Slack Reclamation for Real-Time Task Scheduling over Dynamic Voltage Scaling Multiprocessors *

Jian-Jia Chen, Chuan-Yue Yang, and Tei-Wei Kuo
Department of Computer Science and Information Engineering
Graduate Institute of Networking and Multimedia
National Taiwan University, Taipei, Taiwan, ROC
Email:{r90079, r92032, ktw}@csie.ntu.edu.tw

Abstract

In the past decades, a number of research results have been reported for energy-efficient task scheduling over uniprocessor and multiprocessor environments. While researchers have started the exploring of slack reclaiming for tasks during run time, little work has been done for multiprocessor cases. This paper proposes a set of multiprocessor energy-efficient task scheduling algorithms with different task remapping and slack reclaiming schemes, where tasks have the same arrival time and share a common deadline. Tasks are reassigned to processors dynamically, and the slack time is reclaimed to slow down the execution speeds of the remaining tasks for energy efficiency. Extensive simulations were performed to provide insights. The energy consumption could be reduced up to 29% in the experiments, compared to the previous work.

Keywords: Slack Reclamation, Energy-Efficient Scheduling, Real-Time Task Scheduling, Multiprocessor Scheduling.

1 Introduction

With the advanced technology in VLSI circuit designs, many modern processors could now operate at various supply voltages, where different supply voltages lead to different processing speeds. Many computer systems, especially embedded systems, are now equipped with voltage-scaling processors and adopt various energy-efficient strategies in managing their subsystems intelligently. Well-known example processors for such systems are Intel StrongARM SA1100 processor [18] and the Intel XScale [19]. The power consumption is usually a convex and increasing function of processor speeds and is highly dependent on the hardware designs and implementations. The slower the speed is, the less the power consumption is. As pointed out in [2], multiprocessor implementations of real-time systems could be even more energy-efficient than uniprocessor implementations, due to the convexity of power consumption functions.

Energy-efficient real-time task scheduling is to derive a schedule for real-time tasks with the minimization on the energy consumption such that their timing constraints can be met. For aperiodic real-time task scheduling, Yao, et al. [22] and Bansal, et al. [6] proposed off-line optimal (and on-line competitive) scheduling algorithms to minimize the energy consumption of task executions in a uniprocessor environment. When periodic real-time task scheduling is considered, Aydin, et al. [3, 15] proposed energy-efficient scheduling algorithms based on the earliest deadline first scheduling strategy. When all of the power consumption functions of tasks are the same, Chen, et al. [8, 21] proposed approximation algorithms to schedule frame-based tasks over multiprocessors with and without independent voltage scaling over processors. Heuristic algorithms for periodic tasks in multiprocessor environments were proposed in [1, 5]. In [10, 11, 24], energy-efficient scheduling algorithms based on list heuristics are proposed to schedule real-time tasks with precedence constraints. Mishra, et al. [16] explored energy-efficient scheduling issues with the considerations of task communication delay. And in [14, 17, 23], some real cases were studied.

Most previous study on multiprocessor energy-efficient scheduling assumed that the actual execution time of a task is equal to the worst-case execution time, such as those in [1, 5, 8, 21]. This work is motivated by the scheduling of tasks in reality, where tasks might complete earlier than their worst-case execution times [4, 25]. We focus our study on the scheduling of frame-based real-time tasks, in which all of the tasks arrive at the same time and share a common deadline. The closest related work was done by Zhu, et al.

* Support in parts by research grants from ROC National Science Council (NSC-94-2213-E-002-007 and 94-2752-E-002-008-PAE).
In the study, an algorithm with a global queue of ready tasks is presented for the selection of a candidate such that the slack time, due to the early completion of another task, is used to slow down the execution speed of the selected task.

In this paper, we propose a slack reclamation scheme to reduce the energy consumption during the run time. Given a task set, we first derive a schedule in an off-line fashion, where a schedule consists of task assignments over processors and task execution speed assignments. The schedule is then revised, whenever the slack reclamation scheme is applied, during the run time. The scheme consists of two phases: the task remapping phase and the slack reclaiming phase. It is applied repeatedly whenever a task completes its execution. In the task remapping phase, the current schedule is first revised such that tasks that have not started their executions yet might be moved to different processors for executions. In the slack reclaiming phase, the slack time is then used to slow down the execution speeds of some tasks.

We propose two dynamic task remapping algorithms and four slack reclaiming algorithms in the proposed scheme. In the simulations, the proposed scheme was shown being able to reduce the energy consumption up to 29%, compared to the previous work [25].

The rest of this paper is organized as follows: Section 2 defines the system models and formulate the energy-efficient scheduling problem. Our proposed scheme is presented in Section 3. Simulation results are shown in Section 4. Section 5 concludes this work.

2 System Models and Problem Definition

This paper targets energy-efficient scheduling over homogeneous multiprocessors, where the power consumption function of each task is the same on every processor. The power consumption function \( P() \) of the dynamic voltage scaling part of a processor is a function of the adopted processor speed \( s \) [7, 20]:

\[
P(s) = C_{ef} V_{dd}^2 s, \tag{1}
\]

where \( s = k \frac{(V_{dd} - V_t)^2}{V_{th}} \), and \( C_{ef}, V_t, V_{dd}, \) and \( k \) denote the effective switch capacitance, the threshold voltage, the supply voltage, and a hardware-design-specific constant, respectively (\( V_{dd} \geq V_t \geq 0, k > 0, \) and \( C_{ef} > 0 \)). Note that the power consumption function is a convex and increasing function of processor speeds. When \( V_t = 0 \), the power consumption function \( P(s) \) could be rephrased as a cubic function of the processor speed \( s \). In this paper, we assume that both \( P(s) \) and \( P(s)/s \) are convex and increasing functions.

In this study, we assume that each processor could operate at any non-negative speed in \([0, S_{max}]\), where \( S_{max} \) is the maximum available processor speed provided by the processor. We also assume that the time and energy overheads on speed (voltage) switching are assumed being negligible, such as those researches in [3, 6, 15, 22, 26]. We consider systems that the speed of each processor could be adjusted independently from each another. For the simplicity of presentation, we set \( S_{max} \) as one and have processor speeds (and related variable values) normalized in the range \([0, 1]\) in the derivation of the to-be-proposed scheme and the simulations. The computation executed in a time interval is linearly proportional to the processor speed, and that the energy consumed for a processor in the execution of a task at the processor speed \( s \) for \( t \) time units is the multiplication of \( t \) and its corresponding power consumption \( P(s) \) at the speed \( s \).

We are interested in the scheduling of frame-based real-time tasks, where each task in a given task set \( T \) is ready at time 0 with a common deadline \( D \). Tasks are independent. The scheduling of each frame of a length \( D \) is thus done repeatedly. We shall finish the executions of all tasks before the end of the frame, i.e., the deadline \( D \). Each task \( \tau_i \) in \( T \) is characterized by its estimated worst-case execution time \( e_i \), at the highest available speed \( S_{max} \), i.e., 1, and its actual execution time \( a_i \) at \( S_{max} \). For the rest of this paper, the worst-case execution time and actual execution time of a task are both referred to those at the speed \( S_{max} \) implicitly. We must emphasize that the processor speeds, the worst-case execution times, the actual task execution times, the power consumption function, and the deadline \( D \) are all normalized whenever they are referenced for the rest of this paper (because \( S_{max} \) is now set as 1).\(^1\) For a task, we assume that its worse-case execution time is known in a priori, while its actual one is determined at the run time. The actual execution time of task \( \tau_i \) at processor speed \( s \) is inversely proportional to processor speed \( s \), i.e., \( a_i/s \). Multiprocessor systems considered in this paper only execute one task on one processor without task migration. That is, once a task starts its execution on some processor, it is not allowed to migrate to another one.

In this work, we consider energy-efficient real-time task scheduling over \( M \) homogeneous processors without task migration. A schedule of a task set \( T \) within a frame is a mapping of the executions of the tasks of \( T \) in a frame to processors in which processor speeds for the corresponding execution intervals of the tasks are defined. A schedule is feasible if no task misses its deadline \( D \), the speed limitation is not violated, and no task migration is observed.

\(^1\)After the normalization, \( D \) becomes \( D \cdot S_{max} \).


3 Our Slack Reclamation Scheme

3.1 Overview

In this section, we propose our slack reclamation scheme to reduce the overall energy consumption of the multiprocessor system. First, we map tasks onto processors and derive a static schedule by considering the worst-case execution times of tasks. The static schedule could be derived by adopting the 1.13-approximation algorithm proposed in [8], which assigns tasks on processors with the smallest utilization \( \sum \frac{T_i}{p} \) in a non-increasing order of their worst-case execution times, and each processor then operates at the minimum available speed to finish all of the tasks assigned to the processor before the deadline \( D \). The to-be-proposed slack reclamation scheme would not work if there does not exist any available processor speed to execute all of the tasks on any processor on time in a derived static schedule. More processors or processors with higher speeds might be needed. Let \( S_{static} \) denote the schedule derived from the 1.13-approximation algorithm, i.e., Algorithm LTF in [8], for the rest of this paper. In schedule \( S_{static} \), the tasks assigned to be executed on processor \( p_m \) are executed in a non-increasing order of their worst-case execution times from time 0 to time \( D \).

The energy consumption could be further reduced by adjusting the processing speeds of tasks in a dynamic manner. Because tasks might complete their executions earlier than their worst-case estimated execution times, the unused time, called slack, could be used to reduce the processing speeds of some remaining tasks while the resulting schedule is still feasible. Suppose that some tasks in \( T \) complete at time instant \( t^* \) within a frame. The scheduler could reschedule the tasks that have not started their executions yet to reduce the energy consumption on the fly. In the current schedule, let the set of tasks assigned to start their executions on processor \( p_m \) no earlier than \( t^* \) be denoted as \( T_m \), and the speed for the executions of the tasks in \( T_m \) be denoted as \( s_m \). Let \( E_m \) denote the load of the tasks in \( T_m \), where the load of a task set is the sum of the worst-case execution times of all of the tasks in the task set, i.e., \( E_m = \sum_{\tau_i \in T_m} \epsilon_i \). The earliest estimated scheduling time instant \( t_m^\ast \) at time instant \( t^* \) on processor \( p_m \) is the time instant at which the currently executing task at \( t^* \) will complete its execution in the worst case on processor \( p_m \) in the current schedule. In other words, \( t_m^\ast \) is the earliest time instant for scheduling based on the current schedule on processor \( p_m \). In order to reduce the energy consumption, our scheme should derive a feasible schedule by revising the task set \( T_m \) and the earliest estimated scheduling time instant \( t_m^\ast \) in a dynamic manner. Note that the current schedule is \( S_{static} \) initially. \( T_m \) is the collection of tasks assigned to execute on processor \( p_m \) initially. \( t_m \) is the worst-case completion time of the task starting its execution on processor \( p_m \) at time 0. \( s_m = \frac{E_m}{t_m} \). Our scheme will revise the schedule whenever a task completes its execution because slack time might be observed.

For the rest of this section, we shall show how to revise the current schedule at \( t^* \) into another feasible schedule. Our proposed slack reclamation scheme at time instant \( t^* \) consists of the task remapping phase and the slack reclaiming phase to reclaim the slack time created at time instant \( t^* \) for saving energy. In the task remapping phase, we reassign the tasks in task set \( \bigcup_m T_m \) onto processors so that the energy consumption could be reduced. In the slack reclaiming phase, we reclaim the slack time to slow down the execution speeds of some tasks to reduce the energy consumption.

3.2 Task Remapping Phase

The purpose of this section is to present dynamic remapping to map tasks in task set \( \bigcup_m T_m \) onto different processors for executions to reduce the energy consumption. Two types of dynamic remapping are proposed: the dynamic task-level remapping and the dynamic processor-level remapping. Dynamic task-level remapping considers the moving of each task on one processor \( p_m \) to another processor \( p_m' \) individually. Dynamic processor-level remapping considers the moving of all of the tasks on one processor \( p_m \) to another processor \( p_m' \) at the same time. The schedule under considerations for task remapping is referred to as the pre-remapping schedule. The schedule resulting from task remapping is referred to as the post-remapping schedule.

For each task that completes its execution at \( t^* \) on processor \( p_m \) (1 \( \leq m \leq M \)), the task remapping phase starts and the earliest estimated scheduling time instant \( t_m^\ast \) is updated as \( t^* \). Scheme 1 is the algorithm for the dynamic remapping scheme, where Step 3 in Scheme 1 invokes Algorithms 3 and 2 for dynamic processor-level and task-level remapping, respectively. Steps 1 and 2 are for initialization of variables, such as \( t_m^\ast \). Scheme 1 remaps tasks in \( \bigcup_m T_m \) into \( M \) disjoint sets by invoking Algorithm 2 or 3 at Step 3 (Please see Sections 3.2.1 and 3.2.2). Steps 4-10 are for the adjustment of the task execution speeds to satisfy the deadline \( D \) and to minimize the energy consumption. Let \( T_m^\ast \) be the collection of tasks assigned on processor \( p_m \) (1 \( \leq m \leq M \)) after task remapping is done, and \( E_m^\ast = \sum_{\tau_i \in T_m^\ast} \epsilon_i \) is the load of task set \( T_m^\ast \). The total worst-case execution time \( t_m^\ast \) of the tasks in \( T_m^\ast \) in the pre-remapping schedule is defined as \( \sum_{\tau_i \in T_m^\ast} s_i, \) where \( s_i \) is the processor speed of the task set \( T_k \) containing task \( \tau_i \) in the pre-remapping schedule. For each processor \( p_m \), if \( t_m^\ast \) is less than \( D \) minus \( t_m \), we update \( t_m^\ast \) as \( D - t_m \); otherwise, \( t_m^\ast \) is updated as \( D - t_m \). \( s_m \) is then updated as \( \frac{E_m}{t_m^\ast} \).
In dynamic task-level remapping, task sets are remapped into another processor for each individual task. In other words, the task sets and have a new processor for dynamic task-level remapping. Algorithm 1:

1. sort task in $T^* = \bigcup_m T_m$ in a non-increasing order of their worst-case execution time at $S_{\text{max}}$.
2. $E_{m}^i \leftarrow 0$ for $m = 1, \ldots, M$;
3. while $T^*$ is not empty do
   4. remove task $\tau_i$ with the greatest $e_i$ from $T^*$ ;
   5. find $p_n$ with the minimum $(D - t_n^*)P\left(\frac{E_n^i}{D - t_n^*}\right)$;
   6. $T_n^i \leftarrow \{\tau_i\} \cup T_n^i, E_n^i \leftarrow e_i + E_n^i, \text{ and } t_n \leftarrow \frac{e_n}{s_m} + t_n$,
   7. if $\frac{E_n^i}{D - t_n^*} > 1$ for some $n$ then
      8. call Algorithm GLF (see Section 3.2.2)

$T_m$ is updated as $T_m^\dag$, and $E_m$ is set as $E_m^\dag$. After the updating of all processor speeds, the post-remapping schedule is derived.

### 3.2.1 Dynamic Task-Level Remapping

In dynamic task-level remapping, task sets $T_m$ are moved among processors to reduce the energy consumption. In this section, we consider to break each task set $T_m$, and move each individual task to a different processor. In other words, the $M$ task sets in the pre-remapping schedule would be remapped into another $M$ task sets and have a new processor in the post-remapping schedule.

Algorithm 2, denoted as Algorithm GTGF, is proposed for dynamic task-level remapping. Algorithm GTGF assigns the task with the maximum worst-case execution time among the remaining tasks to the processor with the minimum increased energy consumption if the task is assigned to execute on the processor. The rationale behind the strategy is similar to that in the packing of boxes into a trunk. Tasks in $T^* = \bigcup_m T_m$ are first sorted and re-indexes in a non-increasing order of their worst-case execution times. In each iteration of the loop from Step 3 to Step 6, task $\tau_i$ with the maximum worst-case execution time in $T^*$ is assigned to the processor $p_n$ with the minimum increased energy consumption. In other words, $p_n$ is the processor with the minimum value $(D - t_n^*)P\left(\frac{E_n^i}{D - t_n^*}\right)$ for $1 \leq n \leq M$. The worst-case execution time $\frac{E_m^i}{s_m}$ of task $\tau_i$ executing at speed $s_m$ is included in the calculation of $t_n$ for processor $p_n$, where $\tau_i$ is in $T_m$ in the pre-remapping schedule, and $s_m$ is defined based on $T_m$. After the loop ends, Step 7 check the feasibility of the resulting remapping, given the maximum processor speed. If it fails, then dynamic processor-level remapping, i.e., Algorithm GLF (which will be introduced in Section 3.2.2), is used (Step 8), instead. The time complexity of Algorithm GTGF is $O(|T^*| |M|)$, provided that tasks in $T$ is sorted in a priori (Note that sorting in the implementation could be skipped since tasks are already sorted in $S_{\text{static}}$).

### 3.2.2 Dynamic Processor-Level Remapping

In dynamic processor-level remapping, the idea is to move the task set $T_m$ with the largest load to the processor with the smallest earliest estimated scheduling time instant. Algorithm GLF first sorts and re-indexes task sets $T_1, T_2, \ldots, T_M$ in a non-increasing order of their loads (Step 1). For each iteration in the loop from Step 3 to Step 5 in Algorithm 3, we assign the task set $T_m$ with the largest load in the remaining task sets to the processor $p_n$ with the smallest earliest estimated scheduling time instant $t_n^*$ among the remaining unassigned processors. The value $t_n$ on processor $p_n$ is then updated as $\frac{E_m^i}{s_m}$. $T_m^\dag$ and $E_m^\dag$ are set as $T_m$ and $E_m$, respectively, before the next iteration. The time complexity of Algorithm GLF is $O(M \log M)$, which is dominated by the sorting of task sets and the procedure in the finding of processor $p_n$ with the minimum $t_n^*$. The feasibility and optimality of Algorithm GLF could be shown by the following theorem:

**Theorem 1** The post-remapping schedule derived by Scheme 1 and Algorithm GLF is feasible if the pre-remapping schedule is feasible.
is reached. 

Algorithm GLF would minimize the energy consumption by executing all of the tasks in \( D \) deadline, if the pre-remapping schedule is not feasible because the tasks executed on processor \( p_k \) could not complete no later than deadline \( D \) without violating speed \( S_{\max} \). A contradiction is reached. \[ \]

**Theorem 2** Given a pre-remapping schedule, the post-remapping schedule derived from Scheme 1 with Algorithm GLF would minimize the energy consumption by executing all of the tasks in \( T_m^* \) at speed \( \frac{E_m^*}{D-D^*} \) in the post-remapping schedule, when the actual execution time of each of the tasks that have not completed before \( t^* \) is equal to the worst-case execution time.

**Proof.** This theorem could be proved by contradiction. Suppose that the post-remapping schedule is not feasible on some specific processor \( p_m \), in which \( \frac{E_{m}^{*}}{D-D^{*}} > 1 \). Because of the task-set remapping policy (i.e., the loop of Algorithm GLF), there must exist another processor \( p_k \) with \( E_k \geq E_m \) and \( t_k^* \geq t_m^* \) in which \( E_k, t_k^* \), and \( t_m^* \) are defined in the pre-remapping schedule. Since \( \frac{E_k}{D-D^*} \geq \frac{E_m}{D-D^*} \), the pre-remapping schedule is not feasible because the tasks executed on processor \( p_k \) could not complete no later than deadline \( D \) without violating speed \( S_{\max} \). A contradiction is reached. \[ \]

3.3 Slack Reclaiming Phase

The slack reclaiming phase is to effectively use the slack time to reduce the speed of each processor. Suppose that the task remapping phase occurs at time point \( t \), i.e., \( t = t^* \). Let \( p_m \) be a processor with a task completing at time \( t^* \). (Note that there could be more than one processor having a task completing at time \( t^* \).) After the task remapping phase, the slack reclaiming phase starts with the post-partition schedule. Let task \( \tau_k \) be the task with the largest worst-case execution time in task set \( T_m \). The slack reclamation phase would focus on how to execute task \( \tau_k \) in the reducing of the energy consumption. Four algorithms are presented in later subsections for the scheduling of \( \tau_k \). After the execution speed of \( \tau_k \) is determined, \( \tau_k \) starts its execution and is removed from task set \( T_m \). The load of \( T_m \) is updated, i.e., \( E_m = E_m - e_k \). The slack reclaiming phase ends.

3.3.1 Fair Slack Reclamation

Algorithm fair slack reclamation (FR) lets all tasks in \( T_m \) share the slack time fairly. As a result, task \( \tau_k \) executes at speed \( \frac{E_m}{D-D^*} \). Note that Algorithm FR is pessimistic in the expectation of further slack time in the future.

3.3.2 Greedy Slack Reclamation

Algorithm greedy slack reclamation (GR) only benefits task \( \tau_k \) with the slack time. As a result, task \( \tau_k \) executes at speed \( \frac{E_m}{D-D^*} \) in time interval \( [t^*, t^* + e_k/s_m] \). \( t^* \) is set
as $t^*_m + e_k/s_m$. Algorithm GR is more optimistic in the observation of slack time in the near future so that the current slack time is used greedily. The time complexity of Algorithms FR and GR is $O(1)$.

### 3.3.3 Hybrid Slack Reclamation

Algorithm hybrid slack reclamation (HR) is motivated by the difficulty in the prediction of the future. Algorithm HR is a combination of Algorithm FR and Algorithm GR, in which the current slack time is fairly shared by a specified number $K$ of tasks in $T_m$. Since tasks on a processor execute in a non-increasing order of their worst-case execution times, let the slack time be fairly shared by the collection $T^K_m$ of tasks in $T_m$ with the $K$ largest worst-case execution times (tie-breaking in the selection of tasks with equal worst-case execution times could be done arbitrarily). $E^K_m$ denotes the load of $T^K_m$. The latest completion time of all of the tasks in $T^K_m$ is $(D - E^K_m)/s_m$ to ensure that all of the tasks in $(T_m \setminus T^K_m)$ could complete no later than deadline $D$ in the worst cases. The slack time $(t^*_m - t^*)$ is shared fairly by tasks in $T^K_m$. Thus task $T_k$ should execute at speed $(D - E^K_m)/s_m$.

When $K$ is set as 1, Algorithm HR becomes Algorithm GR. If $K$ is sufficiently large, Algorithm HR becomes Algorithm FR. The time complexity of Algorithm HR is $O(K)$. However, it is possible to reduce the time complexity to $O(1)$ if $E^K_m$ could be constructed incrementally in $O(1)$ in dynamic processor-level remapping. For the rest of this paper, Algorithm HR with a parameter setting $K$ is abbreviated as Algorithm HR-K.

### 3.3.4 Aggressive Slack Reclamation

Based on the dynamic uniprocessor energy-efficient algorithm presented in [4], we propose an extended algorithm to use the slack time aggressively with reference to the execution history. Algorithm aggressive slack reclamation (AR) assumes that the ratio of the actual execution time to the worst-case execution time of every task is about the same. Let $t^*$ be the starting time of the slack reclaiming phase, and $\lambda$ be the average ratio of the actual execution time to the worst-case execution time of every task that has finished its execution no later than $t^*$. For the national brevity, let $s^{\max}$ denote the maximum value of speed $s_m$, and the largest processor speed in schedule $S_{\text{static}}$. The rationale behind Algorithm AR is to let task $T_k$ use the current slack time and execute at a speed derived based on the expected execution times of tasks in $T_m$, provided that all tasks in $(T_m \setminus \{T_k\})$ could finish their executions at $s^{\max}$ before $D$ even if the execution time of $T_k$ is equal to the worst case. If all of the tasks in task set $(T_m \setminus \{T_k\})$ could complete their executions at speed $s^{\max}$ no later than $D$ when they start executing after time $(t^* + e_k \cdot (D - t^*)/(E_m \cdot \lambda))$, then task $T_k$ is set to execute at speed $(E_m \cdot \lambda/(D - t^*))$, and $t^*_m$ is set as $(t^* + e_k \cdot (D - t^*)/(E_m \cdot \lambda))$. Otherwise, task $T_k$ is set to execute at speed $(e_k/(D - E_m - e_k - t^*))$, and $t^*_m$ is set as $(D - (E_m - e_k)/s^{\max})$ by assuming that all of the tasks in $(T_m \setminus \{T_k\})$ will execute at speed $s^{\max}$, which is the speed for the executions of tasks in $(T_m \setminus \{T_k\})$, is changed to $(E_m - e_k)/((D - t^*_m))$ if $t^*_m$ satisfies the inequality $(E_m - e_k)/(D - t^*_m) > s_m$. (Note that $t^*_m$ is updated as above.) Otherwise, $s_m$ remains the same as that in the post-remapping schedule. The time complexity is $O(1)$.

### 3.4 Remarks

If we reclaim the slack time as that described in Algorithm FR, GR, HR-K, and AR in the slack reclaiming phase, the energy consumption of the post-remapping schedule could be reduced without violating of the common deadline $D$. The four slack reclaiming algorithms could be better illustrated by an example, as shown in Figure 2. The worst-case execution times of $T_1, T_2, T_3$, and $T_4$ are 8, 2, 1, and 1, respectively, where the actual execution times of $T_1, T_2, T_3$, and $T_4$ are 7, 1, 1, and 1, respectively. $D = 12$ and $P(s) = s^j$. The energy consumptions of the schedules derived from Algorithms FR, GR, HR-2, and AR are 8.208, 8.64, 8.1575, and 8.64123, respectively.

There are eight combinations of remapping algorithms and slack reclaiming algorithms in the proposed scheme. The eight algorithms could be denoted as Algorithms GLF+FR, GLF+GR, GLF+HR-K, GLF+AR, GTEF+FR, GTEF+GR, GTEF+HR-K, and GTEF+AR, where the term before ‘+’ denotes the adopted task remapping algorithm, and that after ‘+’ denotes the adopted slack reclaiming algorithm. When Algorithm GLF (Algorithm GTEF) is adopted, the time complexity is $O(M \log M)/O(|\bigcup M|)$ for each run of task remapping, regardless of which slack reclaiming algorithm is used.

### 4 Performance Evaluation

#### 4.1 Simulation Setup

In this section, we provide performance evaluation on the energy consumption of our proposed algorithms, compared to Algorithms GSSR and FGSB presented in [25].

For each task $T_i$ in $T$, the worst-case execution time $e_i$ of $T_i$ at $S_{\text{max}}$ was uniformly distributed in the range of $[1, 50]$. In a task set, the actual execution times of 50% tasks were equal to their worst-case execution times, and those of the other 50% tasks were random variables. The ratio $\beta$ of the actual execution time of $T_i$ to the worst-case execution time of $T_i$ was determined by either a normal distribution or
a uniform distribution. For each workload generated by a normal distribution, we specified a value $\alpha$ so that $\beta_i$ was a normal distribution with a mean value $\alpha$ and a standard deviation 0.1. For each workload generated by a uniform distribution, we specified a value $\alpha$ so that $\beta_i$ was a uniform distribution between $[\alpha, 1]$. The actual execution time of task $\tau_i$ was $\beta_i \cdot e_i$. The simulation results were derived based on 100 independent simulations by varying the value of $\alpha$ from 0.1 to 0.9, stepped by 0.1, with 2, 4, 8, 16, 32 processors and 500 tasks in the simulated task set. The power consumption function used in the simulations was $P(s) = s^{3}$. In the simulations, the common deadline $D$ was set as the maximum $E_m$ in $S_{\text{static}}$ for the feasibility guarantee of $S_{\text{static}}$.

The normalized energy consumption was adopted as the performance metric in our simulations. The normalized energy consumption for an input instance was defined as the ratio of the energy consumption of the on-line schedule derived by the algorithm to that of $S_{\text{static}}$. The less the normalized energy consumption is, the better the algorithm performs for the input instance.

### 4.2 Simulation Results

Due to space limitation, only representative results are shown. Figure 3.(a) and Figure 3.(b) show the simulation results by applying different task remapping algorithms for task sets with normal distribution of $\beta_i$ when $M = 4$ and $M = 32$, respectively, where Algorithm $\text{STATIC+HR-2}$ denotes the algorithm that does not perform any task remapping and takes Algorithm $\text{HR-2}$ in the slack reclaiming phase. As shown in Figure 3, Algorithm $\text{GLF+HR-2}$ always outperformed Algorithm $\text{STATIC+HR-2}$, especially for systems with more processors. In our simulation results, Algorithm $\text{GTEF}$ only outperformed Algorithm $\text{GLF}$ in few cases, such as when $\alpha \geq 0.4$ and $M = 4$. This comes from that Algorithm $\text{GTEF}$ is too conservative in slack time reclaiming due to the fair distribution of the slack time among the unexecuted tasks, especially when the number of available processors increases. Algorithm $\text{GLF}$ is a better choice in the most simulated cases for task remapping if we take both the performance and the timing overheads into considerations.

In order to provide insights for the settings of the user-specified parameter $K$ in Algorithm $\text{HR-K}$, Figure 3.(c) shows the performance of Algorithm $\text{HR-K}$ with different settings on the value of $K$ ($2, 3, 4,$ and $9$) on a four-processor system when the Algorithm $\text{GLF}$ was adopted in the task remapping phase. Both of the characteristics of the task sets and the number of processors in the system would affect the suitability of the value of $K$. In general, setting $K$ as 2 or 3 in Algorithm $\text{HR-K}$ was good enough.

Figure 4 shows the simulation results by applying different slack reclaiming algorithms, compared to Algorithm $\text{GSSR}$ in [25], for different workload distribution and different numbers of processors. In Figure 4.(b), when $\alpha$ was equal to 0.1, the average normalized energy consumptions of Algorithms $\text{GSSR}$ and $\text{GLF+AR}$ were 52.2% and 36.7%, respectively. The improvement of our proposed algorithm on the minimization of energy consumption was about 29%. As shown in our simulation results, since Algorithms $\text{GLF+AR}$ and $\text{GLF+HR-2}$ perform either better or no much worse than the other simulated algorithms for general cases, they could be good choices when the characteristics of the task set are unknown in a priori.

### 5 Conclusion

This paper considers energy-efficient task scheduling for frame-based real-time tasks for homogeneous multiprocessors, where all of the tasks arrive at the same time and share a common deadline. A slack reclamation scheme is proposed to reduce the energy consumption during the run time, where the scheme is applied whenever a task completes its execution. In the task remapping phase, the current schedule is revised such that tasks that have not started their executions yet might be moved to different processors for executions. In the slack reclaiming phase, the slack time is then used to slow down the execution speeds for some tasks. Two dynamic task remapping algorithms and four slack reclaiming algorithms are proposed under the slack reclamation scheme. It was shown that the proposed scheme could reduce the energy consumption of a task set up to 29% in the experiments, compared to the previous work [25]. By adopting the approaches presented in [9, 12, 13, 15], our proposed scheme can be applied for discrete available speeds.

For future research, we shall consider energy-efficient task scheduling with precedence constraints, especially for multiprocessor environments. We shall also consider the possibility of having different power consumption functions for different tasks.

### References


