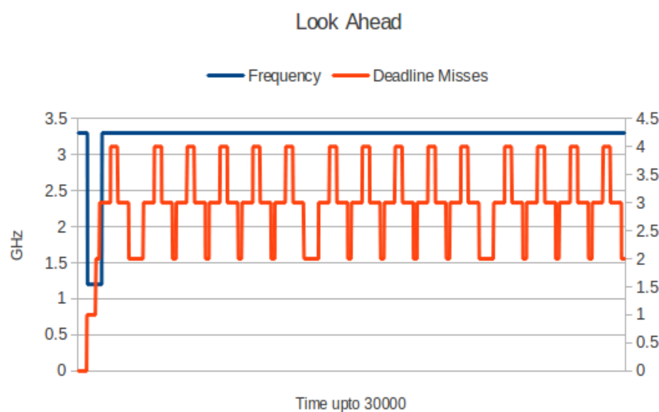
CPU Intensive, Maximum deadline misses allowed = 4 Maximum job drops = 1 alpha = 3 beta =1

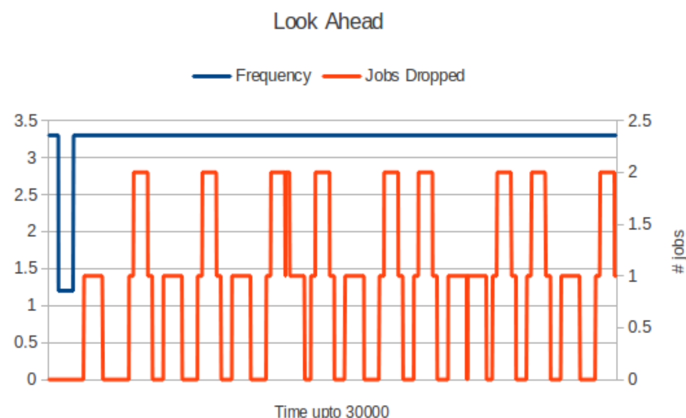




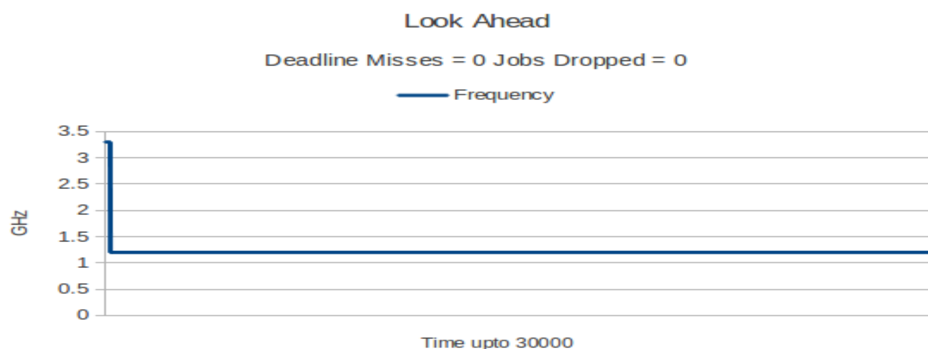
In the above graphs we see that the frequency follows the QoS like in the case of the previous algorithm but we see here that the frequency drops at once. In this algorithm we see that we are operating at lower frequencies for larger portion of the graph as compared to the previous approach. Corresponding to every peak in deadline misses graph is a job drop peak.

Memory Intensive heavy tasks, Maximum deadline misses allowed = 4 Maximum job drops = 1 alpha=3 beta=1





Memory Intensive light tasks, Maximum deadline misses allowed = 4 Maximum job drops = 1 alpha=3 beta=1



Frequency stabilizes faster than that in case of experimental algorithm.

Future Work:

This idea can be improvised if we are able to predict the actual execution times of the jobs to be executed further based on heuristics. We will be able to calculate the expected real slack and make more appropriate frequency decisions.

Frequency decisions made in the above two approaches are a fixed set of decisions. The algorithms can be made more dynamic by having a larger variety of decisions based on finely granular observation the system wrt how early a job finishes its execution, what is the current slack, how many jobs are there in the ready queue, what is QoS_m and QoS_d of the system.

When we drop a job, other jobs and tasks that depend on the job are also affected. We do not take that into consideration in this approach. This would have an additional penalty.

References:

- Dynamic Voltage and Frequency Scaling in Multimedia Servers, Alaa Brihi, Waltenegus Dargie, Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference
- Power-Aware CPU Management in QoS-Guaranteed Systems, Saowanee Saewong, PhD. Thesis, Carnegie Institute of Technology
- http://moss.csc.ncsu.edu/~mueller/rt/rt09/readings/projects/g5/