Global Reliability-Aware Power Management for Multiprocessor Real-Time Systems*

Xuan Qi, Dakai Zhu
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX, 78249
{xqi,dzhu}@cs.utsa.edu

Hakan Aydin
Department of Computer Science
George Mason University
Fairfax, VA 22030
aydin@cs.gmu.edu

Abstract

Recently, the negative effect of the popular power management technique Dynamic Voltage and Frequency Scaling (DVFS) *on the system reliability has been identified. As a result, various* reliability-aware power management (RAPM) *schemes have been studied for uniprocessor real-time systems. In this paper, we investigate* global-scheduling-based RAPM (G-RAPM) *schemes for a set of frame-based real-time tasks running on a homogeneous multiprocessor system. An important dimension of the problem is how to select the appropriate subset of tasks for energy* and *reliability management (i.e., schedule a recovery for each selected task and scale down their executions). We show that making this decision optimally (i.e., the static G-RAPM problem) is NP-hard. Then we propose two efficient G-RAPM heuristics, which rely on* local *and* global *task selections, respectively. Moreover, to reclaim dynamic slack generated at runtime, we extend the* slack sharing *based global dynamic power management scheme to the reliability-aware settings. The proposed schemes are evaluated through extensive simulations. The results show that our static G-RAPM heuristics can preserve system reliability while achieving significant energy savings (within 3% of an upper bound for most cases). Moreover, G-RAPM with global task selection provides better opportunities for dynamic slack reclamation and up to* 15% *more energy savings can be obtained at runtime compared to that of local task selection.*

1 Introduction

Energy management has become an important research area in the last decade, in part due to the proliferation of embedded computing devices and remains as one of the grand challenges for the research and engineering community, both in industry and academia [12]. One common strategy to save energy in computing systems is to operate system components at low-performance (and thus low-power) states, whenever possible. As one of the most effective and widely-deployed power management techniques, *dynamic voltage and frequency scaling (DVFS)* exploits the convex relation between processor dynamic power consumption and processing frequency/supply voltage [3] and scales down simultaneously the processor processing frequency and supply voltage to save energy [22].

For real-time systems where tasks have stringent timing constraints, scaling down system processing frequency (speed) may cause deadline misses and special provisions are needed. In the recent past, many research studies explored the problem of minimizing energy consumption while meeting the deadlines for various real-time task models by exploiting the available static and/or dynamic *slack* in the system [2, 16, 19, 29]. However, recent studies show that DVFS has a direct and adverse effect on the transient fault rates (especially for those induced by electromagnetic interference and cosmic ray radiations) [6, 9, 30]. Therefore, for safety-critical real-time embedded systems (such as nuclear plants and avionics control systems) where reliability is as important as energy efficiency, *reliability-cognizant* energy management becomes a necessity.

One cost-effective approach to tolerate transient faults is the *backward error recovery* technique, in which the system state is restored to a previous *safe state* and the computation is re-performed [18]. By adopting such a recovery approach while considering the negative effects of DVFS on transient faults, we have introduced a *reliability-aware power management (RAPM)* scheme [25]. The central idea of the RAPM scheme is to exploit the available slack to schedule a recovery

*This work was supported in part by NSF awards CNS-0720651, CNS-0855247, CNS-0720647 and CAREER Award CNS-0546244.

task at a task's dispatch time before utilizing the remaining slack for DVFS to scale down the execution of the task and save energy, thereby preserving the system reliability [25]. Following this line of research, several RAPM schemes have been proposed to consider various task models, scheduling policies and reliability requirements [5, 20, 24, 26, 27, 28, 31, 32], all of which have focused on uniprocessor systems. For dependent tasks represented by directed acyclic graphs (DAGs) to be executed on multiprocessor systems, Pop *et al.* developed a constraint-logic-programming (CLP) based solution to minimize energy consumption, which transforms the user-defined reliability goals to tolerate a fixed number of transient faults through re-execution [17].

There are two paradigms in multiprocessor real-time scheduling: the *partitioned* and *global* approaches [7, 8]. With the emergence of multicore processors where processing cores on a chip can share the last level cache, it is expected that the migration cost for global scheduling will be significantly reduced. Different from all the existing work, in this paper, for a set of frame-based real-time tasks that share a common deadline, we study *global scheduling based RAPM (G-RAPM)* schemes to minimize energy consumption while preserving system reliability in multiprocessor real-time systems.

After showing that the static G-RAPM problem is NP-hard, we propose two static G-RAPM heuristic schemes. Depending on how to exploit the system slack and when to select the appropriate subset of tasks for energy and reliability management, the proposed heuristics are characterized by *global* and *local* task selections, respectively. Here, a recovery block will be scheduled for each selected task and the executions of selected tasks will be scaled down accordingly. The remaining tasks that are not selected run at the maximum frequency to preserve reliability. Due to the uneven time allocation for tasks in the G-RAPM schemes, the execution orders (i.e., priorities) of tasks are determined through a *reverse dispatching process* in the global queue to ensure that all tasks can finish their executions in time. Moreover, to reclaim the dynamic slack generated from early completion of tasks' executions and/or unused recovery blocks for more energy savings, we extend our previous work on *slack sharing* in global-scheduling-based dynamic power management to the reliability-aware settings.

The simulation results show that, the proposed G-RAPM schemes can preserve system reliability while achieving significant energy savings (within 3% of the upper bound for most cases) in multiprocessor real-time systems. By giving managed tasks higher priorities, G-RAPM with global task selection provides better opportunities for online dynamic slack reclamation and can obtain about 15% more energy savings compared to that of G-RAPM with local task selection.

The remainder of this paper is organized as follows. Section 2 presents system models considered in this work. Section 3 reviews the key idea of RAPM and formulates the global scheduling based RAPM (G-RAPM) problem for multiprocessor real-time systems, which is shown to be NP-hard. In Section 4, two static heuristic schemes are proposed. The dynamic G-RAPM schemes are presented in Section 5 and Section 6 discusses the simulation results. We conclude the paper in Section 7.

2 System Models

2.1 Power Model

Considering the linear relation between processing frequency and supply voltage [3], the dynamic voltage and frequency scaling (DVFS) technique decreases supply voltage for lower frequency requirements to reduce the system's dynamic power consumption [21]. To avoid ambiguity, for the remainder of the paper, we will use the term *frequency change* to stand for both supply voltage and frequency adjustments. With the ever-increasing static leakage power due to scaled feature size and increased levels of integration, as well as other power consuming components (such as memory), power management schemes that focus on individual components may not be energy efficient at the system level. Therefore, system-wide power management becomes a necessity [1, 13, 15, 19].

In our previous work, we proposed and used a *system-level power model* for uniprocessor systems [1, 30] and similar power models have also been adopted in other studies [13, 15, 19]. In this paper, by following the same principles, the power consumption for a system with k identical processors is expressed as:

$$\begin{aligned} P(f) &= P_s + \sum_{i=1}^{k}(P_{ind} + P_{d,i}) \\ &= P_s + \sum_{i=i}^{k}(P_{ind} + C_{ef} \cdot f_i^m) \end{aligned} \qquad (1)$$

Here, P_s is the *static power* used to maintain the basic circuits of the system (e.g., keeping the clock running), which can be removed only by powering off the whole system. For each processor, there are two parts for its *active power*: the *frequency-independent active power* (P_{ind}, which is a constant and assumed to be the same for all processors) and *frequency-dependent active power* (P_d, which depends on the supply voltage and processing frequency of each processor). When there is no workload on a given processor, we assume that the corresponding P_{ind} value can be effectively removed by putting the processor into power saving sleep states [4]. The effective switching capacitance C_{ef} and the dynamic power exponent m (which is, in general, no smaller than 2) are system-dependent constants [3]. f_i is the processing frequency for the i'th processor. Despite its simplicity, this power model includes all essential power components of a system and can support various power management techniques (e.g., DVFS and power saving sleep states).

Due to the energy consumption related to P_{ind}, it may not be energy-efficient to execute tasks at the lowest available frequency that guarantees timing constraints and an energy-efficient frequency f_{ee}, below which the system consumes more *total energy*, does exist [1, 15, 19, 30]. Considering the prohibitive overhead of turning on/off a system (e.g., tens of seconds), we assume that the system will be on and P_s is always consumed. By putting processors to sleep states for saving energy when idle, we can get the energy-efficient frequency for each processor as $f_{ee} = \sqrt[m]{\frac{P_{ind}}{C_{ef} \cdot (m-1)}}$ [1, 30].

Consequently, for energy efficiency, the processing frequency for any task should be within the range of $[f_{ee}, f_{max}]$. Moreover, we use normalized frequencies in this work and it is assumed that $f_{max} = 1$. For simplicity, the time overhead for adjusting frequency (and supply voltage) is assumed to be negligible[1].

2.2 Fault Model

There are various reasons that could lead to run-time faults, such as hardware failures, electromagnetic interferences, and the effects of cosmic ray radiation. In this paper, we will focus on transient faults, which have been shown to be dominant [14]. The inter-arrival rate of transient faults is assumed to follow Poisson distribution [23]. Moreover, for DVFS-enabled computing systems, considering the negative effect of DVFS on transient faults, the average transient fault rate $\lambda(f)$ at a scaled frequency $f (\leq f_{max})$ (and corresponding supply voltage V) can be given as [30]:

$$\lambda(f) = \lambda_0 \cdot g(f) \tag{2}$$

where λ_0 is the average fault rate at f_{max} (and V_{max}). That is, $g(f_{max}) = 1$. The fault rate will increase at lower frequencies and supply voltages. Therefore, we have $g(f) > 1$ for $f < f_{max}$.

[1]Such overhead can be easily incorporated into the execution time of the applications under consideration when scaling down the processing frequency [2, 29].

More specifically, in this work, we consider an exponential fault rate model, where $g(f)$ is given by [26]:

$$g(f) = 10^{\frac{d \cdot (1-f)}{1-f_{ee}}} \tag{3}$$

Here d (> 0) is a constant, representing the sensitivity of fault rates to DVFS.

Transient faults are assumed to be detected by using *sanity* (or *consistency*) checks at the completion of a task's execution [18]. Once a fault is detected, *backward recovery* technique is employed and a recovery task (in the form of re-execution) is dispatched for fault tolerance [23, 25]. Again, for simplicity, the overhead for fault detection is assumed to be incorporated into the worst-case execution time of tasks.

3 RAPM and Problem Formulation

3.1 RAPM with Recovery Tasks

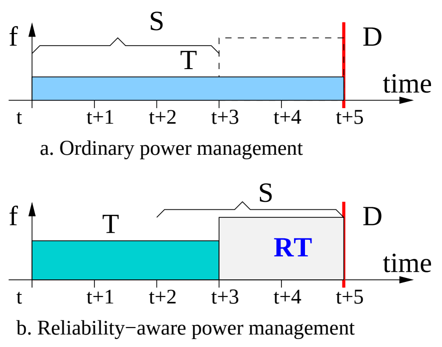

f
S
T
D
time
t
t+1
t+2
t+3
t+4
t+5
a. Ordinary power management
f
T
S
RT
D
time
t
t+1
t+2
t+3
t+4
t+5
b. Reliability−aware power management

Figure 1. Ordinary and Reliability-Aware Power Management [25].

Before formally presenting the problem to be addressed in this paper, we first review the fundamental ideas of RAPM schemes through an example. Suppose that a task T is dispatched at time t with the WCET of 2 time units. If task T needs to finish its execution by its deadline $(t + 5)$, there will be 3 units of available slack. As shown in Figure 1a, without paying special attention to the negative effects of DVFS on task reliability, the *ordinary* (and *reliability-ignorant*) power management scheme will exploit *all* the available slack to scale down the execution of task T for the maximum energy savings. However, such ordinary power management scheme can lead to the degradation of task's reliability by several orders of magnitude [25].

Instead of using *all* the available slack for DVFS to save energy, as shown in Figure 1b, the RAPM scheme reserves a portion of the slack to schedule a *recovery* *task RT* for task T to recuperate the reliability loss due to energy management before scaling down its execution using the remaining slack [25]. The recovery task RT will be dispatched (at the maximum frequency f_{max}) only if a transient fault is detected when task T completes. With the help of RT, the overall *reliability* of task T will be the summation of the probability of T being executed correctly and the probability of having transient fault(s) during T's execution while RT being executed correctly, which has been shown to be no worse than task T's *original* reliability when no power management is applied [25]. That is, regardless of different fault rate increases at scaled processing frequencies, by scheduling an additional recovery task, the RAPM scheme can *guarantee* to preserve the original reliability of a real-time task while still obtaining energy savings using the remaining slack (if any) [25].

3.2 Problem Formulation

In this work, we consider a set of n independent real-time tasks to be executed on a multiprocessor system with k identical processors. The tasks share a common deadline D, which is also the period (or frame) of the task set. The worst-case execution time (WCET) for task T_i at the maximum frequency f_{max} is denoted as c_i $(1 \leq i \leq n)$. When task T_i is executed at a lower frequency f_i, it is assumed that its execution time will scale linearly and task T_i will need time $t = \frac{c_i}{f_i}$ to complete its execution in the worst case. To address the negative effects of DVFS on transient faults and preserve system reliability, a recovery task will be scheduled for each task whose execution will be scaled down. Moreover, it is assumed that any faulty scaled task will be recovered *sequentially* on the same processor and that a given task cannot run in parallel on multiple processors.

Since the system is assumed to be on all the time and the static power P_s is always consumed, we focus on managing the energy consumption related to system active power. At the scaled frequency f_i, the active energy consumption to execute task T_i is given as:

$$E_i(f_i) = (P_{ind} + C_{ef} f_i^m) \cdot \frac{c_i}{f_i} \tag{4}$$

Note that, not all tasks will be selected for power management due to workload constraints and/or energy efficiency. We use $h_i = 1$ to indicate that task T_i is selected for management; otherwise, if T_i is not selected, $h_i = 0$. As discussed before, for tasks that are not selected, they will run at the maximum processing frequency f_{max} to preserve their reliability.

Considering the fact that the probability of having faults during a task's execution is rather small, we focus on the energy consumption for executing all primary tasks and try to minimize the *fault-free* energy consumption. More specifically, the *global scheduling based RAPM (G-RAPM)* problem to be addressed in this paper is to: **find the priority assignment (i.e., execution order of tasks), task selection (i.e., h_i) and the scaled frequencies of tasks (i.e., f_i) to ensure the schedulability of the tasks and at the same time to**:

$$\text{minimize} \sum_{i=1}^{n} E_i(f_i) \tag{5}$$

subject to

$$f_{ee} \leq f_i < f_{max}, \text{if } h_i = 1 \tag{6}$$

$$f_i = f_{max}, \text{if } h_i = 0 \tag{7}$$

Here, Equation (6) restricts the scaled frequency of any task to the range of $[f_{ee}, f_{max})$. Equation (7) states that the *un-selected* tasks will run at f_{max}.

Note that, when there is only one processor (i.e., when $k = 1$), the problem will be reduced to that of optimal RAPM problem for uniprocessor systems, which has been studied in our previous work and shown to be NP-hard [26]. Therefore, for the static G-RAPM problem, finding the optimal solution to minimize the fault-free energy consumption will be NP-hard as well. In what follows, we focus on two static heuristic solutions, which will be evaluated against the theoretically ideal upper bound on energy savings in Section 6.

4 Static Global Scheduling Based RAPM

There are a few inter-related key issues in solving the static G-RAPM problem, such as priority assignment, slack determination, and task selection. Depending on how the available slack is determined and utilized, we study in this section two heuristic schemes that are based on *local* and *global* task selection, respectively.

4.1 G-RAPM: Local Task Selection

It has been shown that the optimal priority assignment to minimize the scheduling length of a set of real-time tasks on multiprocessor systems under global scheduling is NP-hard [7]. Moreover, our previous study revealed that such priority assignment, even if it is found, may not lead to the maximum energy savings due to the runtime behaviours of tasks [29]. Therefore, to get the static mapping of tasks to processors and determine the amount of available slack on each processor, we adopt the *longest-task-first (LTF)* heuristic for the *initial* priority assignment. If the task set is schedulable under the worst-fit and LTF heuristics, the amount of available slack on each processor can be determined. Then, the existing RAPM solutions for uniprocessor systems [26] can be applied for the tasks that are statically mapped on each processor individually.

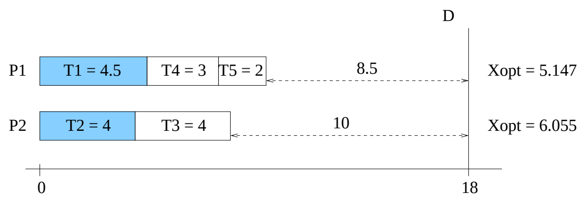


Figure 2. Canonical task-to-processor mapping and local task selection.

For example, consider a task set with five tasks $T_1(4.5)$, $T_2(4)$, $T_3(4)$, $T_4(3)$ and $T_5(2)$, which will be executed on a 2-processor system with the common deadline of 18. The numbers in the parentheses are the WCETs of tasks. With LTF priority assignment and worst-fit assignment, the *canonical* mapping[2] of tasks to processors (assuming all tasks will use their WCETs) is shown in Figure 2. Here, three tasks (T_1, T_4 and T_5) are mapped on processor P_1 with 8.5 units of slack. The other two tasks (T_2 and T_3) are mapped on the second processor P_2 with 10 units of slack.

In [26], we have shown that, for a single processor system with slack S, to maximize energy savings under RAPM, the optimal aggregate workload for the selected tasks should be:

$$X_{opt} = S \cdot \left(\frac{P_{ind} + C_{ef}}{m \cdot C_{ef}} \right)^{\frac{1}{m-1}} \tag{8}$$

Assume that $P_{ind} = 0.1$, $C_{ef} = 1$ and $m = 3$ [26], we can get $X_{opt,1} = 5.147$ and $X_{opt,2} = 6.055$ for the first and second processor in the above example, respectively. Again, if the largest task is selected first [26, 27], we can see that tasks T_1 and T_2 will be selected for management on the two processors, respectively.

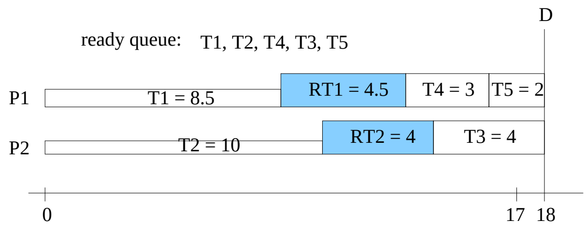


Figure 3. Final static schedule for G-RAPM with local task selection.

After scheduling their recovery tasks RT_1 and RT_2, and scaling down the execution of tasks T_1 and T_2 by utilizing the remaining slack on each processor, the final canonical schedule is shown in Figure 3. Note that, due to the uneven slack allocation among the tasks under the G-RAPM scheme, the execution order (i.e., the order of tasks being dispatched from the global queue) can be different from the one following the initial priority assignment. If we blindly follow the original order of tasks in the global queue, tasks may not be able to finish on time and deadline violation can occur. For instance, after the first four tasks are dispatched and all tasks (including recovery tasks, if any) take their WCETs, Figure 4 shows that there is not enough time on any of the processors and task T_5 will miss the deadline.

[2]The mapping of tasks to processors at runtime may change depending on the actual execution time of tasks.

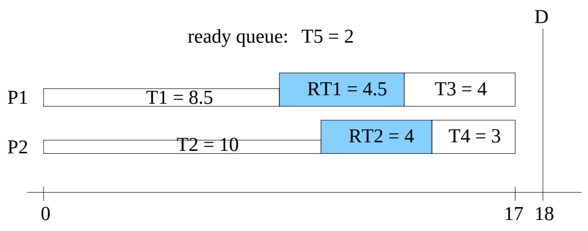


Figure 4. Task T_5 misses the deadline following the original execution order.

Therefore, to overcome such timing anomalies in global scheduling, after obtaining the final canonical schedule from the G-RAPM with local task selection, we should re-assign tasks' priorities (i.e., the order of tasks in the global queue) according to the final schedule. For this purpose, based on the start times of tasks in the final schedule, we can reverse the dispatching process and re-enter the tasks to the global queue from the last task to the first task. For instance, the final order of tasks in the global queue (i.e., their priorities) for the above example can be obtained as shown in Figure 3.

The formal steps of the G-RAPM scheme with local task selection are summarized in Algorithm 1. Here, the first step to get the canonical schedule with any given initial priority assignment of tasks involves ordering tasks based on their priorities (with the complexity of $O(n \cdot log(n))$) and dispatching tasks to processors (with $O(k \cdot n)$ complexity). The second step can be done in $O(n)$ time by selecting tasks on each processor. The last step of getting the final priorities of tasks through reverse dispatching can also be done in $O(n)$ time. Therefore, the overall complexity for Algorithm 1 will be $O(max(n \cdot log(n), k \cdot n))$.

Algorithm 1 G-RAPM with local task selection

```
1: Step 1: Get the canonical schedule from any initial priority assignment (e.g., LTF);
2: if (schedule length > $D$) report failure and exit;
3: Step 2: For each processor $PROC_i$: apply the uniprocessor RAPM scheme in [26] for tasks mapped to that processor (i.e., determine its available slack $S_i$; calculate $X_{opt}$, select tasks and calculate the scaled frequency);
4: Step 3: Get the final execution order (i.e., priority assignment) of tasks in the global queue from the final canonical schedule obtained in Step 2.
```

4.2 G-RAPM: Global Task Selection

Note that, for G-RAPM with local task selection, after obtaining the amount of available slack and the optimal workload desired to be managed for each processor, it is not always possible to find a subset of tasks that have the *exact* optimal workload. Such deviation of the managed workload from the optimal one can accumulate across processors and thus lead to less energy savings. Instead, we can take a global approach when determining the amount of available slack and selecting tasks for management.

For instance, we can see that the overall workload of all tasks in the above example is $W = 17.5$. With the deadline of 18 and two processors, the total available computation time will be $2 \cdot 18 = 36$. Therefore, the total amount of available slack will be $S = 36 - 17.5 = 18.5$. That is, we can view the system by putting the processors side by side sequentially (i.e., the same as having the execution of the tasks on a processor with deadline of 36). In this way, we can calculate that the overall optimal workload of the selected tasks to minimize energy consumption as $X_{opt} = 11.2$. Following the same heuristic as the longest task first, three tasks (T_1, T_2 and T_3, with the aggregated workload of 11.5) are selected to achieve the maximum energy savings.

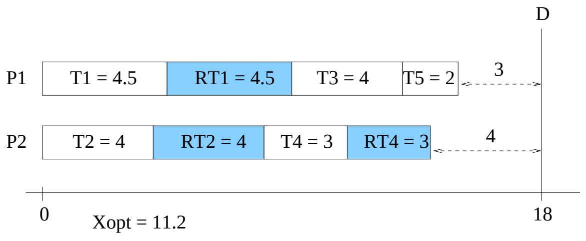


Figure 5. Canonical task-to-processor mapping with global task selection.

However, we may not be able to use the remaining slack to uniformly scale down the execution of the selected tasks. Otherwise, scheduling the task set with managed tasks will require perfect balancing, which may not be feasible. Note also that, the selected tasks should have higher priorities and are mapped in the front of the schedule to provide better opportunity for slack reclamation at runtime. Therefore, before assigning the scaled frequency to the selected tasks, we first map those tasks and their recovery tasks. Then the unselected tasks (again, following the LTF heuristic) are mapped to processors. The resulting canonical mapping for the example is shown in Figure 5.

Here, there are 3 and 4 units of slack that can be utilized to scale down the selected tasks on the processors, respectively. The resulting final schedule is shown in Figure 6. Again, to address the deadline violation problem due to uneven slack allocation, the final priority assignment (i.e., order of tasks in the global queue) should be obtained through the reverse dispatching process as discussed earlier. For the example, with the parameters for the power model given in Section 4.1, it can be calculated that G-RAPM with global task selection can save 32.4% energy when compared to that of no power management, while G-RAPM with local task selection saves only 26.6% (an improvement of 5.8%). Moreover, by scheduling the managed tasks in the front of the schedule, the scheme with global task selection can provide more opportunities to reclaim the dynamic slack from free of the recovery tasks at runtime, which is further evaluated and discussed in Section 6.

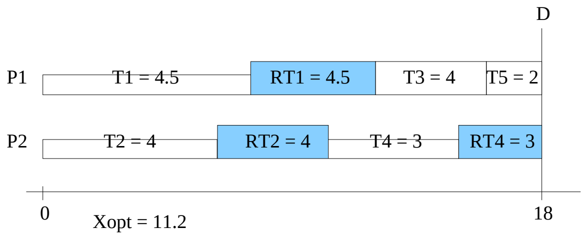


Figure 6. Final canonical schedule with global task selection.

Note that, the global task selection scheme may have a large value for X_{opt}. However, to ensure that any selected task (and its recovery task) can be successfully mapped to a processor, any task T_i with $c_i > \frac{D}{2}$ should not be selected. Taking this point into consideration, the steps for the G-RAPM with global task selection can be summarized in Algorithm 2. Similarly, the complexity of Algorithm 2 can also be found as $O(max(n \cdot log(n), k \cdot n))$.

5 Dynamic G-RAPM Schemes

In general, real-time tasks only take a small portion of their WCETs and dynamic slack can be expected at runtime [10]. Moreover, when the execution of a scaled task completes successfully, the time reserved for its recovery can be freed and becomes dynamic slack as well. Therefore, to exploit such dynamic slack at runtime and obtain better energy savings, we investigate dynamic G-RAPM schemes in this section.

In our previous work, we have studied a global scheduling based power management scheme based on the idea of *slack sharing* for multiprocessor real-time

Algorithm 2 G-RAPM with global task selection

```
1: Step 1:
2: $S = k \cdot D - \sum c_i$; //Calculate global slack
3: Calculate $X_{opt}$;
4: Select tasks (with $c_i < \frac{D}{2}$) for management;
5: Map selected tasks to processors (e.g., by LTF);
6: Map unselected tasks to processors (e.g., by LTF);
7: if (schedule length $> D$) report failure and exit;
8: Step 2:
9: Calculate scaled frequency for selected tasks on
   each processors;
10: Step 3: Get the final execution order (i.e., priority
    assignment) of tasks in the global queue from the
    final canonical schedule obtained in Step 2.
```

systems [29]. The basic idea is to mimic the timing of the canonical schedule at runtime to ensure that all tasks can finish in time. That is, when a task completes early on a processor and generates some dynamic slack, the processor should share part of this slack appropriately with another processor that is supposed to complete its task early in the canonical schedule. After that, the remaining slack can be utilized to scale down the next task's execution to save energy. We have proven that, with proper slack sharing at runtime, all tasks can complete their executions in time under the global scheduling based power management scheme [29].

The slack sharing approach can also be applied on top of the canonical schedules generated from the static G-RAPM schemes (with either local or global task selection). Different from previous work where dynamic slack is only used for scaling down the processing frequency of tasks, the dynamic slack reclamation should be differentiated for scaled tasks that have statically scheduled recovery tasks and the ones that do not have recovery tasks yet.

If the next task to be dispatched is a scaled tasks, the dynamic slack can be utilized to further scale down the processing frequency (as low as f_{ee}) for more energy savings. Otherwise, if the next task has not been scaled down (without any recovery) and the reclaimable dynamic slack is larger than the task's size, a recovery task will be scheduled first and remaining slack is used to scale down the execution of the next task. If the reclaimable dynamic slack is not enough to schedule the recovery task, no power management will be applied to the next task, which should run at the maximum frequency f_{max} to preserve its reliability.

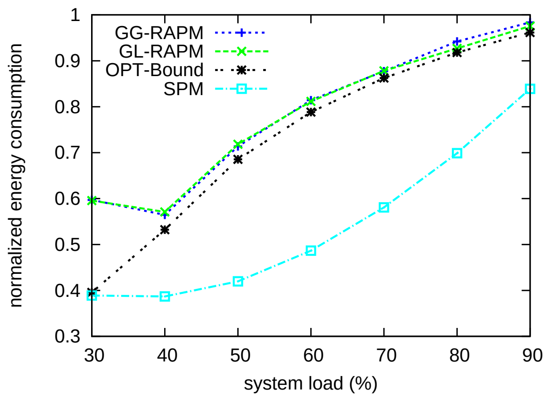
The algorithm is very similar to the dynamic power management algorithm in [29] and is omitted due to space limitation. However, we want to point out that, the dynamic slack reclamation with slack sharing will not extend the completion time of any task (including its recovery task, if available) in the static G-RAPM schedule. Therefore, regardless of task migrations at runtime, all tasks can complete their executions in time.

6 Simulations and Evaluations

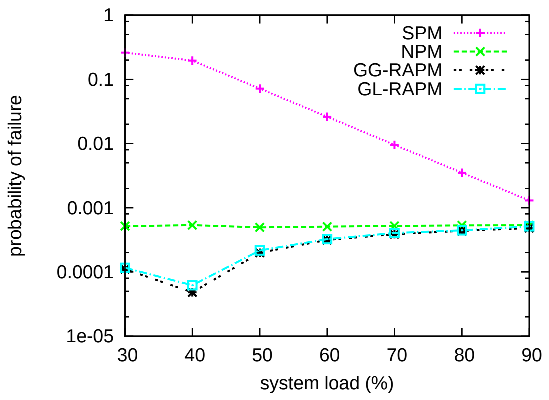
To evaluate the performance of our proposed G-RAPM schemes on energy savings and reliability, we developed a discrete event simulator. For easy of presentation, the scheme of *no power management (NPM)*, which executes all tasks at f_{max} and puts processors to power savings sleep states when idle, is used as the baseline and normalized energy consumption will be reported. Note that, as discussed in Section 2.1, the static power P_s will always be consumed for all schemes, which is set as $P_s = 0.01$. We further assume that $m = 3$, $C_{ef} = 1$, $P_{ind} = 0.1$ and normalized frequency is used with $f_{max} = 1$. In these settings, the energy efficient frequency can be found as $f_{ee} = 0.37$ (see Section 2.1).

For transient faults that follow the Poisson distribution, the lowest fault rate at the maximum processing frequency f_{max} (and corresponding supply voltage) is assumed to be $\lambda_0 = 10^{-5}$. This number corresponds to 10,000 FITs (failures in time, in terms of errors per billion hours of use) per megabit, which is a realistic fault rate as reported in [11, 33]. The exponent in the exponential fault model is assumed to be $d = 3$ (see Equation (3)). That is, the average fault rate is assumed to be 1000 times higher at the energy efficient speed f_{ee} (and corresponding supply voltage). The effects of different values of d have been evaluated in our previous work [25, 26, 30].

We consider synthetic task sets where each task set contains 100 real-time tasks, which will be executed on a system with 16 processors. The cases with other number of tasks and processors have also been evaluated and similar results have been obtained. For each task, its WCET is generated following a uniform distribution within the range of $[10, 100]$. Moreover, to emulate the actual execution time at runtime, we define α_i as the ratio of average over worst-case execution time for task T_i and the actual execution time of tasks follows a uniform distribution with the mean of $\alpha_i \cdot c_i$. The system load is defined as the ratio of overall workload of all tasks over system processing capacity $\gamma = \frac{\sum c_i}{k \cdot D}$, where k is the number of processors and D is the common deadline of tasks. Each result point in the figures is the average of 100 task sets and the execution of each task set is repeated for $5,000,000$ times.



(a) Normalized Energy Consumption



(b) Probability of Failure

Figure 7. Static G-RAPM Schemes

6.1 Static G-RAPM Schemes

First, assuming that all tasks take their WCETs, we evaluate the performance of the static G-RAPM schemes with local and global task selection, which are denoted as *GL-RAPM* and *GG-RAPM* in the figures, respectively. For comparison, the *ordinary static power management (SPM)*, which uniformly scales down the execution of all tasks based on the schedule length, is considered. Moreover, by assuming that there exists a subset of tasks with aggregated workload being exactly as X_{opt} and those tasks are selected and scaled down uniformly (see Section 4.2), the fault-free energy consumption of the task set is calculated, which provides an upper-bound on energy savings for any optimal static solution and denoted as *OPT-Bound*.

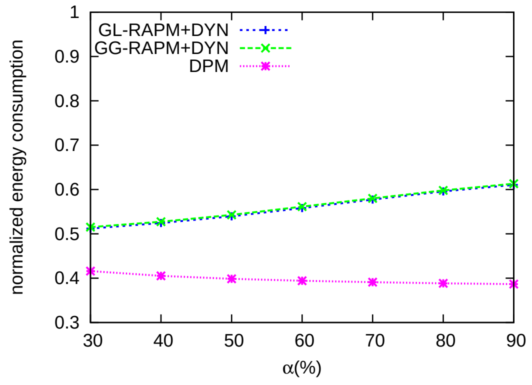
Figure 7(a) first shows the normalized energy consumption of the static G-RAPM schemes under different system loads (which is represented in the X-axis). From the figure we can see that, the static G-RAPM with local and global task selection schemes perform roughly the same in terms of energy savings. It comes from the fact that, in most cases, the managed workload of the selected tasks under both schemes has little difference. As system load increases, less static slack is available and in general more energy will be consumed for all schemes (with less energy savings). For moderate to high system loads, the normalized energy consumption under the static G-RAPM schemes is very close (within 3%) to that of *OPT-Bound*, which is in line with our previous results for uniprocessor systems [26, 27]. However, when system load is low (e.g., $\gamma = 0.3$), almost all tasks will be managed under the static G-RAPM schemes and run at a scaled frequency close to $f_{ee} = 0.37$, which may incur higher probability failure and thus more energy is consumed by the recovery tasks. Therefore, the normalized energy consumption for GL-RAPM and GG-RAPM increases. Compared to that of *OPT-Bound* that does not include energy consumption of recovery tasks, the difference becomes large when $\gamma = 0.3$.

Figure 7(b) further shows the *probability of failure*, which is the ratio of the number of failed tasks (by taking the recovery tasks into consideration) over total number of tasks executed.We can see that the static G-RAPM schemes can preserve system reliability (by having lower probability of failure of executing the tasks) when compared to that of NPM. In contrast, although SPM can save more energy, it can lead to significant system reliability degradation (up to two orders of magnitude) at low to moderate system loads.

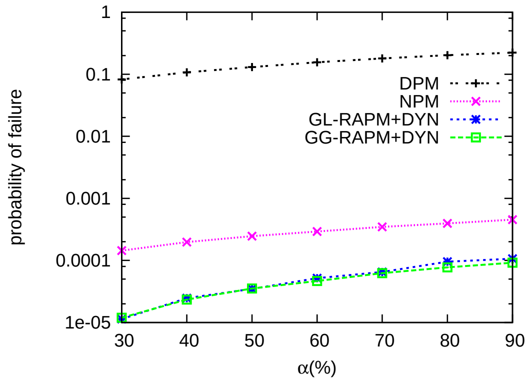
6.2 Dynamic G-RAPM Schemes

In this section, we evaluate the performance of the dynamic slack reclamation schemes by varying α from 0.3 to 0.9. Here, *GL-RAPM+DYN* represents the case of applying dynamic slack reclamation on top of the static schedule generated by the G-RAPM with local task selection. Similarly, *GG-RAPM+DYN* stands for the G-RAPM with global task selection. Again, for comparison, the ordinary *dynamic power management (DPM)* on top of the static schedule from SPM is also included.

First, at low system load $\gamma = 0.4$, Figure 8(a) shows the normalized energy consumption of the dynamic schemes. We can see that applying dynamic slack reclamation on top of two static schedules will achieve almost the same energy savings. The reason is that, when $\gamma = 0.4$, there is around 60% static slack available and the optimal workload to manage is 36%. That is, almost all tasks will be managed statically and run at $f = 0.42$ under both static schemes, which leave little space (with the limitation of $f_{ee} = 0.37$) for further

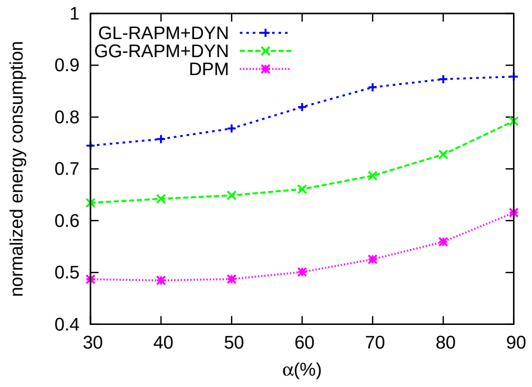


(a) Normalized Energy Consumption

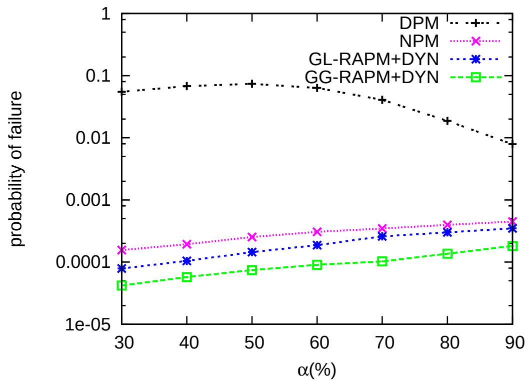


(b) Probability of Failure

Figure 8. Dynamic RAPM Schemes with system load $\gamma = 0.4$



(a) Normalized Energy Consumption



(b) Probability of Failure

Figure 9. Dynamic RAPM Schemes with system load $\gamma = 0.8$

energy savings under dynamic schemes. When α decreases, more dynamic slack will be available and more tasks can be scaled to the energy efficient frequency for slightly more energy savings. Not surprisingly, Figure 8(b) shows that system reliability can be preserved under the dynamic G-RAPM schemes, while the ordinary DPM resulting in increased probability of failure by three orders of magnitude.

Figure 9 further shows the results for a higher system load $\gamma = 0.8$. Here, we can see that applying dynamic slack reclamation to the static schedule of the G-RAPM with global task selection can lead to more energy savings (up to 15%) compared to that of local task selection. The main reason is that, at high system utilization $\gamma = 0.8$, very few tasks can be managed statically. By scheduling these managed tasks at the front of the schedule, GG-RAPM provides more opportunities for remaining tasks to reclaim the dynamic slack and yields more energy savings. Moreover, by managing more tasks at run time, the dynamic scheme on GG-RAPM schedule also has better system reliability as more tasks will have recovery tasks (shown in Figure 9(b)).

7 Conclusions

In this paper, for real-time tasks that share a common deadline, we studied global-scheduling-based reliability-aware power management (G-RAPM) schemes for multiprocessor real-time systems. After showing that the problem is NP-hard, we propose two efficient static heuristics, which rely on *global* and *local* task selections, respectively. To overcome the timing anomaly in global scheduling, the tasks' priorities (i.e., execution order) are determined through a *reverse dispatching process*. Moreover, we extended our previous work on dynamic power management with *slack sharing* to the reliability-aware settings.

Simulation results confirm that, the proposed G-RAPM schemes can preserve system reliability while achieving significant energy savings in multiprocessor real-time systems. The energy savings for the static schemes, for most cases, are within 3% of a theoretically computed ideal upper-bound. Moreover, by assigning higher priorities to scaled tasks with recoveries, the G-RAPM with global task selection provides better opportunities for dynamic slack reclamation at runtime.

References

[1] H. Aydin, V. Devadas, and D. Zhu. System-level energy management for periodic real-time tasks. In *Proc. of The 27^{th} IEEE Real-Time Systems Symposium*, Dec. 2006.

[2] H. Aydin, R. Melhem, D. Mossé, and P. Mejia-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Trans. on Computers*, 53(5):584–600, 2004.

[3] T. D. Burd and R. W. Brodersen. Energy efficient cmos microprocessor design. In *Proc. of The HICSS Conference*, Jan. 1995.

[4] I. Corp. Mobile pentium iii processor-m datasheet. Order Number: 298340-002, Oct 2001.

[5] F. Dabiri, N. Amini, M. Rofouei, and M. Sarrafzadeh. Reliability-aware optimization for dvs-enabled real-time embedded systems. In *Proc. of the 9th int'l symposium on Quality Electronic Design*, pages 780–783, 2008.

[6] V. Degalahal, L. Li, V. Narayanan, M. Kandemir, and M. J. Irwin. Soft errors issues in low-power caches. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 13(10):1157–1166, Oct. 2005.

[7] M. L. Dertouzos and A. K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *IEEE Trans. On Software Engineering*, 15(12):1497–1505, 1989.

[8] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operation Research*, 26(1):127–140, 1978.

[9] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner. Razor: circuit-level correction of timing errors for low-power operation. *IEEE Micro*, 24(6):10–20, 2004.

[10] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Proc. of The Int'l Conference on Computer-Aided Design*, pages 598–604, 1997.

[11] P. Hazucha and C. Svensson. Impact of cmos technology scaling on the atmospheric neutron soft error rate. *IEEE Trans. on Nuclear Science*, 47(6):2586–2594, 2000.

[12] http://public.itrs.net. International technology roadmap for semiconductors. 2008. S. R. Corporation.

[13] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. In *Proc. of The 14^{th} Symposium on Discrete Algorithms*, 2003.

[14] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh. Measurement and modeling of computer reliability as affected by system activity. *ACM Trans. Comput. Syst.*, 4(3):214–237, 1986.

[15] R. Jejurikar and R. Gupta. Dynamic voltage scaling for system wide energy minimization in real-time embedded systems. In *Proc. of the Int'l Symposium on Low Power Electronics and Design (ISLPED)*, pages 78–81, 2004.

[16] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proc. of the eighteenth ACM symposium on Operating systems principles*, pages 89–102, 2001.

[17] P. Pop, K. H. Poulsen, V. Izosimov, and P. Eles. Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems. In *Proc. of the 5th IEEE/ACM int'l conference on Hardware/software codesign and system synthesis*, pages 233–238, 2007.

[18] D. K. Pradhan. *Fault Tolerance Computing: Theory and Techniques*. Prentice Hall, 1986.

[19] S. Saewong and R. Rajkumar. Practical voltage scaling for fixed-priority rt-systems. In *Proc. of the 9^{th} IEEE Real-Time and Embedded Technology and Applications Symposium*, 2003.

[20] R. Sridharan, N. Gupta, and R. Mahapatra. Feedback-controlled reliability-aware power management for real-time embedded systems. In *Proc. of the 45th annual Design Automation Conference*, pages 185–190, 2008.

[21] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Proc. of The First USENIX Symposium on Operating Systems Design and Implementation*, Nov. 1994.

[22] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced cpu energy. In *Proc. of The 36^{th} Symposium on Foundations of Computer Science*, 1995.

[23] Y. Zhang, K. Chakrabarty, and V. Swaminathan. Energy-aware fault tolerance in fixed-priority real-time embedded systems. In *Proc. of the 2003 IEEE/ACM int'l conference on Computer-aided design*, 2003.

[24] B. Zhao, H. Aydin, and D. Zhu. Enhanced reliability-aware power management through shared recovery technique. In *Proc. of the Int'l Conf. on Computer Aidded Design (ICCAD)*, 2009.

[25] D. Zhu. Reliability-aware dynamic energy management in dependable embedded real-time systems. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2006.

[26] D. Zhu and H. Aydin. Energy management for real-time embedded systems with reliability requirements. In *Proc. of the Int'l Conf. on Computer Aidded Design*, 2006.

[27] D. Zhu and H. Aydin. Reliability-aware energy management for periodic real-time tasks. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2007.

[28] D. Zhu, H. Aydin, and J.-J. Chen. Optimistic reliability aware energy management for real-time tasks with probabilistic execution times. In *in the Proc. of the 29th IEEE Real-Time Systems Symposium (RTSS)*, 2008.

[29] D. Zhu, R. Melhem, and B. R. Childers. Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. *IEEE Trans. on Parallel and Distributed Systems*, 14(7):686–700, 2003.

[30] D. Zhu, R. Melhem, and D. Mossé. The effects of energy management on reliability in real-time embedded systems. In *Proc. of the Int'l Conf. on Computer Aidded Design*, 2004.

[31] D. Zhu, X. Qi, and H. Aydin. Priority-monotonic energy management for real-time systems with reliability requirements. In *Proc. of the IEEE International Conference on Computer Design (ICCD)*, 2007.

[32] D. Zhu, X. Qi, and H. Aydin. Energy management for periodic real-time tasks with variable assurance requirements. In *Proc. of the IEEE Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

[33] J. F. Ziegler. Trends in electronic reliability: Effects of terrestrial cosmic rays. available at http://www.srim.org/SER/SERTrends.htm, 2004.