Noname manuscript No.
(will be inserted by the editor)

Global Scheduling Based Reliability-Aware Power Management for Multiprocessor Real-Time Systems

Xuan Qi · Dakai Zhu · Hakan Aydin

Received: date / Accepted: date

Abstract *Reliability-aware power management (RAPM)* has been a recent research focus due the negative effects of the popular power management technique dynamic voltage and frequency scaling (DVFS) on system reliability. As a result, several RAPM schemes have been proposed for uniprocessor real-time systems. In this paper, for a set of frame-based independent real-time tasks running on multiprocessor systems, we study *global scheduling based RAPM (G-RAPM)* schemes. Depending on how recovery blocks are scheduled and utilized, both *individual-recovery* and *shared-recovery* based G-RAPM schemes are investigated. An important dimension of the G-RAPM problem is how to select the appropriate subset of tasks for energy <u>and</u> reliability management (i.e., scale down their executions while ensuring that they can be recovered from transient faults). We show that making such decision optimally (i.e., the static G-RAPM problem) is NP-hard. Then, for the individual-recovery based approach, we study two efficient heuristics, which rely on *local* and *global* task selections, respectively. For the shared-recovery based approach, a linear search based scheme is proposed. The schemes are shown to guarantee all task deadlines. Moreover, to reclaim the dynamic slack generated at runtime from early completion of tasks and unused recoveries, we also propose online G-RAPM schemes which exploit the *slack-sharing* idea studied in our previous work. The proposed schemes are evaluated through extensive simulations. The results show the effectiveness of the proposed schemes in yielding energy savings while simultaneously preserving system reliability and timing constraints. For static cases, as the shared-recovery based scheme can leave more slack and manage more tasks, it can save more energy compared to that of individual-recovery based schemes (especially for the cases with modest system loads). Moreover, by reclaiming dynamic slack generated at runtime, online G-RAPM schemes are shown to yield better energy savings.

This work was supported in part by NSF awards CNS-0720651, CNS-0720647, CNS-1016855, CNS-1016974, and NSF CAREER Awards CNS-0546244 and CNS-0953005.

Xuan Qi
Dept. of Computer Science, University of Texas at San Antonio, One UTSA Circle, San Antonio, TX 78249.

Dakai Zhu, corresponding author (dzhu@cs.utsa.edu)
Dept. of Computer Science, University of Texas at San Antonio, One UTSA Circle, San Antonio, TX 78249.

Hakan Aydin
Department of Computer Science, George Mason University, 4400 University Drive, Fairfax, VA 22030.

Keywords Energy management · Dynamic Voltage anf Frequency Scaling · Reliability management · Multiprocessor Real-Time Systems

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 [24]. One common strategy to save energy in computing systems is to operate the 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 [7] and scales down simultaneously the processing frequency and supply voltage to save energy [40].

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 [5,31,35,50]. 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) [15,21,51]. 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.

A 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 [34]. 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 [46]. The central idea of the RAPM scheme is to exploit the available slack to schedule a recovery task at a task's dispatch time before utilizing the remaining slack for DVFS to scale down the task execution and save energy, thereby preserving the system reliability [46]. Following this line of research, several RAPM schemes have been proposed for various task models, scheduling policies, and reliability requirements [13,36,45,47–49,54,55], all of which have focused on uniprocessor systems. For a system with multiple DVFS-capable processing nodes, Pop *et al.* developed a constraint-logic-programming (CLP) based solution to minimize energy consumption for a set of dependent tasks represented by directed acyclic graphs (DAGs), where the user-defined reliability goals is transformed to tolerating a fixed number of transient faults through re-execution [33].

Different from all the existing work, in this paper, for a set of independent 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, where both *individual-recovery* and *shared-recovery* based schemes are investigated. Note that there are two different paradigms in multiprocessor real-time scheduling: the *partitioned* and *global* approaches [16,17]. For partitioned scheduling, tasks are statically mapped to processors and a task can only run on the processor to which it is assigned to. After mapping tasks to processors, applying the existing uniprocessor RAPM schemes on each processor can be straightforward. In contrast, for global scheduling, tasks can run on any processor and even migrate between processors at run-time depending on tasks' dynamic behaviours, which makes the RAPM problem more challenging. Moreover, 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 (which has been the traditional argument against global scheduling) can be significantly reduced. Hence, in this paper, we investigate global scheduling based RAPM schemes for multiprocessor real-time systems.

First, for individual-recovery based approach (where a recovery task is scheduled for each selected task and the execution of selected tasks is then scaled down accordingly), we show that the static G-RAPM problem is NP-hard. Then, we propose two heuristic schemes, which are characterized by *global* and *local* task selections, respectively, depending on how to exploit the system slack and when to select the appropriate subset of tasks for energy and reliability management. Observing 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. For shared-recovery based approach, where tasks whose executions are scaled down on one processor share a common recovery block, a linear search based scheme is explored to find out the subset of tasks that should be managed. Note that, the unselected tasks will run at the maximum frequency to preserve system reliability.

In addition, to reclaim the dynamic slack generated from early completion of tasks and/or free of 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 proposed G-RAPM schemes are evaluated through extensive simulations. The results show the effectiveness of the proposed G-RAPM schemes in preserving system reliability while achieving significant energy savings for multiprocessor real-time systems. For static cases, since the shared-recovery based scheme can leave more slack and manage more tasks, it can save more energy compared to that of individual-recovery based schemes (especially for the cases with modest system loads). By reclaiming dynamic slack generated at runtime, dynamic G-RAPM schemes could further scale down the execution of selected tasks or manage more tasks, which in turn leads to higher energy savings.

The remainder of this paper is organized as follows. Section 2 reviews the related work. Section 3 presents the system models and formulates the global scheduling based RAPM (G-RAPM) problem after reviewing the key idea of RAPM. In Section 4, based on the idea of individual-recovery, both static heuristics and dynamic schemes are proposed. Section 5 presents the shared-recovery based static and online adaptive G-RAPM schemes. Section 6 discusses the simulation results. We conclude the paper in Section 7.

2 Related Work

Based on the *dynamic voltage and frequency scaling (DVFS)* technique [38], which trades processing speed for energy savings, energy management has been studied extensively for single-processor systems [4,26,31]. For multiprocessor systems, based on the partitioned scheduling policy, Aydin *et al.* studied the problem of how to partition real-time tasks among multiple processors to minimize the dynamic energy consumption [6]. Their results show that, when earliest deadline first (EDF) scheduling is adopted on each processor, balancing the workload among all the processors evenly gives the optimal energy consumption and the general partition problem for minimizing energy consumption in multiprocessor real-time system is NP-hard in the strong sense [6]. The work was extended to consider rate monotonic scheduling (RMS) in their later work [1]. Anderson and Baruah investigated how to synthesize a multiprocessor real-time system with periodic tasks such that the energy consumption is minimized at run-time [2]. Chen *et al.* proposed a series of approximation algorithms to improve energy-efficiency of multiprocessor real-time systems, where both frame-based tasks and periodic tasks have been considered, with and without leakage power consideration [9,10,8,39]. In our previous work, based on global scheduling, power-aware algorithms have been developed for real-time multiprocessor systems, which exploits the slack reclamation and slack sharing for energy saving [50,53]. More recently, Choi and Melhem studied the interplay between parallelism of an application, program performance, and energy consumption [11]. For an application with given ratio of serial and parallel portions and the number of processors, the authors derived optimal frequencies allocated to the serial and parallel regions in an application to either minimize the total energy consumption or minimize the energy-delay product.

The joint consideration of energy management and fault tolerance has attracted attention in recent years. For independent periodic tasks, using the primary/backup model, Unsal *et al.* proposed an energy-aware software-based fault tolerance scheme. The scheme postpones as much as possible the execution of backup tasks to minimize the overlap between primary and backup executions, in an attempt to reduce energy consumption [37]. For Duplex systems (where two concurrent hardware platforms are used to run the same software for fault detection), Melhem *et al*. explored the optimal number of checkpoints, *uniformly* or *non-uniformly* distributed, to achieve minimum energy consumption [30]. Elnozahy *et al*. proposed an *Optimistic-TMR* (OTMR) scheme to reduce the energy consumption for traditional TMR (Triple Modular Redundancy, in which three hardware platforms are used to run the same software simultaneously to detect and mask faults) systems by allowing one processing unit to slow down provided that it can catch up and finish the computation before the deadline if there is a fault [20]. In [52], Zhu *et al.* further explored the optimal frequency setting for OTMR and presented detailed comparisons among Duplex, TMR and OTMR on reliability and energy consumption. Combined with voltage scaling techniques, Zhang *et al.* have proposed an adaptive checkpointing scheme to tolerate a fixed number of transient faults and save energy for serial applications [41]. The work was further extended to periodic real-time tasks in [42]. Izosimov *et al.* obtain and solve an optimization problem for mapping a set of tasks with reliability constraints, timing constraints and precedence relations to processors and for determining appropriate fault tolerance policies (re-execution and replication) [28]. However, the existing research on co-management of energy and reliability either focused on tolerating fixed number of faults [28,20,30] or assumed a constant fault rate [41,43].

More recently, DVFS has been shown to have a direct and negative effect on system reliability due to increased number of transient faults (especially the ones induced by cosmic ray radiations) at lower supply voltages [15,21,51]. Taking such effects into consideration, Zhu has studied a reliability-aware power management (RAPM) scheme that can preserve system reliability while exploiting slack time for energy savings [46]. The central idea of RAPM is to reserve a portion of the available slack to schedule a *recovery task* for the task whose execution is scaled down, to recuperate the reliability loss due to the energy management [46]. The scheme is further extended to consider multiple tasks with a common deadline [47], periodic real-time tasks [48,54] as well as different reliability requirements [44,45,49,55].

Ejlali *et al.* studied schemes that combine the information (about hardware resources) and temporal redundancy to save energy and to preserve system reliability [19]. By employing a feedback controller to track the overall miss ratio of tasks in soft real-time systems, Sridharan *et al.* [36] proposed a reliability-aware energy management algorithm to minimize the system energy consumption while still preserving the overall system reliability. Pop *et al.* studied the problem of energy and reliability trade-offs for distributed heterogeneous embedded systems [32]. The main idea is to transform the user-defined reliability goals to the objective of tolerating a fixed number of transient faults by switching to pre-determined contingency schedules and re-executing individual tasks. A constrained logic programming-based algorithm is proposed to determine the voltage levels, process start time and message transmission time to tolerate transient faults and minimize energy consumption while meeting the timing constraints of the application. Dabiri *et al.* [14] considered the problem of assigning frequency/voltage to tasks for energy minimization subject to reliability and timing constraints. More recently, Ejlali *et al.* study a standby-sparing hardware redundancy technique for fault-tolerance [18]. Following the similar idea in OTMR [20], the standby processor is operated at low power state whenever possible provided that it can catch up and finish the tasks in time. This scheme is evaluated through simulations and show better energy performance when compared to that of the backward recovery based approach [18].

Different from the existing work, by focusing on global scheduling, we study the RAPM problem for a set of independent frame-based tasks that share a common deadline and run on a multiprocessor real-time system. To find the proper priority and frequency assignment for the tasks, we propose both individual-recovery and shared-recovery based G-RAPM schemes. The online schemes with dynamic slack reclamation using slack-sharing technique are also studied.

3 System Models and Problem Formulation

In this section, we first present the power and fault models considered in this work and state our assumptions. Then, the task and application model are discussed and the problem to be addressed in this paper is formulated after reviewing the key idea of reliability-aware power management (RAPM).

3.1 Power Model

Considering the linear relation between processing frequency and supply voltage [7], the *dynamic voltage and frequency scaling (DVFS)* technique reduces the supply voltage for lower frequency requirements to reduce the system's dynamic power consumption [38]. 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 [3,25,29,35].

In our previous work, we proposed and used a *system-level power model* for uniprocessor systems [3,51] and similar power models have also been adopted in other studies [25, 29,35]. In this paper, we consider a shared-memory system with k identical processors and, by following the same principles, the power consumption of the system can be expressed as:

$$\begin{aligned} P(f_1,\ldots,f_k) &= P_s + \sum_{i=1}^{k} \hbar_i (P_{ind} + P_{d,i}) \\ &= P_s + \sum_{i=i}^{k} \hbar_i (P_{ind} + C_{ef} \cdot f_i^m) \end{aligned} \tag{1}$$

Above, 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. When the i'th processor executes a tasks, it is said to be *active* (i.e., $\hbar_i = 1$). In that case, its *active power* has two components: 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). Otherwise, 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 (i.e., $\hbar_i = 0$) [12]. The effective switching capacitance C_{ef} and the dynamic power exponent m (which is, in general, no smaller than 2) are system-dependent constants [7]. 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 with multiple processors and can support various power management techniques (e.g., DVFS and power saving sleep states).

Due to the energy consumption related to the frequency-independent active power 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 [3,29,35,51]. 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 $f_{ee} = \sqrt[m]{\frac{P_{ind}}{C_{ef}\cdot(m-1)}}$ [3,51].

We further assume that the frequency of the processors can vary *continuously* from the minimum frequency f_{min} to the maximum frequency f_{max}. For processors with only a few discrete frequency levels, we can either use two adjacent frequency levels to emulate the desired frequency [26], or use the next higher discrete frequency to ensure the solution's feasibility. Moreover, we use normalized frequencies in this work and it is assumed that $f_{max} = 1$. From the above discussion, for energy efficiency, the processing frequency for any task should be limited to the range of $[f_{low}, f_{max}]$, where $f_{low} = max\{f_{min}, f_{ee}\}$. In addition, the time overhead for adjusting frequency (and supply voltage) is assumed to be incorporated into the worst-case execution time of tasks, which can also be handled by reserving a small share of slack before utilizing them for DVFS and recovery as discussed in [5,50].

3.2 Fault and Recovery Models

Various reasons can 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 [27]. The inter-arrival rate of transient faults is assumed to follow Poisson distribution [43]. 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 [51]:

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

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 $f < f_{max}$.

More specifically, in this work, we consider an exponential fault rate model, where $g(f)$ is given by [47]:

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

Here d (> 0) is a constant, representing the sensitivity of fault rates to DVFS. That is, the highest fault rate will be $\lambda_{max} = \lambda_0 \cdot 10^d$, which corresponds to the lowest energy efficient frequency f_{low}.

Transient faults are assumed to be detected by using *sanity* (or *consistency*) checks at the completion of a task's execution [34]. 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 [43,46]. Again, for simplicity, the overhead for fault detection is assumed to be incorporated into the worst-case execution time of tasks.

3.3 Task Model and 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. This simplified task model enables us to identify and tackle several open issues related to global scheduling based RAPM. In our future work, we plan to explore dependent real-time tasks represented by a directed acyclic graph (DAG), where the dependency between tasks will make the system slack be accesible by different tasks in non-equal amounts.

Reliability-Aware Power Management (RAPM): 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 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 [46].

Instead of using *all* the available slack for DVFS to scale down the execution of task T and 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 [46]. The recovery task RT will be dispatched (at the maximum frequency f_{max}) only if a transient

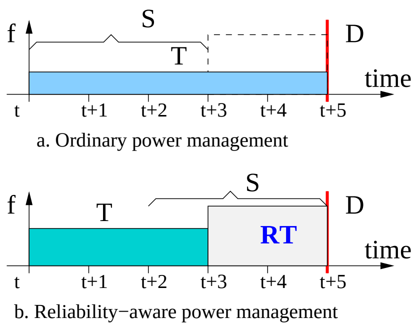

f
S
T
D
time
t
t+1
t+2
t+3
t+4
t+5
a. Ordinary power management
f
T
S
RT
D
time
t
t+1
t+2
t+3
t+4
t+5
b. Reliability−aware power management

Fig. 1 Ordinary and Reliability-Aware Power Management [46].

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 (which is defined as the probability of having no fault during T's execution when no power management is applied), regardless of different fault rate increases at scaled processing frequencies [46].

Problem Formulation: From the above discussion, to address the negative effects of DVFS on transient faults, a recovery task needs to be scheduled before the deadline of any task T_i which is dispatched at a scaled frequency $f < 1$, so as to preserve its original reliability. Although it is possible to preserve the overall system reliability while sacrificing the reliability for some tasks, for simplicity, we focus on maintaining the original reliability of each task in this work. 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. Note that, due to workload constraints and/or energy efficiency consideration, occasionally some tasks may not be selected for power management. The binary variable x_i is used to denote whether T_i is selected for management. Specifically, $x_i = 1$ indicates that task T_i is selected for management; otherwise, if T_i is not selected, $x_i = 0$. For tasks that are not selected, they will run at the maximum processing frequency f_{max} to preserve their reliability.

Recall that the system is assumed to be on all the time and the static power P_s is always consumed. Therefore, we will focus on managing the energy consumption related to system active power. At the frequency f_i, the active energy consumption to execute task T_i is:

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

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. As the scheduling policy, we consider *static-priority* global scheduling where the priorities of tasks form a total order and remain constant at run-time, once they are statically determined.

Suppose that the worst-case completion time of task T_i (and its recovery in case of the primary scaled execution of T_i fails) under a given schedule is ct_i. 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., x_i) and the scaled frequencies of selected tasks (i.e., f_i) to:

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

subject to

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

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

$$\max\{ct_i | i = 1, \ldots, n\} \leq D \tag{8}$$

Here, Equation (6) restricts the scaled frequency of any selected task to the range of $[f_{low}, f_{max})$. Equation (7) states that the *un-selected* tasks will run at f_{max}. The last condition (i.e., Equation (8)) ensures the schedulability of the task set under the given global scheduling algorithm with the priority assignments and task selections.

Depending on how recovery tasks are scheduled and utilized, we can have a separate recovery task for each selected task (i.e., the *individual recovery* approach) or have several selected tasks on the same processor share one recovery task (i.e., the *shared recovery* approach). In what follows, in increasing level of sophistication and complexity, we will investigate *individual-recovery* and *shared-recovery* based G-RAPM schemes, where both static and dynamic schemes are studied.

4 G-RAPM with Individual Recovery

The first and intuitive approach to preserve the system reliability is to have a separate recovery task *statically* scheduled for each selected task whose execution is to be scaled down. In this section, after showing the static individual recovery based G-RAPM problem is NP-hard, we present two static heuristic schemes as well as an online scheme that can efficiently reclaim dynamic slack generated at runtime.

4.1 Static Individual Recovery Based G-RAPM Schemes

Note that, when the system has only one processor (i.e., $k = 1$), the static individual recovery based G-RAPM problem will reduce to the static RAPM problem for uniprocessor systems, which has been studied in our previous work and shown to be NP-hard [47]. Therefore, finding the optimal solution for the static individual recovery based G-RAPM problem to minimize the fault-free energy consumption will be NP-hard as well. Note that, there are a few inter-related key issues in solving this problem, such as priority assignment, slack determination, and task selection. Depending on how the available slack is determined and utilized to select tasks, in what follows, we study two heuristic schemes that are based on *local* and *global* task selection, respectively. The performance of these schemes will be evaluated against the theoretically ideal upper bound on energy savings in Section 6.

4.1.1 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 [16]. 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 [50]. 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 [47] can be applied for the tasks that are statically mapped on each processor individually.

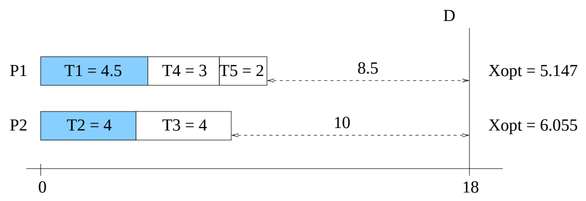


Fig. 2 Canonical task-to-processor mapping and local task selection. Shaded tasks are selected.

For example, consider a task set with five tasks $T_1(4.5)$, $T_2(4)$, $T_3(4)$, $T_4(3)$ and $T_5(2)$, that are 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[1] 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 [47], we have shown that, for a single processor system where the amount of available slack is S, to maximize energy savings under RAPM, the optimal aggregate workload for the selected tasks should be:

$$X_{opt} = S \cdot \left(\frac{P_{ind} + C_{ef}}{m \cdot C_{ef}} \right)^{\frac{1}{m-1}} \tag{9}$$

Assume that $P_{ind} = 0.1$, $C_{ef} = 1$ and $m = 3$ [47], 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 [47,48], we can see that tasks T_1 and T_2 will be selected for management on two processors, respectively.

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 in the final canonical schedule (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, with the original LTF order of tasks and the calculated scaled frequencies for selected tasks, Figure 4 shows the partial online schedule after the first four tasks are dispatched where all dispatched tasks (including recovery tasks, if any) are assumed to take

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

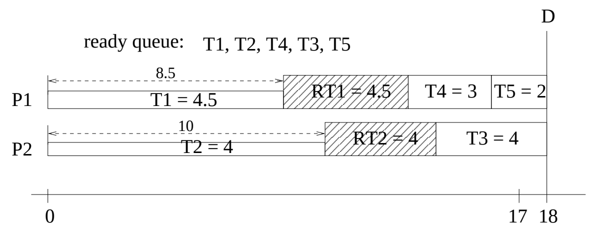


Fig. 3 Final static schedule for local task selection with recovery tasks and scaled execution.

their WCETs. We can easily see that there is not enough time on any of the processors and task T_5 will miss the deadline.

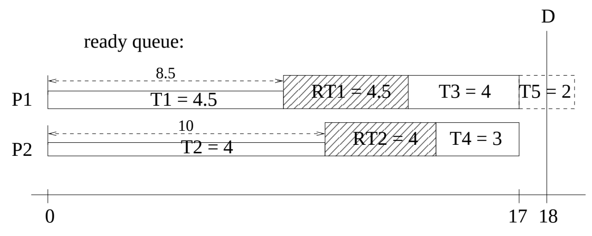


Fig. 4 Task T_5 misses the deadline with the original task order (priority assignment).

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 (with the latest start time) to the first task (with the earliest start time). 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 individual recovery based 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 feasible 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 Individual Recovery based G-RAPM with local task selection

```
1: Step 1: Get the canonical schedule from any initial feasible 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 [47] for tasks mapped to
   that processor (i.e., determine its available slack $S_i$; calculate $X_{opt}$, select tasks and calculate the scaled
   frequency for selected tasks);
4: Step 3: Get the final execution order (i.e., priority assignment) of tasks based on the start time of tasks in
   the final canonical schedule obtained in Step 2, where a task with early start time should a higher priority.
```

4.1.2 Global Task Selection

In local task selection scheme, 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 longest task first heuristic, three tasks (T_1, T_2 and T_3, with the aggregated workload of 11.5) are selected to achieve the maximum energy savings.

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 load balancing among the processors, which may not be feasible. Note that, when the scaled execution of a selected task completes successfully, its recovery task can be dropped and the corresponding time becomes to be dynamic slack. Therefore, to provide better opportunity for dynamic slack reclamation at runtime, we should first map those selected tasks as well as their recovery tasks to processors (again, following the LTF heuristic). After that, the unselected tasks are mapped to processors accordingly. The resulting canonical mapping for the above example is shown in Figure 5.

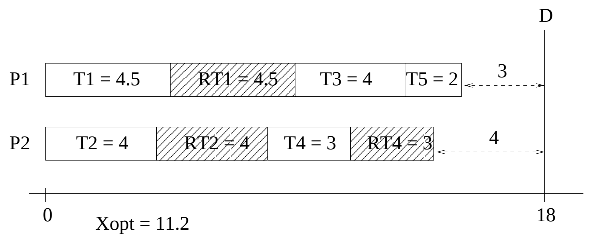


Fig. 5 Canonical task-to-processor mapping with global task selection.

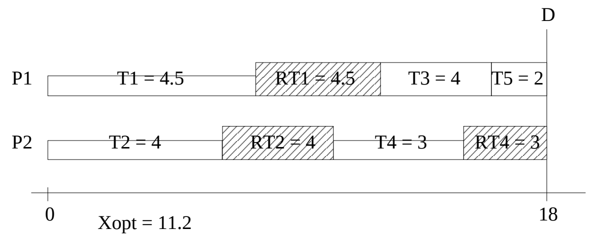


Fig. 6 Final canonical schedule with global task selection.

Here, there are 3 and 4 units of slack that can be utilized to scale down the selected tasks on the two 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 based on the start time of tasks in the final schedule. For this example, considering the scaled execution of globally selected tasks, the final order (i.e., priority) of tasks in the global queue can be found as T_1, T_2, T_4, T_3 and T_5.

For energy savings, with the parameters for the power model given in Section 4.1.1, one can compute that the individual recovery based G-RAPM scheme with global task selection can save 32.4% energy when compared to that of no power management (NPM). In contrast, the scheme with local task selection saves only 26.6% (i.e., global task selection obtains 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 the unused recovery tasks, which is further evaluated and discussed in Section 6.

Note that, the scheme with 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 $O(max(n \cdot log(n), k \cdot n))$.

Algorithm 2 Individual Recovery based 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 based on the *start time* of tasks
    in the final canonical schedule obtained in Step 2, where a task with early start time has a higher priority.
```

4.2 Online Individual Recovery Based G-RAPM Scheme

It is well-known that real-time tasks typically take a small fraction of their WCETs [22]. Moreover, the recovery tasks are executed only if their corresponding scaled primary tasks fail at a small probability. Therefore, significant amount of dynamic slack can be expected at run time from early completion of tasks and/or free of the time reserved for recovery tasks, which provides abundant opportunities to further scale down the execution of selected tasks (as low as f_{low}) for additional energy savings or to manage more tasks for enhancing system reliability. However, as shown in our previous work, simple greedy dynamic slack reclamation under global scheduling for multiprocessor systems may lead to timing anomaly and cause deadline misses [50] and special care is needed to reclaim such slack at runtime.

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 systems [50]. The basic idea of slack sharing is to mimic the timing of the static feasible 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 (if any) can be utilized to scale down the next task's execution and save energy.

The idea of slack sharing can also be applied on top of the canonical schedules generated from the static individual recovery based G-RAPM schemes (with either local or global task selection) at runtime. Different from the work in [50] where dynamic slack is only used to further scale down the execution of tasks, after sharing the slack appropriately, the dynamic slack reclamation should be differentiated for scaled tasks that already have statically scheduled recovery tasks and the ones that do not have recovery tasks yet.

Algorithm 3 summarizes the steps of the online individual recovery based G-RAPM scheme. The input for the algorithm is the global *Ready-Q*, which holds all tasks in the order of tasks' priorities, as well as the statically determined scaled frequency of tasks. Similar to the algorithms in [50], the *expected finish time* (EFT) of a processor is defined as the latest time when the processor will complete the execution of its currently running task (including recovery task) in the worst case, which is essentially the same as the *finish time* of the task (including its recovery task) in the static canonical schedule. f_i is the scaled frequency for task T_i.

At the beginning or after the processor $PROC_x$ completes the execution of its task, it will first share part of its slack (if any) with the processor who supposes to complete early (lines 5 to 7). Then, it will fetch the next ready task T_i in the global *Ready-Q*. If T_i a scaled task, the expected finish time for processor $PROC_x$ will move forward including the recovery time for task T_i scheduled offline, and the dynamic slack will be utilized to re-calculate the scaled frequency for task T_i (lines 13 and 14). Notice that the scaled frequency f_i will be limited by f_{low} to ensure energy efficiency. Otherwise, if T_i was not selected offline and does not have its recovery scheduled yet, depending on the amount of available dynamic slack, task T_i can either be scaled down after scheduling its recovery (lines 17 to 19) or it will be executed at the maximum frequency f_{max} to preserve its reliability (line 22). Then, the processor $PROC_x$ will execute task T_i at frequency f_i and recover its execution properly if T_i has its recovery (lines 26 and 27).

Note that, with proper slack sharing at runtime, dynamic slack reclamation under the global scheduling based power management scheme does not extend the completion time of any task compared to that of tasks in the static canonical schedule, which in turn ensures

Algorithm 3 Online Individual Recovery Based G-RAPM Algorithm

```
1: Input: priorities and scaled frequencies of tasks obtained from the static individual recovery based G-
   RAPM schemes (with either local or global task selection);
2: /* Suppose that the current processor is $PROC_x$; */
3: /* and the current time is $t$; */
4: while (Ready-Q $\neq \emptyset$) do
5:     Find processor $PROC_y$ with the minimum $EFT_y$;
6:     if ($EFT_x > EFT_y$) /*if $PROC_y$ supposes to finish early;*/ then
7:         Switch $EFT_x$ and $EFT_y$; /*$PROC_x$ shares its slack with $PROC_y$;*/
8:     end if
9:     $T_i$ = Next ready task in Ready-Q;
10:    $EFT_x + = \frac{c_i}{f_i}$; //move forward the expected finish time on $PROC_x$;
11:    $Slack = EFT_x - t$; //current available time for task $T_i$;
12:    if ($T_i$ already has its recovery task) then
13:        $EFT_x + = c_i$; //incorporate $T_i$'s recovery time into expected finish time;
14:        $f_i = \max\{f_{low}, \frac{c_i}{Slack}\}$; //re-calculate $T_i$'s scaled frequency;
15:    else
16:        /*task $T_i$ has no recovery task yet;*/
17:        if ($Slack > 2 \cdot c_i$) //slack is enough for a new recovery task; then
18:            Schedule a recovery task $RT_i$ for task $T_i$;
19:            $f_i = \max\{f_{low}, \frac{c_i}{(Slack - c_i)}\}$; //calculate scaled frequency
20:        else
21:            /*otherwise, no enough slack for a recovery task;*/
22:            $f_i = f_{max} = 1.0$;
23:        end if
24:    end if
25:    Execute task $T_i$ at frequency $f_i$;
26:    if ($T_i$ fails and has a recovery task $RT_i$) then
27:        Execute recovery task $RT_i$ at frequency $f_{max}$;
28:    end if
29: end while
```

that all tasks can complete their executions in time [50]. Following the same reasoning, we can have the following theorem.

Theorem 1 *For a given set of tasks whose priorities and scaled frequencies have been determined by an offline individual recovery based G-RAPM scheme, any task (including its recovery task, if available) under Algorithm 3 will finish no later than its finish time in the static canonical schedule generated offline.*

Corollary 1 *Under the individual recovery based G-RAPM scheme, the original system reliability is preserved.*

5 G-RAPM with Shared Recovery Tasks

Observing the conservatism of the RAPM schemes with individual recovery tasks, a *shared recovery* based RAPM scheme has been studied for uniprocessor systems in our recent work [45]. Note that, in uniprocessor systems, the scaled tasks are executed *sequentially* and their recovery tasks will not be invoked simultaneously. Therefore, instead of scheduling a separate recovery task for each selected task, the shared recovery scheme only reserve slack for one recovery block, which is *shared* among all selected tasks, and more slack will be available for DVFS to save more energy. The superiority of the shared recovery scheme for uniprocessor systems on energy savings as well as reliability enhancement has been shown in our previous work [45].

However, extending the idea of shared recovery to RAPM schemes for multiprocessor systems introduces non-trivial issues considering the potentially *simultaneous failures* during the *concurrent* scaled execution of selected tasks. In what follows, we first discuss a simple shared recovery based G-RAPM scheme and prove its feasibility to meet tasks' deadline while preserving system reliability. Then, a static linear search based method to find the size of the shared recovery and the proper subset of tasks for management is addressed. The online adaptive shared recovery based G-RAPM scheme is studied at the end of this section.

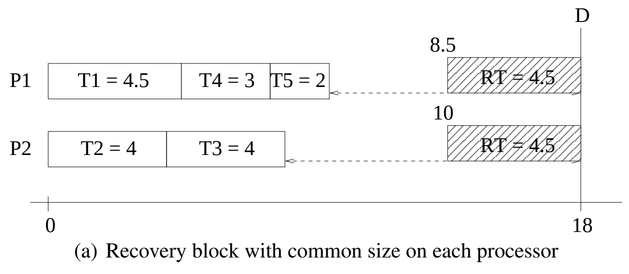
5.1 Common Size Shared Recovery on Each Processor

In the shared recovery RAPM scheme for uniprocessor systems [45], some of the static slack is first reserved as a recovery block, which can be shared by all selected tasks and has the same size as the largest selected tasks. The remaining slack is used to scale down the execution of the selected tasks. The selected tasks are executed at the scaled frequency as long as no fault occurs. However, when a faulty scaled execution of a selected task is observed and recovered using the recovery block, the static shared recovery RAPM scheme will switch to a *contingency schedule* and execute all remaining tasks at the maximum frequency to preserve system reliability and guarantee the timing constraints.

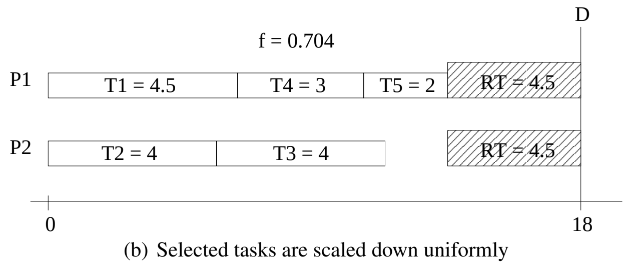
As mentioned early, considering the possibility of having simultaneous faults during the concurrent scaled execution of selected tasks in multiprocessor systems, having *only one* recovery block cannot guarantee system reliability preservation and timing constraints. Intuitively, each processor in the system needs to have a recovery block that can be shared by the selected tasks statically mapped to the processor. However, it is not trivial to find the proper size of the recovery block on each processor as tasks may run on a different processor at runtime under global scheduling, especially considering the dynamic behaviour of real-time tasks. Moreover, to ensure timing constraints, *coordination* is needed among the processors in global scheduling on the recovery steps once a fault occurs.

In this work, to ensure that the largest selected task can be recovered on *any* processor, we study a simple shared recovery based G-RAPM scheme, where the *recovery block on each processor is assumed to have the same size as that of the largest selected tasks*. Before formally presenting the algorithm, we use the aforementioned example to illustrate the basic ideas. Figure 7(a) shows the static schedule, where there are 8.5 and 10 units of slack on the first and second processor, respectively. Suppose that all tasks are selected for management. After reserving the shared recovery block on each processor, where both recovery blocks have the size of 4.5 (the maximum size of selected tasks), the remaining slack are used to scaled down the selected tasks *uniformly* to the frequency $f = 0.704$, as shown in Figure 7(b). Note that, it is possible to further scale down tasks T_2 and T_3 on the second processor. However, exploiting such possibility brings additional complexity and we will adopt the common scaled frequency for simplicity.

By having a shared recovery block on each processor, more slack will be available for DVFS to manage more tasks and save more energy. For the above example, with the same parameters for the power model as the ones given in Section 4.1.1, the energy savings under the shared recovery based G-RAPM scheme can be calculated as 41.3% (assuming no fault occurs), which is a significant improvement over the individual recovery based G-RAPM schemes (28.6% and 32.4% for local and global task selection, respectively).



(a) Recovery block with common size on each processor



(b) Selected tasks are scaled down uniformly

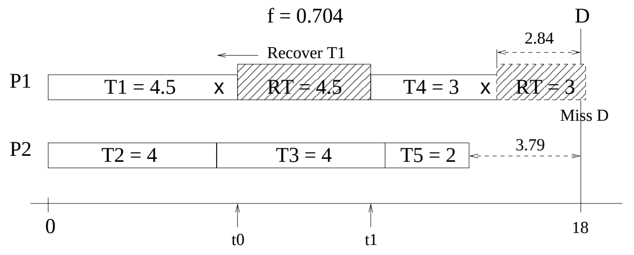
Fig. 7 The static canonical schedule under the shared recovery based G-RAPM scheme

Another important aspect of the shared recovery based G-RAPM scheme is on its handling of faulty tasks. When the faulty scaled execution of a selected task is detected on one processor, other processors need to be notified as well to coordinate the adoption of a contingency schedule. First, for the concurrently running tasks on other processors, they can continue their executions at the scaled frequency, which can be recovered later in case faults occur during their scaled execution. Note that, as shown next, with one recovery block on each processor, the shared recovery based G-RAPM scheme ensures that any faulty execution of the concurrently running tasks can be recovered in time without violating tasks' timing constraints. However, for the remaining tasks that are still in the global ready queue, similar to the shared recovery RAPM scheme for uniprocessor systems, they should be dispatched at the maximum frequency for reliability preservation.

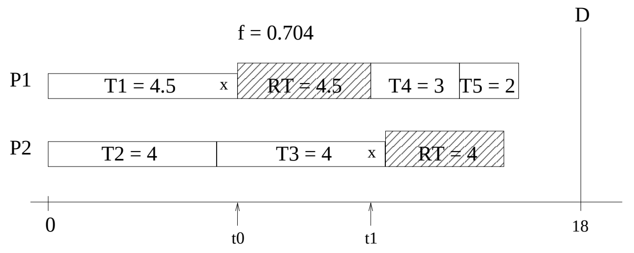
For the above example, suppose that task T_1 fails and is recovered on the first processor. If task T_4 is still dispatched at the scaled frequency and fails as shown in Figure 8(a), we can see that it cannot be recovered in time, which in turn leads to system reliability degradation. On the other hand, if T_4 runs at f_{max}, even if the task T_3 that runs on the second processor concurrently with T_1 continues its execution at the scaled frequency, it can still be recovered in time after the scaled execution fails. Moreover, the remaining tasks T_4 and T_5 can also finish in time as long as they are dispatched at the maximum frequency as shown in Figure 8(b).

The outline of the shared recovery based G-RAPM algorithm is summarized in Algorithm 4. Here, a global boolean variable (flag) $FaultOccurence$ is used to indicate whether faults have been detected during the scaled execution of selected tasks in the current frame or not. Note that, the large tasks that are not selected for management are dispatched at the maximum frequency f_{max}, hence their original reliability is preserved by definition, since DVFS is not applied.

The input for Algorithm 4 is assumed to be a set of feasible priority and frequency assignment for tasks. That is, if tasks are dispatched from the global queue following their



(a) Faulty scaled execution of remaining tasks cannot be recovered in time



(b) Concurrent failures (of T_1 and T_3) can be recovered in a timely manner

Fig. 8 Faulty execution handling in the static shared recovery based G-RAPM scheme.

Algorithm 4 : Outline of the shared recovery based G-RAPM algorithm

```
1: Input: Feasible priority and frequency assignments for tasks;
2: **For task completion events:**
3: **if** (the completed task is executed at scaled frequency) AND (execution fails) **then**
4:     Set the global flag: $FaultOccurence = true$;
5:     Re-invoke the faulty task for re-execution at the maximum frequency $f_{max}$;
6: **end if**
7: **For task dispatch events:**
8: **if** ($FaultOccurence == false$) **then**
9:     Dispatch the header task $T_i$ in *Ready-Q* at its assigned frequency $f_i$;
10: **else**
11:     Dispatch the header task $T_i$ in *Ready-Q* at the maximum frequency $f_{max}$;
12: **end if**
13: **All tasks complete and current frame ends:**
14: Reset the global flag: $FaultOccurence = false$;
```

priorities at their assigned frequencies, the static *canonical* schedule where all tasks are assumed to take their WCETs will finish $c_{max}^{selected}$ time units (the size of the largest selected task, which is also the size for recovery blocks) before the deadline. When there is no fault, from [50], we know that under global scheduling all tasks will be dispatched no late than their start times in the canonical schedule.

After a faulty scaled task is detected and recovered, its recovery will finish no later than $c_{max}^{selected}$ time units after its finish time in the canonical schedule. The same applies to the concurrent running scaled tasks on other processors regardless of faults occuring or not during their scaled execution. Recall that all remaining tasks in the global queue will be dispatched at f_{max} and take less time compared to the case in the canonical schedule. Therefore, all the remaining tasks will be dispatched at a time no more than $c_{max}^{selected}$ after their start times in the canonical schedule, which in turn ensures that they can complete before the deadline. Moreover, by recovering all faulty scaled tasks and dispatching remaining tasks at f_{max}, the reliability of all tasks will be preserved. The conclusions are summarized in the following theorem.

Theorem 2 *For a given set of feasible priority and frequency assignments to tasks, where the canonical schedule (assuming all tasks take their WCETs and no fault occurs) finishes* $c_{max}^{selected}$ *time units (the size of the largest selected task, which is also the size for recovery blocks) before the deadline, Algorithm 4 ensures that the reliability of all tasks will be preserved and all tasks (including their recoveries, if any) will finish in a timely manner before their respective deadlines.*

5.2 A Linear Search Heuristic for Task Selection

In last section, we assume that a set of *feasible* assignment for tasks' priorities and frequencies is given as the input for Algorithm 4. However, finding the appropriate priority and frequency assignment for the tasks to get the maximum energy savings is not trivial, especially for tasks with large variation on their WCETs. Note that, for the shared recovery G-RAPM scheme discussed in the above section, the size of the recovery blocks on all processors is determined by the largest task to be selected and managed. Therefore, for better energy savings, we may exclude a few large tasks from management to reduce the size of the recovery blocks, which in turn leaves more slack on all processors for DVFS to scale down the selected tasks.

Moreover, for those unselected large tasks, if we schedule them together with selected tasks on all processors, the total spare system capacity will be distributed across all processors and the resulting amount of slack on each processor may not be enough to schedule a common size recovery block. In this paper, for simplicity, we will dedicate a few processors to handle these unselected large tasks at the maximum frequency f_{max}. The selected tasks will be mapped to the remaining processors following a given heuristic (such as LTF and worst-fit) and the common size recovery blocks are reserved accordingly. Then, the uniform scaled frequency can be determined.

The outline for the linear search heuristic is as follows. Initially, we assume that all tasks will be managed. If the available static slack is enough for a common size recovery block on each processor, the remaining slack will be used to determine the scaled frequency and the amount of energy savings will be calculated. Otherwise, if the system load is high and the static slack is not enough to schedule the common size recovery blocks, all tasks will be tentatively assigned the frequency f_{max}. Then, in each round, the largest task will be excluded from the consideration, which enables us to gradually reduce the size of the required recovery blocks and more energy. Note that, as more large tasks are excluded, more dedicated processors will be needed to process them. The remaining processors will be used to execute the managed tasks. Through this procedure, we can find out the number of large tasks that should be excluded as well as the scaled frequency and execution order (i.e., priority) of the managed tasks for the best energy savings.

5.3 Online Adaptive G-RAPM with Shared Recovery

Although the shared recovery based G-RAPM scheme discussed above can preserve system reliability while guaranteeing tasks' timing constraints, it has also some *conservativism* since it dispatches the remaining tasks at the maximum frequency f_{max} once a transient fault has been detected at the end of a scaled execution. Such conservatism becomes a more severe problem at low system loads, where *early* faulty executions due to the lower frequency setting for selected tasks lead to more tasks being executed at f_{max} and less energy savings. The simulation results in Section 6 underline this problem. Therefore, to have more tasks be executed at scaled frequency for more energy savings while ensuring reliability preservation and timing constraints of tasks, online adaptive schemes with shared recovery warrant some investigation.

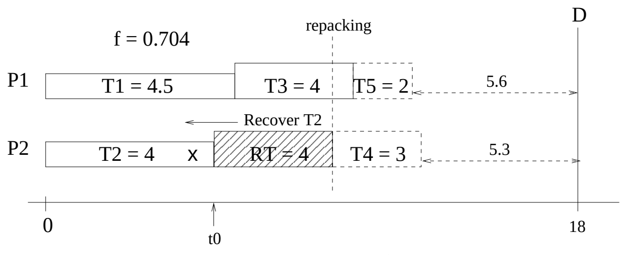
In the online shared recovery RAPM scheme for uniprocessor systems [45], the size of the shared recovery block and scaled frequency of remaining tasks will be adjusted after a faulty scaled execution is recovered or tasks complete early and more dynamic slack is generated. Such adaptation has been shown to be very effective for more energy savings and reliability enhancement. Following the same direction, we can also adaptively adjust the frequency setting for remaining tasks at runtime instead of following the static conservative decision to execute them at f_{max} after a fault occurs.

Once again, the adaptation for online shared recovery based G-RAPM scheme for multiprocessor systems involves coordination among the processors. Note that, under the shared recovery based G-RAPM scheme, once a faulty scaled execution of a selected task occurs, the system will run in the *recovery mode* to ensure that the faulty execution is recovered. Moreover, it should also ensure that the scaled executions of the concurrently running tasks on other processors complete successfully or are recovered in case of run-time transient faults. After all scaled executions of the concurrently running tasks (including their recoveries in case of transient faults) have completed, instead of letting all processors execute the newly-dispatched tasks at f_{max}, we can perform an online *re-packing process* for the current and remaining tasks to find the appropriate new recovery block size and scaled frequency. For cases where the available slack in the system after re-packing is not enough for a common size recovery block on each processor, we can repeat such re-packing process whenever a task completes its execution early and more dynamic slack is generated.

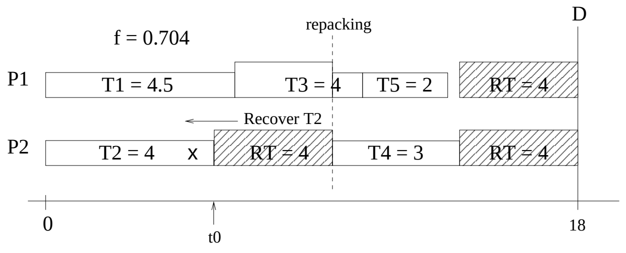
Figure 9 illustrates how the online adaptive shared recovery G-RAPM scheme works with the previous example. Here, tasks are assumed to take their WCETs. After the faulty scaled execution of task T_2 is recovered on the second processor (the first processor executes task T_3 at f_{max} at that time following the contingency schedule), tasks T_4 and T_5 will be re-mapped to the processors. With the assumption that tasks will take their WCETs, there will be 5.6 and 5.3 units of slack available on these two processors, respectively, as shown in Figure 9(a).

Considering both the currently running task T_3 and the remaining tasks T_4 and T_5, the size of the shared recovery blocks reserved on each processor should be 4 (although only part of task T_3 is scaled down, a full recovery is needed to preserve its original reliability). Then, the remaining slack can be used to get the scaled frequency for tasks T_3, T_4 and T_5 as shown in Figure 9(b). Therefore, instead of executing task T_4 at f_{max}, the second processor can execute it at the scaled frequency with the newly reserved shared recovery block to ensure its reliability preservation.

Algorithm 5 shows the outline of the online adaptive shared recovery G-RAPM scheme. Here, a global flag is used to indicate the current running state of the system: *normal scaled execution* (with $RunMode = normal$) or *contingency full speed execution* (with $RunMode = contingency$). In addition, each processor has its local running state indicator: *scaled execution*, *recovery execution* or *full speed execution*. Such states of processors are used to help determine when the re-packing process should be performed after a faulty



(a) Re-pack the remaining tasks T_4 and T_5 after T_2's recovery completes



(b) Scale down the remaining tasks after new recovery blocks are reserved

Fig. 9 Illustration of the online adaptive shared recovery G-RAPM scheme with the above example.

execution has been detected. According to the initial frequency assignment, the system's running states are first initialized at the very beginning (lines 2 to 6).

When a task T_i completes its execution on processor $PROC_x$, it will be recovered if its execution has been scaled down and a fault is detected. The system's running states are set to *contingency mode* accordingly (lines 8 to 11). Otherwise, if task T_i completes its execution successfully, we first check to see if the new scaled frequency should be calculated (i.e., the system runs at the contingency mode and all processors have completed the recovery execution). If yes, the new scaled frequency will be calculated through the re-pack process (line 14). The new frequency will be set appropriately depending on whether there is enough slack for a new scaled frequency or not (lines 15 to 24). In case the recovery process has not completed on other processors (line 25), the system still needs to run at contingency mode and processor $PROC_x$ will pick up the next task and execute it at the maximum frequency f_{max} (lines 26 and 27). Finally, if the system runs at the *normal* model, the processor $PROC_x$ will fetch and execute the next task after reclaiming the dynamic slack (if any) using the slack-sharing technique (lines 29 to 32).

Note that the adaptation on calculating the new scaled frequency through the re-pack process will not extend the worst-case schedule (including recovery blocks) beyond the deadline. Moreover, the online dynamic slack reclamation will not extend the finish time tasks as well. Therefore, under Algorithm 5, all tasks' reliabilities will be preserved since any scaled execution of tasks will be protected by a recovery block and all tasks will complete their executions (including recovery) before the deadline.

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. The *no power management (NPM)* scheme,

Algorithm 5 : Outline of the online adaptive shared recovery based G-RAPM scheme

```
1: Input: Feasible priority and frequency assignments for tasks;
2: if (Initial frequency assignment $f < f_{max}$) then
3:     Set $RunMode = normal$ and $Flag_j = scaled$; // case of sufficient slack
4: else
5:     Set $RunMode = contingency$ and $Flag_j = FullSpeed$;
6: end if
7: Event: when task $T_i$ completes on processor $PROC_x$:
8: if ($T_i$ run at scaled frequency $f_i < f_{max}$) AND (execution fails) then
9:     Set the global flag: $RunMode = contingency$;
10:    Set the local flag for $PROC_x$: $Flag_x = recovery$;
11:    Re-invoke task $T_i$ for re-execution at the maximum frequency $f_{max}$;
12: else if ($T_i$ completes its execution successfully) then
13:    if ($RunMode == contingency$) AND (all other processors have $Flag_i == FullSpeed$) then
14:        Re-pack the remaining tasks to processors to get new scaled frequency $f$;
15:        if ($f < f_{max}$) //slack is enough for a new scaled frequency then
16:            Set $f$ as the frequency for the remaining tasks;
17:            Notify all other processors the new scaled frequency $f$;
18:            For all processors, set $Flag_i = scaled$;
19:            Set $RunMode = normal$;
20:            Fetch the header task $T_k$ from Ready-Q and execute it at $f_k$;
21:        else
22:            Set $Flag_x = FullSpeed$; //no enough slack, run at $f_{max}$;
23:            Fetch the header task $T_k$ from Ready-Q and execute it at $f_{max}$;
24:        end if
25:    else if ($RunMode == contingency$) AND (exist $Flag_y == recovery||scaled$) then
26:        Set $Flag_x = FullSpeed$; //recovery process is not done yet
27:        Fetch the header task $T_k$ from Ready-Q and execute it at $f_{max}$;
28:    else if ($RunMode == normal$) then
29:        Set $Flag_x = scaled$; //normal scaled execution;
30:        Fetch the header task $T_k$ from Ready-Q;
31:        Reclaim dynamic slack using the slack-sharing technique and re-calculate $f_k$;
32:        Execute task $T_k$ at the scaled frequency $f_k$;
33:    end if
34: end if
```

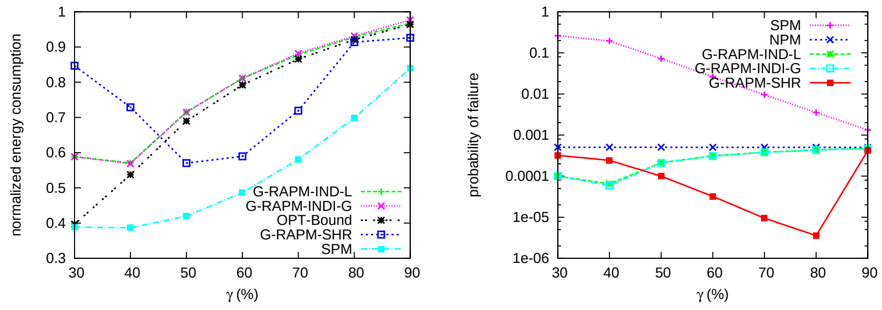
which executes all tasks at f_{max} and puts processors to power savings sleep states when idle, is used as the baseline and normalized energy consumption will be reported. Note that, as discussed in Section 3.1, the static power P_s will always be consumed for all schemes, which is set as $P_s = 0.01 \cdot k$ (k is the number of processors). 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 lowest energy efficient frequency can be found as $f_{low} = f_{ee} = 0.37$ (see Section 3.1).

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 [23,56]. 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 lowest energy efficient speed f_{low} (and corresponding supply voltage). The effects of different values of d have been evaluated in our previous work [46,47,51].

We consider systems with 4, 8 and 16 processors. The number of real-time tasks in each synthetic task set varies from 40 to 800, which results in roughly 10 to 50 tasks per processor. 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- to 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 to the 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. In what follows, we focus on the results for systems with 4 and 16 processors. Similar results for Simulations with 8 processors have yielded similar results.

6.1 Static G-RAPM Schemes

First, assuming that all tasks take their WCETs, we evaluate the performance of the static G-RAPM schemes. The first two are individual recovery based G-RAPM schemes with local and global task selection, which are denoted by *G-RAPM-IND-L* and *G-RAPM-IND-G* in the figures, respectively. The shared recovery based G-RAPM scheme is denoted as *G-RAPM-SHR*. For comparison, the *ordinary static power management (SPM)*, which uniformly scales down the execution of all tasks based on the schedule length, is also considered. Moreover, by assuming that there exists a subset of managed tasks with aggregated workload exactly equal to X_{opt} (see Section 4), the fault-free energy consumption of the task set is calculated, which provides an upper-bound on energy savings for any optimal static individual recovery based G-RAPM solution. That scheme is denoted as *OPT-Bound*.



(a) Normalized Energy Consumption (b) Probability of Failure

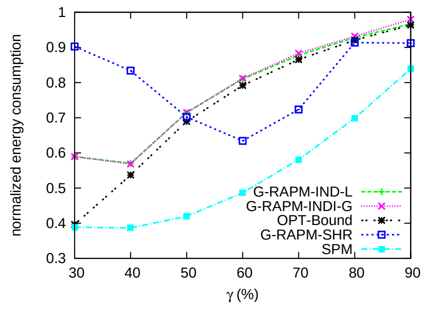
Fig. 10 Static G-RAPM schemes for systems with 4 processors.

Figure 10(a) first shows the normalized energy consumption of the static G-RAPM schemes under different system loads (which is represented in the X-axis) for a system with 4 processors. Here, each task set contains 40 tasks. From the figure we can see that, for individual recovery based G-RAPM schemes, local and global task selection perform roughly the same in terms of energy savings. This is because, in most cases, the managed workload of the selected tasks under both schemes shows little difference. As the system load increases, less static slack is available and in general more energy will be consumed (with less energy savings). For moderate to high system loads, the normalized energy consumption under the static individual recovery based G-RAPM schemes is very close (within 3%) to that of *OPT-Bound*, which is in line with our previous results for uniprocessor systems [47,48]. However, when the system load is low (e.g., $\gamma = 30\%$), almost all tasks will be managed under both individual recovery based G-RAPM schemes and run at a scaled frequency close to $f_{low} = 0.37$, which may incur higher probability failure and thus more energy is consumed by the recovery tasks. Therefore, the normalized energy consumption for both *G-RAPM-IND-L* and *G-RAPM-IND-G* increases. Compared to that of *OPT-Bound* that does not include energy consumption of recovery tasks, the difference becomes large when $\gamma = 30\%$. When compared to that of the ordinary (but reliability-ignorant) static power management (SPM) scheme, from 15% to 30% more energy will be consumed by the individual recovery based G-RAPM schemes.

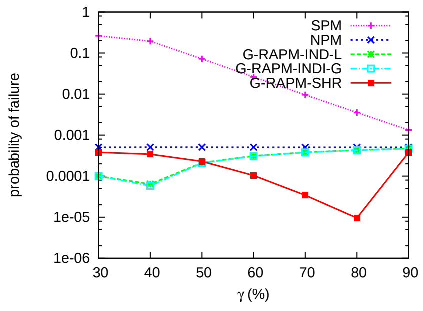
For the shared recovery based G-RAPM scheme, the energy savings under different system loads exhibit interesting patterns. When system load is high (e.g., $\gamma = 80\%$ or $\gamma = 90\%$) where the amount of static slack is limited, there is no much opportunity for energy management and shared recovery based G-RAPM scheme performs very close to that of the individual recovery based G-RAPM schemes. As system load becomes lower (e.g.,$\gamma = 50\%$ or $\gamma = 60\%$), since the amount of static slack reserved for recovery blocks is limited by the largest managed task, more static slack will be available for DVFS to scale down the execution of managed tasks at lower frequency, which results in better energy savings compared to that of the individual recovery based G-RAPM scheme. However, at very low system load (e.g., $\gamma = 30\%$), the scaled frequency for managed tasks will be close to f_{low} and the probability of having transient faults during the execution of any task becomes high. Recall that, whenever a scaled task incurs a fault, the static shared recovery based G-RAPM scheme will switch to a contingency schedule where all remaining tasks in the current frame will be executed at f_{max} and consume more energy. Therefore, the higher probability of incurring a fault during the scaled execution of first few tasks at low system loads results in much less energy savings for the static shared recovery based G-RAPM scheme.

Figure 10(b) further shows the *probability of failure*, which is the ratio of the number of failed tasks (by taking the recovery tasks into consideration) to the total number of executed tasks. We can see that all the static G-RAPM schemes can preserve system reliability (by having lower probability of failure during the execution of the tasks) when compared to that of NPM. In contrast, although the ordinary (but reliability-ignorant) 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.

For systems with 16 processors where each task set has 160 tasks, similar results have been obtained and are shown in Figure 11.



(a) Normalized Energy Consumption



(b) Probability of Failure

Fig. 11 Static G-RAPM schemes for systems with 16 processors.

6.2 Dynamic G-RAPM Schemes

In this section, we evaluate the performance of the online G-RAPM schemes with dynamic slack reclamation. Here, *G-RAPM-IND-L+DYN* represents the case of applying dynamic slack reclamation on top of the static schedule generated by the individual recovery based G-RAPM scheme with local task selection. Similarly, *G-RAPM-IND-G+DYN* stands for individual recovery based G-RAPM with global task selection and *G-RAPM-SHR+DYN* for online adaptive shared recovery based G-RAPM scheme. Again, for comparison, the ordinary *dynamic power management (DPM)* on top of the static schedule from SPM is also included. To obtain different amount of dynamic slack, we vary α from 30% to 90%.

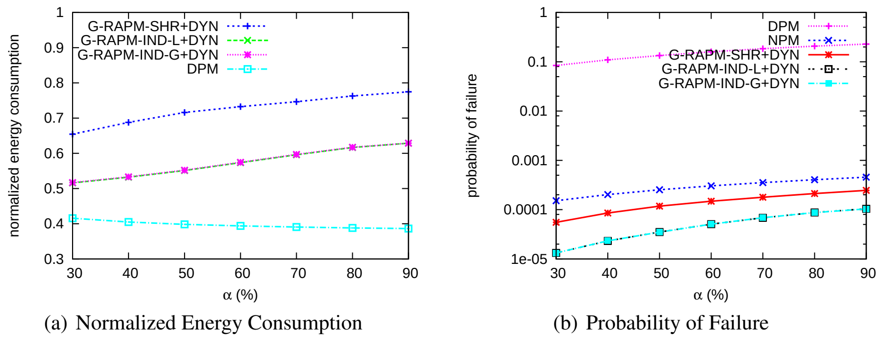


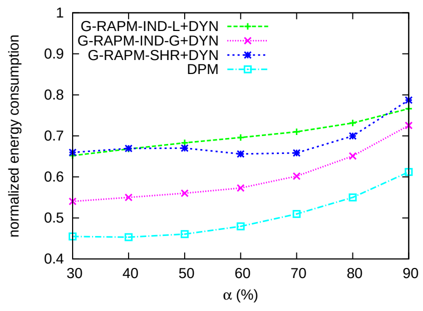
Fig. 12 Dynamic G-RAPM Schemes for a system with 4 processors and system load $\gamma = 40\%$.

At low system load $\gamma = 40\%$, Figure 12(a) first shows the normalized energy consumption of the dynamic G-RAPM schemes for a system with 4 processors. Again, there are 40 tasks in each task set. We can see that, for the individual recovery based G-RAPM schemes, applying dynamic slack reclamation on top of the static schedules will achieve almost the same energy savings. The reason is that, when $\gamma = 40\%$, there is around 60% static slack available and the optimal workload to manage for individual recovery based G-RAPM schemes is 36%. That is, almost all tasks will be managed statically and run at $f = 0.42$ under both individual recovery based G-RAPM schemes, which leave little space (with the limitation of $f_{low} = 0.37$) for further energy savings at runtime. When α decreases, more dynamic slack will be available and more tasks can be scaled to the lowest energy efficient frequency f_{low} for slightly more energy savings. Compared to that of *DPM*, from 10% to 22% more energy is consumed under the individual recovery based G-RAPM schemes.

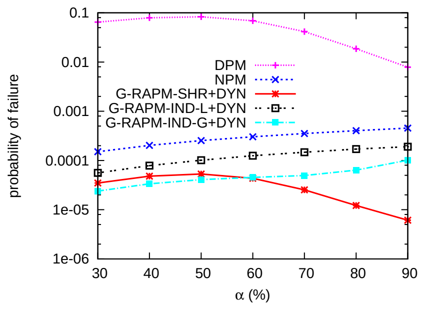
Surprisingly, even with online adaptation, the online adaptive shared recovery based G-RAPM scheme performs consistently worse and consumes around 18% more energy compared to that of individual recovery based G-RAPM schemes. This is due to the required synchronous handling of faults (which occur quite frequently under low system loads) and the frequent contingency execution of tasks at the maximum frequency.

Not surprisingly, Figure 12(b) shows that system reliability can be preserved under all the online dynamic G-RAPM schemes. Although the ordinary DPM can obtain more energy savings, it can lead to increased probability of failure by three orders of magnitude.

Figure 13(a) further shows the results for the same system at a higher system load $\gamma = 80\%$. Here, we can see that, for individual recovery based G-RAPM schemes, ap-



(a) Normalized Energy Consumption

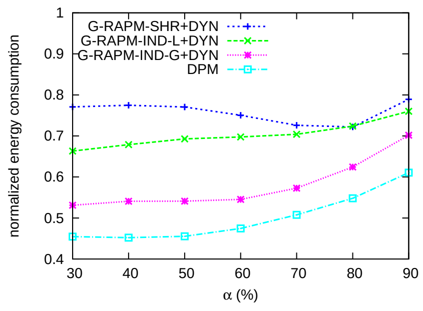


(b) Probability of Failure

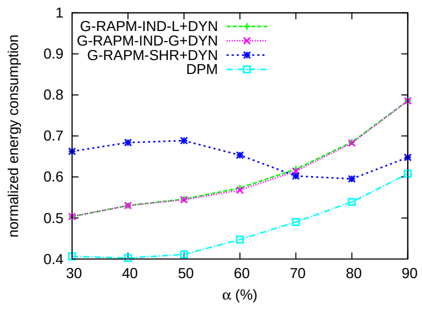
Fig. 13 Dynamic G-RAPM Schemes for a system with 4 processors and system load $\gamma = 80\%$.

plying dynamic slack reclamation to the static schedule of global task selection can lead to more energy savings (up to 13%) compared to that of local task selection. The main reason is that, at high system load $\gamma = 80\%$, very few tasks can be managed statically. By intentionally scheduling these managed tasks at the front of the schedule, *G-RAPM-IND-G+DYN* provides more opportunities for remaining tasks to reclaim the dynamic slack and yields more energy savings at runtime. Moreover, by managing more tasks at run time, *G-RAPM-IND-G+DYN* also has better system reliability as more tasks will have recovery tasks when compared to that of the scheme with local task selection (shown in Figure 13(b)).

Moreover, at high system load, tasks are executed at higher frequencies with small probability of failure, which in turn requires fewer synchronous handling of faulty tasks under the online adaptive shared recovery based G-RAPM scheme. Therefore, the performance difference between the online shared recovery and individual recovery based G-RAPM schemes becomes less.



(a) 16-processor and 160 tasks per task set



(b) 4-processor and 200 tasks per task set

Fig. 14 Dynamic G-RAPM Schemes with system load $\gamma = 80\%$.

Figure 14(a) further shows the normalized energy consumption for a system with 16 processors and each task set having 160 tasks at system load $\gamma = 80\%$. The results are quite similar to that of the 4-processor system as shown in Figure 13(a). Interestingly, as the number of tasks in each task set increases, Figure 14(b) shows that the performance difference between the two individual recovery based G-RAPM schemes diminishes. The reason is that, when more tasks are available, the managed tasks are more likely to be scattered in the static schedule under local task selection, which leads to similar opportunity for dynamic slack reclamation as that of global task selection and thus similar overall energy savings. Moreover, as the amount of slack reserved for recovery blocks under the shared recovery based G-RAPM scheme becomes relatively small when there are more tasks, we can see that the online adaptive shared recovery G-RAPM scheme performs better when no much dynamic slack is available (i.e., $\alpha = 90\%$). However, as α becomes smaller and more dynamic slack is available, the resulting low frequency from dynamic slack reclamation will lead to more frequent contingency execution of tasks at f_{max} for the online adaptive shared recovery G-RAPM scheme and thus more energy consumption.

7 Conclusions

In this paper, for independent real-time tasks that share a common deadline, we studied global-scheduling-based reliability-aware power management (G-RAPM) schemes for multiprocessor systems. We consider both *individual-recovery* and *shared-recovery* based G-RAPM schemes. For the individual-recovery based G-RAPM problem, 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*. For the shared recovery based G-RAPM problem, a simple G-RAPM scheme with common size recovery blocks on each processor as well as an online adaptive scheme are investigated. Moreover, we extend our previous work on dynamic power management with *slack sharing* to the reliability-aware settings.

Simulation results confirm that, all the proposed G-RAPM schemes can preserve system reliability while achieving significant energy savings in multiprocessor real-time systems. For individual-recovery based static G-RAPM schemes, the energy savings are within 3% of a theoretically computed ideal upper-bound for most cases. Moreover, by assigning higher priorities to scaled tasks with recoveries, the global task selection heuristic provides better opportunities for dynamic slack reclamation at runtime compared to that of the local task selection. For the shared-recovery based G-RAPM scheme, it performs best at the modest system loads. However, due to the requirements of synchronous handling of faulty tasks among the processors and the contingency execution of tasks at the maximum frequency, the online adaptive shared recovery G-RAPM generally save less energy compared to its counterpart with dynamic slack reclamation.

References

1. AlEnawy, T.A., Aydin, H.: Energy-aware task allocation for rate monotonic scheduling. In: RTAS '05: Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium, pp. 213–223 (2005)
2. Anderson, J.H., Baruah, S.K.: Energy-efficient synthesis of periodic task systems upon identical multiprocessor platforms. In: ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04), pp. 428–435 (2004)
3. Aydin, H., Devadas, V., Zhu, D.: System-level energy management for periodic real-time tasks. In: Proc. of The 27th IEEE Real-Time Systems Symposium (2006)
4. Aydin, H., Melhem, R., Mossé, D., Mejia-Alvarez, P.: Dynamic and aggressive scheduling techniques for power-aware real-time systems. In: Proc. of The 22th IEEE Real-Time Systems Symposium (2001)

5. Aydin, H., Melhem, R., Mossé, D., Mejia-Alvarez, P.: Power-aware scheduling for periodic real-time tasks. IEEE Trans. on Computers **53**(5), 584–600 (2004)
6. Aydin, H., Yang, Q.: Energy-aware partitioning for multiprocessor real-time systems. In: Proc. of the 17^{th} International Parallel and Distributed Processing Symposiu m (IPDPS), Workshop on Parallel and Distributed Real-Time Systems (WPDRTS) (2003)
7. Burd, T.D., Brodersen, R.W.: Energy efficient cmos microprocessor design. In: Proc. of The HICSS Conference (1995)
8. Chen, J.J.: Multiprocessor energy-efficient scheduling for real-time tasks with different power characteristics. In: ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing, pp. 13–20 (2005)
9. Chen, J.J., Hsu, H.R., Chuang, K.H., Yang, C.L., Pang, A.C., Kuo, T.W.: Multiprocessor energy-efficient scheduling with task migration considerations. In: ECRTS '04: Proceedings of the 16th Euromicro Conference on Real-Time Systems, pp. 101–108 (2004)
10. Chen, J.J., Hsu, H.R., Kuo, T.W.: Leakage-aware energy-efficient scheduling of real-time tasks in multiprocessor systems. In: RTAS '06: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 408–417 (2006)
11. Cho, S., Melhem, R.G.: On the interplay of parallelization, program performance, and energy consumption. IEEE Transactions on Parallel and Distributed Systems **21**(3), 342–353 (2010)
12. Corp., I.: Mobile pentium iii processor-m datasheet. Order Number: 298340-002 (2001)
13. Dabiri, F., Amini, N., Rofouei, M., Sarrafzadeh, M.: Reliability-aware optimization for dvs-enabled real-time embedded systems. In: Proc. of the 9th int'l symposium on Quality Electronic Design, pp. 780–783 (2008)
14. Dabiri, F., Amini, N., Rofouei, M., Sarrafzadeh, M.: Reliability-aware optimization for dvs-enabled real-time embedded systems. In: Proc. of the 9th int'l symposium on Quality Electronic Design (ISQED), pp. 780–783 (2008)
15. Degalahal, V., Li, L., Narayanan, V., Kandemir, M., Irwin, M.J.: Soft errors issues in low-power caches. IEEE Trans. on Very Large Scale Integration (VLSI) Systems **13**(10), 1157–1166 (2005)
16. Dertouzos, M.L., Mok, A.K.: Multiprocessor on-line scheduling of hard-real-time tasks. IEEE Trans. On Software Engineering **15**(12), 1497–1505 (1989)
17. Dhall, S.K., Liu, C.L.: On a real-time scheduling problem. Operation Research **26**(1), 127–140 (1978)
18. Ejlali, A., Al-Hashimi, B.M., Eles, P.: A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems. In: Proc. of the 7th IEEE/ACM Int'l conference on Hardware/software codesign and system synthesis (CODES), pp. 193–202 (2009)
19. Ejlali, A., Schmitz, M.T., Al-Hashimi, B.M., Miremadi, S.G., Rosinger, P.: Energy efficient seu-tolerance in dvs-enabled real-time systems through information redundancy. In: Proc. of the Int'l Symposium on Low Power and Electronics and Design (ISLPED) (2005)
20. Elnozahy, E.M., Melhem, R., Mossé, D.: Energy-efficient duplex and tmr real-time systems. In: Proc. of The 23^{rd} IEEE Real-Time Systems Symposium (2002)
21. Ernst, D., Das, S., Lee, S., Blaauw, D., Austin, T., Mudge, T., Kim, N.S., Flautner, K.: Razor: circuit-level correction of timing errors for low-power operation. IEEE Micro **24**(6), 10–20 (2004)
22. Ernst, R., Ye, W.: Embedded program timing analysis based on path clustering and architecture classification. In: Proc. of The Int'l Conference on Computer-Aided Design, pp. 598–604 (1997)
23. Hazucha, P., Svensson, C.: Impact of cmos technology scaling on the atmospheric neutron soft error rate. IEEE Trans. on Nuclear Science **47**(6), 2586–2594 (2000)
24. http://public.itrs.net: International technology roadmap for semiconductors. (2008). S. R. Corporation.
25. Irani, S., Shukla, S., Gupta, R.: Algorithms for power savings. In: Proc. of The 14^{th} Symposium on Discrete Algorithms (2003)
26. Ishihara, T., Yauura, H.: Voltage scheduling problem for dynamically variable voltage processors. In: Proc. of The Int'l Symposium on Low Power Electronics and Design (1998)
27. Iyer, R.K., Rossetti, D.J., Hsueh, M.C.: Measurement and modeling of computer reliability as affected by system activity. ACM Trans. Comput. Syst. **4**(3), 214–237 (1986)
28. Izosimov, V., Pop, P., Eles, P., Peng, Z.: Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems. In: Proc. of the conference on Design, Automation and Test in Europe (DATE), pp. 864–869 (2005)
29. Jejurikar, R., Gupta, R.: 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), pp. 78–81 (2004)
30. Melhem, R., Mossé, D., Elnozahy, E.M.: The interplay of power management and fault recovery in real-time systems. IEEE Trans. on Computers **53**(2), 217–231 (2004)
31. Pillai, P., Shin, K.G.: Real-time dynamic voltage scaling for low-power embedded operating systems. In: Proc. of the eighteenth ACM symposium on Operating systems principles, pp. 89–102 (2001)

32. Pop, P., Poulsen, K., Izosimov, V., Eles, P.: 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 (CODES+ISSS), pp. 233–238 (2007)
33. Pop, P., Poulsen, K.H., Izosimov, V., Eles, P.: 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, pp. 233–238 (2007)
34. Pradhan, D.K.: Fault Tolerance Computing: Theory and Techniques. Prentice Hall (1986)
35. Saewong, S., Rajkumar, R.: Practical voltage scaling for fixed-priority rt-systems. In: Proc. of the 9^{th} IEEE Real-Time and Embedded Technology and Applications Symposium (2003)
36. Sridharan, R., Gupta, N., Mahapatra, R.: Feedback-controlled reliability-aware power management for real-time embedded systems. In: Proc. of the 45th annual Design Automation Conference (DAC), pp. 185–190 (2008)
37. Unsal, O.S., Koren, I., Krishna, C.M.: Towards energy-aware software-based fault tolerance in real-time systems. In: Proc. of The International Symposium on Low Power Electronics Design (ISLPED) (2002)
38. Weiser, M., Welch, B., Demers, A., Shenker, S.: Scheduling for reduced cpu energy. In: Proc. of The First USENIX Symposium on Operating Systems Design and Implementation (1994)
39. Yang, C.Y., Chen, J.J., Kuo, T.W.: An approximation algorithm for energy-efficient scheduling on a chip multiprocessor. In: DATE '05: Proceedings of the conference on Design, Automation and Test in Europe, pp. 468–473 (2005)
40. Yao, F., Demers, A., Shenker, S.: A scheduling model for reduced cpu energy. In: Proc. of The 36^{th} Symposium on Foundations of Computer Science (1995)
41. Zhang, Y., Chakrabarty, K.: Energy-aware adaptive checkpointing in embedded real-time systems. In: Proc. of the conference on Design, Automation and Test in Europe (2003)
42. Zhang, Y., Chakrabarty, K.: Task feasibility analysis and dynamic voltage scaling in fault-tolerant real-time embedded systems. In: Proc. of IEEE/ACM Design, Automation and Test in Europe Conference(DATE) (2004)
43. Zhang, Y., Chakrabarty, K., Swaminathan, V.: 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)
44. Zhao, B., Aydin, H., Zhu, D.: Reliability-aware dynamic voltage scaling for energy-constrained real-time embedded systems. In: Proc. of the IEEE International Conference on Computer Design (ICCD) (2008)
45. Zhao, B., Aydin, H., Zhu, D.: Enhanced reliability-aware power management through shared recovery technique. In: Proc. of the Int'l Conf. on Computer Aidded Design (ICCAD) (2009)
46. Zhu, D.: 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)
47. Zhu, D., Aydin, H.: Energy management for real-time embedded systems with reliability requirements. In: Proc. of the Int'l Conf. on Computer Aidded Design (2006)
48. Zhu, D., Aydin, H.: Reliability-aware energy management for periodic real-time tasks. In: Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) (2007)
49. Zhu, D., Aydin, H., Chen, J.J.: 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)
50. Zhu, D., Melhem, R., Childers, B.R.: 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)
51. Zhu, D., Melhem, R., Mossé, D.: The effects of energy management on reliability in real-time embedded systems. In: Proc. of the Int'l Conf. on Computer Aidded Design (2004)
52. Zhu, D., Melhem, R., Mossé, D., Elnozahy, E.: Analysis of an energy efficient optimistic tmr scheme. In: Proc. of the 10^{th} Int'l Conference on Parallel and Distributed Systems (2004)
53. Zhu, D., Mossé, D., Melhem, R.: Power aware scheduling for and/or graphs in real-time systems. IEEE Trans. on Parallel and Distributed Systems **15**(9), 849–864 (2004)
54. Zhu, D., Qi, X., Aydin, H.: Priority-monotonic energy management for real-time systems with reliability requirements. In: Proc. of the IEEE International Conference on Computer Design (ICCD) (2007)
55. Zhu, D., Qi, X., Aydin, H.: 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)
56. Ziegler, J.F.: Trends in electronic reliability: Effects of terrestrial cosmic rays. available at http://www.srim.org/SER/SERTrends.htm (2004)