Reliability-Aware Energy Management for Periodic Real-Time Tasks

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

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

Abstract

The prominent energy management technique in real-time embedded systems, Dynamic Voltage and Frequency Scaling (DVFS), was recently shown to have direct and adverse effects on system reliability. In this work, we propose static and dynamic *reliability-aware energy management* schemes for a set of *periodic* real-time tasks to minimize the *system-wide* energy consumption while preserving system reliability. Focusing on EDF scheduling, we first show that the problem is NP-hard and propose two *task-level*, static, utilization-based heuristics. Then, we develop a *job-level* dynamic (on-line) scheme by building on the idea of *wrapper-tasks*, to monitor and manage dynamic slack efficiently in reliability-aware settings. Our schemes incorporate recovery tasks/jobs to the schedule as needed for reliability preservation, while still using the remaining slack for energy savings. Simulation results show that all the proposed schemes can achieve significant energy savings while preserving the system reliability. The energy savings obtained by the static heuristics are shown to be close to those of the static optimal solution by a margin of 5%. Further, by effectively using the run-time slack, the dynamic schemes are able to yield energy savings similar to those of ordinary (but *reliability-ignorant*) energy management algorithms, but without suffering from drastically decreased system reliability figures.

1 Introduction

The phenomenal improvements in the performance of computing systems have resulted in drastic increases in power densities. For battery-operated devices with limited energy budget, energy has been recognized as a first-class system resource [29]. Many hardware and software techniques have been proposed to manage power consumption in modern computing systems and power aware computing has recently become an important research area. One common strategy to save energy is to run the system components at low-performance operation points, whenever possible. For example, DVFS scales down the CPU frequency and supply voltage simultaneously to save energy [28].

For real-time systems where tasks have stringent timing constraints, scaling down the clock frequency (processing speed) may cause deadline misses and special provisions are needed. In the recent past, several research studies explored the problem of minimizing energy consumption while meeting all the deadlines for various real-time task models. These include also a number of power management schemes which exploit the available static and/or dynamic *slack* in the system [1, 22, 24].

Reliability and fault tolerance have always been major factors in computer system design. Due to the effects of hardware defects, electromagnetic interferences and/or cosmic ray radiations, faults may occur at run-time, especially in systems deployed in dynamic/vulnerable environments. With the continued scaling of CMOS technologies and reduced design margins for higher performance, it is expected that, in addition to the systems that operate in electronics-hostile environments (such as those in outer space), practically all digital computing systems will be much more vulnerable to *transient faults* [8]. The *backward error recovery* techniques, which restore the system state to a previous safe state and repeat the computation, can be used to tolerate transient faults [23]. In real-time systems, backward recovery techniques often rely on *temporal redundancy*, manifested in the form of slack time.

It is worth noting that both DVFS and backward recovery techniques are based on (and compete for) the active use of the system slack. Thus, there is an interesting trade-off between energy efficiency and the system reliability. Moreover, DVFS has been shown to have a direct effect on the rate increases of transient faults, especially for those induced by cosmic ray radiations [34], which further complicates the problem. Hence, 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 becomes a necessity.

Although fault tolerance and energy management have been well studied in the context of real-time systems independently, only a few studies investigated the implications of having both fault tolerance and energy efficiency requirements very recently [7, 21, 27, 30]. As an initial study, we previously proposed a *reliability-aware power management (RA-PM)* scheme that dynamically schedules a recovery job at task dispatch time, hence preserving the system reliability [32]. The scheme is further extended to multiple aperiodic tasks that share a common deadline [33]. However, preemptive scheduling, which is common for *periodic* task systems, has not been considered.

In this work, we study both static and dynamic RA-PM schemes for a set of periodic real-time tasks scheduled by preemptive Earliest-Deadline-First (EDF) policy. Specifically, we consider the problem of exploiting the spare CPU capacity for energy savings while preserving the system reliability. We show that the optimal static RA-PM problem is *NP-hard* and propose two efficient heuristics for selecting a subset of tasks to use the spare capacity for the objectives of energy and reliability management. Moreover, we develop a *job-level* dynamic RA-PM algorithm, that tracks and manages the dynamic slack which may be generated at run-time, again for these dual objectives. The latter algorithm is built on the *wrapper task* mechanism: the key idea is to *conserve* the dynamic slack allocated to scaled tasks for recovery, which is essential for preserving reliability. To the best of our knowledge, this is the first research effort that provides a comprehensive energy management framework for *periodic* real-time tasks *while preserving the system reliability*.

The remainder of this paper is organized as follows. The system model and problem formulation are presented in Section 2. In Section 3, we present the task-level, utilization-based static RA-PM schemes. The *wrapper-task* concept is introduced and the job-level dynamic RA-PM scheme is presented in Section 4. Simulation results are presented and discussed in Section 5. Section 6 reviews the closely related work and Section 7 concludes the paper.

2 System Model and Problem Description

2.1 Application Model

We consider a set of independent periodic real-time tasks $\Gamma = \{T_1, \ldots, T_n\}$. The task T_i is characterized by a pair (p_i, c_i), where p_i represents its period and c_i denotes its worst case execution time (WCET). The j^{th} *job* of T_i, which is referred to as J_{ij}, arrives at time $(j-1) \cdot p_i$ and has a deadline of $j \cdot p_i$.

In DVFS settings, it is assumed that the WCET c_i of task T_i is given under the maximum processing speed f_{max}.

For simplicity, we assume that the execution time of a task scales *linearly* with processing speed[1]. That is, at speed f, the execution time of task T_i is assumed to be $c_i \cdot \frac{f_{max}}{f}$.

The system utilization is defined as $U = \sum_{i=1}^{n} u_i$, where $u_i = \frac{c_i}{p_i}$ is the utilization of task T_i. The tasks are to be executed on a uni-processor system according to the preemptive EDF policy. Using the well-known feasibility condition for EDF [20], we assume that $U \leq 1$.

2.2 Power Model

The relation between the supply voltage and operating frequency is known to be almost linear [4]. DVFS reduces supply voltages for lower frequencies [28] and we will use the term *frequency change* to stand for both supply voltage and frequency adjustments. Considering the ever-increasing static leakage power due to scaled feature size and increased levels of integration [17] as well as the power-saving states provided in modern power-efficient components (e.g., CPU [6] and memory [18]), in this work, we adopt the simple *system-level power model* proposed in [34], where the power consumption P of a computing system is given by:

$$P = P_s + \hbar(P_{ind} + P_d) = P_s + \hbar(P_{ind} + C_{ef} f^m) \tag{1}$$

Here, P_s is the *static power*, which can be removed only by powering off the whole system. It includes the power to maintain basic circuits and keep the clock running. P_{ind} is the *frequency-independent active power*, which is a constant and corresponds to the power that is independent of CPU processing speed. It can be efficiently removed by putting systems into sleep state(s) [6, 18]. P_d is the *frequency-dependent active power*, which includes processor's dynamic power and any power that depends on system processing speeds [4, 18].

When there is a computation in progress, the system is *active* and $\hbar = 1$. Otherwise, when the system is in power-saving sleep mode or turned off, $\hbar = 0$. The effective switching capacitance C_{ef} and the dynamic power exponent m (in general, $2 \leq m \leq 3$ [4]) are system-dependent constants and f is the processing frequency. For simplicity, normalized frequencies are used (i.e. $f_{max} = 1.0$).

Despite its simplicity, the above power model captures the essential components for system-wide energy management. Note that *energy* is the time integral of power. For a given job, the energy consumption to execute it will be $E = P \cdot t$, where P is the power level and t is the job's execution time. Intuitively, lower frequencies result in less frequency-dependent active energy consumption. But with reduced speeds, the job runs longer and thus consumes more static and frequency-independent active energy. Therefore, a minimal *energy-efficient frequency* f_{ee}, below which DVFS starts to consume more total energy, does exist [13, 17, 24]. From the above equation, one can find that f_{ee} is given as[2] [34]:

$$f_{ee} = \sqrt[m]{\frac{P_{ind}}{C_{ef} \cdot (m-1)}} \tag{2}$$

Consequently, we assume that the CPU frequency is never reduced below the threshold f_{ee} for energy efficiency. We

[1]A number of studies have indicated that the execution time of tasks does not scale linearly with reduced processing speed due to accesses to memory [26] and/or I/O devices [3]. However, exploring the full implications of this observation is beyond the scope of this paper and it left as future work.

[2]Considering the prohibitive overhead of turning on/off a system (e.g., tens of seconds), we assume that the system will not be turned off during the interval considered and P_s is always consumed.

develop our framework by assuming that the frequency can vary continuously from f_{ee} to f_{max}. However, we also discuss the implications of having discrete speed levels in Section 5.3.

2.3 Fault Model

During a job's execution, a fault may occur due to various reasons, such as hardware failures, software errors, electromagnetic interferences as well as the effects of cosmic ray radiations. The *transient* faults occur much more frequently than *permanent* faults [15], especially with the continued scaling of CMOS technologies and reduced design margins [8]. Consequently, in this paper, we focus on transient faults and explore backward recovery techniques to recovery them. It is assumed that the faults are detected using *sanity* (or *consistency*) checks at the completion of a job's execution, and if needed, the recovery task is dispatched, by taking the form of re-execution [23].

In our previous work [34], we have studied the negative effects of DVFS on transient faults induced by cosmic ray radiations. Assuming that transient faults follow Poisson distribution [30], the average transient fault rate for systems running at frequency f (and corresponding supply voltage) can be expressed as [34]:

$$\lambda(f) = \lambda_0 \cdot g(f) \tag{3}$$

where λ_0 is the average fault rate corresponding to the maximum frequency f_{max}. That is, $g(f_{max}) = 1$. With reduced processing speeds and supply voltages, the *critical charge*, which is the smallest charge needed to cause a soft error, generally decreases and leads to increased fault rates [25]. Therefore, we have $g(f) > 1$ for $f < f_{max}$.

Moreover, considering the relationship between transient fault rates, critical charge, supply voltages and the number of particles in the cosmic rays [11, 25, 36], we have derived an *exponential* fault rate model: $\lambda(f) = \lambda_0 \cdot g(f) = \lambda_0 10^{\frac{d(1-f)}{1-f_{min}}}$, where the exponent d (> 0) is a constant which indicates the sensitivity of fault rates to DVFS [34]. The maximum fault rate is assumed to be $\lambda_{max} = \lambda_0 10^d$, which corresponds to the minimum frequency f_{ee} (and corresponding supply voltage).

2.4 Problem Description

Our primary objective in this paper is to develop power management schemes for periodic real-time tasks executing on a uni-processor system and to preserve system reliability at the same time. Define the *reliability* of a real-time job as the *probability of being correctly executed before its deadline*. One of the key findings reported in [34] is that the reliability of any job whose execution is scaled through DVFS decreases drastically due to the increased fault rates <u>and</u> extended execution time.

Without loss of generality, we assume that the system reliability is *satisfactory* when no power management scheme is applied, even under the worst-case scenario (i.e., all tasks take their WCETs). Note that the reliability of a real-time system depends on the *correct* execution of *all* jobs within their deadlines. In order to preserve system reliability, for simplicity, we focus on maintaining the reliability of *individual* jobs in this work. For the cases where recovery jobs are used to achieve the specified reliability, such recovery jobs can be considered as normal jobs and their reliabilities are also preserved. Specifically, **for a periodic real-time task set with utilization U, we consider the problem of how to use the spare CPU utilization $1 - U$, as well as the dynamic slack generated at run-time, for maximizing energy savings while keeping the reliability of any job of task T_i no less than R_i^0 ($i = 1, \ldots, n$)**, where $R_i^0 = e^{-\lambda_0 c_i}$ (from

Poisson fault arrival pattern and the average fault rate λ_0 [32]) is the original reliability for jobs of task T_i, when there is no power management and the jobs uses their WCETs.

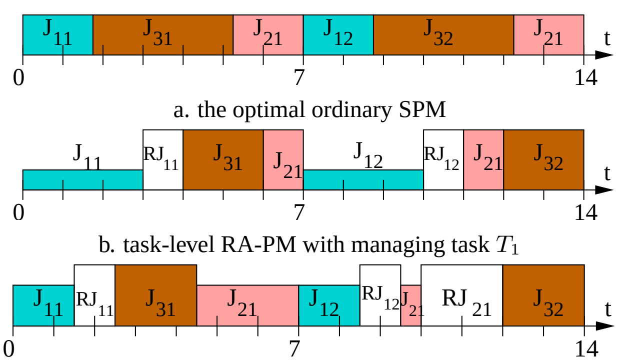
2.5 Reliability-Aware Power Management (RA-PM)

Conventionally, DVFS-based *ordinary* power management schemes exploit all the available (dynamic or static) slack for energy management and are, consequently, *reliability-ignorant* (in the sense that no attention is paid to the potential effects of DVFS on task reliabilities). Instead of using *all* the available slack for DVFS to save energy, one can reserve a portion of the slack to schedule one *recovery job* RJ for any job J whose execution is scaled down, to recuperate the reliability loss due to the energy management [32]. The recovery job RJ will be dispatched (at the maximum frequency f_{max}) only if a transient fault is detected when J completes. The recovery is in the form of re-execution and RJ has the same WCET as that of J [23].

With the help of RJ, the overall *reliability* R of job J will be the summation of the probability of J being executed correctly and the probability of having transient fault(s) during J's execution while the recovery job RJ being executed correctly. Therefore, **if the amount of available slack is *more* than the WCET of a job, by scheduling a recovery job (e.g., re-execution), one can guarantee to preserve the reliability of a real-time job while still obtaining energy savings using the remaining slack, regardless of different fault rate increases and scaled processing speeds** [32].

In increasing level of sophistication and implementation complexity, we first introduce the *task-level static* schemes and then *job-level dynamic* schemes in the next two sections.

3 Task-Level Static Schemes



a. the optimal ordinary SPM

b. task-level RA-PM with managing task T_1

c. task-level RA-PM with managing tasks T_1 and T_2

Figure 1: Static schemes for three tasks $\{T_1(1,7), T_2(2,14), T_3(2,7)\}$.

To start with, we can consider static RA-PM schemes that make their decisions at the *task-level*. In this approach, for simplicity, all the jobs of a task have the same treatment. That is, if a given task is selected for energy management, all its jobs will run at the same scaled frequency; otherwise, they will run at f_{max}. From the above discussion, to recuperate reliability loss due to scaled execution, each *scaled job*[3] will need a corresponding recovery job within its deadline, should a fault occur.

[3]We use the expression *scaled job* to refer to any job whose execution is slowed down through DVFS, for energy management purposes.

To provide the required recovery jobs, we can construct *recovery tasks (RT)* by exploiting the spare CPU capacity (or, *static slack*). The recovery task will have the same timing parameters (i.e., WCET and period) as those of the task to be scaled. By incorporating the recovery tasks, EDF could schedule a recovery job for any scaled job within its deadline and preserve its reliability. Here, the recovery job will be assigned a lower priority level than the corresponding (*primary*) job with the same deadline, and it will be activated only when the primary job incurs a fault.

As a concrete example, suppose that we have a periodic task set of three tasks $\Gamma = \{T_1(1,7), T_2(2,14), T_3(2,7)\}$ with system utilization as $U = \frac{4}{7}$. Without considering system reliability, the *optimal* ordinary static power management (SPM) under EDF will scale down all tasks at the speed $f = U \cdot f_{max} = \frac{4}{7}$ as shown in Figure 1a [1, 22]. In the figure, the X-axis represents time and the height of task boxes represents processing speed. Due to the periodicity, only the schedule within the least common multiple (LCM) of tasks' periods is shown. However, by uniformly scaling down the execution in this way, the reliability figures of all the tasks (and that of the system) would be significantly reduced [34].

When applying static RA-PM, we first compute the spare capacity as $1 - U = \frac{3}{7}$. After constructing the recovery task $RT_1(1,7)$, which has the same WCET and period as the task T_1 with the utilization as $ru_1 = \frac{1}{7}$, the overall system utilization will be $U' = U + ru_1 = \frac{5}{7}$. If we allocate the remaining spare capacity (i.e., $1 - U' = \frac{2}{7}$) to task T_1, all jobs of T_1 can be scaled down to the speed of $\frac{1}{3}$. With the recovery task RT_1 and the scaled execution of T_1, the *effective* system utilization is *exactly* 1 and the modified task set is schedulable under EDF as shown in Figure 1b. From the figure, we can see that every scaled job of task T_1 has a corresponding recovery job within its deadline. Therefore, all the jobs of task T_1 could preserve their reliability level R_1^0. Notice that, the jobs of tasks T_2 and T_3 still run at f_{max} and, hence, their reliability figures are preserved at the levels of R_2^0 and R_3^0, respectively.

Therefore, by incorporating a recovery task for each task to be managed, the task-level utilization-based static RA-PM scheme could preserve system reliability while obtaining energy savings. In [33], we reported that it is not optimal (in terms of energy savings) for the RA-PM scheme to utilize all the slack for a single task in case of *aperiodic* tasks. Similarly, we may use the spare capacity for *multiple* periodic tasks for better energy savings. For instance, Figure 1c shows the case where both T_1 and T_2 are scaled to speed $\frac{2}{3}$ after constructing the recovery tasks RT_1 and RT_2. For illustration purposes, we assume that the system power is given by a cubic function. Simple algebra shows that, managing only task T_1 could achieve the energy savings of $\frac{8}{9}E$, where E is the energy consumed by all jobs of task T_1 within LCM under no power management. In comparison, the energy savings would be $\frac{11}{9}E$ if both T_1 and T_2 are managed, which is a significant improvement.

Intuitively, when more tasks are to be managed, more computation can be scaled down for more energy savings. However, more spare capacity will be reserved for recovery tasks, which, in turn, reduces the remaining spare capacity for DVFS to save energy. A natural question to ask is, for a periodic task set with multiple real-time tasks, whether there exists a fast (i.e. polynomial-time) optimal solution (in term of energy savings) for the problem of task-level utilization-based static RA-PM. Unfortunately, the answer is negative, as we argue below.

3.1 Intractability of Task-Level Utilization-Based RA-PM

The inherent complexity of the optimal static RA-PM problem warrants an analysis. Suppose that the system utilization of the task set is U and the spare capacity is $sc = 1 - U$. If a subset Φ of tasks are selected for management with total utilization $X = \sum_{T_i \in \Phi} u_i < sc$, after accomodating all recovery tasks, the remaining spare capacity (i.e., $sc - X$) could be used to scale down the selected tasks for energy management. Considering the convex relation between power and processing speed (see Equation 1), the solution that minimizes the energy consumption is to uniformly scale down all jobs of the selected tasks, where the scaled processing speed will be $f = \frac{X}{X+(sc-X)} = \frac{X}{sc}$. Therefore, without considering the execution of recovery jobs, the amount of total *fault-free* energy consumption within LCM would be:

$$
\begin{aligned}
E_{LCM} &= LCM \cdot P_s + LCM(U-X)(P_{ind} + c_{ef} \cdot f_{max}^m) \\
&\quad + LCM \cdot sc \left(P_{ind} + c_{ef} \cdot \left(\frac{X}{sc} \right)^m \right)
\end{aligned} \quad (4)
$$

where the first part is the energy consumption due to static power, the second part is for jobs of unselected tasks and the third part is for scaled jobs of the selected tasks. Simple algebra shows that, when $X_{opt} = sc \cdot (\frac{P_{ind}+C_{ef}}{m \cdot C_{ef}})^{\frac{1}{m-1}}$, E_{LCM} will be minimized.

If $X_{opt} \geq U$, all tasks should be scaled down appropriately to minimize energy consumption. Otherwise, the problem becomes essentially a task selection problem, where the summation of the selected tasks' utilization should be *exactly* equal to X_{opt}, if possible. In other words, such a choice would definitely be the optimal solution.

Theorem 1 *For a set of periodic tasks, the problem of the task-level utilization-based static RA-PM is NP-hard.*

Proof We consider a special case of the problem with $m = 2$, $C_{ef} = 1$ and $P_{ind} = 0$; that is, $X_{opt} = \frac{sc}{2}$. We show that even this special instance is intractable, by transforming the PARTITION problem, which is known to be NP-hard [10], to that special case.

In PARTITION, the objective is to find whether it is possible to partition a set of n integers $a_1, \ldots, a_n$ (where $\sum_{i=1}^{n} a_i = S$) into two disjoint subsets, such that the sum of numbers in each subset is exactly $\frac{S}{2}$.

Given an instance of the PARTITION problem, we construct the corresponding static RA-PM instance as follows: we have n periodic tasks, where $c_i = a_i$ and $p_i = 2 \cdot S$. Note that, in this case, $U = \sum \frac{c_i}{p_i} = \frac{1}{2}$, $sc = 1 - U = \frac{1}{2}$. Observe that, the energy savings will be maximized if it is possible to find a subset of tasks whose total utilization is exactly $X_{opt} = \frac{sc}{2} = \frac{1}{4}$. Since $p_i = 2S \ \forall i$, this is possible if and only if one can find a subset of tasks Φ such that $\sum_{i \in \Phi} c_i = \frac{S}{2}$. But this can happen only if the original PARTITION problem admits a YES answer. Therefore, if RA-PM problem had a polynomial-time solution, one could also solve the PARTITION problem in polynomial-time, by constructing the corresponding RA-PM problem, and checking if the maximum energy savings that can be obtained correspond to the amount we could gain through managing exactly $X_{opt} = \frac{sc}{2} = 25\%$ of the periodic workload.

■

3.2 Heuristics for Task-Level Utilization-Based RA-PM

Considering the intractability of the problem, we propose two simple heuristics for selecting tasks for energy management: *Largest-utilization-first (LUF)* and *Smallest-utilization-first (SUF)*. Suppose that the tasks in a given periodic task set are indexed in the non-decreasing order of their utilizations (i.e., $u_i \leq u_j$ for $1 \leq i < j \leq n$). SUF will select the first k tasks, where k is the largest integer that satisfies $\sum_{i=1}^{k} u_i \leq X_{opt}$. Similarly, LUF will select the last k tasks, where k is the smallest integer that satisfies $\sum_{i=k}^{n} u_i \leq X_{opt}$.

Here, SUF tries to manage as many tasks as possible, since any managed jobs could achieve better reliability [32]. However, at some point, when the remaining spare capacity is not enough to accomodate a recovery task for the task with the next smallest utilization, SUF may waste significant portion of the spare capacity. LUF tries to select larger utilization tasks first, where the amount of wasted spare capacity is at most the smallest utilization among all tasks.

The potential drawback of LUF is that, sometimes, relatively few tasks might be managed for energy savings. These heuristics are evaluated in Section 5.

4 Job-Level Dynamic RA-PM

In our backward recovery framework, the recovery jobs are executed only if their corresponding scaled jobs fail. Otherwise, the CPU time reserved for recovery jobs are removed (freed) and become dynamic slack at run-time. Moreover, it is well-known that real-time tasks typically take a small fraction of their WCETs [9]. Therefore, significant amount of dynamic slack can be expected at run time, which should be exploited to further save energy and/or enhance system reliability by managing individual *jobs*.

Unlike the greedy RA-PM scheme which allocates all available dynamic slack for the next ready task when the tasks share a common deadline [32], in periodic execution settings, the run-time dynamic slack will be generated at different priorities and may not be always reclaimable by the next ready job [1]. Moreover, possible preemptions that a job experiences *after* it has reclaimed some slack further complicates the problem. This is because, once a job's execution is scaled through DVFS, additional slack *must* be reserved for potential recovery operations to preserve system reliability. *Hence, maintaining the reclaimed slack until the job completes successfully is essential in reliability-aware settings*.

The slack management problem has been studied extensively (e.g., Slack Stealing [19], CASH-queue [5] and α-queue [1] approaches) for different purposes. By borrowing and also extending some fundamental ideas from these studies, we provide a new framework which guarantees the *conservation* of the reclaimed slack, thereby maintaining the reliability figures.

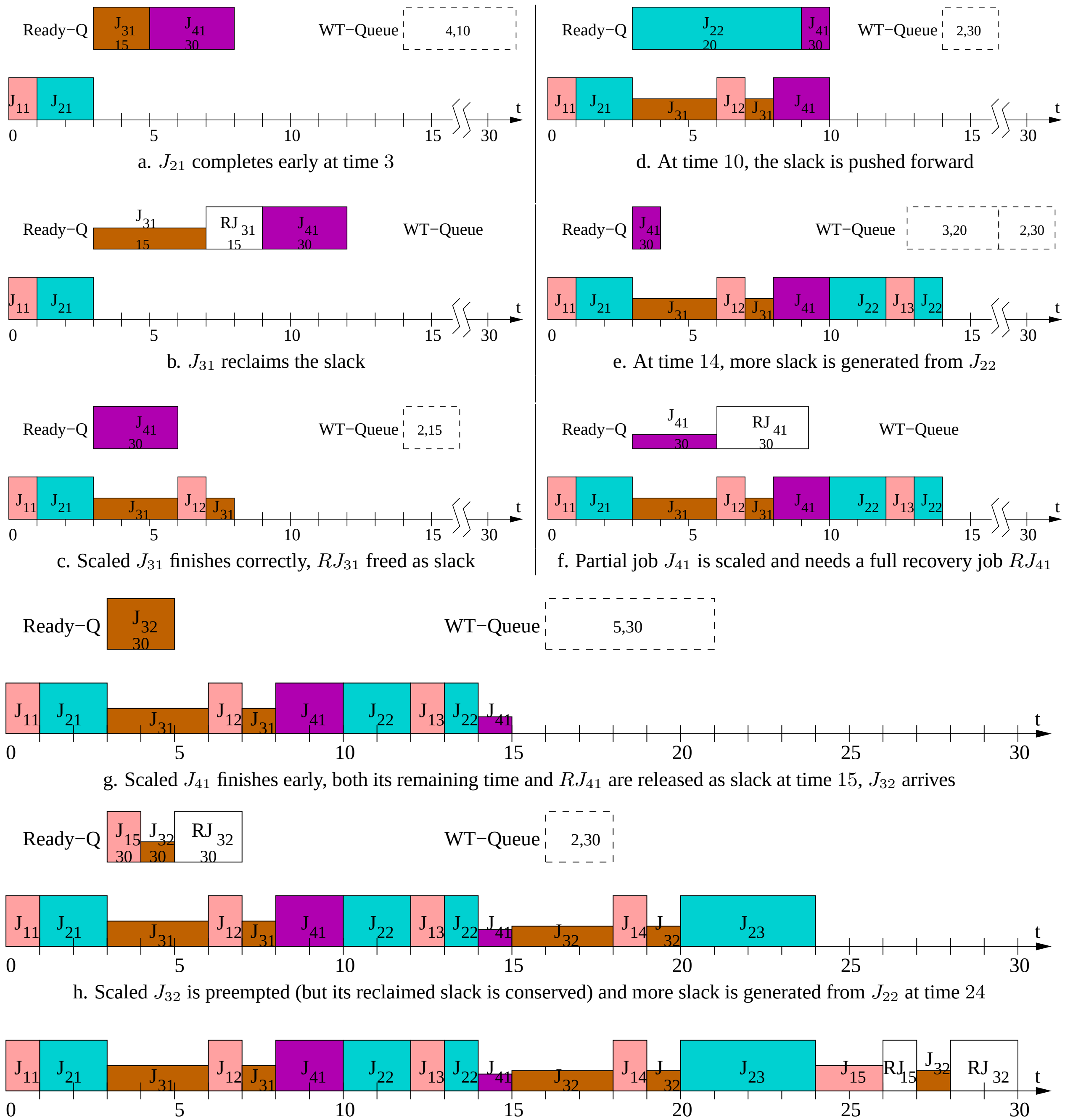
Specifically, in this work, we propose the *wrapper-task* mechanism to track/manage dynamic slack. For any dynamic slack generated at run-time, a new wrapper-task will be created with the following two timing parameters: a *size* that equals the amount of dynamic slack generated and a *deadline* that is the deadline of the job whose early completion gave rise to this slack.

A wrapper-task is destroyed when all the slack it represents is *reclaimed* or *wasted*. Otherwise, it will compete for CPU along with normal real-time jobs. When a wrapper-task has the highest priority (i.e., the earliest deadline) and is scheduled, it will "fetch" the highest priority job in the ready queue (if any) and *wrap* the job's execution during the interval when the wrapper-task occupies the CPU. If no such job is is ready, the CPU will become idle, the wrapper-task is said to "execute no-ops" and the corresponding dynamic slack is consumed/wasted during this time interval.

4.1 An Example for Wrapper-Task Mechanism

Before formally presenting the algorithm, we first illustrate the idea of wrapper-tasks through a detailed example. We consider a task-set with four periodic real-time tasks $\Gamma = \{T_1(1,6),\, T_2(6,10),\, T_3(2,15),\, T_4(3,30)\}$. For the jobs within LCM (= 30), suppose that J_{21}, J_{22}, J_{23} and J_{41} take 2, 3, 4 and $2\frac{1}{3}$ time units, respectively, and all other jobs take their WCETs.

Recall that EDF scheduling is used. For jobs with the same deadline, the one of the smaller index task is assumed to have higher priority. When J_{21} completes early at time 3, 4 units of dynamic slack will be generated and the system state is shown in Figure 2a. Here, a wrapper-task (shown as *dotted rectangle*) is created to represent the slack, which is labeled by two numbers: a *size* (e.g., 4) and a *deadline* (e.g., 10). Similar to ready jobs that are kept in the ready queue



a. J_{21} completes early at time 3

b. J_{31} reclaims the slack

c. Scaled J_{31} finishes correctly, RJ_{31} freed as slack

d. At time 10, the slack is pushed forward

e. At time 14, more slack is generated from J_{22}

f. Partial job J_{41} is scaled and needs a full recovery job RJ_{41}

g. Scaled J_{41} finishes early, both its remaining time and RJ_{41} are released as slack at time 15, J_{32} arrives

h. Scaled J_{32} is preempted (but its reclaimed slack is conserved) and more slack is generated from J_{22} at time 24

i. J_{15} reclaimed the new slack and was scaled down; when it fails, RJ_{15} is executed; J_{32} and RJ_{32} meet their deadlines as well

Figure 2: Using wrapper-tasks to manage dynamic slack

(*Ready-Q*) (where the deadlines are indicated by the numbers at the bottom of the job boxes), wrapper-tasks are kept in a *WT-Queue* in increasing order of their deadlines.

It is known that, the slack that a job J_x can reclaim (i.e. the *reclaimable* slack) should have a deadline no later than the deadline of J_x [1]. From our previous discussion, to recuperate reliability loss due to energy management, a recovery job needs to be scheduled within J_x's deadline. Hence, a non-scaled job will reclaim the slack only if the amount of reclaimable slack is larger than the job size.

Thus, at time 3, J_{31} reclaims the available slack and scales down its execution as shown in Figure 2b. Here, a recovery job RJ_{31} is created. The scaled execution of J_{31} uses the time slots of the reclaimed slack and is scaled at speed $\frac{2}{4} = \frac{1}{2}$, while RJ_{31} will take J_{31}'s original time slots. Both J_{31} and RJ_{31} could finish their execution within J_{31}'s deadline in the worst case.

Suppose that the scaled J_{31} finishes its execution correctly at time 8, after being preempted by J_{12} at time 6. The recovery job RJ_{31} will be removed from *Ready-Q* and all its time slots will become slack as shown in Figure 2c. But this slack is not sufficient for reclamation by J_{41}. However, since the corresponding wrapper-task has higher priority, it is scheduled and wraps the execution of J_{41}. When the wrapper-task finishes at time 10, a *new* wrapper-task with the same size is created, but with the deadline of J_{41}. It can also be viewed as J_{41} borrowing the slack for its execution and returning it with the extended deadline (i.e., the slack is *pushed forward*). The schedule and queues at time 10, after J_{22} arrives, are shown in Figure 2d.

When J_{22} completes early at time 14 (after being preempted by J_{13} at time 12), 3 units of slack is generated with deadline of 20, as shown in Figure 2e. Now, we have two pieces of slack (represented by two wrapper-tasks, respectively) with different deadlines.

Note that, as faults are assumed to be detected at the end of a job's execution, a *full* recovery job is needed to recuperate the reliability loss due to even *partially scaled* execution[4]. Thus, when the *partially-executed* J_{41} reclaims all the available slack (since both wrapper-tasks have deadlines no later than J_{41}'s deadline), a full recovery job RJ_{41} is created and inserted into *Ready-Q*. J_{41} uses the remaining slack to scale down its execution appropriately as shown in Figure 2f.

When the scaled J_{41} finishes early at time 15, both its unused CPU time and RJ_{41} are freed as slack. After the arrival of J_{32} at time 15, the schedule and queues are shown in Figure 2g. Here, J_{32} will reclaim the slack and be scaled with speed $\frac{2}{5}$ after reserving the slack for the recovery job RJ_{32}. After the scaled J_{32} is preempted by J_{14} and J_{23}, at time 18 and 20, respectively, and J_{23} completes early at time 24, Figure 2h shows the newly generated slack and state of *Ready-Q*, which contains J_{15} (with arrival time 24). Note that, the recovery job RJ_{32} (i.e., the slack time) is conserved even after J_{32} is preempted by higher priority jobs.

J_{15} reclaims the new slack. Suppose that both of the scaled jobs J_{15} and J_{32} fail, then, RJ_{15} and RJ_{32} will be executed as illustrated in Figure 2i. It can be seen that all jobs (including recovery jobs) finish their executions on time and no deadline is missed.

4.2 Job-Level Dynamic RA-PM Algorithm (RA-DPM)

As the example illustrated, in addition to *Ready-Q* that is used to hold the ready jobs, a wrapper-task queue (i.e., *WT-Queue*) is used to track/manage available dynamic slack. The rules for managing dynamic slack with wrapper-tasks are as follows:

- **Rule 1 (slack generation):** When new slack is generated due to *early completion of jobs* or *removal of recovery jobs*, a new wrapper-task is created. However, it may be merged with an existing element in *WT-Queue* if they have the same deadline. That is, all wrapper-tasks in *WT-Queue* represent slack with different deadlines. Wrapper-tasks in *WT-Queue* are kept in the increasing order of their deadlines.

[4]Although checkpointing could be used for partial recovery [30, 31], we have shown that checkpoints with single recovery section cannot guarantee to preserve task reliability [32].

- **Rule 2 (slack reclamation):** The slack is reclaimed when: (a) a non-scaled job has the highest priority in *Ready-Q* and its reclaimable slack is larger than the job size; or (b) the highest priority job in *Ready-Q* has been scaled (i.e., its recovery job has been reserved) but its speed is higher than f_{ee} and there is reclaimable slack. After the slack is reclaimed, the corresponding wrapper-tasks are removed from *WT-Queue* and destroyed.
- **Rule 3 (slack forwarding/wasting):** the wrapper-tasks of non-reclaimed slack compete for CPU along with ready jobs. When a wrapper-task has higher priority (i.e., earlier deadline) and wraps the execution of a job, the corresponding slack is *pushed forward*; otherwise, if a wrapper-task executes no-ops, the corresponding slack is wasted. Note that, when wrapped execution is interrupted by higher priority jobs, only part of slack (which is consumed by the wrapped execution) will be pushed forward, while the remaining part has the original deadline.

Algorithm 1 EDF-based RA-DPM Algorithm

1: **Step 1:**
2: Suppose that t_{past} is the elapsed time since last scheduling point; J and WT are current job and wrapper-task (could be $NULL$ if no such job or wrapper-task); $J.rem$ and $WT.rem$ are remaining time requirements; and $J.d$ and $WT.d$ are the deadlines;
3: **if** (J!=NULL and $J.rem - t_{past} > 0$) {
4: $J.rem -= t_{past}$;
5: **if** (J completes)
6: CreateWT($J.rem$, $J.d$);//slack of early completion
7: **else** *Enqueue(J, Ready-Q)*;}
8: **if** (WT!=NULL and $WT.rem - t_{past} > 0$) {
9: $WT.rem -= t_{past}$; *Enqueue(WT, WT-Queue)*;}
10: **if** (WT!=NULL and J!=NULL)
11: CreateWT(t_{past}, $J.d$);//push forward slack;
12: **if** (J is scaled and succeeds){
13: RemoveRecoveryJob(J,*Ready-Q*);
14: CreateWT($J.c$, $J.d$);//slack from recovery job;}
15: **Step 2:**
16: **for** (all newly arrived job NJ){ $NJ.rem = NJ.c$;
17: $NJ.f = f_{max}$; *Enqueue(NJ, Ready-Q)*;}
18: **Step 3:**//in the following, J and WT will represent the next job and wrapper-task to be processed, respectively;
19: J=*Dequeue(Ready-Q)*;
20: **if** (J!=NULL) ReclaimSlack(J, *WT-Queue*);
21: WT=*Header(WT-Queue)*;
22: **if** (J!=NULL){
23: **if** ($WT != NULL$ and $WT.d < J.d$)
24: //WT wraps J's execution (a timer is needed)
25: WT = *Dequeue(WT-Queue)*;
26: **else** $WT = NULL$;//normal execution of J
27: Execute(J);}
28: **else if** (WT!=NULL)
29: WT = *Dequeue(WT-Queue)*;//WT executes no-ops

Considering that the execution of real-time jobs may be wrapped by a higher priority wrapper-task, the outline of EDF-based RA-DPM algorithm is shown in Algorithm 1. Note that, RA-DPM may be invoked by three types of events: *job arrival*, *job completion* and *wrapper-task completion* (a timer can be used to signal the completion of a wrapper-task to the operating system). As common routines, we use *Enqueue(J, Q)* to add a job/wrapper-task to the corresponding queues and, *Dequeue(Q)* to fetch the highest priority (i.e., the header) job/wrapper-task and remove it from the queue. Moreover, *Header(Q)* is used to retrieve the header job/wrapper-task without removing it from the queue.

At each scheduling point, as the first step (from line 3 to line 14), the remaining execution time information of the currently running job and wrapper-task (if any) are updated. If they did not finish, put them back to *Ready-Q* and

WT-Queue (lines 7 and 9), respectively. When a wrapper-task (WT) is used and wraps the execution of J (line 11), as discussed before, the corresponding amount of slack (i.e., t_{past}) is pushed forward by creating a new wrapper-task with the deadline of the currently wrapped job. Otherwise, the slack is consumed (wasted).

If the current job completes early (line 6) or its recovery job is removed due to the primary job's successful scaled execution (lines 13 and 14), new slack is generated and corresponding wrapper-tasks are created. Note that, only if the deadline of newly created wrapper-task is different from the ones in *WT-Queue*, can it be added to *WT-Queue*; otherwise, it will be merged with the one that has the same deadline.

Secondly, if new jobs arrive at the current scheduling point, they are added to *Ready-Q* according to their EDF priority (line 17). The remaining timing requirements will be set as their WCETs at the speed f_{max}. The last step is to choose the next highest priority ready job J (if any) for execution (lines 19 to 29). J first tries to reclaim the available slack (line 20; details are shown in Algorithm 2). Then, depending on the priority of the remaining wrapper-tasks, the execution of J may be wrapped by a wrapper-task (line 25) or executed normally (line 26). When a wrapper-task has the highest priority but no job is ready, the wrapper-task executes no-ops (line 29).

Algorithm 2 ReclaimSlack(J, *WT-Queue*)

1: **if**(J is a recovery job) return; //recovery job is not scaled
2: **Step 1:** //collect reclaimable slack
3: $slack = 0$;
4: **for**($WT \in$*WT-Queue*)
5: **if** ($WT.d \leq J.d$) $slack += WT.rem$;
6: **Step 2:** //scale down J if the slack is enough
7: **if** ($!J.scaled$ && $slack <= J.c$) return;
8: **if** ($!J.scaled$) $slack -= J.c$; //reserve for recovery
9: $tmp = \min(f_{ee}, \frac{J.rem * J.f}{slack + J.rem} f_{max})$;
10: $slack = \frac{J.rem * J.f}{tmp} - J.rem$; //slack needed for PM
11: $J.f = tmp$; //new speed
12: **if** ($!J.scaled$){CreateRecoveryJob(J);$slack += J.c$;}
13: $J.scaled = true$; //label as scaled
14: //remove reclaimed slack from *WT-Queue*;
15: **while** ($slack > 0$){
16: WT =*Header(WT-Queue)*;
17: **if** ($slack \geq WT.rem$){
18: WT =*Dequeue(WT-Queue)*;$slack -= WT.rem$;}
19: **else**{$WT.rem -= slack$; $slack = 0$;}
20: }

Algorithm 2 shows the further details of slack reclamation. As mentioned previously, recovery jobs are executed at f_{max} and are not scaled (line 1). For a job J, by traversing *WT-Queue*, we can find out the amount reclaimable slack (lines 3 and 5). If J is not a scaled job (i.e., its recovery job is not reserved yet) and the amount of reclaimable slack is no larger than the size of J (i.e., $J.c$), the available slack is not enough for reclamation (line 7). Otherwise, after properly reserving the slack for recovery (line 8), J's new speed is calculated, which is bounded by f_{ee} (line 9; as discussed in Section 2). The actual amount of slack used by J includes those for energy management (line 10) as well as the slack for recovery job (where the recovery job is created and added to *Ready-Q* in line 12). For the reclaimed slack, the corresponding wrapper-task(s) will be removed from *WT-Queue* and destroyed (lines 15 to 20), which ensures that this slack is *conserved* for the scaled job, even if higher-priority jobs preempt the scaled job's execution later.

4.3 Analysis of RA-DPM

Note that, when all jobs in a task set present their WCETs at run time, there will be no dynamic slack and no wrapper-task will be created. In this case, RA-DPM will perform the same as EDF and generate the same worst case schedule, which is feasible by assumption. However, as some jobs complete early, RA-DPM will undertake slack reclamation and/or wrapped execution, and one needs to show that the feasibility is preserved even after these changes in CPU time allocation of jobs.

In RA-DPM, the slack is reclaimed for *dual* purposes of scheduling a recovery block and slowing down the execution to save energy with DVFS. Similarly, the slack may be added to the *WT-Queue* as a result of early completion of a job/recovery block, or release of the recovery block (in case of a successful, non-faulty completion of a job). However, the feasibility of the resulting schedule is orthogonal to these details; hence, we will not further concern about whether the slack is obtained from a main job or a recovery block, and for what purpose (i.e. recovery or DVFS) it is used.

Recall that, the elements of *WT-Queue* represent the slack of tasks that complete early. These slack elements, while being reclaimed, may be entirely or partially re-transformed to actual workload. Our strategy will consist of proving that, *at any time t during execution, the remaining workload could be feasibly scheduled by EDF, even if all the slack elements in WT-Queue were to be re-introduced to the system*, with their corresponding deadlines and remaining worst-case execution times (sizes). This, in turn, will allow us to show the feasibility of the actual schedule, since the above-mentioned property implies the feasibility even with an over-estimation of the actual workload, for any time t.

Before presenting the proof for the correctness of RA-DPM, we first introduce the concept of *processor demand* and the fundamental result in the feasibility analysis of task systems scheduled by preemptive EDF [2, 16].

Definition 1 *The* processor demand *of a real-time job set Φ in an interval $[t_1, t_2]$, denoted as $h_\Phi(t_1, t_2)$, is the sum of computation times of all jobs in Φ with arrival times greater than or equal to t_1 and deadlines less than or equal to t_2.*

Theorem 2 ([2, 16]) *A set of independent real-time jobs Φ can be scheduled (by EDF) if and only if $h_\Phi(t_1, t_2) \leq t_2 - t_1$ for all intervals $[t_1, t_2]$.*

■

Let us denote by $J(r, e, d)$ a job J that is released at $t = r$, and that must complete its execution by the deadline d, with worst-case execution time e. We next prove the following lemma that will be instrumental in the rest of the proof.

Lemma 1 *Consider a set Φ_1 of real-time jobs which can be scheduled by preemptive EDF in a feasible manner. Then, the set Φ_2, obtained by replacing $J_a(r_a, e_a, d_a)$ in Φ_1 by two jobs $J_b(r_a, e_b, d_b)$ and $J_c(r_a, e_c, d_c)$, is still feasible if $e_b + e_c \leq e_a$, and $d_a \leq d_b \leq d_c$.*

Proof

Since the EDF schedule of Φ_1 is feasible, from Theorem 2, we have $h_{\Phi_1}(t_1, t_2) \leq t_2 - t_1 \forall\ t_1, t_2$. We need to show that $h_{\Phi_2}(t_1, t_2) \leq t_2 - t_1 \forall\ t_1, t_2$.

It is well-known that, when evaluating the processor demand for a set of real-time jobs, one can safely focus on intervals that start at a *job release time* and end at a *job deadline* [2, 16]. Noting that the only difference between Φ_1 and Φ_2 consists in substituting two jobs J_b and J_c for J_a, we first observe that $h_{\Phi_2}(r_x, d_y) = h_{\Phi_1}(r_x, d_y) \leq d_y - r_x$, whenever r_x is a job release time strictly greater than r_a, or d_y is a job deadline strictly smaller than d_a. Hence, we need to consider only the intervals $[r_x, d_y]$ where $r_x \leq r_a$ and $d_y \geq d_a$. By taking into account the fact that $d_a \leq d_b \leq d_c$, the following properties can be easily derived for all possible positionings of d_y with respect to these three deadlines:

- $h_{\Phi_2}(r_x, d_y) = h_{\Phi_1}(r_x, d_y) - (e_a - e_b - e_c)$ if $d_c \leq d_y$,
- $h_{\Phi_2}(r_x, d_y) = h_{\Phi_1}(r_x, d_y) - (e_a - e_b)$ if $d_a \leq d_b \leq d_y < d_c$,
- $h_{\Phi_2}(r_x, d_y) = h_{\Phi_1}(r_x, d_y) - e_a$ if $d_a \leq d_y < d_b \leq d_c$.

Since $e_a \geq e_b + e_c$ by assumption, in all three cases, $h_{\Phi_2}(r_x, d_y) \leq h_{\Phi_1}(r_x, d_y) \leq d_y - r_x$, and the job set Φ_2 is also feasible.

■

Now, we introduce some additional notations and definitions for the execution state of RA-DPM, at time t.

- $J_R(t)$ denotes the set of ready jobs at time t. Each job $J_i \in J_R(t)$ has a corresponding remaining worst-case execution time e_i at time t and deadline d_i. Note that J_i can be seen as released at time t, and with worst-case execution time e_i and deadline d_i.
- $J_F(t)$ denotes the set of jobs that will arrive *after* t, with their corresponding worst-case remaining execution times and deadlines.
- $J_W(t)$ denotes the set of jobs obtained through the *WT-Queue*. Specifically, for every slack element in *WT-Queue* with size s_i and deadline d_i, $J_W(t)$ will include a job $J_i(t, s_i, d_i)$.

Definition 2 *The* Augmented Remaining Workload *of RA-DPM at time t, denoted by* ARW(t)*, is defined as* $J_R(t) \bigcup J_F(t) \bigcup J_W(t)$.

Informally, *ARW(t)* denotes the workload obtained by re-introducing *all* the slack elements in *WT-Queue* to the ready-queue, with their corresponding deadlines. This is clearly an *over-estimation* of the actual workload at time t, since the amount of workload re-introduced by slack reclamation can never exceed $J_W(t)$.

Theorem 3 ARW(t) *can be scheduled by EDF in a feasible manner during the execution of RA-DPM, for every t.*

Proof The statement is certainly true at $t = 0$, when the *WT-Queue* is empty, and the workload can be scheduled in a feasible manner even under the worst-case conditions. So, assume that the statement holds $\forall\, t \leq t_1$.

Note that for $t = t_1, t_1 + 1, \ldots$, *ARW(t)* remains feasible as long as there is no slack reclamation or 'wrapped execution'. This is because, under these conditions, the task with highest priority in the ready queue is executed at every time slot according to EDF – and being an optimal preemptive scheduling policy, it preserves the feasibility of the remaining workload. Also note that, if the ready queue is empty for a given time slot, then the slack at the head of *WT-Queue* is consumed, which corresponds to the fact that *ARW(t)* is updated dynamically according to EDF execution rules.

Let t_2 be the first time instant after t_1, if any, where RA-DPM performs a slack reclamation or starts the "wrapped execution". We denote the head of *WT-Queue* by H at $t = t_2$, with deadline d_H and size e_H. We will show that *ARW()* remains feasible after such a point in both scenarios, completing the proof.

- **Case 1:** At $t = t_2$, slack reclamation is performed through the *WT-Queue*. Assume k units of slack is transferred from H to the job J_A which is about to be dispatched, with deadline $d_A \geq d_H$ and remaining worst-case execution time e_A. Note that this slack transfer can be seen as replacing $J_H(t_2, e_H, d_H)$ in *ARW(t_2)* by two new jobs $J_{H_1}(t_2, k, d_A)$ and $J_{H_1}(t_2, e_H - k, d_H)$; and by virtue of Lemma 1, *ARW(t)* remains feasible after the slack transfer. If, the slack is transferred from multiple elements in *WT-Queue* successively, then we can repeat the argument for the second, third,... elements in the same order.

- **Case 2:** At $t = t_2$, a 'wrapped execution' starts, to end at $t = t_3 > t_2$. We will show that *ARW(t)* remains feasible for $t_2 \leq t \leq t_3$, completing the proof.

  The wrapped execution (i.e., slack forwarding) in the interval $[t_2, t_3]$ is functionally equivalent to the following: in every time slot $[t_i, t_{i+1}]$ in the interval $[t_2, t_3]$, one unit of slack from H (the head of *WT-Queue*) is replaced by another item in *WT-Queue* with size 1, and deadline d_{A_i}, which is the deadline of job J_{A_i} that executes on the CPU in the interval $[t_i, t_{i+1}]$. On the other hand, when seen from the perspective of changes in *ARW(t)*, this is equivalent to the reclaiming by J_{A_i} one unit of slack from H in slot $[t_i, t_{i+1}]$ –even though, in actual execution, this slack unit will not be used because of wrapped execution. As a conclusion, *ARW(t)* remains feasible at every time slot in the interval $[t_2, t_3]$ as slack reclamation on *ARW(t)* was shown to be safe in Case 1 above.

■

Since *ARW(t)* is an over-estimation of the actual workload, we obtain the following conclusion.

Corollary 1 *RA-DPM preserves the feasibility of any periodic real-time task set under preemptive EDF.*

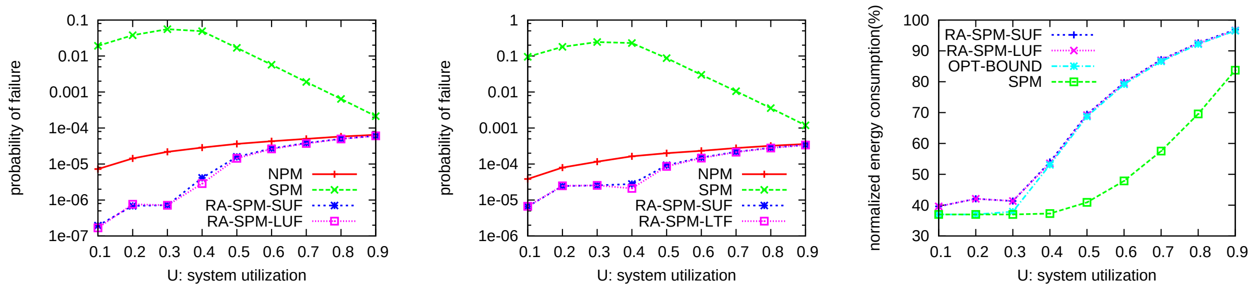
■

4.4 Complexity of RA-DPM

Note that, the deadlines of wrapper-tasks correspond to the deadlines of jobs in the task set considered. At any time t, there are at most n different deadlines corresponding to jobs with release times on or before t *and* deadlines on or after t. That is, the number of wrapper-tasks in *WT-Queue* is at most n. Therefore, slack reclamation can be performed (by traversing *WT-Queue*) in time $O(n)$. Hence, the complexity of RA-DPM is $O(n)$ at each scheduling point.

5 Simulation Results and Discussion

To evaluate the performance of our proposed schemes, we developed a discrete event simulator using C++. In the simulations, we consider six different schemes. First, the scheme of *no power management (NPM)*, which executes all tasks/jobs at f_{max} and puts system to sleep states when idle, is used as the baseline for comparison. The *ordinary static power management (SPM)* scales all tasks uniformly at speed $f = U \cdot f_{max}$ (where U is the system utilization). For the task-level static RA-PM, after obtaining the optimal utilization (X_{opt}) that should be managed, two heuristics are considered: *smaller utilization task first (RA-SPM-SUF)* and *larger utilization task first (RA-SPM-LUF)*. For dynamic schemes, we implemented our *job-level dynamic RA-PM (RA-DPM)* and the *cycle conserving EDF (CC-EDF)* [22], a well-known but reliability-ignorant energy management algorithm.



a. reliability for $p \in [10, 20]$ b. reliability for $p \in [20, 200]$ c. energy for $p \in [10, 20]$

Figure 3: Reliability and energy for static schemes.

We focus on active power and assume $P_{ind} = 0.1$, $C_{ef} = 1$ and $m = 3$. Considering normalized frequency with $f_{max} = 1$, the minimum energy efficient frequency is $f_{ee} = 0.37$ (see Section 2). Transient faults are assumed to arrive according to a Poisson distribution with an average fault rate as $\lambda_0 = 10^{-6}$ at f_{max} (and corresponding supply voltage), which corresponds to 100,000 FITs (failure in time, in terms of errors per billion hours of use) per megabit and is a reasonable fault rate as reported [11, 36]. To take the effects of DVFS on fault rates into consideration, we adopt the exponential fault model developed in [34] and assume that the exponent $d = 2$. That is, the average fault rate is assumed to be 100 times higher at the lowest speed f_{ee} (and corresponding supply voltage). The effects of different values of d have been evaluated in our previous work [32, 33, 34].

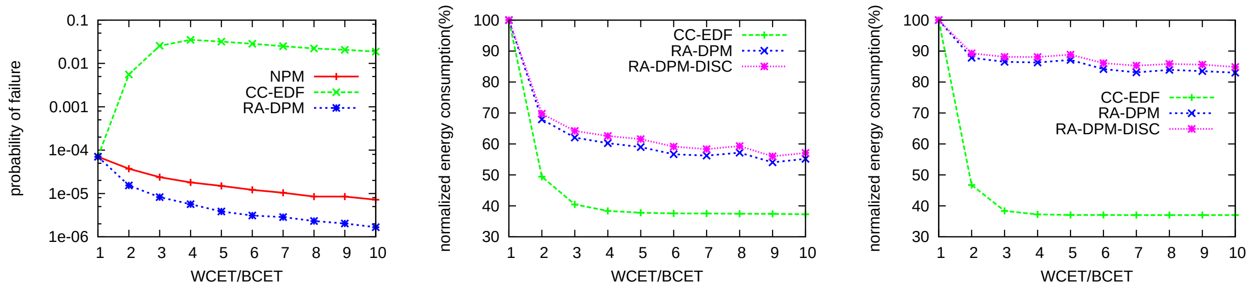
We consider synthetic real-time task sets where each task set contains 20 periodic tasks. The periods of tasks (p) are uniformly distributed within the range of $[10, 20]$ (for short period tasks) or $[20, 200]$ (for long period tasks). The WCETs of tasks are uniformly distributed in the range of 1 and their periods. Finally, the WCETs of tasks are scaled by a constant such that the system utilization of tasks reaches a desired value [22]. The variability in the actual workload is controlled by the $\frac{WCET}{BCET}$ ratio (that is, the worst-case to best-case execution time ratio), where the actual execution time of tasks follows a normal probability distribution function with mean and standard deviation being $\frac{WCET+BCET}{2}$ and $\frac{WCET-BCET}{6}$, respectively [1].

We simulate the task set's execution for 10^7 and 10^8 time units, for short- and long-period task sets, respectively. That is, approximately 20 million jobs are executed at each run. Moreover, for each result point in the graphs, 100 task sets are generated and the presented results correspond to the average.

5.1 Performance of Task-Level Static Schemes

For different system utilization (i.e., spare capacity), we first evaluate the performance of the task-level static schemes. It is assumed that all jobs take their WCETs. Figure 3a first shows the probability of failure (i.e., $1-$*reliability*) for NPM and static schemes for task sets with short periods (i.e., $p \in [10, 20]$). Here, the probability of failure shown is the ratio of the number of failed jobs over the total number of jobs executed.

From the figure, we can see that, as system utilization increases, for NPM, the probability of failure increases slightly. The reason for this is that, with increased total utilization, the computation requirement for each task increases and tasks run longer, which increases the probability of being subject to transient fault(s). The probability of failure for SPM increases dramatically due to increased fault rates as well as extended execution time. Note that, the minimum energy



a. reliability for $p \in [10, 20]$ b. energy for $p \in [10, 20]$ c. energy for $p \in [20, 200]$

Figure 4: Reliability and energy for dynamic schemes.

efficient frequency is $f_{ee} = 0.37$. For very low system utilization (i.e., $U < 0.37$), SPM executes all tasks with f_{ee}. The probability of failure increases slightly with increased utilization due to the same reason as for NPM. However, when system utilization is higher than 0.37, the processing speed of SPM increases with increased utilization, which has lower failure rates and results in decreased probability of failure.

For reliability-aware SPM schemes (i.e., RA-SPM-SUF and RA-SPM-LUF), by incorporating a recovery task for each task to be scaled, the probability of failure is lower than that of NPM and system reliability is preserved, which confirms the theoretical result obtained in Section 3. Note that, with 20 tasks in a task set, the utilization for each task is a small number and is close to each other. Therefore, RA-SPM-SUF and RA-SPM-LUF perform roughly the same.

The probability of failure for long-period task sets is shown in Figure 3b, where all schemes have similar behavior to that of short-period task sets. However, for the same system utilization, long-period task sets will have longer execution time (almost 10 times longer), which leads to roughly 10 times larger probability of failure.

Figure 3c further shows the normalized energy consumption for short-period tasks with NPM as a baseline. Here, reliability-aware SPM schemes consume roughly 20% more energy than that of ordinary SPM due to less spare capacity available for energy management. Moreover, the figure also shows the energy consumption for *OPT-BOUND*, which is calculated as the fault-free energy consumption with the assumption that the managed tasks have the accumulated utilization *exactly* as X_{opt}. Clearly, *OPT-BOUND* provides a performance bound for the *optimal* static solution. Thus, from the figure, we can see that the normalized energy consumption for two heuristics will be within 5% of that for the optimal solution. For long-period tasks, the normalized results are similar, and are not included due to space limitations.

5.2 Performance of Job-Level Dynamic Schemes

With system utilization fixed at $U = 1.0$, we vary $\frac{WCET}{BCET}$ ratio and evaluate the performance of the dynamic schemes. Figure 4a first shows the probability of failure for short period tasks. Here, we can see that, as $\frac{WCET}{BCET}$ ratio increases, more dynamic slack is available and the probability of failure for CC-EDF decreases radically due to scaled execution. Again, by reserving slack for recovery jobs, RA-DPM preserves system reliability by having a lower probability of failure than that of NPM. The result for long period tasks is similar.

Figure 4b shows the normalized energy consumption for short period tasks. Initially, as the ratio of $\frac{WCET}{BCET}$ increases, more dynamic slack is available and normalized energy consumption decreases. Due to limitation of f_{ee}, when $\frac{WCET}{BCET} > 5$, the normalized energy consumption for both schemes stays roughly the same and RA-DPM consumes about 15%

more energy than CC-EDF. RA-DPM performs much worse than CC-EDF for long period tasks (in terms of energy) as shown in Figure 4c. One possible explanation could be that, slack is pushed forward too much by the long period tasks, which prevents other jobs from reclaiming them, and may be wasted.

5.3 Effects of Discrete Speeds

So far, we have assumed that the clock frequency can be scaled in continuous manner. However, on current DVFS-enabled processors (e.g., Intel XScale [12]), there are only a finite number of speed levels. Nevertheless, our schemes can be easily adapted to discrete speed settings. After obtaining the scaled speed (e.g., Algorithm 2 line 9), we can either use two adjacent frequency levels to emulate the task's execution at that speed [14], or use the next higher discrete speed that will always ensure to maintain the feasibility of the algorithm. Assuming Intel XScale model [12] with 5 speeds $\{0.15, 0.4, 0.6, 0.8, 1.0\}$ and using the next higher speed, we re-ran the simulations. The results for normalized energy consumption are represented as *RA-DPM-DISC* and shown in Figure 4bc. From the figures, we can see that, the cases for discrete speeds consume at most 2% more energy than that of continuous speed. The reason is that, with the next higher discrete speed, the unused slack is not wasted but actually saved for future usage.

6 Closely Related Work

In [27], Unsal *et al.* proposed to postpone the execution of backup tasks to minimize the overlap of primary and backup execution and thus the energy consumption. The optimal number of checkpoints, evenly or unevenly distributed, to achieve minimal energy consumption while tolerating one fault was explored by Melhem *et al.* in [21]. Elnozahy *et al.* proposed an *Optimistic TMR* scheme that reduces the energy consumption for traditional TMR systems by allowing one processing unit to slow down provided that it can catch up and finish the computation before the application deadline [7]. The optimal frequency settings for OTMR was further explored in [35]. Assuming a Poisson fault model, Zhang *et al.* proposed an adaptive checkpointing scheme that dynamically adjusts checkpoint intervals for energy savings while tolerating a fixed number of faults for a single task [30]. The work is further extended to a set of periodic tasks [31].

Most of the previous research either focused on tolerating fixed number of faults [7, 21] or assumed constant fault rate [30, 35] when applying DVFS for energy savings. However, it has shown that there is a direct and negative effect of voltage scaling on the rate of transient faults [8, 34]. In our recent work, we presented a reliability-aware power management scheme based on single-task model [32], which is extended to a set of aperiodic tasks sharing a common deadline [33].

7 Conclusions

Although an efficient energy management technique, dynamic voltage and frequency scaling (DVFS) was recently shown to have negative impact on settings where transient faults become more prominent with continued scaling of CMOS technologies and reduced design margins. For mission critical applications, where system reliability may be more important than energy consumption, *reliability-cognizant* energy management becomes a necessity. Based on our previous research, which concluded that system reliability can be preserved by scheduling suitable recovery tasks before applying DVFS [32], in this work, we proposed for the first time a reliability-aware energy management (RA-PM) framework for *periodic* tasks. Focusing on EDF scheduling, we first studied *task-level* utilization-based static RA-PM schemes that exploit the spare capacity in the system. We showed the *intractability* of the problem and proposed two efficient heuristics. Moreover, we proposed the *wrapper-task* mechanism for efficiently managing dynamic slack and presented a *job-level* dynamic RA-PM scheme. The scheme is able to *conserve* the slack reclaimed by a scaled job, which is an essential requirement for reliability preservation, across preemption points.

The proposed schemes are evaluated through simulations with synthetic real-time workloads. The results show that, compared to those of ordinary energy management schemes, the new schemes could achieve significant amount of energy savings while preserving system reliability. However, ordinary energy management schemes that are *reliability-ignorant*, often lead to drastically decreased system reliability.

References

[1] H. Aydin, R. Melhem, D. Mossé, and P. Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proc. of IEEE Real-Time Systems Symposium*, 2001.

[2] S. Baruah, R. Howell, and L. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2, 1990.

[3] E. Bini, G.C. Buttazzo, and G. Lipari. Speed modulation in energy-aware real-time systems. In *Proc. of the* 17^{th} *Euromicro Conference on Real-Time Systems (ECRTS)*, 2005.

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

[5] M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proc. of Real-Time Systems Symposium*, 2000.

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

[7] E. (Mootaz) Elnozahy, R. Melhem, and D. Mossé. Energy-efficient duplex and tmr real-time systems. In *Proc. of The* 23^{rd} *IEEE Real-Time Systems Symposium*, Dec. 2002.

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

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

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Mathematical Sciences Series. Freeman, 1979.

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

[12] http://developer.intel.com/design/intelxscale/.

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

[14] T. Ishihara and H. Yauura. Voltage scheduling problem for dynamically variable voltage processors. In *Proc. of The Int'l Symposium on Low Power Electronics and Design*, 1998.

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

[16] K. Jeffay and D. L. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proc. of the IEEE Real-Time Systems Symposium*, Dec. 1993.

[17] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proc. of the* 41^{st} *annual Design automation conference*, 2004.

[18] A. R. Lebeck, X. Fan, H. Zeng, and C. S. Ellis. Power aware page allocation. In *Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[19] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proc. of IEEE Real-Time Systems Sympoisum*, 1992.

[20] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM*, 20(1):46–61, 1973.

[21] 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.

[22] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proc. of* 18^{th} *ACM Symposium on Operating Systems Principles*, Oct. 2001.

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

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

[25] N. Seifert, D. Moyer, N. Leland, and R. Hokinson. Historical trend in alpha-particle induced soft error rates of the alpha^{TM} microprocessor. In *Proc. of the* 39^{th} *Annual International Reliability Physics Symposium*, 2001.

[26] K. Seth, A. Anantaraman, F. Mueller, and E. Rotenberg. Fast: Frequency-aware static timing analysis. In *Proc. of the* 24^{th} *IEEE Real-Time System Symposium*, 2003.

[27] O. S. Unsal, I. Koren, and C. M. Krishna. Towards energy-aware software-based fault tolerance in real-time systems. In *Proc. of The Int'l Symposium on Low Power Electronics Design*, 2002.

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

[29] Heng Zeng, Xiaobo Fan, Carla Ellis, Alvin Lebeck, and Amin Vahdat. Ecosystem: Managing energy as a first class operating system resource. In *Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

[30] 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.

[31] Y. Zhang, K. Chakrabarty, and V. Swaminathan. Energy-aware fault tolerance in fixed-priority real-time embedded systems. In *Proc. of Int'l Conference on Computer Aided Design*, Nov. 2003.

[32] 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*, 2006.

[33] D. Zhu and H. Aydin. Energy management for real-time embedded systems with reliability requirements. In *Proc. of the International Conference on Computer Aidded Design (ICCAD)*, Nov. 2006.

[34] 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 Conference on Computer Aidded Design*, 2004.

[35] D. Zhu, R. Melhem, D. Mossé, and E.(Mootaz) Elnozahy. Analysis of an energy efficient optimistic tmr scheme. In *Proc. of the* 10^{th} *Int'l Conference on Parallel and Distributed Systems*, 2004.

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