A Global Optimal Scheduling Algorithm for Multiprocessor Low-Power Platforms

Shelby FUNK
University of Georgia (UGA)
Athens GA, USA
shelby@cs.uga.edu

Chiahsun HO
University of Georgia (UGA)
Athens GA, USA
ho@cs.uga.edu

Vandy BERTEN
Université Libre de Bruxelles (ULB)
Brussels, Belgium
vandy.berten@ulb.ac.be

Joël GOOSSENS
Université Libre de Bruxelles (ULB)
Brussels, Belgium
joel.goossens@ulb.ac.be

ABSTRACT
This paper presents a real-time scheduling algorithm which globally schedules any feasible periodic task set — i.e., is optimal — on a multiprocessor platform. Beside respecting the deadlines, our algorithm, based on DP-Wrap algorithm, drastically reduces the power consumption of the device by slowing down the processors as much as possible.

We present our algorithm in two versions (if frequency changes are allowed at any time, or if there are some restrictions about it), prove its (schedulability) correctness, and provide several simulations attesting its low-power efficiency.

1. INTRODUCTION
Nowadays, it is straightforward that energy efficiency is a crucial aspect of embedded systems where a huge number of small and specialized autonomous devices are interacting together through many kinds of media (wired/wireless network, bluetooth, GSM/GPRS, infrared, etc.). Moreover, we know that the uniprocessor paradigm will no longer hold in those devices. Even today, many mobile phones are equipped with several processors.

In this work, we are interested in multiprocessor energy efficient systems. We provide a new global scheduling algorithm, adapted from a well-known optimal algorithm (DP-Wrap [16]) to account for energy consumption. Our adaptation keeps its optimality (from the schedulability point of view), but strongly improves its energy efficiency.

1.1 Related work

Optimal Multiprocessor Scheduling.

Many optimal scheduling algorithms for implicit deadline on multiprocessor have been designed over the years. A seminal solution concerns periodic tasks: Proportionate Fairness (PFair) was, as its name implies, based on a fair distribution of the processing capacity between tasks (i.e., each task being executed proportionally to its utilization factor) [5, 6]. Many PFair algorithms were designed over the years (e.g., PD [7], PD2 [27], ER-PD [18]).

More recently, it has been noted that imposing a fair progress in task executions at each and every time instant was too restrictive; imposing the fairness constraint only at task deadlines suffices for the optimality [29, 16]. From this observation, a new family of schedulers called DP-Fair [16] (or Boundary fair [29]) has been proposed. One may cite DP-Wrap [16], LLREF [12] or BF [29] as some algorithms following this refined concept. In addition, Andersson et al. developed EKG [3] and EKG for sporadic tasks [2], which categorize tasks into migratory and non-migratory. Migratory tasks are scheduled according to DP-Fair approach and non-migratory tasks are scheduled according to EDF policy.

Even more recently, two works propose optimal schedulers without any fairness. The first one, the RUN algorithm [25], reduces the multiprocessor scheduling problem to a uniprocessor schedule using a dualization technique. The second one, U-EDF [23], is an horizontal EDF generalization for multiprocessor (while global EDF is a vertical and sub-optimal EDF generalization).

DVFS on multiprocessors.

Energy consumption is a crucial aspect of embedded systems, in particular for battery-operated ones. Modern processors enable Dynamic Voltage/Frequency Scaling (DVFS) which achieve a quadratic energy savings by applying a linear decrease of the voltage/frequency of the CPUs.

Scheduling techniques for DVFS uniprocessors are numerous, see e.g. [17, 20, 21, 24, 28]. Concerning multiprocessors, where the interest of DVFS is even more important (see arguments of [1]), most of the work concerns partitioning, when task migration is forbidden [4, 11]. Few works concern global scheduling: [11, 30, 5, 22].

1.2 Model and Definitions

In this work, we consider a set of $n$ strictly periodic tasks running on $m$ identical CPUs. Those CPUs will be DVFS
Dynamic and Voltage Frequency Scaling), which provide several execution modes. A smart usage of those modes, typically going from small frequency with low energy consumption to high frequency and high consumption, can save energy, especially when the load of the system is not too high.

In the following, time is expressed in seconds (s), and the work is expressed as the amount of cycles that can be executed in one second on a CPU at its highest speed. We denote this unit of work as \( w \). With this definition, we will see below that the maximal speed of a processor, expressed as the amount of units of work processed per unit of time, is 1.

### 1.2.1 System characteristics

We will denote by \( t \) the current time. The time-line will be divided in time slices (TS), denoting the interval between two consecutive deadlines (of any task in the system). \( t_j \) stands for the \( j^{th} \) deadline, and \( \sigma_j \) the \( j^{th} \) time slice, going from \( t_{j-1} \) (incl.) to \( t_j \) (excl.). We consider \( f_0 = 0 \). \( L_j \) denotes the length of the time slice \( (t_j - t_{j-1}) \). For simplicity we often drop the subscript and name \( L = \sigma_j - \sigma_{j-1} \). Throughout this paper, the index “\( j \)” will always be used to denote time slice numbers.

### 1.2.2 Task system characteristics

The task system \( J \) contains \( n \) tasks, denoted \( \tau_1, \tau_2, \ldots, \tau_n \). \( \tau_i \) creates a job every \( p_i \) time units, with a worst case work requirement of \( e_i \) units of work. We consider that the period equals the deadline. Each job must be finished before its deadline \( d_i(t) \), which is the arrival time of the next job.

At time \( t \), \( \eta_i(t) \) denotes the amount of work that the current job of the task \( \tau_i \) has already processed, and \( r_i(t) \) the remaining worst case work. Then, \( r_i(t) = e_i - \eta_i(t) \). The commonly used utilization factor \( u_i \) is \( e_i / p_i \). The in our work, we introduce a new kind of utilization factor: \( u'_i(t) \) denotes the ratio of processor that needs to be assigned to task \( \tau_i \) in order to meet its deadline. By definition, \( u'_i(t_j) = u_i \) for any \( t_j \) corresponding to an arrival of task \( \tau_i \). Furthermore, if at some time \( t \), with some task \( \tau_i \), we have \( u'_i(t) > 1 \), the system cannot guarantee the deadline of this task \( \tau_i \) will be met (while it may happen that the deadline is not missed if the task does not use its worst case execution time). Then, \( u'_i(t) \leq 1, \forall t, \forall \tau_i \), is a necessary condition of schedulability (which is stronger than the classical condition \( u_i \leq 1 / V(\tau_i) \)). Throughout we will always use \( i \) to denote task index.

### 1.2.3 Processor characteristics

The system is composed of a set of \( m \) identical processors \( \pi_1, \ldots, \pi_m \). Each of them can be set to use a given frequency, expressing the number of units of work processed per second. With our definition of the unit of work, a processor speed is always below or equal to 1. We denote by \( f_k(t) \) the speed (or frequency) chosen for \( \pi_k \) at time \( t \). Given an ideal speed \( s \), \([s]_j \) is the slowest speed above (or equal to) \( s \), and \([s]_j \) is the fastest speed below (or equal to) \( s \). Those two notations are defined for any value in \([0, 1]\).

If a task \( \tau_i \) has a utilization factor of \( u_i \), and we give it a processor \( \pi_k \) with speed \( f_k = u_i \), then the task will have to continuously run on this CPU.

### 1.3 DP-Wrap algorithm

This work presents a DVFS extension of the DP-Wrap scheduling algorithm. We begin by presenting DP-Wrap for identical multiprocessors and describing properties of the algorithm. DP-Wrap schedules tasks on a time-slice basis. At the beginning of each time slice, the DP-Wrap scheduler allocates an execution time for the next time slice and assigns the workload to the processors using McNaughton’s wrap-around algorithm [21].

At each time slice boundary \( t \), DP-Wrap ensures each task’s remaining execution time is proportional to its uti-
lization. Specifically,
\[ r_i(t) = u_i \times (d_i(t) - t) \forall \tau_i \in J. \] (1)

Any schedule satisfying this property at each time slice boundary will be valid (i.e., meet all deadlines) on m processors if and only if the following conditions hold for \( J \)
\[ u_i \leq 1, \forall \tau_i \in J \text{ and } \sum_{\tau_i \in J} u_i \leq m. \]

The DP-Wrap algorithm enforces Equation (1) by allocating each task \( \tau_i \) to execute for \( u_i \times L \) time units during the time slice. Once the execution times have been determined, DP-Wrap must assign the workload to the processors. We let \( \ell_{i,1}(t) \) denote the remaining execution of \( \tau_i \) allocated to processor \( \pi_k \) during the time slice, which we call \( \tau_i \)'s local remaining work on \( \pi_k \).

To start, tasks are assigned to processor \( \pi_k \) one by one as long as the total assigned workload is smaller than \( L \). Let \( \tau_i \) be the last task assigned to \( \pi_k \). Then for all \( x < i \), we let \( \ell_{i,1}(t_j) = u_x \times L \). DP-Wrap will then allocate as much of \( \tau_i \)'s execution to \( \pi_1 \) as possible and allocate the remaining execution to \( \pi_2 \) as follows
\[ \ell_{i,1}(t_j) = u_i \times L - \sum_{x<i} u_x \times L \text{ and } \]
\[ \ell_{i,2}(t_j) = u_i \times L - \ell_{i,1}(t_j). \]

We say \( \tau_i \) is “wrapped” from processor \( \pi_1 \) to processor \( \pi_2 \). This procedure is repeated on the remaining processors until all tasks are assigned to the processors.

Because the tasks’ execution times are proportional to their utilization values, the time slice schedules are all scaled versions of one another. If we “mirror” every other time slice then the executing task will not change at time slice boundaries and each migrating task will migrate only once per time slice. Moreover, because we know which processor will execute the migrating task next, there need not be any communication delay when a processor starts to execute a migrating task. For instance in Figure 1 task \( \tau_3 \) starts on \( \pi_3 \), then has some time where it can be transferred, and then is started on \( \pi_2 \) to finish its work for the current time slice. If there is no time to migrate off-line, that probably means that its utilization is very close to \( s \), and this task should have then received a full processor.

We now examine how to extend the DP-Wrap scheduling algorithm to DVFS processors. We begin by assuming that all jobs execute for their worst-case execution times. We then explore how to account for early job completions.

2. TASK PLACEMENT AND SPEED SELECTION USING WCET

When assuming all jobs execute for their worst case execution times, the DP-Wrap scheduler will allocate tasks’ execution times to be proportional to their utilization, as in the original DP-Wrap algorithm. Therefore, if \( t \) is any time slice boundary \( u_i'(t) = u_i \forall \tau_i \in J \). Hence, once again all time slices have the same basic schedule, but scaled differently. When we consider early completion times, this may no longer be the case.

As long as the relation between speed and energy consumption (both dynamic and static leakage current) is convex, it is optimal to select a speed that will allow the processors speeds to remain as constant as possible. This constant optimal speed is given by \( s = \frac{\sum_{\tau_i \in J} u_i'(t)}{m} = \frac{U'(J,t)}{m} \) (if migration energy cost can be neglected, which is mainly the case here, as we showed before). Notice that \( s \leq 1 \), otherwise the system is not schedulable on \( m \) processors.

Unfortunately, this speed \( s \) is usually not available. It is well known that the most efficient way to simulate this speed is to use \( |s| \) for some portion \( \rho \leq 1 \) of the time slice, and then \( |s| \) for the remainder of the time slice, where:
\[ |s| \times \rho + |s| \times (1 - \rho) = s. \]

Hence,
\[ \rho = \frac{|s| - s}{|s| - |s|}. \] (2)

Ideally, we will try to schedule each time slice of length \( L \) so the task system runs \( m \times \rho \times L \) cycles at speed \( |s| \), and the remaining cycles at speed \( |s| \). Unfortunately, this approach will not always work. If some \( \tau_i \in J \) has utilization \( u_i > |s| \), then these ideal speed settings will cause \( \tau_i \) to miss every deadline. Therefore, both the maximum utilization and the total utilization of \( J \) must be considered when selecting processor speeds.

We can use McNaughton’s RESCHEDULE algorithm (Algorithm 1) to determine the speeds required for a given task set. If \( J \)'s maximum task utilization is larger than \( s \), RESCHEDULE assigns the task with the largest utilization to its own processor, setting the processor speed to the utilization of the given task, and removes the task from consideration. This process is repeated until the maximum utilization does not exceed \( s \). After the dense tasks have all been assigned to their own processors, the remaining tasks are assigned to the remaining processors in the standard wrap-around fashion.

Algorithm 1: McNaughton’s Reschedule algorithm
1. \( J^{\text{rem}} = J; \)
2. \( k = 1; \)
3. while \( \exists \tau_i \in J^{\text{rem}} : u_i > U(J^{\text{rem}}) - k \) do
4. \( J_k \leftarrow \{ \tau_i \in J^{\text{rem}} \}; \)
5. \( u_k = u_i; \)
6. \( J^{\text{rem}} = J^{\text{rem}} \setminus \tau_i; \)
7. \( k = k + 1; \)
8. Set \( f_k, \ldots, f_n \) to \( s = \frac{U(J^{\text{rem}})}{m - k + 1} \) and assign \( J^{\text{rem}} \) to these processors in a wrap-around fashion;

Of course, we must adjust the RESCHEDULE algorithm because it uses a continuous speed spectrum. Below, we propose two ways of being as close as possible as this ideal speed selection. In the no constraint method, we allow the frequency to change at any time. In the constrained method, we assume that frequency can only change at time slice boundaries.

2.1 No constraint

1 RESCHEDULE was actually developed to determine the minimum makespan of a set of jobs. Algorithm 1 reframes McNaughton’s algorithm for speed-scaling.
Algorithm 2 is based on the Reschedule algorithm, but uses discrete speed settings. Because the speeds in Algorithm 2 usually do not exist, we need to use two speeds, $\lfloor s \rfloor_f$ for $\rho \times L$ units of time (using Equation 2), and $\lceil s \rceil_f$ for $(1 - \rho) \times L$ units of time, such that the total amount of processed work is the same as if we had used speed $s$. Once all the dense tasks have been assigned to a processor, we compute the optimal speed $s$ for the remaining tasks, and compute the two speeds for the remaining processors. We can now apply the classical wrap-around algorithm [16], with time slice of width $s$.

Algorithm 2: Task placement and speed selection without constraints

1. $k = 1$;
2. while $\exists \tau_i \in J_{\text{rem}} : u_i'(t) > \frac{U'(J_{\text{rem}}, t)}{m - i + 1}$ do
3. $\mathcal{J}_k \leftarrow (\tau_i, u_i')$; // $\tau_k$ is too dense
4. $s = u_i'(t)$; // It receives its own CPU
5. $\rho \leftarrow \frac{[\lfloor s \rfloor_f - s]}{\lceil s \rceil_f}$; // Ideal speed
6. $f_0[0, 0] = [\lfloor s \rfloor_f]$; // Simulate speed $s$
7. $f_0[0, 1] = [\lceil s \rceil_f]$; // with two speeds
8. $J_{\text{rem}} \leftarrow J_{\text{rem}} \setminus \tau_i$;
9. $s \leftarrow \frac{U'(J_{\text{rem}}, t)}{m - k + 1}$; // Ideal speed
10. $\rho \leftarrow \frac{[\lfloor s \rfloor_f - s]}{\lceil s \rceil_f}$;
11. foreach $k' \in \{k, \ldots, m\}$ do
12. $f_0[0, 0] = [\lfloor s \rfloor_f]; f_0[0, 1] = [\lceil s \rceil_f]$;
13. Allocate $J_{\text{rem}}$ on the $m - k + 1$ last CPUs by using wrap-around approach;


Lemma 1. Algorithm 2 will not introduce overlap.

Proof. Observe that a task migrates only if it is scheduled in the wrap-around phase of the algorithm. Therefore, any such task must have a utilization value smaller than $s$. Clearly, if all the processors executed at speed $s$, there would be no task parallelism. We need to show that parallelism is not introduced by using the speeds $[\lfloor s \rfloor_f]$ and $[\lceil s \rceil_f]$.

Consider any task $\tau_i$ that migrates from $\tau_k$ to $\tau_{k+1}$. Because $u_i \leq s$, we know

$$u_i \times L \leq (\rho \times [\lfloor s \rfloor_f] + (1 - \rho) \times [\lceil s \rceil_f]) \times L.$$ 

Because processors $\tau_k$ and $\tau_{k+1}$ change speeds simultaneously, it is not possible for $\tau_i$'s schedule to overlap.

Figures 1a and 1b show an example of a time slice, with 4 processors and 5 tasks. Notice that tasks $\tau_1$ and $\tau_2$ have a too large utilization factor to allow a uniform speed for all tasks. Indeed, the ideal speed would be $U'(t)/m = 13/16$, which is too low for $u_1$. Then, task $\tau_1$ receives its own processor, and we now consider the remaining tasks. The ideal speed for tasks $\tau_2, \ldots, \tau_5$ on 3 processors would be $12.33/16$, which is too slow for $\tau_2$, which also receives its own processor. The ideal speed for the remaining tasks on the two last processors is $12/16$, which is fast enough for all the remaining tasks.

2.2 Speed changes only between time slices

We now like to reconsider the task placement and speed scheduling assuming that changing the CPU speed can only be done at time slice boundaries. As in the previous approach, we would like to have all processors having an average speed of $s$. However, we may not be able to achieve exactly this average without switching speeds.

A first solution would be to run (at least) $k$ CPUs at speed $[\lfloor s \rfloor_f]$, and the remaining $m - k$ CPUs at speed $[\lceil s \rceil_f]$, where $k$ is the smallest value such that

$$k \times [\lfloor s \rfloor_f] + (m - k) \times [\lceil s \rceil_f] \geq m \times s.$$ 

Hence,

$$k = \left\lceil \frac{m \times s}{[\lceil s \rceil_f] - [\lfloor s \rfloor_f]} \right\rceil.$$ 

However, we face the same problem as in the previous solution: if some tasks have a utilization higher than $[\lfloor s \rfloor_f]$, this method does not work.

We propose a method solving this problem in Algorithm 3. We consider the tasks by decreasing $u(t)$, and fill the processors one by one. Once again, we must consider both the maximum utilization and the average per-processor utilization to determine the ideal speed setting $s$ for each processor. However, we must set the processor speed to $[\lfloor s \rfloor_f]$ for the entire time slice. Once the speed is determined, we assign tasks to the processor. As soon as a processor is full, we go to the next one, and find the ideal speed for the remaining tasks (which will be lower or equal to the previous processor, as we consider the tasks by decreasing utilization factor).

In Algorithm 3 $\delta$ represents the ratio of the time slice which is already assigned to tasks for the current processor $\tau_k$. For the very first processor, the minimal speed (if all the processor were using the same speed) is $\frac{U'(t)}{m}$, if this value is above the densest task (i.e., $\tau_1$), and $u_i'(t)$, otherwise. For the remaining tasks, as long as the densest remaining task $\tau_t$ can still fit on the current processor $\tau_k$, we add it to the processor (line 7) and update $\delta$ (line 8). Otherwise, we have to split the task onto processors $\tau_k$ and $\tau_{k+1}$. We first put on $\tau_k$ what still fits (line 10), and then compute the
speed of processor $\pi_{k+1}$. In the simplest case (denoted $s_1$), we simply divide the unassigned workload by the number of available processors. But there are two cases where this speed cannot be used. First, if this speed is lower than the utilization factor of the densest remaining task, we need to use this factor as the speed ($s_2$). Second, choosing a too low frequency could create an overlap between the execution of $\tau_i$ on $\pi_k$ and on $\pi_{k+1}$. We avoid this by adapting the speed ($s_3$). Finally, the speed of the next processor is set to the maximum of (the ceiling of) the three values $s_1, s_2$ and $s_3$.

Algorithm 3: Task placement and speed selection with speed changes at time slice boundaries

\begin{algorithm}
\begin{algorithmic}
\State $k = 1$; // Current processor
\State $i = 1$; // Current task
\State $\delta = 0$; // Filled part of $\pi_k$
\State $f_k = \lceil \max\{u_1', \frac{\ell_k}{m}\}\rceil$; // Speed of $\pi_k$
\State // Tasks are ordered by decreasing $u'$
\For{$i \in 1, \ldots, n$}
\If{$\delta + u' < f_k$}
\If{Enough room on $\pi_k$ for $\tau_i$}
\State $J_k \triangleq (\tau_i, u')$;
\State $\delta = \delta + u'$;
\Else
\State // Task $\tau_i$ is split on $\pi_k$ and $\pi_{k+1}$
\State $J_k \triangleq (\tau_i, f_k - \delta)$; // put the part that can be put on $\pi_k$
\State $s_1' = \frac{u'_i - (f_k - \delta) + \sum_{i' = i+1}^m u'_{i'}}{m - k}$; // ideal speed
\State $s_2' = u_{i+1}'$; // $s$ too slow to $\tau_{k+1}$
\State $s_3' = \frac{u'_i - f_k}{\delta} - 1$; // $s$ causes overlap
\State $+ k$; // Go to the next processor
\State $f_k' = \max\{s_1, s_2, s_3\}$;
\State $J_k \triangleq (\tau_i, u'_i - (f_{k-1} - \delta))$;
\State // Put the remaining part of $\tau_i$
\State $\delta = u'_i - (f_{k-1} - \delta)$;
\EndIf
\EndIf
\EndFor
\end{algorithmic}
\end{algorithm}

3. ACCOUNTING FOR EARLY COMPLETION

In the above section, we assumed that all jobs execute for their full worst-case execution time. In reality, jobs will rarely execute for that long. When tasks finish early, we may be able to adjust the schedule in response to the reduced workload. This strategy is called slack reclamation. In addition, the scheduler may optimistically allocate less than $u_i(t) \times L$ time units to a time slice in anticipation of an early task completion. This strategy is called aggressive speed selection. We will consider these strategies in this section.

Unlike the previous strategies, when slack reclamation or aggressive speed selection is used, it is possible that at a time slice boundary $t$ there exist tasks $\tau_i$ such that $u'_i(t) \neq u_i$. If this is the case, we may need to use a different strategy to allocate local execution times at the beginning of each time slice. Once the local execution times are determined, we can still set CPU speeds using the methods presented in Algorithms 2 and 3.

3.1 Slack reclaiming

When a task $\tau_i$ completes early (i.e., completes execution in less than $e_i$ time units), it introduces slack into the system. Specifically, if $\tau_i$ completes at time $t$ and $\eta_i(t) < e_i$, then the system load is reduced by $(e_i - \eta_i(t))$ time units. The slack normally increases the amount of processor idle time. Instead, the schedule may be altered to use the slack, making the whole system more flexible.

Intuitively the slack can be reclaimed in two ways: Intra-frame or Inter-frame. Intra-frame slack reclamation allows the scheduler to reclaim slack within a time slice when a task completes its execution early. By contrast, Inter-frame slack reclamation allows the scheduler to reclaim slack from tasks that completed early during some previous time slice. Algorithms 2 and 3 automatically perform Inter-frame reclamation by using $u_i(t)$ when allocating $\tau_i$'s local execution time. If a task completes early, its remaining execution time is set to 0. Therefore, this task's workload is removed from consideration when setting the processor speeds. For Inter-frame slack reclamation, we can increase the local execution of other tasks in order to keep the processor busy for the full $L$ time units, if possible.

The intra-frame slack reclamation algorithm (given in Algorithm 4) will be called within a time slice whenever a task completes early. Once it is called, it will distribute the released slack to other tasks (extend other tasks’ local execution work $\ell_{i,k}(t)$). For simplicity, this algorithm restricts the “extended” tasks to be non-migratory tasks assigned to the same processor as the early completion task so as not to create additional migrations. It will firstly select tasks with some local remaining work to extend. This will avoid adding more context switches to the schedule. If no such tasks exist, the algorithm will then select tasks with no more local remaining work to extend.

Algorithm 4: Intra-frame slack reclamation

\begin{algorithm}
\begin{algorithmic}
\State \textbf{input}: current time $t'$, TS boundary prior than $t'$, processor $k$, early completion task $\tau_i$
\State \textbf{output}: tasks allocated according to their index
\State 1 \textbf{Assume} all tasks are allocated according to their index in ascending order
\State 2 \textbf{slack} = $\ell_{i,k}(t')$;
\State 3 $l = i + 1$;
\While{$\exists \tau_i \in J_k(t)$ and $\tau_i$ is not migratory and slack > 0}
\State 5 $\alpha = \min(\eta_i(t'), \ell_{i,k}(t'))$;
\State 6 \textbf{slack} = slack - $\alpha$;
\State 7 $\ell_{i,k}(t') = \ell_{i,k}(t') + \alpha$;
\State 8 $l = l + 1$;
\EndWhile
\State 10 $l = i + 1$;
\While{$\exists \tau_i \in J_k(t)$ and $\tau_i$ is not migratory and slack > 0}
\State 12 $\alpha = \min(\eta_i(t'), \ell_{i,k}(t'))$;
\State 13 \textbf{slack} = slack - $\alpha$;
\State 14 $\ell_{i,k}(t') = \ell_{i,k}(t') + \alpha$;
\State 15 $l = l - 1$;
\EndWhile
\end{algorithmic}
\end{algorithm}
We need to show that this algorithm will correctly schedule the current time slice and that all future time slices will still be schedulable. The schedule of the current time slice is correct if (i) no tasks execute in parallel and (ii) all tasks are able to execute for their allocated local execution time. By construction, our approach only increases the local execution times of non-migratory tasks. Therefore, the adjustment of local execution times within the time slice will not introduce any task parallelism within the time slice. Also, all local execution times are increased only enough to use the released slack. Therefore, the total allocated work in each time slice cannot increase above $L_j$.

It remains to be shown that this change will not cause any deadline misses. We begin with the following lemma.

**Lemma 2.** Let $J$ be any task set feasible on $m$ processors. Consider any time slice $\sigma_j$ such that $u'_i(t_j-1) \leq u_i$ for all $\tau_i \in J$. Assume the time slice is scheduled using Algorithm 4 in conjunction with either Algorithm 2 or Algorithm 3. Then the schedule of the time slice will be correct and $u'_i(t_j) \leq u_i$ for all $\tau_i \in J$.

**Proof.** We first show that the schedule of the time slice will be correct. We have already shown that Algorithm 4 will correctly adjust the schedule of a time slice. It remains to show that the task allocation algorithm will build a correct schedule for the time slice – i.e., there is no task parallelism and all tasks are able to complete their allocated local execution times.

At the beginning of the time slice, the task allocation algorithm schedules each task $\tau_i$ to execute for $u'_i(t_j-1) \times L$ time units on processor(s) with average speed $s \geq u'_i(t_j-1)$. Note that $1 \geq u_i \geq u'_i(t_j-1)$. Therefore, it is possible to set the processor speed to some $s \geq u'_i(t_j-1)$. We can use the same argument as we did in Lemma 2 to prove there will be no task parallelism.

We know the processors can execute all tasks for their allocated execution times provided the total allocated time is at most $L \times m$. We now show this must be true.

\[
\sum_{i=1}^{n} \sum_{k=1}^{m} \ell_i,k(t_j-1) = \sum_{i=1}^{n} u'_i(t_j-1) \times L \\
\leq \sum_{i=1}^{n} u_i \times L \\
\leq m \times L.
\]

Therefore, the schedule of the time slice is correct.

We now show that $u'_i(t_j) \leq u_i$ for all $\tau_i \in J$. If $d_i(t_j) = t_j$, then $\tau_i$ releases a new job at time $t_j$. In this case, $u'_i(t_j) = u_i$. If $d_i(t_j) > t_j$, then $u'_i(t_j) = \frac{r_i(t_j)}{d_i(t_j) - t_j}$. There are three possibilities: (i) $\tau_i$ completed its execution early during the time slice, (ii) $\tau_i$ executed for exactly $u'_i(t_j-1) \times L$ time units during the time slice, (ii) due to slack reclamation, $\tau_i$ executed for more than $u'_i(t_j-1) \times L$ time units during the prior time slice. In the first case, $r_i(t_j) = 0$. Therefore $u'_i(t_j) = 0 < u_i$. Below, we show that $u'_i(t_j) \leq u_i$ in the second and third cases as well.

\[
u'_i(t_j) = \frac{r_i(t_j)}{d_i(t_j) - t_j} \\
\leq \frac{r_i(t_j - 1) - u'_i(t_j - 1) \times L}{d_i(t_j) - t_j} \\
\leq u_i \times (d_i(t_j - 1) - t_j - 1) - u_i \times (t_j - t_j - 1) \\
\leq d_i(t_j - 1) - t_j \\
= u_i.
\]

With this Lemma, we can prove that the system schedule is correct when we implement slack reclamation with Algorithms 2 or 3.

**Theorem 1.** Let $J$ be any task set feasible on $m$ processors. Assume $J$ is scheduled using either Algorithm 2 or Algorithm 3 with slack reclamation. Then the resulting schedule will be correct.

**Proof.** Initially, $u'_i(0) = u_i$ for all $\tau_i \in J$. By induction, Lemma 2 demonstrates that at every time slice $[t_j-1, t_j)$, $u'_i(t) \leq u_i$ for all $\tau_i \in J$ and $\tau_i$ will execute for at least $u'_i(t_j-1) \times L$ time units during the time slice. Let $\tau_i$ be any task that has a deadline at time $t_j$. Then $\tau_i$ will execute for at least $u'_i(t_j-1) \times L = \frac{r_i(t_j-1)}{d_i(t_j-1) - t_j - 1} \times L = r_i(t_j-1)$ time units during the time slice. Therefore, all tasks meet their deadlines.

### 3.2 Aggressive speed selection

In the discussion above, we have set CPU speeds assuming all tasks will execute to their worst case execution time. In this section, we consider deliberately under-allocating local task execution times, anticipating early task completions. Such an approach would modify the calculation of $t_i,k(t)$ in the task allocation and speed setting algorithms (Algorithms 2 and 3). The remainder of these algorithms can be used without change. In this section, we examine how to safely allocate local task execution times while allowing tasks to execute at a rate slower than their utilization values.

In this scenario, each task $\tau_i$ has an expected execution time $e_{\text{exp},i}$ in addition to its worst-case execution time $e_i$. Using $e_{\text{exp},i}$ and $e_i$, we can find $\tau_i$’s expected remaining execution time $e_{\text{exp},i}(t)$ and expected remaining utilization ratio $u'_{\text{exp},i}(t)$ as follows:

\[
r_{\text{exp},i}(t) = \begin{cases} 
    e_{\text{exp},i} - \eta(t) & \text{if } e_{\text{exp},i}(t) > \eta(t) \\
    e_i - \eta(t) & \text{otherwise}
\end{cases} \\
u'_{\text{exp},i}(t) = \frac{r_{\text{exp},i}(t)}{d_i(t) - t}.
\]

Alternatively, we could use a more complicated approach for determining the expected remaining execution time. For example, the execution time could be expressed in the form of a probability distribution function (pdf). In this case, the expected remaining execution could be evaluated using a conditional probability. Below, we present the analysis
assuming a single expected execution time. This analysis could easily be extended to accommodate other models.

Initially, the system will use the $u_{exp}(t)$ to determine local execution times. If the task executes for longer than the expected execution time or if the feasibility of the system can no longer be guaranteed, then the scheduler will increase the task allocations. A key issue is determining a feasibility test that will allow tasks to execute as slowly as possible while still guaranteeing that all tasks will be able to meet their deadlines even if the system suffers worst-case demand.

Note that it is not sufficient to ensure that $\sum u'_i(t) \leq m$. Consider the following example: A task $\tau_n$ completes at time $t' < d_n(t')$. Let $t_2 = d_n(t)$ and assume there are multiple time slices between $\tau_n$’s early completion and its deadline. If we simply assure that $\sum u'_i(t) \leq m$ at each time slice boundary, we could under-allocate the remaining tasks so much that $\sum u'_i(t_j-1) = m$. Even if all processors execute at speed 1 during the time slice $\sigma_j$, we will have $\sum u'_i(t_j) = m + u_n$. This is because tasks $\tau_1$ through $\tau_{n-1}$ will require the full capacity of the m processors ($u'_i(t_j) = u'_i(t_j-1)$). On the other hand, $\tau_n$ completed early and released a new job at time $t_j$. Therefore $u'_i(t_j-1) = 0$ and $u'_i(t_j) = u_n$.

The above discussion tells us that we need to consider more than just the worst-case execution times of the currently released jobs when under-allocating local execution times. In addition, we need to consider the worst-case load of upcoming jobs. Specifically, after allocating execution times for the time slice $\sigma_j$, we need to make sure that the resulting workload at time $t_j$ will not cause any deadline misses even if all tasks execute for their worst-case execution times. Therefore, we can verify the validity of our allocation choices for $t_j-1$ by checking the resulting worst-case load during the intervals $[t_j,d_i(t_j)]$ for $i = 1, \ldots, n$.

This process involves three steps:

- Finding a method for determining optimistic local execution times $\ell_i(t_j-1)$ for the time slice $[t_j-1,t_j]$.

- Developing a test for checking if these local execution times are “safe” i.e., if the system will continue to be feasible at time $t_j$ even if all tasks execute for their worst-case execution times.

- Find a method for allocating less optimistic local execution times for time slice $\sigma_j$ if the optimistic ones result in the worst-case remaining system load being infeasible.

As mentioned above, we would like our optimistic local execution times to be based on the expected execution times rather than the worst case execution times. Specifically, we would like to allocate $u'_{exp}(t_j-1) \times L$ units of execution time for $\tau_i$ during a time slice of length $L$. However, we must ensure that $\tau_i$’s worst case remaining work can be completed before its deadline. Therefore, we must allocate enough execution time at $t_j-1$ to ensure all tasks with $d_i(t_j-1) = t_j$ meet their deadlines and tasks with $d_i(t_j-1) > t_j$ have enough time to compute their remaining work.

These observations give the following approach for finding our optimistic local execution times:

$$
l_1 = u'_{exp}(t_j-1) \times L
$$

$$
l_2 = r_i(t_j-1) - (d_i(t_j-1) - t_j-1 - L)
$$

$$
\ell_i(t_j-1) = \max\{l_1, l_2\}
$$

(3)

We now need to develop a test to determine if the above choices for $\ell_i(t_j-1)$ are safe. As discussed above, we must consider the current load and the upcoming load to ensure safety. To this end, we present the following average load function, $\hat{u}_i(t_1,t_2)$, for $\tau_i$ over a given interval $[t_1, t_2]$ and the total average load function $\hat{U}(t_1, t_2)$.

$$
\hat{u}_i(t_1,t_2) = \begin{cases} 
\frac{r_i(t_j-1)}{d_i(t_j-1) - t_j} - \frac{t_j - d_i(t_j-1)}{d_i(t_j-1) - t_j} & \text{if } d_i(t_j) \geq t_2 \\
\frac{t_j - d_i(t_j-1)}{d_i(t_j-1) - t_j} & \text{if } d_i(t_j) < t_2
\end{cases}
$$

$$
\hat{U}(t_1,t_2) = \sum_{i=1}^{n} \hat{u}_i(t_1,t_2)
$$

This function evaluates the average system utilization based on its worst-case load over the interval $[t_1, t_2]$. Note that $\hat{u}_i(t_1,t_2) \leq 1$ because

$$
\min\{u'_i(t_1), u_i\} \leq \hat{u}_i(t_1,t_2) \leq \max\{u'_i(t_1), u_i\} \forall \tau_i \in \mathcal{J}.
$$

The lemma below demonstrates that the worst case average load can be kept from increasing if the local execution times satisfy one of two conditions. The first condition executes the time slice at its current worst-case average load level. However, if there are a few tasks with $u'_i(t_j-1) >> u_i$, it might not be possible to attain this goal. This leads to the second condition, which executes all tasks as much as possible.

**Lemma 3.** Consider two deadlines $t_{j-1}$ and $t_j$, where $t_2 = d_i(t_j)$. If $\tau_i$, if $\hat{U}(t_{j-1}, t_j) \leq m$ and the local execution times $\ell_i(t_{j-1})$ satisfy one of the following conditions

- $\sum_{j=1}^{m} \ell_i(t_{j-1}) \geq L \times \hat{U}(t_{j-1}, t_j)$

- $\ell_i(t_{j-1}) \geq \min\{r_i(t_{j-1}), L \times \hat{u}_i(t_{j-1}, t_j)\}$, or

- $\ell_i(t_{j-1}) = \min\{r_i(t_{j-1}), L\}$.

then

$$
\hat{U}(t_{j-1}, t_j) \leq \hat{U}(t_{j-1}, t_j).
$$

**Proof.** We consider each case separately.

- **Case 1:** Consider any task $\tau_i$ such that $d_i(t_{j-1}) \leq t_j$. Then,

$$
\hat{u}_i(t_{j-1}, t_j) = \frac{r_i(t_{j-1}) - \ell_i(t_{j-1}) + u_i(t_j - d_i(t_{j-1}))}{t_j - t_j} = \frac{r_i(t_{j-1}) + u_i(t_j - d_i(t_{j-1}))}{t_j - t_j} \times \left(1 + \frac{t_j - t_{j-1}}{t_j - t_j}\right)
$$

$$
\max\{r_i(t_{j-1}), u_i(t_j - d_i(t_{j-1}))\} \times \frac{\ell_i(t_{j-1})}{t_j - t_j} = \hat{u}_i(t_{j-1}, t_j) \times \left(1 + \frac{L}{t_j - t_j}\right).
$$

Note that if $d_i(t_{j-1}) = t_j$ then $\hat{u}_i(t_j) = u_i$, so the above expression holds. Similarly, if $d_i(t_{j-1}) > t_j$, then

$$
\hat{u}_i(t_{j-1}, t_j) = \frac{r_i(t_{j-1}) - \ell_i(t_{j-1})}{d_i(t_{j-1}) - t_j} = \frac{r_i(t_{j-1})}{d_i(t_{j-1}) - t_j} - \frac{\ell_i(t_{j-1})}{d_i(t_{j-1}) - t_j} = \hat{u}_i(t_{j-1}, t_j) \times \left(1 + \frac{L}{d_i(t_{j-1}) - t_j}\right)
$$

$$
\ell_i(t_{j-1}) \times \frac{\ell_i(t_{j-1})}{d_i(t_{j-1}) - t_j}.
$$

(4)
Therefore, the total load is
\[
\hat{U}(t_j, t_k) \leq \hat{U}(t_{j-1}, t_k) + \sum_{i=1}^{n} \frac{L \times \hat{u}_i(t_{j-1}, t_k)}{\max\{t_x, d_i(t_{j-1})\} - t_j} - t_j
\]
\[= \hat{U}(t_{j-1}, t_k) + \sum_{i=1}^{n} \frac{L \times \hat{u}_i(t_{j-1}, t_k) - \ell_i(t_{j-1})}{\max\{t_x, d_i(t_{j-1})\} - t_j}.
\]

Observe
\[
\sum_{i=1}^{n} \frac{L \times \hat{u}_i(t_{j-1}, t_k) - \ell_i(t_{j-1})}{\max\{t_x, d_i(t_{j-1})\} - t_j} - t_j
\]
\[= \sum_{i=1}^{n} \frac{L \times \hat{u}_i(t_{j-1}, t_k) - \ell_i(t_{j-1})}{t_x - t_j}
\times \left(1 + \frac{\max\{t_x, d_i(t_{j-1})\} - t_x}{\max\{t_x, d_i(t_{j-1})\} - t_j}\right)
\]
\[= \sum_{i=1}^{n} \frac{L \times \hat{u}_i(t_{j-1}, t_k) - \ell_i(t_{j-1})}{t_x - t_j}
+ \sum_{d_i(t_{j-1}) > t_x} (L \times \hat{u}_i(t_{j-1}, t_k) - \ell_i(t_{j-1}))
\times \frac{d_i(t_{j-1}) - t_x}{d_i(t_{j-1}) - t_j}
\]
\[\leq \frac{L \times \hat{U}(t_{j-1}, t_k) - \sum_{i=1}^{n} \ell_i(t_{j-1})}{t_x - t_j}
\]
\[\leq 0.
\]

The first inequality holds because \(\hat{u}_i(t_{j-1}, t_k) = \frac{r_i(t_{j-1})}{d_i(t_{j-1})}\) when \(d_i(t_{j-1}) > t_x\). Therefore, \(L \times \hat{u}_i(t_{j-1}, t_k) \leq r_i(t_{j-1})\) and, by our choices for \(\ell_i(t_{j-1})\), it must be the case that \(\ell_i(t_{j-1}, t_x) \geq L \times \hat{u}_i(t_{j-1}, t_k)\). The second inequality holds because the total allocated execution time cannot be smaller than \(L \times \hat{U}(t_{j-1}, t_k)\).

This proves the first case. **Case 2:** If we cannot increase the local execution times enough to reach the target load, we use the maximum possible allocations for all tasks. We now show that if \(\ell_i(t_{j-1}) = \min\{r_i(t_{j-1}), L\}\) then \(\hat{u}_i(t_j, t_k) \leq \hat{u}_i(t_{j-1}, t_k)\).

We first consider the case where \(\ell_i(t_{j-1})\) is increased to \(r_i(t_{j-1})\). If \(d_i(t_{j-1}) > t_x\) then
\[\hat{u}_i(t_j, t_k) = u_i(t_x - d_i(t_{j-1}))
\leq r_i(t_{j-1}) + u_i(t_x - d_i(t_{j-1}))
\]
\[= r_i(t_{j-1}) + u_i(t_x - d_i(t_{j-1}))
\]
\[= \hat{u}_i(t_{j-1}, t_k).
\]

We now consider the case where \(\ell_i(t_{j-1})\) is increased to \(L\). Observe that, regardless of whether \(d_i(t_{j-1}) > t_x\) or \(d_i(t_{j-1}) \leq t_x\), the formula for \(\hat{u}_i(t_j, t_k)\) decreases both the numerator and the denominator by \(L\). For example, if \(d_i(t_{j-1}) > t_x\), then
\[\hat{u}_i(t_j, t_k) = \frac{r_i(t_{j-1}) - L}{t_x - t_j} = \frac{r_i(t_{j-1}) - L}{t_x - t_x - t_j}.
\]

It is easy to verify that \(b > a \Rightarrow \frac{a}{b} < \frac{a}{b'}\). Therefore, when \(\ell_i(t_{j-1}) = L\), \(\hat{u}_i(t_j, t_k) \leq \hat{u}_i(t_{j-1}, t_k)\), as desired.

Because all values of \(\hat{u}_i\) decrease when \(\ell_i(t_{j-1})\) is set to its maximal value, it must be the case that \(\hat{U}(t_j, t_k) \leq \hat{U}(t_{j-1}, t_k)\), which, by assumption, is bounded above by \(m\).

Therefore, if \(\hat{U}(t_j, t_k) \leq m\) and we find that using optimistic execution times at time \(t_{j-1}\) will cause \(\hat{U}(t_j, t_k)\) to be larger than \(m\), we can replace our optimistic execution times with the ones presented in the lemma to ensure \(\hat{U}(t_j, t_k) \leq m\). Unfortunately, this strategy takes \(O(n^2)\) time because we need to check \(\hat{U}(t_j, t_k) \leq m\) for all upcoming deadlines \(t_x\). Observe that,
\[
\max_{1 \leq x \leq n} \left\{ \sum_{i=1}^{n} \hat{u}_i(t_j, t_k) \right\} \leq \frac{\sum_{i=1}^{n} \hat{u}_i(t_j, t_k)}{ \sum_{i=1}^{n} \ell_i(t_{j-1})}
\]

By the definition of \(\hat{u}_i(t_j, t_k)\), if \(t_x \leq d_i(t_{j-1})\), \(\hat{u}_i(t_j, t_k) = u_i(t_j)\). As \(t_x\) increases beyond \(d_i(t_{j-1})\), the value of \(\hat{u}_i(t_j, t_k)\) increases if \(u_i(t_j) < u_i\) and decreases if \(u_i(t_j) > u_i\). Therefore, we define \(\hat{u}_i(t_j)\) as follows
\[
\hat{u}_i(t_j) = \max_{1 \leq x \leq n} \left\{ \hat{u}_i(t_j, t_x) \right\} = \begin{cases} u_i(t_j) & \text{if } u_i(t_j) > u_i \\ \hat{u}_i(t_j, D_j) & \text{otherwise} \end{cases}
\]

where \(D_j = \max\{d_i(t_{j-1}) \mid u_i(t_j) < u_i\}\) is the latest deadline of any task that is “behind” in the schedule at time \(t_j\). Then, if we ensure that
\[
\hat{U}(t_j) \leq \frac{n}{\sum_{i=1}^{n} \ell_i(t_{j-1})} \leq m \quad \text{and } D_j \leq D_{j-1}
\]

we are guaranteed to have \(\hat{U}(t_j, t_k) \leq m\) for all upcoming deadlines \(t_x\). Note that if \(u_i(t_{j-1}) \leq u_i\) then \(\hat{u}_i(t_{j-1})\) cannot exceed \(u_i\) if \(r_i\) executes for \(\hat{u}_i(t_{j-1}) \times L\) time units during \(\sigma_j\). This leads us to the following steps for optimistic allocation of local execution times.

- Allocate \(\ell_i(t_{j-1})\) optimistically as described above.
- Find \(\hat{U}^*(t_j)\) assuming for the above execution times are used.
- If \(\hat{U}^*(t_j) \leq m\), proceed with the time slice execution.
- Otherwise let \(\ell_i(t_{j-1}) = \min\{\hat{u}_i(t_{j-1}) \times L, r_i(t_{j-1})\}\) and let \(\ell_i^{max}(t_{j-1}) = \min\{L, r_i(t_{j-1})\}\).
- If \(\sum_{i=1}^{n} \ell_i^{max}(t_{j-1}) \leq L \times \hat{U}^*(t_j)\), let \(\ell_i(t_{j-1}) = \ell_i^{max}(t_{j-1})\). Otherwise, \(\ell_i(t_{j-1}) = \ell_i^{max}(t_{j-1})\) and increase values of \(\ell_i(t_{j-1})\) (starting with tasks having \(u_i(t_{j-1}) > u_i\)) until \(\sum_{i=1}^{n} \ell_i(t_{j-1}) = L \times \hat{U}^*(t_j)\).

This strategy allows the scheduler to under-allocate tasks while ensuring the worst-case load can always be met. We note that this strategy may be adapted for other time-slice-based scheduling algorithms.
One disadvantage of this strategy is that the time slices are no longer scaled copies of one another. Therefore, we can no longer reduce overheads by mirroring the time slices, leading to additional preemptions, migrations and (possibly) frequency switches. In future, we plan to explore methods to reduce overheads imposed by our algorithm, such as finding allocations to reduce migration or ones that will work for several time slices. Such an approach may mean we execute slightly faster than necessary in order to benefit from the reduced system overhead.

### 4. EXPERIMENTS

We considered multiprocessor platforms containing $m = 2, 4, 8, 16$, and 32 identical processors. The number of tasks we consider is dependent on the number of processors in the current scenario. Motivated by [13], we define $\alpha = \frac{n}{m}$ and simulate each $\alpha = 2, 5, 10, 15, 20$. Finally, we allow total system utilization $U(J)$ to range from 0.025 * $m$ to $m$ in step of 0.05 * $m$.

According to [14], we use Stafford’s Randfixedsum algorithm to generate unbiased sets of utilizations for multiprocessor platforms. We randomly assigned period from 5 ms to 200 ms for each task. For each value of $m, \alpha$ and $U(J)$, we randomly generated 250 task sets and simulated their schedules, calculating the percentage of energy savings as compared to a system executing all processors at $s = 1$. Thus, we generated a total of 13,125 task sets.

Since workload varies according to completion times, we investigated the impact of early-completion. In our experiments, we give each task two distinct execution times: minimum execution time ($e_{\text{min}}$) and maximum execution time ($e_{\text{max}}$). The actual execution time is randomly assigned between $e_{\text{min}}$ and $e_{\text{max}}$, and we let the expected execution time equal $e_{\text{exp}} = \frac{e_{\text{min}} + e_{\text{max}}}{2}$. We ran three completion-time scenarios. In the early-completion scenarios, the range for the actual completion times is $[e_{\text{min}}, e_{\text{min}} + \Delta/3)$, where $\Delta = e_{\text{max}} - e_{\text{min}}$. The mid-completion and late-completion scenarios generate completion times in the middle third and upper third of the interval $[e_{\text{min}}, e_{\text{max}}]$, respectively.

We simulated each task set 18 times. Simulations varied depending on speed assignment strategy and workload variation. Speeds were assigned according to the following methods:

- $\text{wsc}$: Speed can change only on time slice boundaries,
- $\text{woc}$: Speed can change during time slice, and
- $\text{hcs}$: Speed can only be the highest available speed. Workloads varied according to the following: with SR: the slack reclamation run allows system readjust jobs’ allocation to account for early-completion and without SR: the system does not readjust jobs’ allocation time.

Figure 2 illustrates our results for $m = 8$ processors with $\alpha = 5$ (40 tasks) with early and late completion times. We observe that the slack reclamation makes only a small difference in the savings. The ability to change speeds within the time slice does provide advantages, though. This difference is more marked when there are fewer tasks. We found that the other scenarios had similar results. In general, all our approaches were able to provide significant savings. Due to lack of space, the complete experiment result can be found in the extended version [15].

### 5. CONCLUSION

This paper has presented a DVFS extension to the DP-Wrap scheduling algorithm. The algorithm adjusts execution times within a time slice to take advantage of released slack. Additionally, we present a method for diverging from the standard DP-Wrap allocation in anticipation of a early task completion times. The methods presented in this paper provide additional flexibility to the DP-Wrap scheduling algorithm. Using these methods, we are able to reduce processor speeds more flexibly. We have demonstrated our approach can provide significant savings in energy consumption.

We note that the ability to diverge from the utilization-based execution times provide the algorithm additional flexibility, which could be used towards a variety of goals (e.g., reducing migrations). Furthermore, these approaches may be adapted to other time-slice-based multiprocessor scheduling algorithms.

### 6. REFERENCES


