2012

An Optimal Real-Time Voltage and Frequency Scaling for Uniform Multiprocessors

Gabriel A. Moreno, *Software Engineering Institute*
Dionisio de Niz, *Software Engineering Institute*

An Optimal Real-Time Voltage and Frequency Scaling for Uniform Multiprocessors

Gabriel A. Moreno and Dionisio de Niz
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA, USA
{*gmoreno,dionisio*}*@sei.cmu.edu*

***Abstract*—Power consumption is an increasing concern in real-time systems that operate on battery power or require heat dissipation to keep the system at its operating temperature. Today, most processors allow software to change their frequency and voltage of operation to reduce their power consumption. Frequency scaling in real-time systems must be done in a way that ensures that the tasks' deadlines are met. In this paper we present the *Growing Minimum Frequency* (GMF) algorithm for voltage and frequency scaling in uniform multiprocessors for real-time systems. This algorithm runs in polynomial time and computes the optimal voltage and frequency assignment, achieving better power efficiency than previous algorithms. We present the optimality proof and evaluate the practical improvement over previous algorithms with simulated tasksets. Our evaluation shows up to 30% power efficiency improvement over previous algorithms.**

***Keywords*-real-time; frequency scaling; uniform multiprocessor; power efficiency;**

I. Introduction

Power consumption is an increasing concern in real-time systems that operate on battery power or require heat dissipation to keep the system at its operating temperature. Today, most processors allow software to change their voltage of operation to reduce their power consumption. Given that the maximum frequency (f) of these processors depend on the supply voltage (V), these changes adjust both the frequency and the voltage together in what is commonly known as voltage and frequency scaling (VFS). The power consumption of the circuit technology used in these processors (CMOS) is proportional to the product of the frequency multiplied by the square of the voltage ($P \propto fV^2$ [1]). As a result, voltage and frequency scaling can lead to significant power consumption savings. For brevity, in the rest of the paper we use frequency scaling to refer to voltage and frequency scaling. Since the algorithm presented in this paper is used to determine the optimal frequency assignment at either design time or task admission time, the power consumption remains constant during the execution of the taskset. For that reason, power and energy minimization are equivalent when we limit ourselves to only processor power (as we do in this paper). The analysis in this paper is done in terms of power.

Frequency scaling in real-time systems require the re-evaluation of whether real-time tasks can complete their execution before their deadlines every time the frequency (and speed) is changed. This verification is simplified when we use optimal algorithms such as the Earliest-Deadline First (EDF) [2] in uniprocessors with periodic tasksets.

Recently, a number of research projects (e.g. [3], [4], [5]) have focused on system-wide energy management. These projects take into account not only the energy consumed by the processor but also the energy consumed by the devices in the system (e.g. screen, memory, etc.). This trend signals the maturity of the research in single-core VFS. However, this is not the case for multiprocessors where not even scheduling has reached full maturity. As a result, in this work we focus exclusively on VFS for multiprocessors and leave system-wide energy management for future work.

Frequency scaling in multiprocessor systems can be performed by reducing the frequency of all the processors to the same level. This scheme in known as uniform frequency scaling. Unfortunately, this approach cannot reduce the frequency below the frequency needed by the task with the largest utilization. For instance, if such a task has an utilization of 100% no reduction is actually possible even if we have more than one processor. This is because this task needs CPU cycles in a sequential fashion (i.e., cannot be parallelized) at the top frequency.

Given the limitations imposed by the highest-utilization and other high-utilization tasks (a.k.a. heavy tasks), recent approaches use non-uniform frequency scaling where different processors can be assigned different frequencies. Such an scheme is also applicable for multiprocessors where the frequency and voltages can be adjusted independently in each processor [6]. Multiprocessors systems where the processors can have different speeds are known as *uniform multiprocessors*.[1] In this paper we present our optimal non-uniform multiprocessor frequency scaling algorithm for real-time system called *Growing Minimum Frequency (GMF)*. GMF takes advantage of recent developments in optimal multiprocessor scheduling when processors can be assigned different speeds (known as uniform multiprocessors).

[1] Although it is unfortunate that *uniform* implies different allowed speeds in multiprocessors, but same speeds in frequency scaling, we have opted for keeping the terms already used in the literature.

A. Related Work

Previous work on real-time voltage and frequency scaling algorithms can be divided in uniprocessor and multiprocessor schemes. Among the uniprocessor schemes techniques for fixed-priority scheduling were presented in [7]. In this paper the authors present an algorithm for fixed-priority scheduling. The focus of the paper is on the practicality of the algorithm and the simplicity and pervasiveness of fixed-priority scheduling. In [8], Pillai and Shin present a dynamic voltage scaling algorithm and its implementation in the OS for hard real-time systems. This algorithm is implemented on top of an Earliest-Deadline First (EDF) scheduler. In [9] Cheol-Hoon and Shin present an on-line voltage scaling scheme extending [8] for non-periodic tasksets. While it is possible to use uniprocessor schemes in a partitioned scheduling approach, our work takes advantage of global scheduling to achieve optimality.

Among the frequency scaling algorithms for multiprocessors [10] presents a scheme for tasksets with probabilistic workload. In this paper the authors focus on a partitioned scheme and use a load balancing approach that has been proven to minimize the power consumption. However, this scheme is not optimal for deterministic workloads. Another scheme for multiprocessors is presented in [11]. In this paper the authors model the allocation cost of processors and develop a cost-minimization algorithm. In [12] the authors model the problem as slack-sharing to implement the frequency scaling. This scheme is dynamic and does not achieve static optimality. In [13] the authors create an optimal static algorithm for continuous frequency scaling in multiprocessors based on the LNREF scheduling algorithm. This is the work closest to ours. However, they focus on the creation of partitions that, in the end produces internal fragmentation when frequencies can only be varied in discrete steps. In [14] the same authors extend their algorithm to dynamic voltage scaling.

The rest of the paper is organized as follows. Section II discusses the bottleneck problem imposed by heavy tasks on frequency scaling algorithms. Section III presents our optimal static frequency scaling algorithm for multiprocessors. Section IV present the optimality proofs of our algorithm. In Section V we present our evaluation. Finally, Section VI presents our conclusions.

II. Frequency-Scaling Bottlenecks

In a previous work Funaoka et al. [13] discovered that when a global scheduler is limited to use the same frequency for all the processors it can cause inefficiencies due to the presence of heavy tasks. Specifically, the authors discovered that when the task with the highest utilization (the heaviest task) has a utilization that is higher than the sum of the utilization of the other tasks divided by the number of processors available, the heavy task becomes the bottleneck. In other words, if the processor frequency required by the heaviest task is higher than the frequency required by all the other tasks, the former must be chosen in order to guarantee meeting the deadline of the heaviest task.

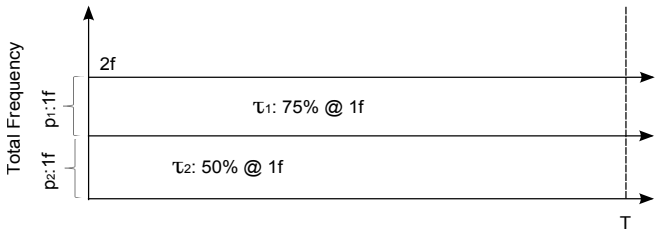

Figure 1. Uniform Frequency Scaling – No Scaling

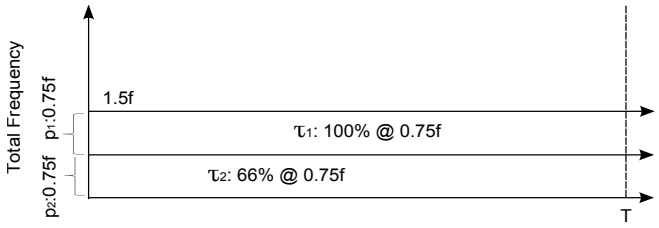

Figure 2. Uniform Frequency Scaling at 75%

Let us illustrate the bottleneck effect with the example in Figure 1. This figure presents the frequency of the processors and its accumulated frequency in the vertical axis and the time in the horizontal axis. In addition, the execution time of each of the two tasks in the figure (that have the same period[2] T) is represented by a rectangle. The height of the rectangle roughly represents the total frequency consumed by the task (that can be less or more than what a single processor offers), and the width represents the time it takes to execute. The label inside the rectangles shows the percentage of time that the tasks needs to be active, which is equivalent to the width divided by T. Figure 1 depicts the processors running at their maximum frequency f. Figure 2 on the other hand, depicts the processors running at 75% of f, which is the minimum uniform scaling possible. This is because at this frequency task τ_1 already needs to run 100% of the time. However, τ_2 still only runs 66% of the time and the rest of the cycles in p_2 are idle, wasting energy.

A. Non-Uniform Frequency Scaling

Non-uniform frequency scaling allows each processor to have its own frequency setting. This leads to additional frequency reductions in the processors not used by the bottleneck task. This is illustrated in Figure 3. This figure shows the non-uniform frequency scaling of the processors for the example presented in Figure 1. In this case, the frequencies of both processors are reduced to the minimum possible (75% and 50%) in order to keep them busy 100% of the time and avoid idle cycles.

[2]Same period tasks are used to simplify the visual depiction but in reality they can have any period.

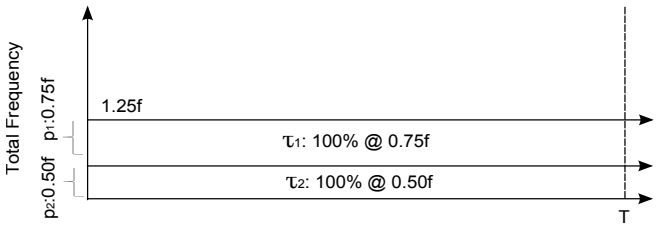

Figure 3. Non-Uniform Frequency Scaling

Funaoka et al. [13] designed the *DecideIndependentFrequency* (DIF) algorithm in order to overcome the heavy task bottlenecks. This algorithm classifies tasks into heavy and light giving their own processor to the heavy tasks. A task is classified as heavy if its utilization is greater than the sum of the utilization of all the lighter tasks divided by the number of processors not previously assigned to heavy tasks. The heavy-or-light classification is performed on a taskset ordered in non-increasing order of utilization starting with all the tasks classified as light. Then, the tasks are tested one at a time to check whether they are heavy or light. Every time a task turns out to be heavy, it is separated from the set of light tasks and it is given its own processor. This process is repeated until no task can be classified as heavy. In the end, the set of tasks classified as light are scheduled in a pool of all the processors not assigned to heavy tasks with an optimal global scheduling algorithm (LNREF in [13]).

The frequency of each heavy-task processor is adjusted to the minimum possible that still makes the task schedulable. The frequency of the pool of processors for the light tasks, on the other hand, is determined with the *DecideUniformFrequency* [13] frequency scaling algorithm (all the processors are assigned the same frequency). This algorithm selects the uniform frequency as either the maximum utilization of the tasks assigned to it or the sum of all the tasks' utilizations divided by the number of processors, whichever is largest, where a frequency of 1 correspond to processors executing at 100% of the maximum speed. This frequency is used if the frequency can be adjusted continuously, but since processors support a discrete set of frequencies, in practice, the next higher available frequency is used.

Table I presents a sample taskset and the output of DIF with four processors with discrete frequency steps of 25%, 50%, 75%, and 100% of the maximum frequency. It is worth noting that only two tasks are classified as heavy, running in their own processors, and the rest are classified as light running in a processor pool.

Figure 4 presents a visual representation of Table I using the same abstraction used to discuss bottlenecks. This figure allows us to identify two types of idle cycles resulting from this frequency assignment and taskset partitioning. First, idle cycles in heavy-task processors. These cycles cannot be used by any other task. And, secondly, idle cycles in the light

Table I
FREQUENCY SCALING EXAMPLE WITH DECIDEINDEPENDENTFREQUENCY

Processor	Frequency	Task	Utilization	Heavy/Light
P_1	1.0	τ_1	1.0	Heavy
P_2	1.0	τ_2	0.9	Heavy
P_3	0.75	τ_3	0.6	Light
P_4	0.75	τ_4	0.5	Light
		τ_5	0.1	Light

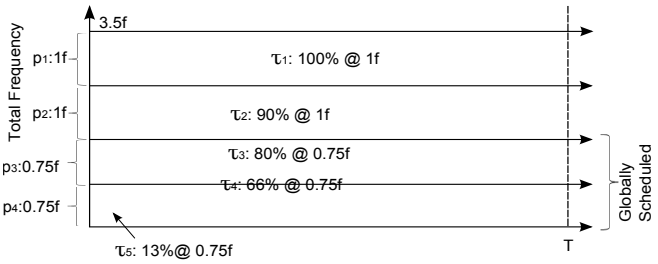

Figure 4. Sub-Optimal Non-Uniform Frequency Scaling

processor pool. These cycles can be used by other light tasks but cannot be used by heavy tasks. In other words, idle cycles are fragmented inside partitions and cannot be used by tasks outside these partitions. This is commonly known as *internal fragmentation*.

Funaoka et al. proved that DIF is optimal if the frequency of the processors can be adjusted continuously. Unfortunately, such an algorithm is not optimal if the frequency can only be adjusted to a limited number of discrete frequencies. The intuition behind this is that if the frequency of the processors can be adjusted to perfectly fit the required utilization, the idle cycles in the heavy-task processors are eliminated, and the idle cycles left in the light-task processor pool are all in the same partition and not fragmented.

The authors proposed an exhaustive algorithm for the discrete case that looks for all possible ways to partition the taskset (into co-scheduled tasks) to be scheduled on a virtual processor (a processor pool scheduled with a global optimal scheduler) with the appropriate frequencies.

Table II shows the result of the exhaustive algorithm when assigning the frequencies of the taskset presented in Table I. The visual depiction of this result is presented in Figure 5. Note that in this case task τ_4 runs in its own processor and the processor frequency is adjusted to make task τ_4 use 100% of the processor.

It is worth noting that the exhaustive method reduces the inefficiencies by repartitioning exhaustively to find the best fit given that the processors within each partition are assigned the same frequency. In Section III we will show that GMF does not need this separation thanks to the use of an optimal multiprocessor scheduler that allows processors to have different frequencies while scheduling the taskset in

Table II
FREQUENCY SCALING EXAMPLE WITH EXHAUSTIVE SEARCH

Processor	Frequency	Task	Utilization	Partition
P_1	1.0	τ_1	1.0	Partition 1
P_2	1.0	τ_2	0.9	Partition 2
P_3	0.5	τ_4	0.5	Partition 3
P_4	0.75	τ_3	0.6	Partition 4
		τ_5	0.1	

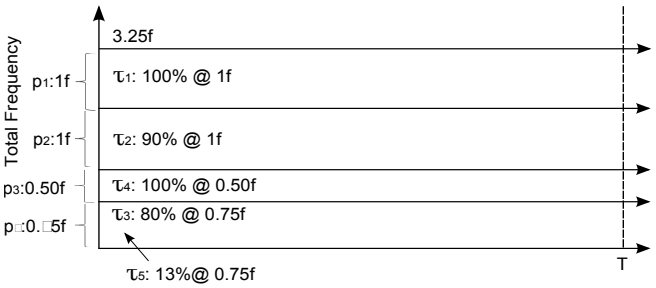

Figure 5. Exhaustive Non-Uniform Frequency Scaling

a single pool of processors.

III. GROWING MINIMUM FREQUENCY (GMF) ALGORITHM

In this section we present the Growing Minimum Frequency (GMF) algorithm in detail. GMF is a non-uniform frequency scaling algorithm for multiprocessors that uses the U-LLREF [15] optimal multiprocessor scheduling algorithm for uniform multiprocessors. GMF takes advantage of the incremental test in U-LLREF. Specifically, U-LLREF verifies the schedulability of a subset of tasks in a subset of processors, growing these subsets one element at a time. GMF follows this test and increments the frequency of the subset of processors being tested by U-LLREF just enough to satisfy the current subset of tasks. GMF is optimal on processors with uniform frequency steps, i.e., the separation between any two consecutive frequency set points is given by a constant δ.

Before we get into the details of our algorithm we first define our system model and describe the U-LLREF algorithm.

A. System Model

A system is composed of a computing platform Π and a task set Γ. The computing platform consists of m uniform processors with a number of frequency set points. It is defined as $\Pi = (\delta, f_{min}, A)$ where δ is the frequency step between two consecutive frequencies, f_{min} is the minimum frequency setting of a processor, and $A = \{f_1, f_2, \ldots, f_m\}$ is a particular frequency assignment for the m processors. Without loss of generality we normalize the frequency settings so that the maximum frequency is 1 and use them as the normalized speed of the processors. The set A is assumed to be ordered in non-increasing order of frequency such that $i < j \implies f_i \geq f_j$.[3] Each of these frequencies is restricted to take a value $f_i = f_{min} + k_i\delta \leq 1$ with $k_i \in \mathbb{N}_0$. The normalized frequency assignment for the processors are used as processor speeds to calculate the capacity of the processors and evaluate the schedulability of the system.

We denote the power consumption of processor i as $P(f_i)$. Using a similar approximation as previous authors [1], we calculate this power consumption as $\mathcal{C}f_i^3$. The total power consumption of a frequency assignment A is denoted as $P(A)$ and is calculated as $\sum_{i=1}^{m} P(f_i)$. Similarly, the power consumption of the first i processors for assignment A is denoted as $P_i(A)$.

The taskset $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is the set of n periodic tasks with deadlines at the end of the period. The utilization of a task is defined as $u_i = \frac{C_i}{T_i}$ with C_i as the worst-case execution time and T_i as its period.

B. U-LLREF

U-LLREF is an extension to the LLREF [16] algorithm for uniform multiprocessors. LLREF is used to schedule tasksets in identical multiprocessors (same instruction set and same speed) executing all tasks at a constant rate. To do this, LLREF divides the schedule of the taskset into two execution time planes: Time and Local (TL-planes). TL-planes are intervals of time between consecutive deadlines. For each of these intervals, LLREF calculates an amount of computation necessary for each task to keep up with the fluid schedule, i.e., the execution time in this interval in order to keep a constant rate of execution for the task. LLREF then assigns the m tasks with the largest amount of computation to execute on the m processors. Such an assignment is recalculated at two events: (1) when a task completes its amount of computation for the plane, and (2) when a task reaches its zero-laxity instant, i.e., the instant when the task needs to start executing continuously and to completion in order to meet its deadline.

U-LLREF extends LLREF to schedule a taskset on a uniform multiprocessor allowing processors to have different speeds. In order to do this, U-LLREF uses a scheduling test that verifies that a subset of tasks fits in the capacity of a subset of processors. The tasks and processors are ordered in non-increasing order of utilization/capacity. The subset of tasks and processors starts with just one element (the first of each set). Then if the total utilization of the tasks is smaller or equal to the total capacity of the subset of processors, the number of elements on the set is increased by one and the test is performed again. These tests continue until $min(m-1, n)$ number of elements is reached in the set. Then, if $n > m$ a final test comparing the capacity of all

[3]Note that, since the m processors are identical, frequency f_i does not necessarily have to be assigned to the ith physical processor.

the processors in the set with the total utilization of the tasks is done. The system is schedulable only if all the tests are satisfied. These test are encoded in the following equations[4]:

$$\begin{array}{l} \sum_{i=1}^{k} u_i \leq \sum_{i=1}^{k} f_i \quad \text{for } k = 1, \ldots, \min(m-1, n) \\ \sum_{i=1}^{n} u_i \leq \sum_{i=1}^{m} f_i \end{array} \tag{1}$$

It is worth noting that U-LLREF assumes (as do other global schedulers) that the cost of migrating a task from one processor to another is negligible. We keep this assumption and leave the evaluation of the migration cost for future work.

C. The Algorithm

GMF works as follows. First, it sorts the tasks in non-increasing order of utilization and initializes the frequency of each processor to the lowest supported frequency. The core of the algorithm consists on taking the first i (starting at 1) tasks (i-taskset) and the first i processors (i-processors), and comparing the sum of the utilization of tasks in the i-taskset to the sum of the frequencies of the i-processors. If the sum of the utilization of the subset of tasks is larger, then we increase the frequency of the processor with the lowest frequency among the i-processors (the slowest processor) by one δ step. We keep increasing the frequency of the slowest processor (which can change after each δ increment) until the sum of the frequencies of the i-processors is larger than or equal to the sum of the utilizations of the i-taskset (the ith U-LLREF equations from 1). If all the processors in the subset reach the maximum frequency before satisfying this condition, then the taskset is ruled not schedulable. Otherwise, when the condition is satisfied, i is incremented by one and the frequency-increment cycle is repeated, thereby considering one more task and one more processor. This process is repeated incrementing i until it reaches m and all the processors have been considered. The last iteration uses the complete set of processors and the complete set of tasks (not just the first m). This algorithm is presented in Algorithm 1.

GMF does not have the fragmentation problems that previous algorithms suffer. Specifically, because the taskset is not partitioned into subsets (e.g. heavy and light tasks), there is no idle time left by one subset that cannot be used by another subset.

Let us illustrate the fragmentation-free nature of GMF with an example. Consider the taskset in Table II. GMF produces the same frequency assignment as the exhaustive algorithm executing the following steps. First the frequency of all processors are initialized to 0.25. Next, the first processor and the heaviest task are considered ($i = 1$). The frequency of just this processor is incremented three times

[4]We replaced the original normalized speed (s_i) for the normalized frequency (f_i) to improve readability.

Algorithm 1 GMF $(\Pi : (f_{min}, \delta, A : \{f_1, \ldots, f_m\}), \Gamma : \{\tau_1, \ldots, \tau_n, \})$

```
1: sort tasks such that u_1 ≥ ··· ≥ u_n
2: for all i ∈ [1...m] do
3:    f_i ← f_min
4: end for
5: U_sum ← 0
6: for i = 1 to min(m, n) do
7:    if i < m then
8:       U_sum ← U_sum + u_i
9:    else
10:      U_sum ← Σ_{j=1}^{n} u_j
11:   end if
12:   while U_sum > Σ_{j=1}^{i} f_j do
13:      slowest ← min(arg min_{j∈[1...i]} f_j)
14:      if f_slowest = 1 then
15:         return ∅    // not schedulable
16:      end if
17:      f_slowest ← f_slowest + δ
18:   end while
19: end for
20: return {f_1, ..., f_m}
```

until it reaches 1 and can satisfy task τ_1 with $u_1 = 1$. Then i is incremented to two ($i = 2$) and the two heaviest tasks are tested against the two fastest processors. Because the sum of the frequencies of the processors is smaller than the sum of the utilization of the tasks ($(1 + 0.25) < (1 + 0.9)$) the frequency of the slowest processor ($i = 2$) is incremented multiple times until such a condition is false (when $f_2 = 1$). At this point i is incremented to 3 and the schedulability test compares the three first processors and tasks ($(1+1+0.25) < (1 + 0.9 + 0.6)$). Again the processor with the slowest frequency ($f_3 = 0.25$) is incremented up to $f_3 = 0.50$ to satisfy our test. Now the fourth processor and task are considered and f_4 is incremented once to pass their schedulability test ($(1 + 1 + 0.50 + 0.50) = (1 + 0.9 + 0.6 + 0.5)$). In the final step all the tasks are considered and tested for schedulability ($(1+1+0.50+0.50) < (1+0.9+0.6+0.5+0.1)$) requiring one increment to the frequency of the slowest processor f_4 (or f_3 since they are equal) to reach 0.75 making the taskset schedulable ($(1+1+0.75+0.5) \geq (1+0.9+0.6+0.5+0.1)$).

While the final frequency assignment of GMF and the exhaustive algorithm from [13] is the same, because GMF uses U-LLREF it allows any task to use idle cycles from any processor in the system. This is depicted in Figure 6.

Figure 6 shows how, even though the frequencies of the processors are assigned considering only subsets of tasks (and only one task at the beginning), every task can use the cycles from any processor in the system. This is because all processors are scheduled with U-LLREF. The difference between GMF and the exhaustive method from [13] becomes

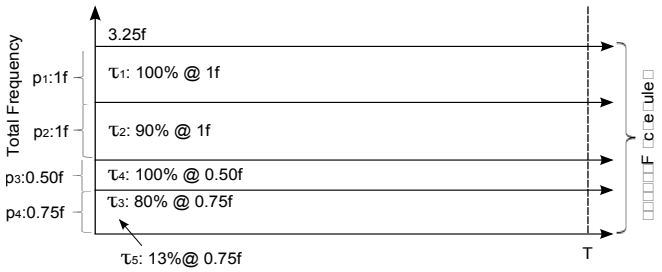

Figure 6. GMF Sample Frequency Assignment

Table III
FREQUENCY ASSIGNMENT FOR $u_5 = 0.25$ WITH THE EXHAUSTIVE ALGORITHM

Processor	Frequency	Task	Utilization	Partition
P_1	1.0	τ_1	1.0	Partition 1
P_2	1.0	τ_2	0.9	Partition 2
P_3	0.75	τ_3	0.6	Partition 3
P_4	0.75	τ_4	0.5	Partition 4
		τ_5	0.25	

evident when we increment the utilization of task τ_5 to $u_5 = 0.25$. In this case, the exhaustive method still creates partitions giving tasks τ_1, τ_2, and τ_3 their own individual partitions and bundling τ_4 and τ_5 in a common partition as shown in Table III.

Figure 7 depicts these partitions in a frequency/time graph indicating how partitions 1 and 4 are fully utilized while partitions 2 and 3 contain idle cycles. If these cycles could have been used by other tasks outside the partitions (say by τ_5) it would have not been necessary to increase the frequency f_4 to 0.75.

In contrast, in GMF only the last increment to i changes. Specifically, when $i = 5$ task τ_5 is considered with $u_5 = 0.25$. In this case the frequency increment still results in $f_4 = 0.5$ and the schedulability test is successful ($(1 + 1 + 0.75 + 0.5) = (1 + 0.9 + 0.6 + 0.5 + 0.25)$).

Figure 8 presents the frequency/time graph that shows how the spreading of the idle cycles from one processor to all task enables GMF to avoid fragmentation within partitions.

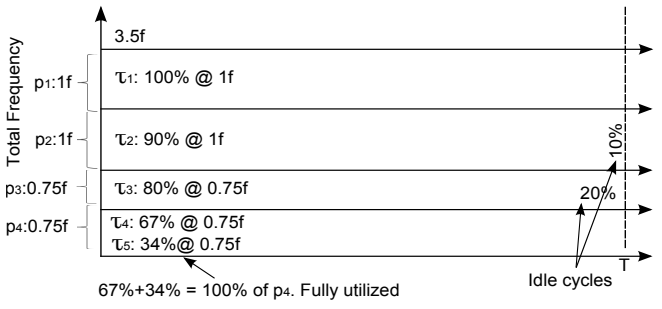

Figure 7. Sub-optimal Assignment with Exhaustive Algorithm

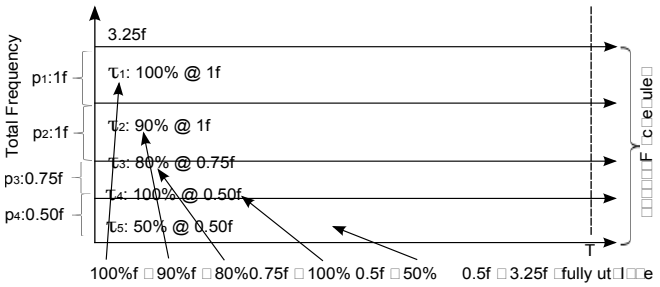

Figure 8. Optimal Assignment with GMF Algorithm

In addition to computing a more power efficient frequency assignment, GMF has the advantage that its complexity is polynomial as opposed to NP-Hard in the case of the exhaustive algorithm.

IV. GMF OPTIMALITY

We say that a frequency assignment $A = \{f_1, \ldots, f_m\}$ is optimal if the task set Γ can be scheduled by U-LLREF on platform Π with frequency assignment A, and there is no frequency assignment $A' = \{f'_1, \ldots, f'_m\}$ such that $P(A') < P(A)$ and task set Γ can be scheduled by U-LLREF on platform Π with frequency assignment A'.

A frequency assignment A is i-optimal, where $i \leq m$, if the first i frequencies of A are an optimal frequency assignment for the first $\min(i, n)$ tasks of Γ on the first i processors of platform Π. A frequency assignment is 0-optimal if each processor is assigned the lowest possible frequency (f_{min}).

The capacity of the platform with frequency assignment A is $S(A) = \sum_{f_i \in A} f_i$.

Now we prove three lemmas that are used to prove the optimality of our algorithm.

Lemma 1. *An increment to a frequency f_i by δ incurs a larger power increase the larger f_i gets.*

Proof: This is a general acceleration property of functions that have positive first and second derivatives.

Given that $P(f) = \mathcal{C}f^3$, we have that

$$P'(f) = 3\mathcal{C}f^2$$
$$P''(f) = (P'(f))' = 6\mathcal{C}f$$

where both $\mathcal{C}$ and f are positive. With the positive first derivative the function is increasing, and with the positive second derivative the slope is also increasing. That is, $P(f)$ is a monotonically increasing convex function, and consequently, the same frequency increase produces a larger increase in power when the base frequency is larger. ■

Lemma 2. *If A is an i-optimal solution, and A' is an $(i+1)$-optimal solution, the first i frequencies of A' are bound from below by the first i frequencies of A. That is, $\forall j \leq i : f'_j \geq f_j$.*

Proof: For A' to be $(i+1)$-optimal, it has to satisfy the first $i+1$ equations of the U-LLREF scheduling test. Since the capacity added by processor $i+1$ does not have any effect on the satisfaction of the first i equations of the U-LLREF scheduling test, we can focus on the first i processors to prove that their speeds are bound by A. Assume for contradiction that $\exists j \leq i | f'_j < f_j$. There are only two possible cases.

- No other processor frequency f'_k such that $f'_k \in A' \wedge k \leq i$ is increased to compensate for the reduction of f'_j. Then either the frequency reduction causes one of the first i equations of the scheduling test not to be satisfied and A' is not $(i+1)$-optimal, or $P_i(A') < P_i(A)$, and consequently A was not i-optimal, both of which are contradictions.
- The frequency of one or more other processors is incremented to compensate for the reduction in a way that satisfies the first i equations of the U-LLREF scheduling test. Suppose that only one processor frequency f_k is increased to compensate for the decrease of f_j; that is, $\exists f'_k \in A', f_k \in A | f'_k > f_k$. It must be that $k < j$, otherwise the jth U-LLREF equation would not be satisfied. Since, $k < j$, we have that $f_k \geq f_j$. With $f'_k > f_k \geq f_j > f'_j$. By Lemma 1, the power difference between f_j and f'_j is smaller than the power difference between f'_k and f_k. Consequently, the power saving of the frequency reduction is not enough to compensate for the frequency increase. Since A' requires more power to provide the same capacity, it is not optimal, which is a contradiction. Now, suppose that a subset of processor frequencies f_k with $1 \leq k < j$ were increased to compensate for the frequency decrease of f_j. Then, for each processor frequency f_k we can apply the same argument presented above for a partial decrease/increase portion.

■

Lemma 3. *Given a frequency assignment A with capacity $S(A)$, and $A^{(k)}$, which is the most power efficient assignment with capacity $S(A)+k\delta$ bound from below by A, the most power efficient assignment with capacity $S(A)+(k+1)\delta$ bound from below by A, $A^{(k+1)}$, is obtained by incrementing the slowest speed of $A^{(k)}$.*

Proof: Assume for contradiction that

$$\exists B^{(k+1)} | S(B^{(k+1)}) = S(A) + (k+1)\delta \\ \wedge P(B^{(k+1)}) < P(A^{(k+1)})$$

We now show that this is false. Starting from an assignment A with capacity $S(A)$, usually there are several possible sequences of k single frequency increases that can be taken to reach another assignment of capacity $S(A)+k\delta$. Each single frequency increase adds the same amount δ to the capacity, but requires a possibly different power increase ΔP_j. Let $\Delta P = \{\Delta P_1, \Delta P_2, \ldots\}$ be the set of power increases corresponding to the available frequency increases among all the processors beyond the frequency assignment A, where $i < j \Leftrightarrow \Delta P_i \leq \Delta P_j$. Note that ΔP may contain more than one power increase for the same processor, corresponding to successive frequency increases in that processor. Achieving a frequency increase of $k\delta$ requires taking a valid subset of size k of the power increases (i.e., the sth increase for a processor can only be taken if it is the first increase for that processor or if the $(s-1)$th was taken previously). Since $A^{(k)}$ is the most efficient allocation providing a $k\delta$ increase over A, it must have taken the smallest k power increases $\{\Delta P_j | j \in \{1, \ldots, k\}\}$. Otherwise, the selection of a power increase $\Delta P_r | r > k$ instead of a $\Delta P_s | s \leq k$ implies $\Delta P_r > \Delta P_s$, which would add $\Delta P_s - \Delta P_r$ more power. Given that the frequency increment from $A^{(k)}$ to $A^{(k+1)}$ is obtained incrementing the slowest frequency of $A^{(k)}$, by Lemma 1, the power increase would be the smallest, that is ΔP_{k+1} because the k smallest power increases have already been taken. That is, the frequency assignment $A^{(k+1)}$ is obtained by taking the $k+1$ frequency increases with the smallest power increase. Consequently, any other subset of ΔP of size $k+1$ that the frequency assignment $B^{(k+1)}$ could take requires at least the same amount of power increase. That is, $P(B^{(k+1)}) \geq P(A^{(k+1)})$, completing the contradiction. ■

Let us now prove the optimality of GMF.

Theorem 1. *Algorithm 1 is optimal in the sense that it computes the most power efficient frequency assignment to schedule a task set with U-LLREF on a multi-processor platform with uniform frequency steps.*

Proof: We show that at the end of its execution, the algorithm has computed the most power efficient frequency assignment that satisfies the U-LLREF scheduling test if the task set is schedulable.

Line 1 sorts the tasks in descending order of utilization, which is the assumption of the U-LLREF scheduling test. Lines 2-4 initialize the frequency assignment with each processor at the slowest possible frequency. That is, the initial frequency assignment is 0-optimal.

Let i be the number of times the loop body of the outer loop (lines 6-19) has been executed. The invariant for the outer loop is that the frequency assignment is i-optimal. In the base case, when $i = 0$, the invariant holds because the initial assignment is 0-optimal.

We now show that after each execution of the body of the outer loop, the invariant is maintained. Let A be the $(i-1)$-optimal assignment computed by the previous iteration of the outer loop or the initial assignment. By lines 7-11

$$U_{sum}(i) = \begin{cases} \sum_{j=1}^{\min(i,n)} u_j & \text{when } i < m \\ \sum_{j=1}^{n} u_j & \text{when } i = m \end{cases}$$

Inner loop invariant. Let k be the number of times the inner loop body has been executed. The frequency assignment $A^{(k)}$ satisfies the following invariants:

1) $A^{(k)}$ is bounded from below by A [5]
2) $S(A^{(k)}) = S(A) + k\delta$
3) $P(A^{(k)}) \leq P(B) \ \forall B | S(B) = S(A^{(k)}) \wedge B$ is bounded from below by A

Inner loop base case. The algorithm does not actually keep a copy of frequency assignment $A^{(k)}$ separate from A. When $k = 0$, $A^{(k)} = A$. Invariants 1 and 2 hold trivially. Invariant 3 holds because $S(A^{(k)}) = S(A)$ and A is optimal for that capacity.

Inner loop induction. The index of the processor with the slowest frequency assignment is $slowest = \arg\min_{j \in \{1 \ldots i\}} f_j$ (line 13). Recall that frequencies are ordered in non-decreasing order such that f_1 refers to the slowest frequency, f_2 to the second slowest, and so on. Assume that at least one of the first i processors is still not assigned the maximum frequency, otherwise Γ is not schedulable on the platform (line 15). By Lemma 3, increasing the slowest frequency (line 17) results in the most power efficient frequency assignment with capacity $S(A) + (k+1)\delta$, which is bound from below by A. That is, the loop invariants hold after going through the body of the loop.

Inner loop termination. The loop terminates as soon as the ith U-LLREF equation is satisfied (line 12), which is evaluated after each single-step increment. When the loop terminates, the speed assignment computed by the algorithm is the most power efficient that satisfies the first i U-LLREF equations.

If $n \geq m$, the outer loop executes m times, the last one with $i = m$. Since of the invariant of the outer loop is maintained, the final allocation is m-optimal, thus optimal. If $n < m$, the resulting frequency assignment is n-optimal, and because the frequency of all processors $j | n < j \leq m$ is the slowest possible frequency, the frequency assignment is optimal. ■

V. Evaluation

To evaluate the performance of the GMF algorithm, we randomly generated 15,000 tasksets and computed the power they would consume with different VFS algorithms when run on two platforms with different frequency settings, both with 4 processors. The tasksets were generated with different utilization levels (up to 100% of the processors) to give room to the VFS algorithm to scale down the frequencies. These utilization levels range from 0.5 up to 4 (100% of all four processors) with increments of 0.25. Then, for each utilization level we generated 1,000 tasksets. Within each taskset we generated tasks with a utilization generated randomly from a uniform distribution within the interval [0.01, 1] adjusting the last task's utilization to add up to the target utilization.

[5]This implies that the first $i - 1$ equations of the U-LLREF conditions are satisfied

Table IV
Processors for Evaluation Platforms

Platform 1

Frequency	Volts
1	5
0.75	4
0.5	3

Platform 2

Frequency	Volts
1	3.5
5/6	2.8
4/6	2.2
3/6	1.6
2/6	1.4

For each taskset we run four VFS algorithms: DIF, GMF, Exhaustive, and Optimal. The Exhaustive algorithm is the one that searches for the optimal partitioning, and Optimal searches the optimal frequency assignment without partitioning. Then, given each frequency assignment, we calculated the power consumption and normalize it to one.

Figures 9 and 10 show plots of the average power consumption in two different platforms. Table IV shows the (normalized) frequencies supported by each platform with the corresponding voltage required for each speed. The description for platform 2 corresponds frequencies and voltages of an Intel Core 2 Duo T7700 processor. Note, however, that we assumed a 4-core platform for our evaluation instead of a dual-core. The plots on Figures 9 and 10 show that GMF perform better than the other frequency scaling algorithms. Furthermore, since GMF does not partition the processors, it can achieve in polynomial time better power efficiency than Exhaustive, which searches for the solution with an NP-Hard algorithm. In both plots, the GMF curve exactly matches the Optimal curve as expected, given that GMF is optimal. It is worth noting that GMF achieves larger relative power savings in platform 1 than in platform 2, specially at high utilization. This is because when there are fewer frequencies available, the amount of idle time left unused by the partitioned approaches is larger than in the cases in which more available frequencies allow for a finer control on the frequencies of the partitions. By avoiding partitioning completely, GMF leaves less idle time unused, and, consequently, achieves better power efficiency.

The two platforms used for the previous experiments satisfy the assumption of uniform frequency steps that was used to prove the optimality of GMF. In addition, we ran simulations using the characteristics of a processor that has frequency steps that are not uniform. Specifically, we used the characteristics of the Intel XScale processor shown in Table V [17]. Note that in this case, instead of computing power from the voltage for each frequency, we directly show the power from the specifications, which includes both the dynamic and leakage power. Although GMF does not use the power from the table to compute the frequency assignment,

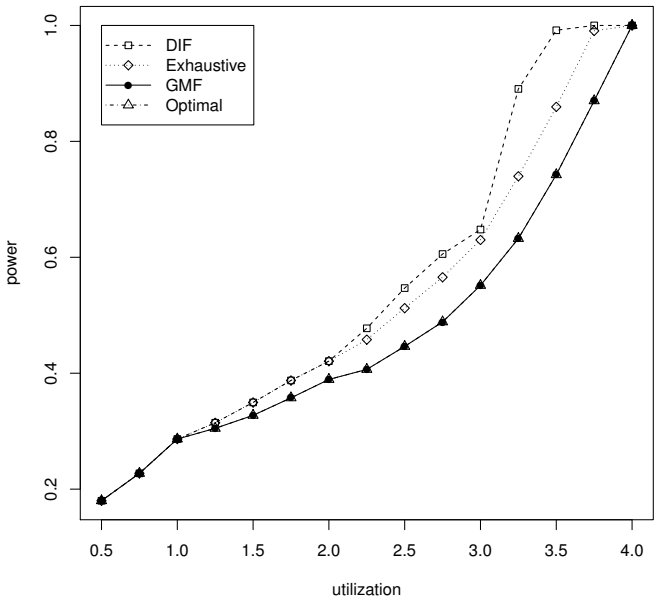

Figure 9. Power Consumption on Platform 1

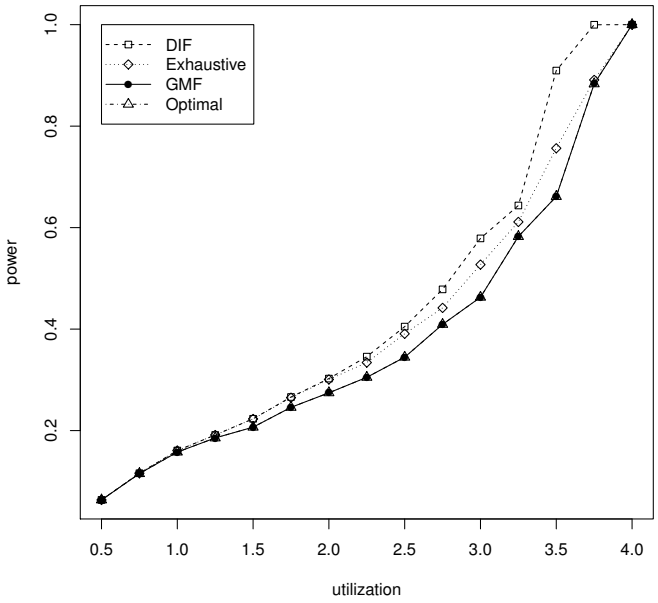

Figure 10. Power Consumption on Platform 2

we use it to compare the power consumption of the solutions calculated by the different algorithms. Figure 11 shows the results obtained running the same simulation described before using a 4-processor platform with the characteristics of Table V. In this case, GMF also computes the optimal frequency assignment. We know there are cases with non-uniform frequency steps in which GMF is not optimal. For example, if the frequency steps and their associated power are such that the utilization of the last task can be satisfied by increasing the frequency of either the slowest processor

Table V
PROCESSOR CHARACTERISTICS FOR PLATFORM 3

Frequency	Power (mWatt)
1	1600
0.8	900
0.6	400
0.4	170
0.15	80

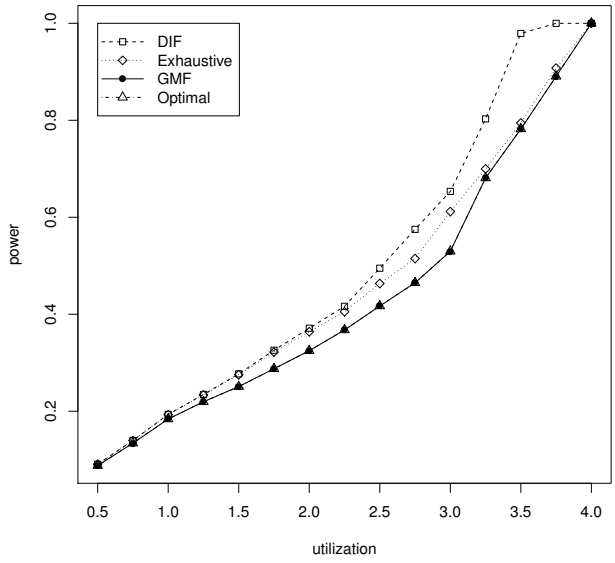

Figure 11. Power Consumption on Platform 3 (Non-Uniform Frequency Step)

i or another processor j, and the power increase is smaller for processor j, which would imply that the next frequency step for j is smaller than that of i. Based on this observation and the results of this last experiment, we believe that GMF is also optimal for some relaxation of the uniform frequency steps condition. Hence, we propose the following conjecture.

Conjecture 1. *Algorithm 1 is optimal in the sense that it computes the most power efficient frequency assignment to schedule a task set with U-LLREF on a multi-processor platform with frequency steps such that the power increase in each frequency increasing step does not decrease.*

VI. CONCLUSIONS

In this paper we presented the Growing Minimum Frequency (GMF) frequency scaling algorithm for real-time systems for uniform multiprocessors. GMF computes the optimal frequency assignment necessary for the multiprocessor to guarantee the deadlines of a real-time taskset with the minimum power consumption. GMF relies on the optimal

global scheduler for uniform multiprocessor U-LLREF to implement a greedy strategy that always increments the minimum frequency. This strategy allows us to keep the complexity of GMF polynomial, avoiding the combinatorial nature of previous algorithms (e.g. when searching for the optimal partitioning). We proved the optimality of GMF for processors with uniform frequency steps. This requirement is satisfied by commercially available processors such as the Intel T7700. In addition, we conjectured that GMF is optimal if that assumption is partially relaxed. Finally, we performed an experimental evaluation to compare the power consumption of GMF against three other frequency scaling algorithms. In these experiments we observed that GMF achieves up to 30% better power efficiency than previous algorithms, and confirmed its optimality.

ACKNOWLEDGMENT

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.[6]

REFERENCES

[1] T. Burd and R. Brodersen, "Energy efficient CMOS microprocessor design," in *System Sciences, 1995. Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, vol. 1, January 1995, pp. 288 –297 vol.1.

[2] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, vol. 20, pp. 46–61, January 1973.

[3] V. Devadas and H. Aydin, "On the interplay of voltage/frequency scaling and device power management for frame-based real-time embedded applications," *IEEE Transactions on Computers*, vol. 61, pp. 31–44, January 2012.

[4] X. Zhong and C. Xu, "System-wide energy minimization for real-time tasks: Lower bound and approximation," in *Proceedings of IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2006.

[6]No Warranty. This Carnegie Mellon University and Software Engineering Institute material is furnished on an "as-is" basis. Carnegie Mellon University makes no warranties of any kind, either expressed or implied, as to any matter including, but not limited to, warranty of fitness for purpose or merchantability, exclusivity, or results obtained from use of the material. Carnegie Mellon University does not make any warranty of any kind with respect to freedom from patent, trademark, or copyright infringement.

This material has been approved for public release and unlimited distribution except as restricted by copyright.

The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013 and 252.227-7013 Alternate I.

[5] D. Zhu and H. Aydin, "Energy management for real-time embedded systems with reliability requirements," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2006.

[6] J. Dorsey, S. Searles, M. Ciraula, S. Johnson, N. Bujanos, D. Wu, M. Braganza, S. Meyers, E. Fang, and R. Kumar, "An integrated quad-core Opteron processor," in *Solid-State Circuits Conference, 2007. ISSCC 2007. Digest of Technical Papers. IEEE International*, February 2007, pp. 102 –103.

[7] S. Saewong and R. R. Rajkumar, "Practical voltage-scaling for fixed-priority RT-systems," in *Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '03)*, 2003, p. 106.

[8] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*, 2001, pp. 89–102.

[9] C.-H. Lee and K. Shin, "On-line dynamic voltage scaling for hard real-time systems using the EDF algorithm," in *Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS)*, December 2004, pp. 319 – 335.

[10] C. Xian, Y.-H. Lu, and Z. Li, "Energy-aware scheduling for real-time multiprocessor systems with uncertain task execution time," in *44th ACM/IEEE Design Automation Conference (DAC '07)*, June 2007, pp. 664 –669.

[11] J.-J. Chen and T.-W. Kuo, "Allocation cost minimization for periodic hard real-time tasks in energy-constrained DVS systems," in *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD '06)*, 2006, pp. 255–260.

[12] D. Zhu, R. Melhem, and B. R. Childers, "Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, pp. 686–700, July 2003.

[13] K. Funaoka, S. Kato, and N. Yamasaki, "Energy-efficient optimal real-time scheduling on multiprocessors," in *2008 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, May 2008, pp. 23 –30.

[14] K. Funaoka, A. Takeda, S. Kato, and N. Yamasaki, "Dynamic voltage and frequency scaling for optimal real-time scheduling on multiprocessors," in *International Symposium on Industrial Embedded Systems (SIES)*, June 2008.

[15] S. Funk and A. Meka, "U-LLREF: An optimal scheduling algorithm for uniform processors," in *Workshop on Models and Algorithms for Planning and Scheduling Problems*, June 2009.

[16] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *27th IEEE International Real-Time Systems Symposium (RTSS '06)*, December 2006, pp. 101 –110.

[17] J.-J. Chen and T.-W. Kuo, "Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems," in *Proceedings of the IEEE/ACM International Conference on Computer Aided Design (ICCAD '07)*, 2007, pp. 289–294.