Real-Time Syst (2011) 47: 109–142
DOI 10.1007/s11241-011-9117-x

Global scheduling based reliability-aware power management for multiprocessor real-time systems

Xuan Qi · Dakai Zhu · Hakan Aydin

Published online: 5 February 2011

Abstract *Reliability-aware power management (RAPM)* has been a recent research focus due to 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 studied 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 *and* 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 the timing constraints. 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 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

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.

X. Qi · D. Zhu (✉)
Dept. of Computer Science, University of Texas at San Antonio, One UTSA Circle, San Antonio, TX 78249, USA
e-mail: dzhu@cs.utsa.edu

H. Aydin
Dept. of Computer Science, George Mason University, 4400 University Drive, MS 4A5, Fairfax, VA 22030, USA

system reliability and timing constraints. For the static version of the problem, the shared-recovery based scheme is shown to provide better energy savings compared to the individual-recovery based scheme, in virtue of its ability to leave more slack for DVFS. Moreover, by reclaiming the dynamic slack generated at runtime, online G-RAPM schemes are shown to yield better energy savings.

Keywords Energy management · Dynamic voltage and 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 (ITRS 2008). 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 (Burd and Brodersen 1995). In essence, the DVFS technique scales down simultaneously the processing frequency and supply voltage to save energy (Yao et al. 1995).

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 timing constraints for various real-time task models by exploiting the available static and dynamic *slack* in a system (Aydin et al. 2004; Pillai and Shin 2001; Saewong and Rajkumar 2003; Zhu et al. 2003). However, recent studies show that DVFS has a direct and adverse effect on the rate of transient faults (especially for those induced by electromagnetic interference and cosmic ray radiations) (Degalahal et al. 2005; Ernst et al. 2004; Zhu et al. 2004). Therefore, for safety-critical real-time embedded systems (such as satellite and surveillance systems) where reliability is as important as energy efficiency, *reliability-cognizant* energy management has become an important objective.

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 repeated (Pradhan 1986). By adopting a backward error recovery approach while considering the negative effects of DVFS on transient faults, we have introduced a *reliability-aware power management (RAPM)* scheme (Zhu 2006). The central idea of the RAPM scheme is to exploit the available slack to schedule a recovery task at the dispatch time of a task before utilizing the remaining slack for DVFS to scale down the task's execution and save energy, thereby preserving the system reliability (Zhu 2006). Following this line of research, several RAPM schemes have been proposed for various task models, scheduling policies, and reliability requirements (Dabiri et al. 2008; Sridharan et al. 2008;

Zhao et al. 2009; Zhu and Aydin 2006, 2007; Zhu et al. 2007, 2008a, 2008b), 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 goal is transformed to the objective of tolerating a fixed number of transient faults through re-execution (Pop et al. 2007).

In contrast to the existing work, in this paper, for a set of independent frame-based real-time tasks that share a common deadline, we propose *global scheduling based RAPM (G-RAPM)* schemes to minimize energy consumption while preserving system reliability in *multiprocessor* real-time systems. We consider both *individual-recovery* and *shared-recovery* based schemes. In general, there are two major paradigms in multiprocessor real-time scheduling: the *partitioned* and *global* approaches (Dertouzos and Mok 1989; Dhall and Liu 1978). For partitioned scheduling, tasks are statically mapped to processors and a task can only run on the processor to which it is assigned. After mapping tasks to processors, applying the existing uniprocessor RAPM schemes on each processor can be straightforward. In contrast, in global scheduling, tasks can run on any processor and migrate between processors at run-time depending on tasks' dynamic behaviors, 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 focus on investigating global scheduling based RAPM schemes for multiprocessor real-time systems.

The main contributions of this paper can be summarized as follows. 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 the system slack is distributed and when a subset of tasks are selected 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 or unused recovery blocks, 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 the static problem, the shared-recovery based scheme is shown to provide additional savings compared to the individual-recovery based schemes, due to its potential to leave more slack more DVFS. By reclaiming the dynamic slack generated at runtime, dynamic G-RAPM schemes can further scale down the processing frequency of the selected tasks and manage more tasks, yielding 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 Sect. 4, for the individual-recovery based approach, 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 Sect. 7.

2 Related work

The DVFS technique has been studied extensively in recent past for single-processor real-time embedded systems (Aydin et al. 2001; Ishihara and Yasuura 1998; Pillai and Shin 2001). For multiprocessor systems, Aydin and Yang studied the problem of partitioning periodic real-time tasks among multiple processors to minimize the dynamic energy consumption (Aydin and Yang 2003). Their results show that, when the earliest deadline first (EDF) scheduling is adopted on each processor, balancing the workload among all processors evenly gives the optimal energy consumption and the general problem of minimizing energy consumption in partitioned multiprocessor real-time system is NP-hard in the strong sense even when the feasibility is guaranteed a priori (Aydin and Yang 2003). The work was extended to consider rate monotonic scheduling (RMS) in later work (AlEnawy and Aydin 2005). Anderson and Baruah investigated how to synthesize a multiprocessor real-time system with periodic tasks such that the energy consumption is minimized at runtime (Anderson and Baruah 2004). Chen et al. proposed a series of approximation algorithms to improve energy-efficiency of multiprocessor real-time systems, for both frame-based tasks and periodic tasks, with and without leakage power consideration (Chen et al. 2004, 2006; Chen 2005; Yang et al. 2005). In our previous work, we developed slack reclamation and slack sharing algorithms for energy-aware real-time multiprocessor systems under global scheduling (Zhu et al. 2003, 2004). More recently, Choi and Melhem studied the interplay between parallelism of an application, program performance, and energy consumption (Cho and Melhem 2010). For an application with a given ratio of serial and parallel portions and the number of processors, the authors derived optimal frequencies allocated to the serial and parallel regions of the 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/back-up model, Unsal et al. proposed an energy-aware software-based fault tolerance scheme. The scheme postpones the execution of back-up tasks as much as possible to minimize the overlap between primary and backup executions, in an attempt to reduce energy consumption (Unsal et al. 2002). For duplex systems (where two hardware platforms are used to run the same software concurrently for fault detection), Melhem et al. explored the optimal number of checkpoints, *uniformly* or *nonuniformly* distributed, to achieve the minimum energy consumption (Melhem et al. 2004). Elnozahy et al. proposed an *Optimistic-TMR* (OTMR) scheme to reduce the energy consumption for traditional TMR (Triple Modular Redundancy) systems (Elnozahy et al. 2002). In TMR, three hardware platforms are used to run the same software simultaneously to detect and mask faults by allowing one processing unit to start late at a lower processing frequency provided that it can catch up and finish the computation before the deadline in case there is an error caused by faults during the execution of the other two processing units. In Zhu et al. (2004), further explored the optimal frequency settings for OTMR and presented detailed comparisons among Duplex, TMR and OTMR for reliability and energy consumption figures. Combined with voltage scaling techniques, Zhang et al. proposed an adaptive checkpointing scheme to save energy consumption for serial applications while tolerating a fixed number of transient faults (Zhang and Chakrabarty 2003). The work was further extended to periodic real-time tasks in Zhang and Chakrabarty (2004). Izosimov et al. studied 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) (Izosimov et al. 2005). However, the existing research on co-management of energy and reliability either focused on tolerating a fixed number of faults (Izosimov et al. 2005; Elnozahy et al. 2002; Melhem et al. 2004) or assumed a constant arrival rate for transient faults (Zhang and Chakrabarty 2003; Zhang et al. 2003).

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 (Degalahal et al. 2005; Ernst et al. 2004; Zhu et al. 2004). 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 (Zhu 2006). 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 through DVFS, and thus to recuperate the reliability loss due to the energy management (Zhu 2006). The scheme was further extended to multiple tasks with a common deadline (Zhu and Aydin 2006), periodic real-time tasks (Zhu and Aydin 2007; Zhu et al. 2007), as well as models with different reliability requirements (Zhao et al. 2008, 2009; Zhu et al. 2008a, 2008b).

Ejlali et al. studied a number of schemes that combine the information about hardware resources and temporal redundancy to save energy and to preserve system reliability (Ejlali et al. 2005). By employing a feedback controller to track the overall miss ratio of tasks in soft real-time systems, Sridharan et al. (2008) proposed a reliability-aware energy management algorithm to minimize the system energy consumption while still preserving the overall system reliability. Pop et al. considered the problem of energy and reliability trade-offs for distributed heterogeneous embedded systems (Pop et al. 2007). 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 was 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. (2008) considered the problem of assigning frequency/voltage to tasks for energy minimization subject to reliability and timing constraints. More recently, Ejlali et al. studied a standby-sparing hardware redundancy technique for fault tolerance (Ejlali et al. 2009). Following the similar idea in OTMR (Elnozahy et al. 2002), 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 was shown to have better energy performance when compared to that of the backward recovery based approach (Ejlali et al. 2009).

In this work, we investigate the RAPM problem for a set of independent frame-based tasks that share a common deadline and run on a multiprocessor real-time system under global scheduling. 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. The schemes are evaluated through extensive simulations.

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 almost linear relation between processing frequency and supply voltage (Burd and Brodersen 1995), the *dynamic voltage and frequency scaling (DVFS)* technique reduces the supply voltage and processing frequency to reduce a system's dynamic power consumption (Weiser et al. 1994). To avoid ambiguity, in the remainder of this 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), it has been noted that power management schemes that focus on individual components may not be energy efficient at the system level (Aydin et al. 2006). Hence, several articles considered system-wide power management problems (Aydin et al. 2006; Irani et al. 2003; Jejurikar and Gupta 2004).

In our previous work, we proposed and employed a *system-level power model* for uniprocessor systems (Aydin et al. 2006; Zhu et al. 2004). Similar power models have been also adopted in other studies (Irani et al. 2003; Jejurikar and Gupta 2004). In this paper, we consider a shared-memory system with k identical processors, where each processor has a separate supply voltage that enables them to have different processing frequencies. Following the same principles as in (Aydin et al. 2006; Zhu et al. 2004), the power consumption of a system with k processors can be expressed as:

$$P(f_1, \dots, f_k) = P_s + \sum_{i=1}^{k} \hbar_i (P_{ind} + P_{d,i})$$

$$= P_s + \sum_{i=i}^{k} \hbar_i (P_{ind} + C_{ef} \cdot f_i^m) \tag{1}$$

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 ith processor executes a task, 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., $\hbar_i = 0$) (Intel Corp. 2001). The effective switching capacitance C_{ef} and the dynamic power exponent m (which is, in general, no smaller than 2) are system-dependent constants (Burd and Brodersen 1995). f_i is the frequency for the ith 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).

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. For uniprocessor systems, 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. Hence, an *energy-efficient frequency*, below which the system consumes more *total energy*, has been derived (Aydin et al. 2006; Jejurikar and Gupta 2004; Zhu et al. 2004). In our settings and power model, by putting idle processors to sleep states to save energy, we can derive the energy-efficient frequency expression for each processor as $f_{ee} = \sqrt[m]{\frac{P_{ind}}{C_{ef} \cdot (m-1)}}$.

We further assume that the processing frequency 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 (Ishihara and Yasuura 1998), or use the next higher discrete frequency to ensure the solution's feasibility. Moreover, we use normalized frequencies and assume 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 Aydin et al. (2004), Zhu et al. (2003).

3.2 Fault and recovery models

During the operation of a computing system, both *permanent* and *transient* faults may occur due to, for instance, the effects of hardware defects, electromagnetic interferences or cosmic ray radiations, and thus result in system *errors*. Unlike crash failures that result from permanent faults, a *soft error* caused by transient faults typically does not last for long and disappears when the computation is repeated. In this paper, we focus on transient faults, which have been shown to be dominant (Iyer et al. 1986). Note that, the rate of soft errors caused by transient faults has been assumed to follow the Poisson distribution (Zhang et al. 2003). For DVFS-enabled computing systems, considering the negative effect of DVFS on transient faults, the average rate of soft errors caused by transient faults at a scaled frequency f $(\leq f_{max})$ (and the corresponding supply voltage V) can be given as (Zhu et al. 2004):

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

where λ_0 is the average rate of soft errors at f_{max} (and V_{max}). That is, $g(f_{max}) = 1$. The rate of soft errors caused by transient faults generally increases at lower frequencies and supply voltages. Therefore, we have $g(f) > 1$ for $f < f_{max}$.

In particular, in this work, we consider the exponential rate model for soft errors caused by transient faults, where $g(f)$ is given by (Zhu and Aydin 2006):

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

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

Soft errors caused by transient faults are assumed to be detected by using *sanity* (or *consistency*) checks at the completion of a task's execution (Pradhan 1986). Once a soft error is detected, *backward recovery* technique is employed and a recovery task (in the form of re-execution) is dispatched for fault tolerance (Zhang et al. 2003; Zhu 2006). Note that, transient faults may also affect system tasks (such as scheduler), which can lead to a system failure that cannot be handled through re-execution. For such system tasks, we assume that their executions are performed on a separate special platform with hardware redundancy (such as triple modular redundancy) (Pradhan 1986). Therefore, with the focus on application tasks, 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 $\frac{c_i}{f_i}$ time units 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 will affect tasks' ready times and make the system slack available to different tasks in varying amounts.

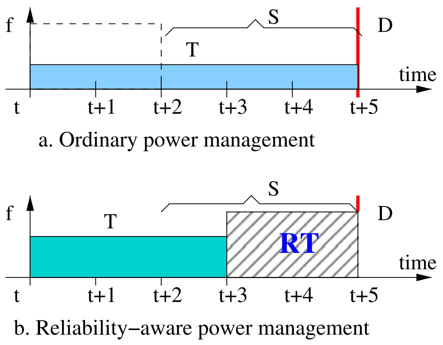

f
S
D
T
time
t
t+1
t+2
t+3
t+4
t+5
a. Ordinary power management
f
S
T
D
RT
time
t
t+1
t+2
t+3
t+4
t+5
b. Reliability−aware power management

Fig. 1 Ordinary and Reliability-Aware Power Management (Zhu 2006)

Reliability-Aware Power Management (RAPM) Before formally presenting our problem, 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 Fig. 1a, without special attention to the negative effects of DVFS on task reliability, the *ordinary* (and *reliability-ignorant*) power management scheme will use *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 (Zhu 2006).

Instead of using *all* the available slack for DVFS to scale down the execution of task T to save energy, as shown in Fig. 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 (Zhu 2006). The recovery task RT will be dispatched (at the maximum frequency f_{max}) only if errors caused by transient faults are 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 incurring errors during T's execution while RT is executed correctly. This overall reliability has been shown to be no worse than task T's *original* reliability (which is defined as the probability of having no error during T's execution when DVFS is not applied), regardless of the rate increases of soft errors at scaled processing frequencies (Zhu 2006).

Problem formulation From the above discussion, to address the negative effects of DVFS on reliability, 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 of 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 energy efficiency considerations, not all tasks may be selected for management. The binary variable x_i denotes whether T_i is *selected*, that is, whether its execution frequency is scaled down or not. Specifically, $x_i = 1$ indicates that task T_i is selected; otherwise (i.e. if T_i is not selected), $x_i = 0$. The tasks that are not selected run at the maximum processing frequency f_{max} to preserve their original 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 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 that the probability of incurring errors during a task's execution is rather small, we focus on the energy consumption for executing all *primary* tasks (i.e. do not account for the energy consumption of the recovery 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 that 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 frequencies of the selected tasks (i.e., f_i) to*:

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

subject to

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

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

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

Here, (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., (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, and develop both static and dynamic schemes.

4 G-RAPM with individual recovery tasks

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 that 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 the dynamic slack.

4.1 Static individual-recovery based G-RAPM schemes

To start with, 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 (Zhu and Aydin 2006). 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 optimal bound on energy savings in Sect. 6.

4.1.1 Local task selection

It has been shown that the optimal priority assignment to minimize the schedule length of a set of real-time tasks on multiprocessor systems under global scheduling is NP-hard (Dertouzos and Mok 1989). 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 behaviors of tasks (Zhu et al. 2003). Therefore, to get an initial mapping of tasks to processors and determine the amount of available slack on each processor, we adopt the *longest-task-first (LTF)* heuristic as the *initial* priority assignment where tasks are dispatched to processors in the non-increasing order of their execution times. Then, the *worst-fit (WF)* heuristic that assigns the next high-priority task to the processor with lowest load (maximum available time) is used. If the task set is schedulable under this *LTF-WF* heuristic, the *initial canonical schedule*, in which all tasks are assumed to use their WCETs and run at f_{max}, can be generated and the amount of available slack on each processor can be determined accordingly.

The existing RAPM solutions for uniprocessor systems (Zhu and Aydin 2006) can then be applied on each processor (with its assigned tasks) to determine the *final canonical schedule and frequency assignments*, from which the *final task priority assignments* can be obtained. In particular, the execution order in the final canonical schedule will provide the priority assignment we are looking for. Once the priorities of tasks are determined, they are assumed to be inserted into a global queue following their priority order. At run-time, whenever a processor P_{next} becomes idle, the next highest-priority task is dispatched on P_{next}. Note that, the actual mapping of

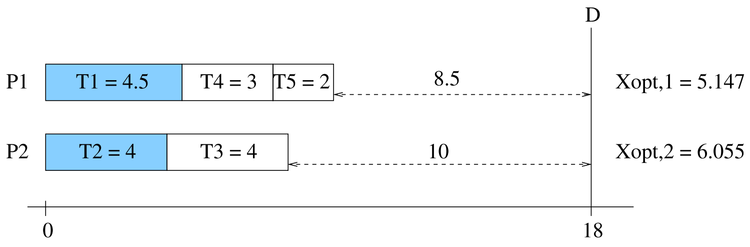


Fig. 2 Initial canonical schedule and the local task selection; shaded tasks are selected

tasks to processors may be different from that in the canonical schedule at runtime since global scheduling is assumed, which depends on the *actual execution times* of individual tasks (Zhu et al. 2003).

To illustrate these steps, we consider a concrete 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 to be executed on a 2-processor system with the common deadline of 18. The numbers in the parentheses are the WCETs of tasks. With the LTF-WF heuristic, the initial canonical schedule is shown in Fig. 2. Here, three tasks (T_1, T_4 and T_5) are mapped on processor P_1 that has 8.5 units of slack. The other two tasks (T_2 and T_3) are mapped on the second processor P_2 that has 10 units of slack.

In Zhu and Aydin (2006), 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:

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

That is, if there exists a subset of tasks such that their total workload is *exactly* X_{opt}, such a subset of tasks would definitely be the optimal selection to maximize energy savings. Unfortunately, finding such a subset of tasks has shown to be NP-hard (Zhu and Aydin 2006, 2007). The subset of selected tasks can be determined by using different heuristics; for example, those that consider the size of tasks (e.g. smallest-task-first or largest-task-first) (Zhu and Aydin 2007). Alternatively, importance of tasks as suggested by the user/designer could be considered as a guideline.

In the example above, assume that $P_{ind} = 0.1$, $C_{ef} = 1$ and $m = 3$ (Zhu and Aydin 2006). Then we obtain $X_{opt,1} = 5.147$ and $X_{opt,2} = 6.055$ for the first and second processors, respectively. If the largest task is selected first (Zhu and Aydin 2006, 2007), we can see that tasks T_1 and T_2 will be selected for management on the 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 Fig. 3, where each processor finishes the execution of its assigned tasks just in time.

Note that, due to the uneven slack allocation among the tasks, 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 (in the initial canonical schedule). If we blindly follow the priority assignment in

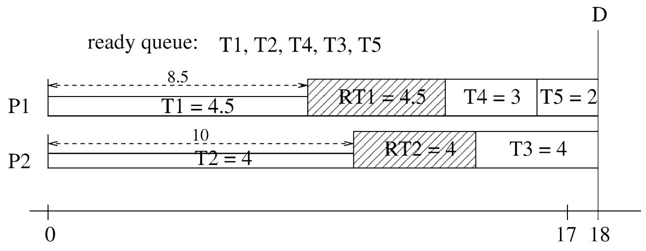


Fig. 3 Final canonical schedule for the local task selection

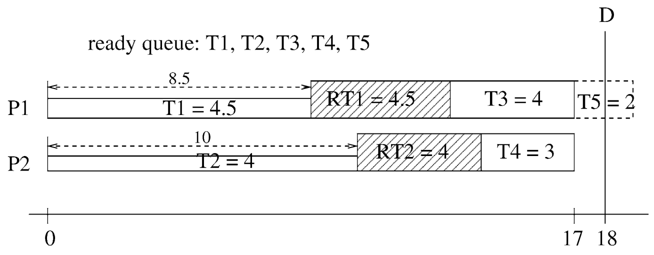


Fig. 4 Task T_5 misses the deadline with the original task order (priority assignment)

the initial canonical schedule where all tasks run at f_{max}, deadline violations can occur after scaling down certain tasks in different proportions. For instance, with the original LTF order of tasks and the calculated scaled frequencies for selected tasks, Fig. 4 shows that, after the first four tasks are dispatched and all tasks (including recovery tasks, if any) are assumed to take their WCETs, there is not enough time on any of the processors and task T_5 will miss its deadline.

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 canonical 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 Fig. 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 initial canonical schedule with any given 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 of solving the RAPM problem and selecting tasks on each processor can be done in $O(n)$ time. The last step of getting the final priorities of tasks through the reverse dispatching process 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 initial canonical schedule from a 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 Zhu and Aydin (2006) for tasks mapped to that processor (i.e., determine its available slack $S_i$; calculate $X_{opt}$; select the managed tasks; calculate the scaled frequency for selected tasks and obtain the final canonical schedule);
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 earlier start time should have a higher priority.
```

4.1.2 Global task selection

In the G-RAPM scheme with local task selection, after obtaining the amount of available slack and the optimal workload desired to be managed for each processor, it is not always possible to find a subset of tasks that have the *exact* optimal workload. Such deviation of the managed workload from the optimal one can accumulate across processors and thus affect 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 is $2 \cdot 18 = 36$. Therefore, the total amount of available slack in the system 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 one 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 $X_{opt} = 11.2$. By following the longest-task-first heuristic, three tasks (T_1, T_2 and T_3) 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. This is because, scheduling the augmented task set with recovery tasks and the scaled execution of the managed tasks may require perfect load balancing among the processors, which may not be feasible. Therefore, the scaled frequencies for the selected tasks can be different, which will be determined after tasks are mapped to individual processors. Note that, when the scaled execution of a selected task completes successfully, its recovery task can be dropped and its unused CPU time becomes dynamic slack. Therefore, to provide better opportunities for dynamic slack reclamation at runtime, we should first map the selected tasks as well as their recovery tasks to processors, followed by the unselected tasks. The resulting canonical schedule for the above example with global task selection is shown in Fig. 5.

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 canonical schedule is shown in Fig. 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)

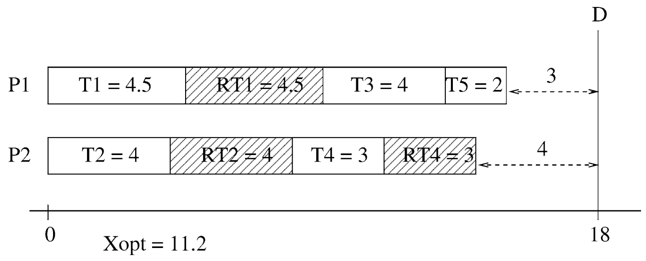


Fig. 5 Initial canonical schedule with global task selection

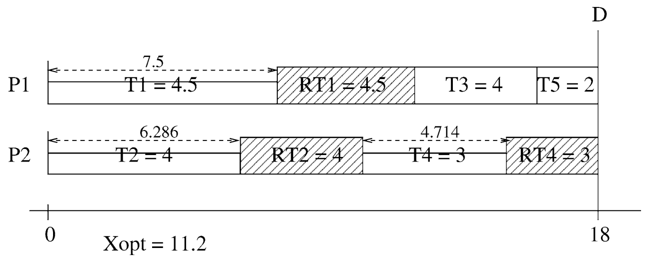


Fig. 6 Final canonical schedule with global task selection

should be obtained through the reverse dispatching process as discussed earlier. For this example, considering the scaled execution of those 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 Sect. 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)* scheme where all tasks run at f_{max}. 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 generated from the unused recovery tasks at runtime, which is further evaluated and discussed in Sect. 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 its WCET $c_i > \frac{D}{2}$ should not be selected. Taking this point into consideration, the steps for the G-RAPM with global task selection are summarized in Algorithm 2. Similarly, the complexity of Algorithm 2 can also be found as $O(max(n \cdot \log(n), k \cdot n))$.

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 (Ernst and Ye 1997). Moreover, the recovery tasks are executed only if their corresponding scaled primary tasks fail, which occurs with a small probability. Therefore,

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 using the LTF heuristic);
6:  Map unselected tasks to processors (e.g., by using the LTF heuristic);
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.
```

significant amount of dynamic slack can be expected at runtime from early completion of tasks and unused recovery tasks. This, in turn, provides abundant opportunities to further scale down the execution of selected tasks for additional energy savings or to manage more tasks to enhance system reliability. However, as shown in our previous work, simple greedy dynamic slack reclamation under global scheduling for multiprocessor systems may lead to timing anomalies. Consequently, this may cause deadline misses and special care is needed to reclaim the dynamic slack safely at runtime (Zhu et al. 2003).

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 (Zhu et al. 2003). The basic idea of slack sharing is to mimic the timing of the canonical schedule (which is assumed to be feasible) at runtime to ensure that all tasks can finish in time. That is, when a task completes 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 earlier 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 slack sharing technique can also be applied on top of the final canonical schedule generated from the static individual-recovery based G-RAPM schemes (with either local or global task selection) at runtime. Different from the work in Zhu et al. (2003) 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 that exploits the slack sharing technique. The input for the algorithm is the global *Ready-Q*, that includes all tasks in their priority order, with information about the statically-determined frequencies. Similar to the algorithms in Zhu et al. (2003), 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

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 ($Ready\text{-}Q \neq \emptyset$) do
5:    Find processor $PROC_y$ with the minimum $EFT_y$;
6:    if ($EFT_x > EFT_y$) /* $PROC_y$ is supposed to finish some task earlier; */ then
7:       Switch $EFT_x$ and $EFT_y$; /* $PROC_x$ shares its slack with $PROC_y$; */
8:    end if
9:    $T_i$ = Next task in $Ready\text{-}Q$;
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 sufficient 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, not sufficient 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
```

completion time of the task (including its recovery task) in the static canonical schedule. All *EFT* values are initially zero for all processors and f_i denotes the scaled frequency of task T_i.

At the beginning or after completing the execution of its current task, the processor $PROC_x$ first shares part of its slack (if any) with another processor that is supposed to complete its current task earlier (lines 5 to 7). Then, the next task T_i in the global *Ready-Q* is considered. If T_i is a scaled task, the expected finish time for processor $PROC_x$ will be moved 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 is limited by f_{low} to ensure energy efficiency. Otherwise, if T_i is not selected offline and does not have its recovery task scheduled yet, depending on the amount of available dynamic slack, task T_i can either be scaled down after scheduling its recovery task (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 and its recovery, if needed (lines 26 and 27).

Note that, with proper slack sharing at runtime, the 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 that all tasks can complete their executions in time (Zhu et al. 2003). Following a similar 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 final canonical schedule generated offline.*

Finally, under the (offline/online) individual-recovery based G-RAPM schemes, any task whose execution is scaled down will have a recovery task, giving the following corollary to conclude this section.

Corollary 1 *Under both static and dynamic individual-recovery based G-RAPM schemes, the original system reliability is preserved.*

5 G-RAPM with shared recovery tasks

To overcome the conservatism of the RAPM schemes with individual recovery tasks, a *shared-recovery* based RAPM scheme has been studied for uniprocessor systems in recent work (Zhao et al. 2009). 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 reserves 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 the previous work (Zhao et al. 2009).

However, extending the shared recovery technique to multiprocessor RAPM schemes introduces non-trivial issues considering the potentially *simultaneous failures* during the *concurrent* scaled execution of selected tasks on different processors. In what follows, we first discuss a simple shared-recovery based G-RAPM scheme and prove its feasibility in terms of meeting the tasks' deadlines while preserving system reliability. Then, a 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 presented at the end of this section.

5.1 Using a uniform-size shared recovery on each processor

In the shared recovery RAPM scheme for uniprocessor systems (Zhao et al. 2009), some of the static slack is first reserved as a recovery block, which can be shared by

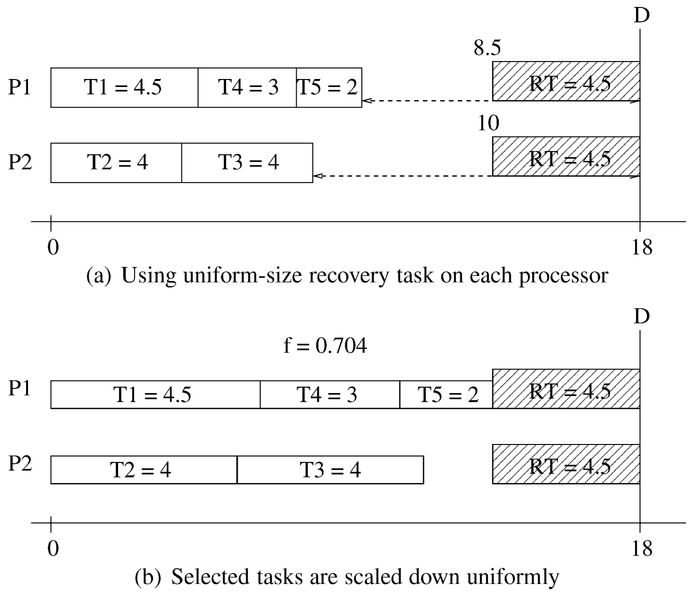


Fig. 7 The canonical schedule under the shared-recovery based G-RAPM scheme

all selected tasks and has the same size as the largest selected task. 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 error occurs. However, when a faulty scaled execution of a selected task is observed and recovered using the shared recovery block, the execution of the remaining tasks will switch to a *contingency schedule*, in which all remaining tasks run at the maximum frequency to preserve system reliability and to guarantee the timing constraints.

However, on multiprocessor systems, multiple scaled tasks may run concurrently. Obviously, having only one shared recovery task cannot guarantee the preservation of the original reliability, as more than a single task may incur soft errors concurrently. Intuitively, each processor in the system needs to have a recovery block that can be shared by the selected tasks that are mapped to that 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 from the one in the canonical schedule at runtime under global scheduling, especially considering the dynamic behaviors (varying actual execution times) of real-time tasks. Moreover, to ensure timing constraints, *coordination* is needed among the processors in global scheduling on the recovery steps once errors are detected.

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 task*. 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 processors, 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 is used to scaled down the selected tasks *uniformly* to the frequency $f = 0.704$, as shown in Fig. 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.

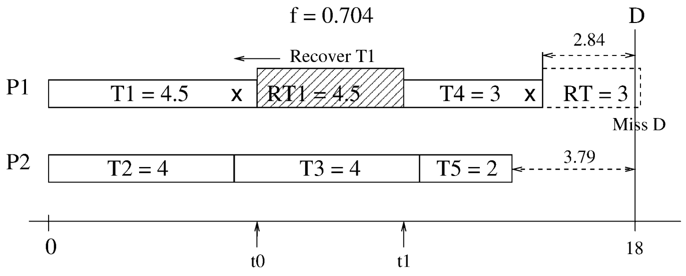
By having a shared recovery block on each processor, more slack will be available for DVFS. In the above example, with the same parameters for the power model as the ones given in Sect. 4.1.1, the energy savings under the shared recovery based G-RAPM scheme can be calculated as 41.3% (assuming no fault 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).

Another important feature of the shared-recovery based G-RAPM scheme is 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. The tasks concurrently running on other processors can continue their executions at the scaled frequency. Moreover, they can be recovered later in case errors occur during their scaled executions. 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, the remaining tasks that are still in the global ready queue, similar to the shared recovery RAPM scheme for uniprocessor systems, should be dispatched at the maximum frequency for reliability preservation.

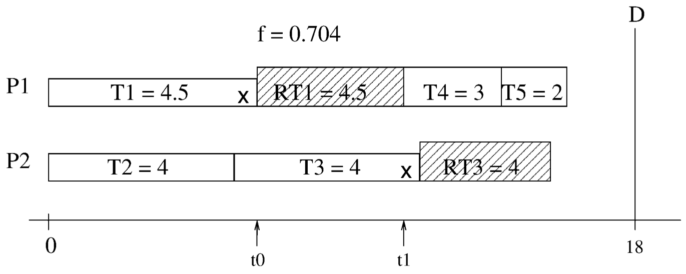
For the above example, suppose that task T_1 fails and is recovered on the first processor. If task T_4 is still dispatched at the scaled frequency and fails as shown in Fig. 8(a), we can see that it cannot be recovered on time, which in turn leads to system reliability degradation. On the other hand, when T_4 runs at f_{max}, the task T_3 that runs on the second processor concurrently with T_1 can continue its execution at the scaled frequency, and be still recovered on time in case of an error. 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 Fig. 8(b).

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

The input for Algorithm 4 is assumed to be a set of feasible global priority and frequency assignment for tasks. That is, if tasks are dispatched from the global queue following their priorities at their assigned frequencies, the 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 error, from Zhu et al. (2003), we know that under global scheduling all tasks will be dispatched no later than their start times in the canonical schedule.



(a) Dispatching remaining tasks at scaled frequency after a fault results in a deadline miss



(b) Concurrent failures (of T_1 and T_3) are recovered in time

Fig. 8 Handling faults 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:  At task completion events:
3:  if (the completed task is executed at scaled frequency) AND (execution fails) then
4:      Set the global flag: $FaultOccurrence = true$;
5:      Re-invoke the faulty task for re-execution at the maximum frequency $f_{max}$;
6:  end if
7:  At task dispatch events:
8:  if ($FaultOccurrence == false$) then
9:      Dispatch the task $T_i$ at the head of $Ready\text{-}Q$ at its assigned frequency $f_i$;
10: else
11:     Dispatch the task $T_i$ at the head of $Ready\text{-}Q$ at the maximum frequency $f_{max}$;
12: end if
13: At every period (frame) boundary:
14: Reset the global flag: $FaultOccurrence = false$;
```

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 scaled tasks running concurrently on other processors regardless of whether errors occur or not during their executions. Recall that all remaining tasks in the global queue will be dispatched at the maximum frequency f_{max} and take no more time compared than the case in the canonical schedule. Therefore, all the remaining tasks will be dispatched at a time no later than $c_{max}^{selected}$ units 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. These 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 error 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 the last section, we assumed that a set of *feasible* global priority assignment for tasks' priorities and frequencies are 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 based 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, if we schedule the unselected large tasks along with the selected tasks on all processors, the total spare system capacity will be distributed across all processors and the resulting available slack on each processor may not be enough to schedule a uniform-size recovery block. In this paper, for simplicity, we dedicate a few processors to handle these unselected large tasks at the maximum frequency f_{max}. The selected tasks are mapped to the remaining processors following a given heuristic (such as LTF and worst-fit) and the uniform-size recovery tasks are reserved accordingly. Then, the uniform scaled frequency can be determined.

The outline of the linear search heuristic can be given as follows. Initially, we assume that all tasks will be managed. If the available static slack is enough for a uniform-size recovery task 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 uniform-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 thus to leave more slack for DVFS. 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

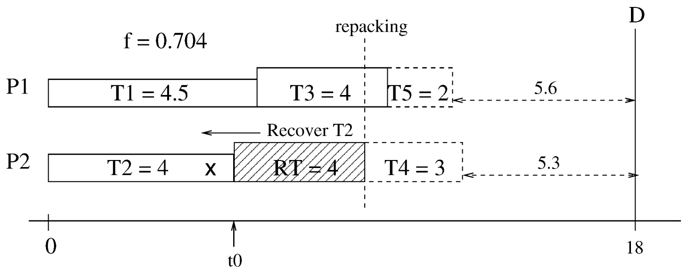
Although the shared-recovery based G-RAPM scheme can preserve the system reliability while guaranteeing tasks' timing constraints, it has also some pessimism since it dispatches all remaining tasks at the maximum frequency f_{max} once an error caused by transient faults has been detected at the end of a scaled execution. Such a drawback becomes a more severe problem at low system loads, where *early* fault detections can cause many follow-up tasks on all processors to run at f_{max} and hurt the energy savings. The simulation results in Sect. 6 underline this problem. Therefore, the online adaptive schemes with shared recovery warrant some investigation, in particular, with the goal of using low frequencies as long as possible without sacrificing the system's reliability.

In the online shared recovery RAPM scheme for uniprocessor systems (Zhao et al. 2009), the size of the shared recovery block and scaled frequency of remaining tasks will be adjusted after a faulty scaled execution has been 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 an error occurs.

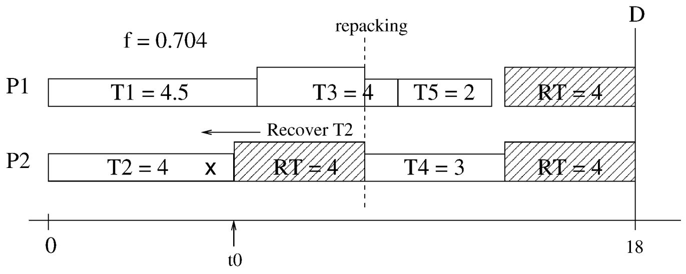
Once again, the adaptation for online shared-recovery based G-RAPM scheme for multiprocessor systems involves coordination among the processors. Note that, under the shared-recovery based G-RAPM scheme, once a faulty scaled execution of a selected task occurs, the system will run in the *recovery mode* 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 or are recovered successfully. After all scaled executions of the concurrently running tasks (including their recoveries in case of errors being detected) 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, where all of them are assumed to take their WCETs. Following the same heuristic (i.e., LTF-WF), such re-packing process can generate a new partial canonical schedule and the amount of slack on each processor can be found. For cases where the available slack in the system after re-packing is not enough for a uniform-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. Otherwise, based on the current and remaining tasks, a new recovery block with appropriate uniform size can be reserved on each processor and scaled frequency can be determined accordingly.

Figure 9 illustrates how the online adaptive shared-recovery based 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 are re-mapped. 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 Fig. 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



(a) Re-packing the tasks T_4 and T_5 after T_2's recovery



(b) Scaling 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

(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 Fig. 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 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 an error 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 so, the new scaled frequency will be calculated through the re-packing process (line 14). The new scaled frequency will be set appropriately depending on whether there is enough slack for a new shared recovery block 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

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

```
1:  Input: A 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: The 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 about the new scaled frequency $f$;
18:             For all processors, set $Flag_i = scaled$;
19:             Set $RunMode = normal$;
20:             Execute the task $T_k$ at the head of $Ready\text{-}Q$ at $f_k$;
21:         else
22:             Set $Flag_x = FullSpeed$; //not enough slack, run at $f_{max}$;
23:             Execute the task $T_k$ at the head of $Ready\text{-}Q$ at $f_{max}$;
24:         end if
25:     else if ($RunMode == contingency$) && ($\exists y\ Flag_y == recovery||scaled$) then
26:         Set $Flag_x = FullSpeed$; //recovery process is not done yet
27:         Execute the task $T_k$ at the head of $Ready\text{-}Q$ at $f_{max}$;
28:     else if ($RunMode == normal$) then
29:         Set $Flag_x = scaled$; //normal scaled execution;
30:         Consider the task $T_k$ at the head of $Ready\text{-}Q$;
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
```

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 in 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).

The adaptation for calculating the new scaled frequency through the re-packing process can be performed with the complexity of $O(x)$, where x is the number of remaining tasks. Although the adaptation is invoked only occasionally after soft errors are detected and recovered, for task sets with large number of tasks, repeatedly performing such re-packing process after the completion of each task can incur significant runtime overhead. To reduce such overhead, we may limit the minimum interval between consecutive invocations of the re-packing process to accumulate enough slack time.

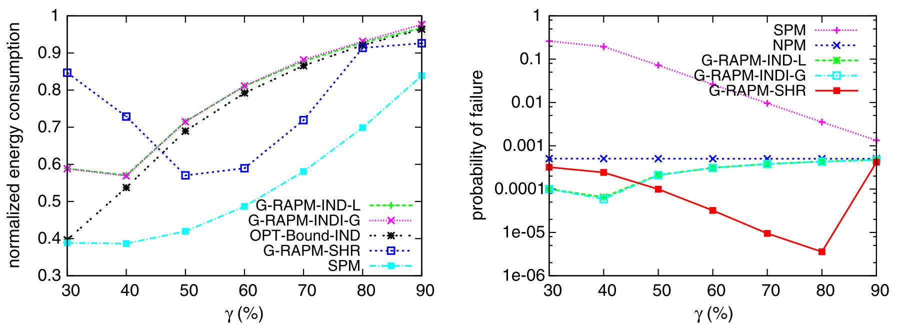
Moreover, the adaptation process does not extend the worst-case schedule (including recovery blocks) beyond the deadline. As the online dynamic slack reclamation will not extend the finish time of tasks as well, all tasks' reliabilities will be preserved since any scaled execution of tasks will be protected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

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, 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 are reported. Note that, as discussed in Sect. 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 $f_{low} = f_{ee} = 0.37$ (Sect. 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 Hazucha and Svensson (2000), Ziegler (2004). The exponent in the exponential fault model is assumed to be $d = 3$ (see (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 (Zhu 2006; Zhu and Aydin 2006; Zhu et al. 2004).

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 (period) of tasks. Each data 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. Simulations with 8 processors have yielded similar results.



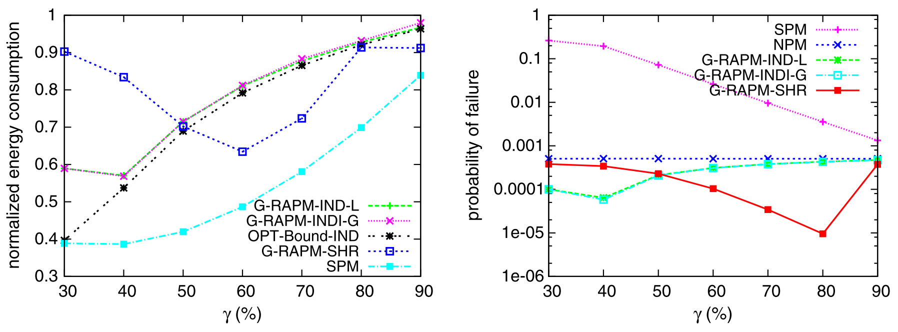
(a) Normalized energy consumption (b) Probability of failure

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

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 aggregate workload exactly equal to X_{opt} (see Sect. 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-IND*.

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. Smaller values of γ indicate more system slack. 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. 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-IND*, which is in line with our previous results for uniprocessor systems (Zhu and Aydin 2006, 2007). 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 of 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-IND* 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



(a) Normalized energy consumption (b) Probability of failure

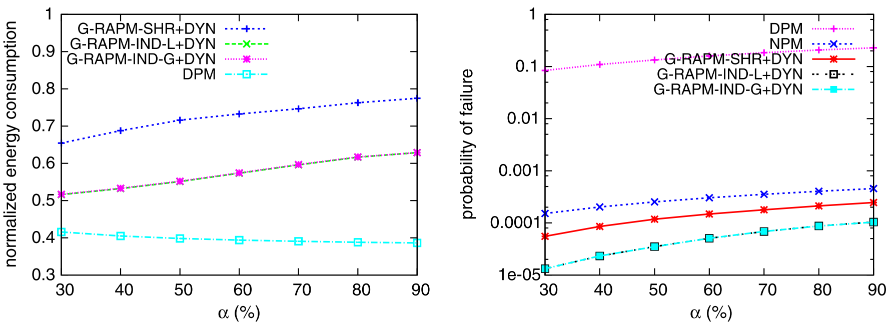
Fig. 11 Static G-RAPM schemes for systems with 16 processors

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., $\gamma = 80$–90%) where the amount of static slack is limited, there are not much opportunities for energy management and the 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., $\gamma = 50$–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 an error, 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 reported in Fig. 11.



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

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

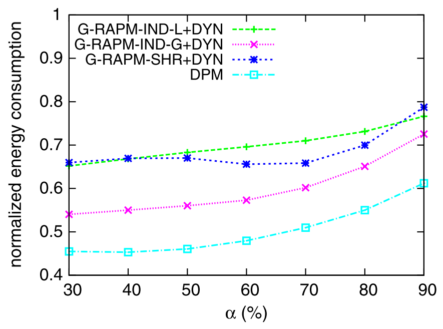
6.2 Dynamic G-RAPM schemes

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%.

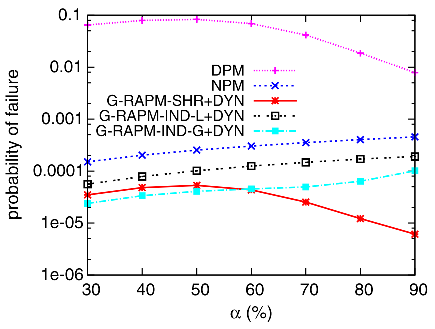
At low system load $\gamma = 40\%$, Fig. 12(a) first shows the normalized energy consumption of the dynamic G-RAPM schemes for a system with 4 processors. Again, there are 40 tasks in each task set. Note that, there is more dynamic slack for smaller values of α. We can see that, for the individual-recovery 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 higher 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, Fig. 12(b) shows that system reliability can be preserved under all the online dynamic G-RAPM schemes. Although the ordinary DPM can obtain



(a) Normalized Energy Consumption



(b) Probability of Failure

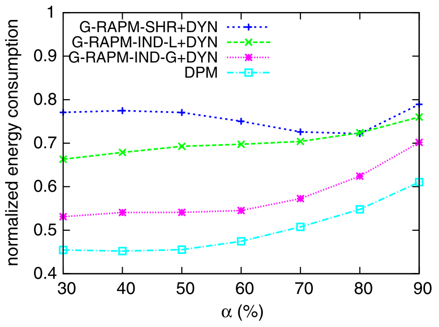
Fig. 13 Dynamic G-RAPM schemes for a system with 4 processors and system load $\gamma = 80\%$

more energy savings, it can lead to increased probability of failure by three orders of magnitude.

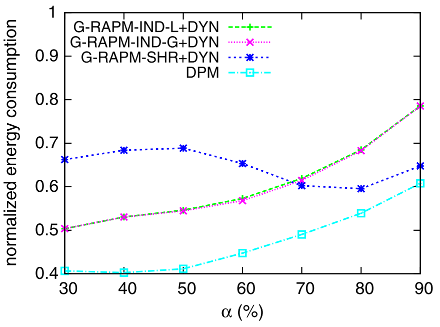
Figure 13(a) further shows the results for the same system at a higher system load $\gamma = 80\%$. Here, we can see that, for individual-recovery based G-RAPM schemes, applying dynamic slack reclamation to the static schedule of global task selection can lead to more energy savings (up to 13%) compared 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 Fig. 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 noticeable.

Figure 14(a) further shows the normalized energy consumption for a system with 16 processors at the system load $\gamma = 80\%$, when each task set has 160 tasks. The results are quite similar to that of the 4-processor system as shown in Fig. 13(a). Interestingly, as the number of tasks in each task set increases, Fig. 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 opportunities for dynamic slack reclamation as those of the 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 reclama-



(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 $\gamma = 80\%$

tion will lead to more frequent contingency execution of tasks at f_{max} for the online adaptive shared-recovery based 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 uniform-size recovery tasks 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 based G-RAPM generally saves less energy compared to its counterpart with dynamic slack reclamation.

References

AlEnawy TA, Aydin H (2005) 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

Anderson JH, Baruah SK (2004) 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

Aydin H, Devadas V, Zhu D (2006) System-level energy management for periodic real-time tasks. In: Proc of the 27th IEEE real-time systems symposium

Aydin H, Melhem R, Mossé D, Mejia-Alvarez P (2001) Dynamic and aggressive scheduling techniques for power-aware real-time systems. In: Proc of the 22th IEEE real-time systems symposium

Aydin H, Melhem R, Mossé D, Mejia-Alvarez P (2004) Power-aware scheduling for periodic real-time tasks. IEEE Trans Comput 53(5):584–600

Aydin H, Yang Q (2003) Energy-aware partitioning for multiprocessor real-time systems. In: Proc of the 17th international parallel and distributed processing symposium (IPDPS), Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)

Burd TD, Brodersen RW (1995) Energy efficient cmos microprocessor design. In: Proc of the HICSS conference

Chen JJ (2005) 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

Chen JJ, Hsu HR, Chuang KH, Yang CL, Pang AC, Kuo TW (2004) Multiprocessor energy-efficient scheduling with task migration considerations. In: ECRTS '04: Proceedings of the 16th euromicro conference on real-time systems, pp 101–108

Chen JJ, Hsu HR, Kuo TW (2006) 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

Cho S, Melhem RG (2010) On the interplay of parallelization, program performance, and energy consumption. IEEE Trans Parallel Distrib Syst 21(3):342–353

Intel Corp. (2001) Mobile pentium iii processor-m datasheet. Order Number: 298340-002

Dabiri F, Amini N, Rofouei M, Sarrafzadeh M (2008) Reliability-aware optimization for dvs-enabled real-time embedded systems. In: Proc of the 9th int symposium on quality electronic design (ISQED), pp 780–783

Degalahal V, Li L, Narayanan V, Kandemir M, Irwin MJ (2005) Soft errors issues in low-power caches. IEEE Trans Very Large Scale Integr 13(10):1157–1166

Dertouzos ML, Mok AK (1989) Multiprocessor on-line scheduling of hard-real-time tasks. IEEE Trans Softw Eng 15(12):1497–1505

Dhall SK, Liu CL (1978) On a real-time scheduling problem. Oper Res 26(1):127–140

Ejlali A, Al-Hashimi BM, Eles P (2009) A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems. In: Proc of the 7th IEEE/ACM int conference on hardware/software codesign and system synthesis (CODES), pp 193–202

Ejlali A, Schmitz MT, Al-Hashimi BM, Miremadi SG, Rosinger P (2005) Energy efficient seu-tolerance in dvs-enabled real-time systems through information redundancy. In: Proc of the int symposium on low power and electronics and design (ISLPED)

Elnozahy EM, Melhem R, Mossé D (2002) Energy-efficient duplex and tmr real-time systems. In: Proc of the 23rd IEEE real-time systems symposium

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

Ernst R, Ye W (1997) Embedded program timing analysis based on path clustering and architecture classification. In: Proc of the int conference on computer-aided design, pp 598–604

Hazucha P, Svensson C (2000) Impact of cmos technology scaling on the atmospheric neutron soft error rate. IEEE Trans Nucl Sci 47(6):2586–2594

http://public.itrs.net: International technology roadmap for semiconductors (2008). S. R. Corporation

Irani S, Shukla S, Gupta R (2003) Algorithms for power savings. In: Proc of the 14th symposium on discrete algorithms

Ishihara T, Yasuura H (1998) Voltage scheduling problem for dynamically variable voltage processors. In: Proc of the int symposium on low power electronics and design

Iyer RK, Rossetti DJ, Hsueh MC (1986) Measurement and modeling of computer reliability as affected by system activity. ACM Trans Comput Syst 4(3):214–237

Izosimov V, Pop P, Eles P, Peng Z (2005) 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

Jejurikar R, Gupta R (2004) Dynamic voltage scaling for system wide energy minimization in real-time embedded systems. In: Proc of the int symposium on low power electronics and design (ISLPED), pp 78–81

Melhem R, Mossé D, Elnozahy EM (2004) The interplay of power management and fault recovery in real-time systems. IEEE Trans Comput 53(2):217–231

Pillai P, Shin KG (2001) 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

Pop P, Poulsen K, Izosimov V, Eles P (2007) Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems. In: Proc of the 5th IEEE/ACM int conference on hardware/software codesign and system synthesis (CODES+ISSS), pp 233–238

Pradhan DK (1986) Fault tolerance computing: theory and techniques. Prentice Hall, New York

Saewong S, Rajkumar R (2003) Practical voltage scaling for fixed-priority rt-systems. In: Proc of the 9th IEEE real-time and embedded technology and applications symposium

Sridharan R, Gupta N, Mahapatra R (2008) Feedback-controlled reliability-aware power management for real-time embedded systems. In: Proc of the 45th annual design automation conference (DAC), pp 185–190

Unsal OS, Koren I, Krishna CM (2002) Towards energy-aware software-based fault tolerance in real-time systems. In: Proc of the international symposium on low power electronics design (ISLPED)

Weiser M, Welch B, Demers A, Shenker S (1994) Scheduling for reduced cpu energy. In: Proc of the first USENIX symposium on operating systems design and implementation

Yang CY, Chen JJ, Kuo TW (2005) 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

Yao F, Demers A, Shenker S (1995) A scheduling model for reduced cpu energy. In: Proc of the 36th symposium on foundations of computer science

Zhang Y, Chakrabarty K (2003) Energy-aware adaptive checkpointing in embedded real-time systems. In: Proc of the conference on design, automation and test in Europe

Zhang Y, Chakrabarty K (2004) 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)

Zhang Y, Chakrabarty K, Swaminathan V (2003) Energy-aware fault tolerance in fixed-priority real-time embedded systems. In: Proc of the 2003 IEEE/ACM int conference on computer-aided design

Zhao B, Aydin H, Zhu D (2008) Reliability-aware dynamic voltage scaling for energy-constrained real-time embedded systems. In: Proc of the IEEE international conference on computer design (ICCD)

Zhao B, Aydin H, Zhu D (2009) Enhanced reliability-aware power management through shared recovery technique. In: Proc of the int conf. on computer aidded design (ICCAD)

Zhu D (2006) 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)

Zhu D, Aydin H (2006) Energy management for real-time embedded systems with reliability requirements. In: Proc of the int conf. on computer aidded design

Zhu D, Aydin H (2007) Reliability-aware energy management for periodic real-time tasks. In: Proc of the IEEE real-time and embedded technology and applications symposium (RTAS)

Zhu D, Aydin H, Chen JJ (2008a) Optimistic reliability aware energy management for real-time tasks with probabilistic execution times. In: Proc of the 29th IEEE real-time systems symposium (RTSS)

Zhu D, Qi X, Aydin H (2008b) Energy management for periodic real-time tasks with variable assurance requirements. In: Proc of the IEEE int conference on embedded and real-time computing systems and applications (RTCSA)

Zhu D, Melhem R, Childers BR (2003) Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems. IEEE Trans Parallel Distrib Syst 14(7):686–700

Zhu D, Melhem R, Mossé D (2004) The effects of energy management on reliability in real-time embedded systems. In: Proc of the int conf. on computer aidded design

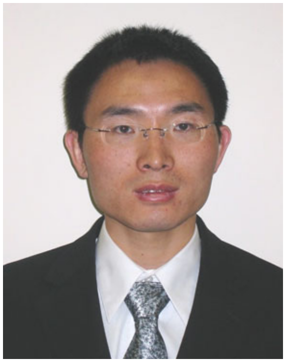
Zhu D, Melhem R, Mossé D, Elnozahy E (2004) Analysis of an energy efficient optimistic tmr scheme. In: Proc of the 10th int conference on parallel and distributed systems

Zhu D, Mossé D, Melhem R (2004) Power aware scheduling for and/or graphs in real-time systems. IEEE Trans Parallel Distrib Syst 15(9):849–864

Zhu D, Qi X, Aydin H (2007) Priority-monotonic energy management for real-time systems with reliability requirements. In: Proc of the IEEE international conference on computer design (ICCD)

Ziegler JF (2004) Trends in electronic reliability: Effects of terrestrial cosmic rays. Available at http://www.srim.org/SER/SERTrends.htm

Xuan Qi received the B.S. degree in computer science from Beijing University of Posts and Telecommunications in 2005. He is now a Ph.D. candidate in Computer Science Department, University of Texas at San Antonio. His research interests include real-time systems, parallel systems, and high performance computing. His current research focuses on energy-efficient scheduling algorithms for multi-processor/multi-core real-time systems with reliability requirements.



Dakai Zhu received the BE in Computer Science and Engineering from Xi'an Jiaotong University in 1996, the ME degree in Computer Science and Technology from Tsinghua University in 1999, the MS and Ph.D. degrees in Computer Science from University of Pittsburgh in 2001 and 2004, respectively. He joined the University of Texas at San Antonio as an assistant professor in 2005. His research interests include real-time systems, power aware computing and fault-tolerant systems. He has served on program committees (PCs) for several major IEEE and ACM-sponsored real-time conferences (e.g., RTAS and RTSS). He was a recipient of the US National Science Foundation (NSF) Faculty Early Career Development (CAREER) Award in 2010. He is a member of the IEEE and the IEEE Computer Society.

Hakan Aydin received the B.S. and M.S. degrees in Control and Computer Engineering from Istanbul Technical University in 1991 and 1994, respectively, and the Ph.D. degree in computer science from the University of Pittsburgh in 2001. He is currently an Associate Professor in the Computer Science Department at George Mason University, Fairfax, Virginia. He has served on the program committees of several conferences and workshops, including the IEEE Real-Time Systems Symposium and IEEE Real-time Technology and Applications Symposium. In addition, he served as the Technical Program Committee Chair of IEEE Real-time and Embedded Technology and Applications Symposium (RTAS'11). He was a recipient of the US National Science Foundation (NSF) Faculty Early Career Development (CAREER) Award in 2006. His research interests include real-time systems, low-power computing, and fault tolerance.