Reliability-Aware Power Management for Parallel Real-Time Applications with Precedence Constraints

Yifeng Guo, Dakai Zhu
Department of Computer Science
University of Texas at San Antonio
San Antonio, TX, 78249
{*yguo,dzhu*}*@cs.utsa.edu*

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

***Abstract*—The negative effects of the *Dynamic Voltage and Frequency Scaling (DVFS)* technique on the system reliability has recently promoted the research on *reliability-aware power management (RAPM)*. RAPM aims at reducing the system energy consumption while preserving the system's reliability. In this paper, we study the RAPM problem for parallel real-time applications for shared memory multiprocessor systems in the presence of precedence constraints. We show that this problem is NP-hard. Depending on how recoveries are scheduled and utilized by a subset of selected tasks, we investigate both *individual-recovery* and *shared-recovery* based RAPM heuristics. Online RAPM schemes that exploit dynamic slack generated at runtime are also considered. The proposed schemes are evaluated through extensive simulations. The results show that all schemes can preserve system reliability under all settings. For modest system loads, similar energy savings are obtained by all static schemes. However, when the system load is low, the shared-recovery based schemes need coordinated recovery operations on all processors and thus save less energy. Moreover, by reclaiming dynamic slack, the online schemes yield better energy savings.**

***Keywords*-DVFS; Transient Faults; Reliability; Multiprocessor Real-Time Systems; Dependent Tasks**

I. Introduction

Power management remains as one of the grand challenges for the research and engineering community, both in industry and academia [9]. In the last decade, several power management techniques have ben proposed. Among these, *Dynamic Voltage and Frequency Scaling (DVFS)* is a widely popular technique with applicability ranging from real-time embedded systems to server systems.

More recently, the research community has started to explore the interplay between power management and other operational objectives, such as reliability and dependability. Reliability, as a traditional system requirement, can be enhanced through various fault tolerance techniques by exploiting different forms of redundancy. *Modular redundancy* can be used to detect/mask permanent faults by executing an application on several processing units in parallel. *Temporal redundancy* can be exploited to re-execute a faulty application that incurs transient faults, thereby increasing the system reliability [17]. Note that, there is an interesting trade-off between system energy efficiency and reliability since both DVFS and backward recovery techniques actively use and compete for free CPU time (or *slack*) in the system. Although both power management [14], [23], [28] and fault tolerance [7], [17], [19] have been studied extensively, the *co-management of energy and reliability* has attracted the researchers' attention only recently.

A number of studies considered incorporating fault tolerance objectives to the DVFS settings for real-time embedded applications [6], [15], [22], [24]. However, none of the above work has considered the negative effects of DVFS on system reliability. In fact, it has been established that the probability of having transient faults increases drastically when systems are operated at lower supply voltages [8], [29]. Taking such negative effects into consideration, a number of *reliability-aware power management (RAPM)* schemes have been proposed for *single-processor systems* [20], [25], [26], [27]. For distributed heterogeneous embedded systems, Pop *et al.* studied a constrained logic programming-based (CLP) scheme to determine the voltage levels, start times, and message transmission times for real-time tasks to achieve a user-defined reliability goal while minimizing energy consumption [16]. Ejlali *et al.* studied a *standby-sparing* hardware redundancy technique to tolerate transient faults while saving energy [5]. In our recent work [18], we have studied global scheduling based RAPM schemes for independent real-time tasks running on multiprocessor systems.

In this paper, we explore the RAPM problem for parallel real-time applications with *precedence constraints* running on a shared-memory multiprocessor system (denoted as the *RAPM-PC* problem). The problem is similar to the one studied in [16], however, we propose schemes to find the energy-efficient mapping of tasks to processors, which is different from the existing solution that assumes a given task-to-processor mapping. This research effort can be considered

978-1-4577-1221-0/11/$26.00 ©2011 IEEE. This work was supported in part by NSF awards CNS-0855247, CNS-1016855, CNS-1016974, and NSF CAREER Awards CNS-0546244 and CNS-0953005.

as an extension of our recent work [18], which addressed RAPM schemes for *independent* tasks on a multiprocessor system. In many applications, real-time tasks are given by precedence constraints that define input-output relationships among them. Therefore, we believe the investigation of the RAPM-PC problem will prove very useful in practice.

The main contributions of this paper can be summarized as follows. We first show that the RAPM-PC problem is NP-hard. Then, depending on how recoveries are scheduled and utilized by a subset of selected tasks, we develop both *individual-recovery* and *shared-recovery* based heuristic schemes. The former strategy involves scheduling a separate recovery task for every selected task, while the latter enables multiple tasks running on the same processor to share a single recovery task. Based on the *slack-sharing* technique [28], online RAPM schemes that exploit dynamic slack generated at runtime are also proposed.

The simulation results show that all schemes can preserve system reliability under all settings. For modest system loads, the static schemes achieve similar energy savings. However, when the system load is low, the individual-recovery based scheme can save more energy since the shared-recovery based schemes need coordinated recovery operations across all processors. Moreover, by reclaiming dynamic slack, the online schemes can save more energy.

The remainder of this paper is organized as follows. Section II presents system models and formulates the RAPM-PC problem. The static heuristic schemes are explored in Section III and online schemes are discussed in Section IV. Section V presents the simulation results and Section VI concludes the paper.

II. System Models and Problem Formulation

In this section, we first present system models and state our assumptions. Then, after reviewing the basic RAPM concepts, we formulate the RAPM-PC problem considered in this paper.

A. Models and Assumptions

Application Model: In this work, we consider a parallel application consisting of n real-time tasks with precedence constraints, represented by a *directed acyclic graph (DAG)* $G = (V, E)$. The set of vertices (nodes) $V = \{T_1, ..., T_n\}$ represents the set of tasks. The set of edges $E = \{E_1, ..., E_m\}$ represents a partial order corresponding to the *precedence constraints* among tasks, with the interpretation that whenever the edge $(T_i, T_j) \in E$, the task T_j cannot start to execute until T_i has been completed [1], [13]. Hence, the *ready time* of a task is defined as the time instant at which all its predecessors have been completed.

The worst-case execution time (WCET) of task T_i under the maximum normalized processing frequency $f_{max} = 1$ is denoted by c_i $(1 \le i \le n)$. When T_i is executed at a lower frequency f_i, its execution time is assumed to be $\frac{c_i}{f_i}$ in the worst case. All tasks in the application need to complete by the deadline D, which is also the *period* (or *frame*) of the application. An example application with six tasks is shown in Figure 1, where the WCET of each task is labeled accordingly and the application deadline is assumed to be $D = 313$.

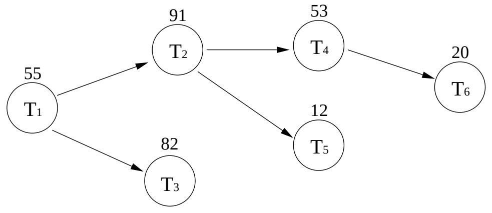


Figure 1. DAG for the example application.

The application will be executed on a shared memory multiprocessor system with k identical processors with negligible communication overhead. In multiprocessor settings, dependent tasks are typically scheduled by the *list scheduling* technique. It is known that finding the optimal execution order (or priority) of tasks to minimize the schedule length is NP-hard [4]. Therefore, when tasks have the same predecessors and become ready simultaneously, we use the *longest task first (LTF)* heuristic to determine their execution orders. For instance, in when both tasks T_2 and T_3 become ready after task T_1 finishes in the application shown in Figure 1, T_2 enters the ready queue first as it has larger WCET. Assuming all tasks take their WCETs, the *canonical execution* of the example application running on two processors under list scheduling with LTF can be found to finish at time 219, well before the application deadline 313. In this paper, we assume that an application's canonical execution under list scheduling with LTF can finish no later than its deadline.

Power Model: The processors in the system are assumed to have DVFS capability, which is a common feature in modern processors. We further assume that the processors can only operate at L different discrete frequency levels $\{f_1, f_2, \ldots, f_L\}$, where $f_1 = f_{min}$ and $f_L = f_{max} = 1.0$ are the minimum and maximum frequency, respectively. Moreover, the time overhead for frequency (and supply voltage) changes is assumed to be incorporated into the tasks' WCETs [28].

With increasing importance of leakage power and emphasis on the need for considering all system components in power management [2], [10], [12], we adopt a system-level power model for multiprocessor systems [18], where the power consumption of a system with k processors can be expressed as:

$$P(F_1, \ldots, F_k) = P_s + \sum_{i=1}^{k} \hbar_i (P_{ind} + C_{ef} \cdot F_i^m) \qquad (1)$$

Here, P_s stands for *static power*, which can be removed only by powering off the whole system. Due to the prohibitive overhead of turning off and on the system in periodic real-time execution settings, we assume that the system is in *on* state at all times and that P_s is always consumed. Hence, we will focus on the energy consumption related to active power, which is given by the second component of the expression above.

When the i^{th} processor executes a task and is *active*, $\hbar_i = 1$; otherwise, $\hbar_i = 0$. P_{ind} is the *frequency-independent active power*, which is the same for all processors. The *frequency-dependent active power* depends on each processor's frequency F_i, and system-dependent constants C_{ef} and m [3]. From this model, we can derive the *energy-efficient frequency* as $f_{ee} = \sqrt[m]{\frac{P_{ind}}{C_{ef} \cdot (m-1)}}$ [18], which is assumed to be smaller than f_{min}.

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 or cosmic ray radiations, which can result in system *errors*. Focusing on transient faults, which have been shown to be dominant [11] especially with scaled technology sizes [8], the average rate of soft errors caused by such faults is assumed to follow the Poisson distribution. Considering the negative effects of DVFS on transient faults, the soft error rate at a scaled frequency f $(< f_{max})$ (and the corresponding supply voltage V) can be modeled as [29]:

$$\lambda(f) = \lambda_0 \cdot 10^{\frac{d \cdot (1-f)}{1-f_{min}}} \qquad (2)$$

where λ_0 is the average error rate at f_{max} and d (> 0) is a constant, representing the sensitivity of soft errors caused by transient faults to DVFS.

It is assumed that soft errors are detected by *sanity* (or, *consistency*) checks at the end of a task's execution [17]. The overhead for fault detection is also assumed to be incorporated into tasks' WCETs. To tolerate soft errors, *backward recovery* technique is employed with recovery tasks, which are assumed to take the form of re-execution [26].

B. RAPM and Problem Formulation

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 and needs to finish by $t+5$. Here, there will be 3 units of available slack. Without special attention to the negative effects of DVFS on task reliability, the *ordinary* power management scheme will use *all* the available slack to scale down the execution of task T for the maximum energy savings as shown in Figure 2a. However, such an approach can lead to a degradation of several orders of magnitude in task's reliability figures [26].

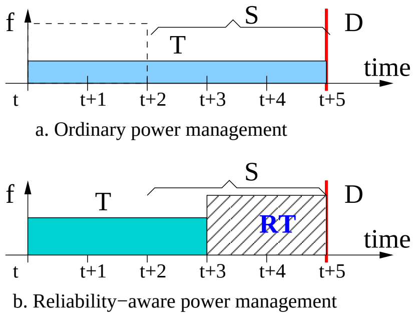

f
S
D
T
time
t
t+1
t+2
t+3
t+4
t+5
a. Ordinary power management
f
S
T
RT
D
time
t
t+1
t+2
t+3
t+4
t+5
b. Reliability−aware power management

Figure 2. RAPM: An example [26].

Instead, as shown in Figure 2b, the RAPM scheme reserves a portion of the slack to schedule a *recovery task* RT for task T before scaling down its execution using the remaining slack [26]. The recovery task RT will be dispatched (at the maximum frequency f_{max}) only if errors are detected when task T completes to recuperate the reliability loss due to DVFS. Here 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 scaled execution while RT is executed correctly, which has been shown to be no worse than task T's *original* reliability (which is defined as the probability of having no error when T is executed at the maximum frequency f_{max}) [26].

Problem Formulation: As in previous work on RAPM [26], [27], we aim to preserve the original reliability of each task even when applying DVFS. As a result, a recovery task needs to be scheduled before the scaled task's deadline to mitigate the negative effects of DVFS on reliability. After an error has been detected in a scaled task, its recovery task is invoked *sequentially* on the same processor and runs at f_{max}. Note that, due to the workload constraints, not all tasks may be selected for scaling down to save energy. We use a binary variable x_i to denote whether T_i is *selected* or not. The tasks that are not selected (with $x_i = 0$) will run at f_{max} to maintain their original reliability; while those selected (with $x_i = 1$) will be eligible to run at a frequency that is lower than f_{max}.

Suppose the processing frequency for task T_i is f_i. Its active energy consumption is:

$$E_i(f_i) = (P_{ind} + C_{ef} f_i^m) \cdot \frac{c_i}{f_i} \qquad (3)$$

Without considering the energy consumed by recovery tasks (which normally have a small probability of being invoked), we focus on the *primary* tasks and aim to minimize the *fault-free* energy consumption. Suppose that the *start time* and *completion time* of T_i (and its recovery task, if any) under list scheduling are st_i and ct_i, respectively. The *RAPM-PC* problem is defined as: **select an appropriate set of tasks (i.e., determine $\{x_i\}$ values) and their scaled frequencies (i.e., $\{f_i\}$) to**:

$$\text{minimize} \sum_{i=1}^{n} E_i(f_i) \tag{4}$$

subject to

$$f_i \in \{f_1, \ldots, f_{L-1}\}, (\text{if } x_i = 1) \tag{5}$$

$$f_i = f_L, (\text{if } x_i = 0) \tag{6}$$

$$st_i \geq ct_j, \forall T_j \in Pred(T_i)\ i = 1, \ldots, n \tag{7}$$

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

Here, the constraints (5) and (6) give the ranges for task frequencies. In Equation (7), $Pred(T_i)$ denotes the set of task T_i's predecessor tasks and the condition enforces the dependencies among tasks. Equation (8) guarantees that all tasks complete their executions before the deadline.

Intractability of RAPM-PC: Note that, when there is no dependency among tasks, the RAPM-PC problem will reduce to the RAPM problem for independent tasks, which has been shown to be NP-hard [18]. Therefore, finding the optimal solution for the RAPM-PC problem is intractable and we will focus on efficient heuristics in this paper. As in [18], we consider both *individual-recovery* (where each selected task has a separate recovery task) and *shared-recovery* (where several selected tasks mapped on the same processor share one recovery task) based approaches. However, our solutions consider the precedence constraints in both static and dynamic phases of the problem.

III. Static RAPM-PC Schemes

In the case of independent tasks, all available slack is to appear at the end of the schedule and can be utilized by all tasks [18] on a given processor. However, with dependent tasks and list scheduling, slack may be available in the middle of the schedule, which can only be reclaimed by a few tasks. Therefore, different from the solution for independent tasks that relies on only workload and task sizes, the solution for dependent tasks should also take tasks' dependence relation into consideration.

A. RAPM-PC with Individual Recovery

We first consider *individual-recovery* based approach. That is, to preserve system reliability, we aim to preserve the original reliability of all tasks by scheduling a *separate* recovery task for any task whose execution is to be scaled down. To guide our task selection process, we bear the intuition that it is typically more beneficial to obtain energy savings through DVFS with minimum increase in the schedule length (as increasing the schedule length would limit the scaling opportunities for other tasks in future iterations). This is also supported by the fact that, with dependent tasks, scaling down a task may not always affect the start time of its successor tasks and the schedule length. As a result, we propose to iteratively consider the *ratio of energy savings over schedule length increase* for each task before deciding which task to scale down in each step. The pseudo-code of our solution is given in Algorithm 1.

Algorithm 1 Static RAPM-PC with individual-recovery

```
1:  For task $T_i$ ($i = 1, \ldots, n$): $l_i = L$; $f_i = f_L$;//initialize
2:  $MoreTaskToScale = true$;
3:  while ($MoreTaskToScale$) do
4:      for (($\forall T_i \in DAG$)&&($l_i > 1$)) do
5:          $c_i' = \frac{c_i}{f_{l_i - 1}} + c_i$; //try to scale down $T_i$
6:          Get the tentative schedule $S'$ by list scheduling;
7:          if ($length(S') \leq D$) then
8:              $ES_i = E_i(f_i) - E_i(f_{l_i - 1})$;
9:              $SI_i = length(S') - length(S)$;
10:         else
11:             $ES_i = -1$;//cannot be scaled down
12:         end if
13:     end for
14:     if ($\exists T_i$ with $ES_i > 0$) then
15:         $MoreTaskToScale = true$;
16:         if ($\exists T_i$ with $SI_i = 0$) then
17:             Find $T_x$ with $ES_x = \max\{ES_i | SI_i = 0\}$;
18:         else
19:             Find $T_x$ with $R_x = \max\{R_i = \frac{ES_i}{SI_i}\}$;
20:         end if
21:         $l_x - -$; // Scale down $T_x$
22:         Get the new schedule $S$ with the new speed of $T_x$;
23:     else
24:         $MoreTaskToScale = false$;
25:     end if
26: end while
```

Initially, all tasks have the maximum frequency $f_{max} = f_L$ (line 1), which is assumed to be schedulable under list scheduling with LTF. Then, for each task, assuming its frequency is scaled down by one level, we can find the tentative new schedule under list scheduling. If the new schedule can still finish in time, the energy savings (ES_i) and schedule length increase (SI_i) are calculated (lines 4 to 13). Finally, the task with the maximum energy saving rate is chosen to actually slow down its execution by one level (lines 14 to 21). Note that, when a task's frequency is reduced by one frequency level, it is possible that the new schedule length does not change (this is because there may be idle intervals in the schedule before the next task starts to run after the completion of its predecessors). For such tasks, the one with maximum energy savings is eventually chosen. The above steps are repeated until no task can be further slowed down one more level while preserving feasibility.

Note that, with n tasks and L frequency levels, the maximum number of iterations for the outer while-loop will be $n \cdot L$. Moreover, to get the largest ratio of energy savings over schedule length increase within each iteration, the inner for-loop has the complexity of $\mathbf{O}(n^2 \cdot k)$. Therefore, the overall complexity of Algorithm 1 can be found to be $\mathbf{O}(L \cdot n^3 \cdot k)$.

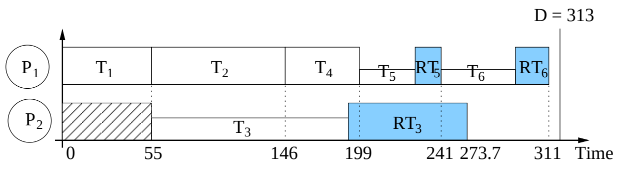


Figure 3. The example: individual recovery.

Consider the example in Figure 1 where we assume that there are four normalized frequency levels $\{0.4, 0.6, 0.8, 1.0\}$. After finishing the above steps, tasks T_3, T_5 and T_6 are selected to scale their frequencies to $0.6, 0.4$ and 0.4, respectively. The resulting schedule with individual recovery tasks is shown in Figure 3, where recovery tasks are represented by darkened rectangles. Here, we can see that all tasks can finish in time. Moreover, assuming $P_{ind} = 0.1$, $C_{ef} = 1$ and $m = 3$, we can see that RAPM-PC with individual recovery can save about 20% energy compared to that of no power management (NPM) where all tasks run at f_{max}.

B. RAPM-PC with Shared Recovery (SHR)

The global shared-recovery based RAPM has been studied for independent tasks on multiprocessor systems recently in [18]. In this section, we extend that solution to the case of tasks with precedence constraints. As observed in [18], due to the potential *simultaneous failures* of tasks running concurrently on different processors, we need to have a recovery task on all processors to guarantee the preservation of all tasks' original reliability figures. Moreover, to ensure that the largest selected task can be recovered on *any* processor, the recovery tasks on all processors have the same size, which is set to the WCET of the largest selected task.

Uniform Scaled Frequency Technique: First, we consider the simplest approach where all selected tasks take the same scaled frequency. Note that, the common size recovery tasks on all processors need to be scheduled at the end of the schedule to recover the potential failure of the last task on each processor. That is, only the slack at the end of the schedule can be utilized for the shared recovery task. Therefore, only the tasks with size not exceeding the smallest slack *at the end of the schedule* on all processors are *eligible* and can be potentially scaled down.

For energy efficiency, we may exclude a few large eligible tasks to get smaller common size recovery tasks while leaving more slack for energy management. By excluding the largest eligible task from management, we can find the lowest uniform frequency for all other smaller eligible tasks while ensuring that all tasks finish on time. By repeating the above process for other eligible tasks, we can finally find the best subset of eligible tasks and their corresponding scaled frequency level to get the best energy savings.

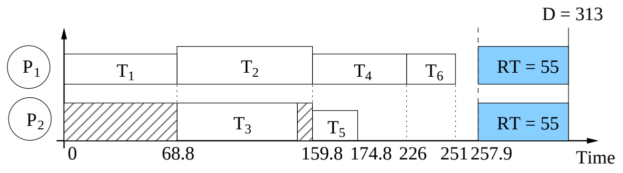


Figure 4. SHR: uniform frequency.

For the example task set, Figure 4 illustrates the schedule with shared recovery tasks. Note that, all tasks are eligible to scale down their execution as the amount of available slack is more than the size of the largest task T_2. However, to get smaller shared recovery tasks and achieve better energy savings, the largest two tasks T_2 and T_3 are excluded in the final solution. That is, tasks T_1, T_4, T_5 and T_6 are uniformly slowed down to frequency 0.8 while the shared recovery tasks take the size of task T_1, which is the largest selected tasks. With the same parameters as before, we can calculate that the shared recovery with uniform scaled frequency can save around 14% energy, compared to NPM.

Nonuniform Scaled Frequency Technique: Although the objective of shared recovery approach is to leave more slack for energy management to obtain better energy savings, the above example shows that the energy savings is less than that of individual recovery based approach. Note that, with uniform scaled frequency, the selected tasks may not be able to efficiently explore the slack in the middle of the schedule. For that purpose, we consider the shared recovery scheme with nonuniform scaled frequency.

Here, the size of the recovery tasks can be determined in the same way as the above scheme. However, the scaled frequency for each selected task is determined following the similar steps as those in Algorithm 1. The difference is that no individual recovery task is needed when a selected task is first scaled down.

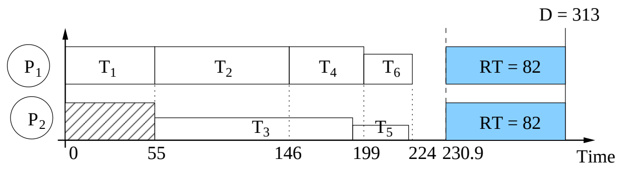
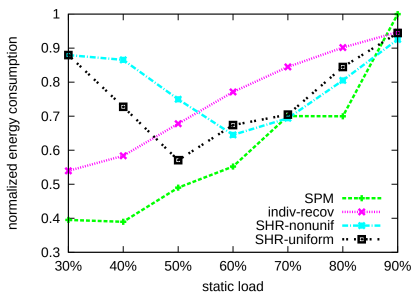
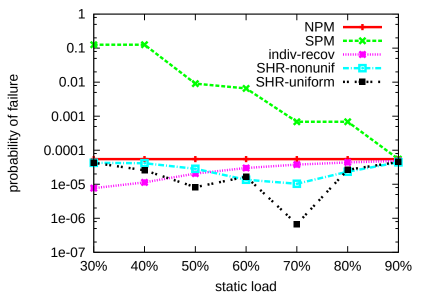


Figure 5. SHR: nonuniform frequencies.

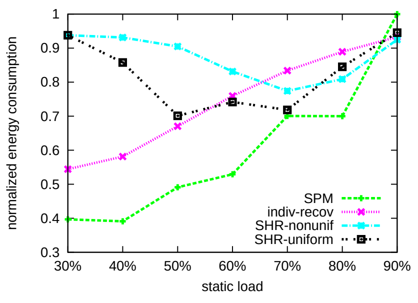
The schedule for the example with shared recovery and nonuniform frequencies is shown in Figure 5. Here, only T_3, T_5 and T_6 are selected to scale down their executions, at frequencies as $0.6, 0.4$ and 0.8, respectively. The shared recovery tasks have the size of task T_3. With better flexibility to choose frequency for selected tasks, the nonuniform frequency approach can save 18% energy compared to



(a) Energy consumption for 4 processors



(b) Probability of failure for 4 processors



(c) Energy consumption for 16 processors

Figure 6. Energy and reliability for static RAPM-PC schemes

NPM, which is slightly better than the case with uniform frequency.

C. Failure Handling for Shared Recovery

Note that, unlike the individual recovery-based scheme where the recovery of any faulty scaled task is independent, for the shared recovery based schemes, coordinated recovery actions are needed among the processors to ensure all tasks can finish in time [18]. Specifically, when errors are detected during the execution of a selected task on one processor, other processors need to be notified to adopt the *contingency schedule*. Here, the concurrently running tasks on other processors can continue their executions at the scaled frequency, but any *newly dispatched* tasks should run at f_{max} for reliability preservation *until the end of frame (period)*. With one recovery task on each processor, the shared recovery based scheme can ensure that any faulty execution of the concurrently running tasks can be recovered in time without violating tasks' timing constraints.

IV. Online RAPM-PC Schemes

Real-time applications typically only use a small fraction of their worst-case execution times. Moreover, the recovery tasks that are statically scheduled will be invoked only if their corresponding scaled primary tasks fail, which can occur with a small probability. Therefore, significant amount of dynamic slack can be expected at run time, which provides abundant opportunities to further scale down the execution of selected tasks to improve energy savings or to manage more tasks to enhance system reliability.

Based on the *slack sharing* technique in our previous work [28], we can apply dynamic slack reclamation on top of the schedules generated from the above static schemes. As shown in [28], as long as the execution order of dependent tasks is the same as in the static schedule (which is assumed to complete before the application's deadline), the slack sharing based dynamic slack reclamation can be applied without violating the deadline of the whole task set. Note that, for the individual recovery based approach, if a task does not have a statically scheduled recovery task, it needs to schedule its recovery task before reclaiming the remaining dynamic slack for energy savings. The steps are similar to those described in [18] and are omitted due to space limitation.

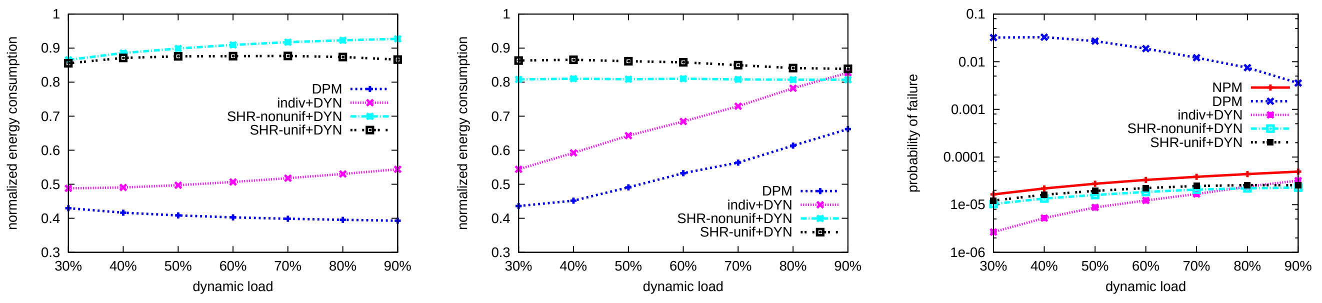
V. Simulation Results and Discussions

We evaluated the performance of the various *RAPM-PC* schemes with a discrete event simulator. The system energy consumption and reliability under the *no power management (NPM)* scheme, which executes all tasks at f_{max} and puts systems to power savings sleep states when idle, is used as the baseline. The parameters for energy and fault models are the same as those in [18]: For systems with 4 and 16 processors, we have $P_s = 0.04$ and $P_s = 0.16$, respectively. In addition, we have $P_{ind} = 0.1$, $C_{ef} = 1$, $m = 3$ and we assume the existence of four normalized frequency levels given by $\{0.4, 0.6, 0.8, 1.0\}$. For transient faults, we assume the exponential model with $\lambda_0 = 10^{-6}$ and $d = 3$.

We used the well-known TGFF library [21] to generate the task graphs. TGFF can generate various dependent task sets (such as tree and series-parallel graphs). For systems with 4 and 16 processors, the generated task sets contains 40 and 160 tasks, respectively. The WCETs of tasks are uniformly distributed in the range of $[10, 100]$. To quantify the amount of static slack at the end of schedule, the system load γ is defined as the ratio of the schedule length over the application's deadline. The actual execution time of each task T_i is generated based on the average-to-worst case execution time ratio α_i. For each setting, 100 task sets are generated and each task set is executed for $50,000,000$ times. The average results are reported.

A. Static RAPM-PC Schemes

For the static schemes where all tasks are assumed to take their WCETs, we evaluate individual recovery based RAPM-PC (*indiv-recov*), shared recovery based RAPM-PC with uniform frequency (*SHR-uniform*), and shared recovery based RAPM-PC with non-uniform frequency (*SHR-nonunif*). The *ordinary* static power management (SPM), that uniformly



(a) Energy consumption with $\gamma = 0.4$ (b) Energy consumption with $\gamma = 0.8$ (c) Probability of failure with $\gamma = 0.8$

Figure 7. Performance of online RAPM-PC schemes for 16 processors

scales down the execution of all tasks based on the system load without scheduling any recoveries, is also considered.

First, Figure 6(a) shows the normalized energy consumption of the static schemes under different system loads for a system with 4 processors. As the system load increases, less static slack is available and both SPM and the individual-recovery based schemes consume more energy. However, for the shared-recovery based schemes, when system load is low (e.g., $\gamma \leq 0.4$), the transient faults that can occur in the early stage of the execution with higher probability force the rest of tasks to run at f_{max}, causing more energy consumption when compared to that of individual-recovery based scheme. The results are in line with those for independent tasks [18]. For the system with 16 processors, similar results are obtained as shown in Figure 6(c).

Figure 6(b) further shows the *probability of failure* for all schemes, which is defined as the number of failed tasks (despite the use of potential recoveries) over the total number of tasks executed. Here, we can see that all static RAPM-PC schemes can preserve system reliability (by having lower probability of failure) when compared to that of *NPM*, confirming our theoretical results. In contrast, although *SPM* can save more energy, it can lead to significant system reliability degradation (up to four orders of magnitude) for low-to-moderate system loads.

B. Online RAPM-PC Schemes

For online schemes, we apply dynamic slack reclamation techniques on the static solutions obtained from the static RAPM-PC schemes. The three variants are denoted as *indiv+DYN*, *SHR-unif+DYN* and *SHR-nonunif+DYN*, respectively. Again, for comparison, we include *DPM* that represents the scheme that reclaims slack available in the SPM schedule. Here, we consider the system with 16 processors with $\gamma = 0.4$ and $\gamma = 0.8$ (which corresponds to low and high system loads, respectively).

First, for the low system load with $\gamma = 0.4$, Figure 7(a) shows the normalized energy consumption of dynamic schemes when varying dynamic load (α) from 0.3 to 0.9 (here, lower values of α indicate more dynamic slack). Note that, with system load $\gamma = 0.4$, all tasks under DPM will be scaled to the minimum frequency 0.4. Although DPM consumes more energy when dynamic load increases, the normalized energy consumption slightly decreases as NPM consumes more energy as well. For the individual-recovery based scheme, as unused recovery tasks can be reclaimed as slack for better energy savings, it consumes only slightly more energy (around 7% to 14%) compared to that of DPM. However, the shared-recovery based schemes consistently perform much worse with 30% to 40% more energy consumption. Again, this is due to the increased probability of faults affecting tasks with many successors at lower frequencies, and the required coordination among processors for fault handling. For high system load with $\gamma = 0.8$, similar results are obtained as shown in Figure 7(b). Note that, for higher value of α, there is less dynamic slack and both individual and shared recovery based schemes perform roughly the same in terms of energy savings.

Not surprisingly, all online RAPM-PC schemes can preserve system reliability as shown in Figure 7(c) when the system load is $\gamma = 0.8$. Again, without taking the negative effects of DVFS on system reliability into consideration, the ordinary *DPM* scheme results in much increased probability of failure (by up to three orders of magnitude).

VI. Conclusions

In this paper, for parallel real-time applications with precedence constraints running on shared memory multiprocessor systems, we investigated the reliability-aware power management (RAPM) schemes that save energy while guaranteeing a certain level of system reliability. Noting that the problem is NP-hard, we studied both individual and shared recovery based heuristics, where a separate recovery task is scheduled for any selected tasks and a few selected tasks on the same processor sharing one recovery task, respectively. Moreover, online schemes based on the *slack-sharing* slack reclamation technique are also considered. Simulation results confirm that the effectiveness of the proposed schemes.

References

[1] I. Ahmad, R. Arora, D. White, V. Metsis, and R. Ingram. *Energy-Constrained Scheduling of DAGs on Multi-core Processors*, pages 592–603. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2009.

[2] H. Aydin, V. Devadas, and D. Zhu. System-level energy management for periodic real-time tasks. In *Proc. of The* 27^{th} *IEEE Real-Time Systems Symposium (RTSS)*, pages 313–322, 2006.

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

[4] M.L. Dertouzos and A.K. Mok. Multiprocessor online scheduling of hard-real-time tasks. *Software Engineering, IEEE Transactions on*, 15(12):1497 –1506, December 1989.

[5] A. Ejlali, B.M. Al-Hashimi, and P. Eles. A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems. In *Proc. of the IEEE/ACM Int'l conference on Hardware/software codesign and system synthesis (CODES)*, pages 193–202, 2009.

[6] E. (Mootaz) Elnozahy, R. Melhem, and D. Mossé. Energy-efficient duplex and tmr real-time systems. In *Proc. of The IEEE Real-Time Systems Symposium*, 2002.

[7] S. Ghosh, R. Melhem, and D. Mossé. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *IEEE Trans. on Parallel and Distributed Systems*, 8(3):272–284, 1997.

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

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

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

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

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

[13] J. Kang and S. Ranka. Dynamic slack allocation algorithms for energy minimization on parallel machines. *J. Parallel Distrib. Comput.*, 70(5):417–430, 2010.

[14] C.M. Krishna and Y.-H. Lee. Voltage-clock-scaling adaptive scheduling techniques for low power in hard real-time systems. In *IEEE Real Time Technology and Applications Symposium*, pages 156–165, 2000.

[15] R. Melhem, D. Mossé, and E. (Mootaz) Elnozahy. The interplay of power management and fault recovery in real-time systems. *IEEE Trans. on Computers*, 53(2):217–231, 2004.

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

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

[18] X. Qi, D. Zhu, and H. Aydin. Global scheduling based reliability-aware power management for multiprocessor real-time systems. *Real-Time Systems: The International Journal of Time-Critical Computing Systems*, 47(2):109–142, 2011.

[19] K. G. Shin and H. Kim. A time redundancy approach to tmr failures using fault-state likelihoods. *IEEE Trans. on Computers*, 43(10):1151 – 1162, 1994.

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

[21] TGFF. http://ziyang.eecs.umich.edu/ dickrp/tgff/.

[22] O. S. Unsal, I. Koren, and C. M. Krishna. Towards energy-aware software-based fault tolerance in real-time systems. In *Proc. of The International Symposium on Low Power Electronics Design (ISLPED)*, Aug. 2002.

[23] X. Wang, M. Chen, C. Lefurgy, and T.W. Keller. Ship: Scalable hierarchical power control for large-scale data centers. In *Proc. of the Int'l Conference on Parallel Architectures and Compilation Techniques*, pages 91–100, 2009.

[24] Y. Zhang and K. Chakrabarty. Energy-aware adaptive checkpointing in embedded real-time systems. In *Proc. of IEEE/ACM Design, Automation and Test in Europe Conference(DATE)*, 2003.

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

[26] D. Zhu. Reliability-aware dynamic energy management in dependable embedded real-time systems. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 397 – 407, 2006.

[27] D. Zhu and H. Aydin. Reliability-aware energy management for periodic real-time tasks. *IEEE Trans. on Computers*, 58(10):1382–1397, 2009.

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

[29] D. Zhu, R. Melhem, and D. Mossé. The effects of energy management on reliability in real-time embedded systems. In *Proc. of the Int'l Conf. on Computer Aidded Design*, pages 35–40, 2004.