Energy-Aware Standby-Sparing Technique for Periodic Real-Time Applications

Mohammad A. Haque, Hakan Aydin
Department of Computer Science
George Mason University
Fairfax VA 22030
mhaque4@gmu.edu, aydin@cs.gmu.edu

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

***Abstract*—In this paper, we present an energy-aware standby-sparing technique for periodic real-time applications. A standby-sparing system consists of a primary processor where the application tasks are executed using Dynamic Voltage Scaling (DVS) to save energy, and a spare processor where the backup tasks are executed at maximum voltage/frequency, should there be a need. In our framework, we employ Earliest-Deadline-First (EDF) and Earliest-Deadline-Late (EDL) scheduling policies on the primary and spare CPUs, respectively. The use of EDL on the spare CPU allows delaying the backup tasks on the spare CPU as much as possible, enabling energy savings. We develop static and dynamic algorithms based on these principles, and evaluate their performance experimentally. Our simulation results show significant energy savings compared to existing reliability-aware power management (RAPM) techniques for most execution scenarios.**

***Index Terms*—Energy Management, Real-Time Systems, Reliability, Dynamic Voltage Scaling, Standby-Sparing, EDF Scheduling.**

I. Introduction

Energy management is a frequent design concern for real-time embedded systems. *Dynamic Voltage Scaling* (DVS) is a widely studied technique to reduce energy consumption, involving simultaneous scaling of the CPU supply voltage and frequency [1]. Another approach is *Dynamic Power Management* (DPM), where the system components are put to idle/sleep states when they are not in use [2]. The real-time execution semantics mandate that the timing constraints (the *feasibility* requirement) be met at run-time even when task response times may increase as a consequence of the energy management techniques [3].

In addition to feasibility and energy management, reliability and fault tolerance are other important objectives for real-time embedded systems. Computer systems are subject to different types of faults [4]. Faults may occur at runtime due to various reasons, including hardware defects, electromagnetic interference, and cosmic ray radiations. *Permanent faults* may bring a system component (e.g. the processor) to halt and cannot be tolerated without some form of hardware redundancy. *Transient faults*, on the other hand, are not persistent – they are often triggered by electromagnetic interference and cosmic ray radiations and disappear within a short time interval. Transient faults are known to occur more frequently [5], [6] and they raise growing concerns with the increase of component density in the CMOS circuits [7]. Restoring the system to a previous safe state and repeating the computation is a common approach to deal with the transient faults [4], [8].

Recent research suggests that the transient fault rates increase significantly in systems where the supply voltage is scaled down to save energy [7], [9]. As a result, a number of energy management techniques that also consider reliability were recently proposed [10]–[13]. These works, generally called as the *Reliability-Aware Power Management (RAPM)* framework, use time-redundancy for both DVS and reliability preservation. RAPM techniques generally assume that the system's *original* reliability, which is defined as the probability of completing all tasks successfully (without incurring transient faults), is acceptable when all the tasks are executed at the maximum CPU frequency. Hence, existing RAPM techniques schedule a separate recovery job for every job that has been slowed (scaled) down. The recovery job is executed only when the corresponding scaled job incurs a transient fault, and at the maximum frequency [13]. It is known that this approach yields a reliability level which is no less than the system's original reliability, thereby *preserving* that level [13]. Most of the existing RAPM techniques are developed for single CPU systems and target preservation of the reliability only with respect to transient faults. As a result, they cannot address permanent faults that can effect the CPU. In addition, due to the need to schedule a separate recovery job for every scaled job on a single CPU, these techniques cannot handle large tasks with utilization greater than 50% (i.e. those tasks have to run at the maximum frequency). Very recently, a technique that explored application of the RAPM on multiprocessor systems in conjunction with global scheduling techniques (allowing migration) has been proposed in [14].

In recent work, Ejlali et al. [15] proposed a novel technique for co-management of energy and reliability in the form of a *standby-sparing* system. The solution combines hardware redundancy with DVS to save energy while preserving the system's original reliability and offer opportunities to tolerate permanent processor faults. Specifically, the technique deploys two processors, named *primary* and *spare*, respectively. The application tasks are executed on the primary processor using DVS. The spare does not use DVS and is reserved for executing *backup* tasks, if needed. Upon the successful completion of a primary task, the corresponding backup task is cancelled and excessive energy consumption is avoided. However, should the primary task fail, the backup runs to completion at the maximum frequency. Hence, the solution statically constructs the schedule on the spare CPU to delay the backups as much as possible to minimize the overlap between the primary and backup copies of the same task. The technique is also based on the observation that many real-time jobs complete early in actual executions [3], [13], [16], [17] and hence the backups are often cancelled without even starting to execute – this helps to improve the energy savings. In addition, more static and dynamic slack for slow-down is available on the primary processor where no CPU time is reserved for recovery.

Despite these innovative aspects, the solution presented in [15] has some important limitations. First, it is limited to *non-preemptive* and *aperiodic* jobs for which a static schedule on a single processor is already given. Then starting from the latest deadline and moving backward, the latest time to initiate backup task on the spare, and the speed for the primary task, are determined in a mechanical way. However, most of the real-time applications on embedded devices are *periodic* in nature and these techniques are not directly applicable for a wide-spectrum of scenarios. In particular, for periodic applications that are invoked repetitively, it is not trivial to determine the latest time for initiating the corresponding backup tasks in the absence of a finite-horizon schedule that can be easily determined for aperiodic jobs. Second, periodic applications are often executed by *preemptive* scheduling policies as non-preemptive execution may result in arbitrarily low system utilization [18]. In preemptive execution settings, the greedy speed assignment technique of [15] cannot be readily applied.

The main contribution of this paper is the proposal for a standby-sparing solution in energy-aware scheduling of preemptive periodic real-time applications. By considering a general system-level power model, we show how the schedules on both CPUs and the frequency assignments for the primary tasks can be determined for periodic preemptive executions. Specifically, we apply the *Earliest-Deadline-First (EDF)* policy on the primary CPU with DVS, while the backups are executed on the spare CPU according to the *EDL (Earliest-Deadline-Late)* policy [19]. Both EDF and EDL assign priorities based on the job deadlines, however EDL *delays* the jobs as much as possible to obtain idle intervals as early as possible in the schedule [19]. We use this important feature of EDL to postpone the execution of backups on the spare CPU and to determine the frequency assignment efficiently for the primary at job dispatch times.

Due to its dual-CPU structure, our solution can withstand the permanent fault of a single CPU. Also, since the backups execute without voltage scaling, the system's original reliability (in terms of resilience with respect to transient faults) is preserved [13]. Finally, the joint use of EDF and EDL on the primary and secondary CPUs allows us to minimize the overlap between the two copies at run-time and reduce the energy cost due to the backup executions. Based on these principles, we present two variants, *Aggressive Standby-Sparing Technique (ASSPT)* and *Conservative Standby-Sparing Technique (CSSPT)*, that differ on the way they use the available slack at run-time for frequency assignment. Our experimental evaluation underlines energy savings potential of our solutions, in addition to their clear benefits on the reliability side.

The rest of this paper is organized as follows. Section II presents our system model. Section III introduces the necessary background information for reliability-aware and standby-sparing scheduling frameworks, as well as the main principles of our approach. Section IV provides the details of our solution. In Section V, we evaluate the energy consumption of our algorithms for various settings. Concluding remarks are provided in Section VI.

II. System Model

We consider a set of periodic real-time tasks $\Psi = \{T_1, ..., T_m\}$. Each periodic task T_i has worst-case execution time c_i under the maximum CPU frequency, and the period p_i. The *hyperperiod* H is defined as the least common multiple of task periods $p_1, \ldots, p_n$. The relative deadline of task T_i is assumed to be equal to its period. The j^{th} job of T_i, denoted as J_{ij}, arrives at time $(j-1)\cdot p_i$ and has a deadline of $j \cdot p_i$. The utilization of a task T_i, u_i, is defined as $\frac{c_i}{p_i}$. The total utilization U_{tot} is the sum of all the individual task utilizations.

We consider a standby-sparing system that consists of a primary CPU and a spare CPU [4], [15]. The main copy of each job J_{ij} runs on the primary CPU, which is assumed to have the DVS capability, while the backup copy, denoted by B_{ij}, runs on the spare CPU without any voltage scaling (i.e. at the maximum frequency). The frequency f of the primary CPU is adjustable up to a maximum frequency f_{max}. We normalize all frequency values with respect to f_{max}, i.e. $f_{max} = 1.0$. A job J_{ij} may take up to $\frac{c_i}{f}$ time units when executed at frequency f. Note that the backup B_{ij} takes at most c_i time units since it is executed without voltage scaling.

We adopt a system-level power model used in previous reliability-aware power management research [13]. Specifically, the power consumption consists of a static and a dynamic component. The static power, P_s, is dominated by the leakage current of the system. The dynamic power P_d includes a frequency-dependent power component P_{dep}, and a frequency-independent power component P_{ind}, driven by the modules such as memory and I/O subsystem in the active state.

$$P = P_s + P_d \tag{1}$$

$$P_d = P_{ind} + P_{dep} \tag{2}$$

The frequency-dependent component P_{dep} of the power consumption can be expressed as:

$$P_{dep} = C_e \cdot V_{dd}^2 \cdot f \tag{3}$$

where, C_e is the system's effective switching capacitance. The processor supply voltage V_{dd} has an almost linear relationship with frequency f. Therefore, Equation (3) can be rewritten as,

$$P_{dep} = C_e \cdot f^3 \tag{4}$$

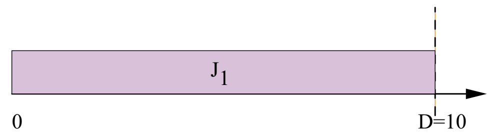


Fig. 1. Static power management for a single task

The energy consumption of a processor from $t = t_1$ to $t = t_2$ is defined as $\int_{t_1}^{t_2} P(t)dt$ where $P(t)$ is the power consumption at time t. As in [13], we assume that due to periodic execution settings, turning on and off the system at run-time is not feasible, hence the static power P_s is not manageable. Hence, our solutions focus on managing the dynamic power P_d. Existing research indicates that arbitrarily slowing down a task is not always energy-efficient [20]–[22], due to the existence of the frequency-independent dynamic power P_{ind}. In other words, there exists a processing frequency below which the total energy consumption of a task increases. This frequency is called the *energy-efficient speed* (f_{ee}) and it can be computed analytically in advance [9], [21].

At job completion times, an *acceptance (or, consistency) test* is performed to determine the occurrence of transient faults [4]. If a fault is detected, we can still continue with the backup job on the spare CPU. Otherwise we can immediately cancel or terminate the backup task. The cost of running the acceptance test is assumed to be included in the worst-case execution time of jobs [13].

III. PRELIMINARIES

In this section, we first present the fundamentals and a comparison of RAPM and energy-aware standby-sparing techniques on a simple example with a single job. Then, we introduce the main principles and mechanism of our solution for general preemptive periodic tasks.

Consider a single job J_1 with deadline $D = 10$ ms and worst-case execution time of $c = 4$ ms under f_{max}. Assuming $f_{ee} < 0.4$, we observe that we can maximize energy savings by executing the job at $f = 0.4$ as shown in Fig. 1. Using the parameters from [9], [13] by setting the transient fault rate at f_{max} to $\lambda_0 = 10^{-7}$, and fault rate sensitivity to voltage scaling $d = 2$, we can compute the *probability of failure* (defined as *1 - reliability*) as 2.15×10^{-8} which represents a reliability degradation of two orders of magnitude compared to the *original* probability of failure figure of 4×10^{-10}.

To preserve the original reliability while applying DVS, RAPM schedules a *recovery job* before the deadline when scaling down the job [13]. The recovery job is executed only when a transient fault is detected at the end of J_1. Given the low probability of occurrence of transient faults, only 4 ms of CPU time is reserved for recovery, implying that it may be executed at f_{max} if needed. Consequently, the execution of J_1 is slowed down only to $f = \frac{4}{6} = 0.66$ in the RAPM solution. [13] indicates that this solution is guaranteed to preserve the original reliability of the job.

Despite this important feature, a number of observations are in order. First, tasks with utilization larger than 50% cannot be managed by RAPM, since it is not possible to assign a full and separate recovery to such jobs. Even for small tasks, reserving recovery time on a single CPU system affects the slow-down and energy savings prospects. Second, by its very nature, RAPM cannot be used to withstand the permanent fault of the single CPU.

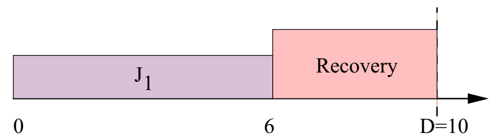


Fig. 2. RAPM for a single task

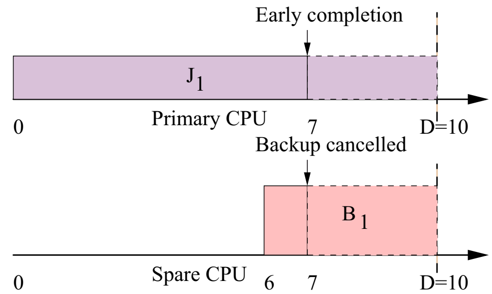


Fig. 3. Standby-sparing system for a single task

The standby-sparing technique [15] is an alternative with some interesting features. As seen in Fig. 3, the technique uses two processors, effectively with the potential of tolerating the permanent fault of a single CPU. The first (primary) processor executes the job J_1 at $f = 0.4$; and the second (spare) processor is used to execute the backup job B_1. By executing B_1, if needed, at the maximum frequency the technique preserves the original reliability as RAPM does. Moreover, observe that the start time of B_1 is delayed as much as possible to minimize the overlap with the primary. This is very helpful for saving energy, since if/when J_1 completes successfully without presenting its worst-case workload (as shown by the completion of J_1 at $t = 7$ in Fig. 3), the backup can be immediately cancelled.

In [15], the authors assumed that a static schedule from $t = 0$ to the latest deadline is already given for a set of *aperiodic* jobs that are executed in *non-preemptive* fashion. This assumption allows determining the scaled frequency on the primary at each job dispatch time through a greedy technique. However, for periodic tasks that may generate large number of jobs executed by a dynamic-priority event-driven policy such as EDF, and that may be subject to preemptions at arbitrary times, this technique is not directly applicable.

As to the potential of the standby-sparing technique for preemptive periodic applications, consider the execution of two periodic tasks T_1 and T_2. The period and worst-case execution time of T_1 are given by 20 and 8, respectively. Similarly, the period and worst-case execution time of T_2 are given by 50 and 20, respectively. The total utilization of the task set is 0.8 and the well-known results [3], [17] suggest that the tasks can be executed at $f = 0.8$ to save energy while

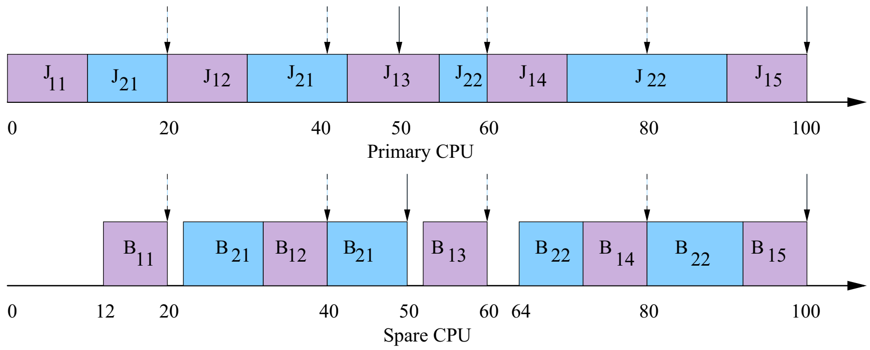


Fig. 4. Standby-sparing system for periodic tasks

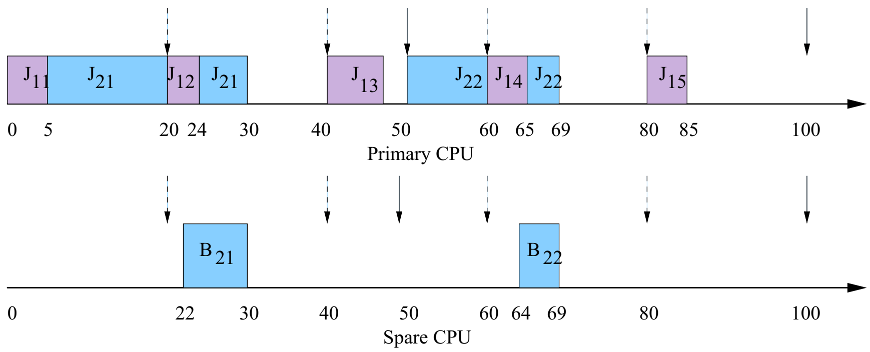


Fig. 5. Taking advantage of early completions (fault-free execution)

meeting all the deadlines by the preemptive Earliest-Deadline-First (EDF) policy during the hyperperiod $H = 100$, which is the least common multiple of p_1 and p_2.

In fact, this is precisely the solution adopted for the primary CPU, as shown on the top schedule of Fig. 4. Note that there are 5 jobs of T_1 and 2 jobs of T_2 within the hyperperiod. The schedule at the bottom of Fig. 4 shows the solution for the spare CPU. Observe that there is a separate backup job B_{ij} corresponding to each primary job J_{ij}. Moreover, the backup jobs are delayed as much as possible, and occasionally are preempted themselves, on the spare CPU. We will shortly discuss how to obtain this ideal "delayed" schedule on the spare. As discussed previously, such a delayed execution pattern is very useful as the backups can be cancelled as soon as the corresponding primaries complete at run-time successfully.

The benefits of this approach are further emphasized by the potential of early completions on the primary: Figure 5 shows a fault-free scenario, where tasks complete early. We observe that the backup tasks can be avoided entirely in many scenarios. For instance, J_{11} completes early at time 5. Since B_{11} was scheduled to start at time 12, it can be cancelled without even starting. As a result, the primary immediately dispatches J_{21}. J_{21} is preempted at time 20, but its backup is still dispatched at $t = 22$, its scheduled startup time. When J_{21} completes early at $t = 30$, we can cancel the remaining part of the backup task, B_{21}, immediately. The rest of the schedules, which contain other early completions, can be easily followed. Comparing the schedules of the spare CPU on Fig. 4 and 5, we note that only two backup jobs (out of seven) are partially executed during the actual execution with early completions. Note that early completions give also potential for dynamic reclaiming [3] on the primary CPU; we will explain in the next section how our technique benefits from this dimension as well.

Now we focus on how to determine the schedule on the spare CPU in a compact way. In one of the influential studies, Chetto et al. investigated some important properties of periodic real-time schedules when priorities are assigned according to the deadlines (the smaller the deadline, the higher the priority) [19]. They distinguished between the traditional EDF (called EDS in [19]) that executes ready tasks as soon as possible, and a variant, called *Earliest Deadline Late (EDL)*, that delays the tasks as much as possible while preserving feasibility. An important advantage of EDL is the fact that it creates idle intervals early in the schedule by delaying periodic tasks. This feature has been exploited for various purposes in the literature; for example, to execute soft real-time jobs as soon as possible without compromising the timeliness of periodic hard real-time tasks [18]. Both EDF and EDL are "optimal" in the sense that all real-time jobs can meet their deadlines under each preemptive policy whenever it is possible to do so.

To compute the idle intervals for any schedule S for a task set ψ, an availability function δ is defined as follows.

$$\delta_{\psi}^{S}(t) = \begin{cases} 1 & \text{if the processor is idle at time t} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

The total amount of time the processor is idle for any time interval $[t_1,\ t_2]$ can be easily obtained by taking the sum of δ_{ψ}^{S} over the interval $[t_1,\ t_2]$ and we denote this sum by $\Delta_{\psi}^{S}(t_1,\ t_2)$.

Theorem 1: (From [19]) Let ψ be any periodic task set and S is a preemptive scheduling policy. At any time instant t,

$$\Delta_{\psi}^{EDL}(0,\ t) \geq \Delta_{\psi}^{S}(0,\ t) \qquad (6)$$

Theorem 1 suggests that, EDL pushes the periodic tasks as much as possible to maximize the idle intervals in the earlier parts of the schedule. *This makes EDL a very good candidate for the spare processor, as we want to delay the backup tasks as much as possible.* Moreover, [19] presents a recursive formulation for determining the idle intervals in the EDL schedules. Let α be a vector $\{a_1, \ldots, a_z\}$ denoting the lengths of consecutive idle intervals during the hyperperiod H in an EDL schedule. It is known that, in an EDL schedule, idle intervals always appear at the release time of jobs [19]. As a result, there are at most $z = \sum_{i=1}^{n} \frac{H}{p_i}$ elements in the idle interval vector α and they can be computed offline for use at run-time, in advance.

IV. Our Solution: Algorithm SSPT

In this section we present the details of our *Standby-Sparing for Periodic Tasks (SSPT)* algorithm. On the primary CPU, the application tasks are executed by the EDF policy, using DVS whenever there is static or dynamic slack. The spare CPU is reserved for the execution of backups following the EDL schedule. The EDL schedule (Section III) is computed in the offline pre-processing phase and the vector of idle intervals α is constructed at the same time. Recall that EDL schedule indicates by how much periodic tasks can be delayed while still meeting their respective deadlines, if necessary by executing at full speed. The backups are dispatched at times indicated by the EDL schedule, and they are cancelled as soon as the corresponding primary completes successfully. If a transient fault is detected at the end of a primary copy, the backup is allowed to complete its execution.

The scaled frequency of each job on the primary is determined at its dispatch time. An important property of our algorithm is that it attempts to reduce the overlap between the primary and backup copies of a given job, since during such an overlap region the energy consumption would be high due to execution at f_{max} on the spare CPU. Specifically, when dispatching the job J_{ij} on the primary at time t, the algorithm computes the aggregate amount of idle intervals from t to d_{ij} in the EDL schedule (where d_{ij} is the deadline of job J_{ij}): this sum represents the *slack* that can be used by the primary copy before its deadline to reduce its frequency below f_{max}. Note that the intervals reserved in the EDL schedule for the backups are not considered during the computation of the available slack for slowdown: this is a critical feature of the algorithm. Figure 6 illustrates the dispatch-time frequency assignment scheme used by the SSPT algorithm. At time t, J_{ij} is dispatched on the primary. There are two idle intervals in the backup schedule before the deadline of J_{ij}. Hence, J_{ij} can reclaim $x_1 + x_2$ units of static slack for slow-down.

This mechanism also allows to integrate the management of dynamic slack, that can arise from early completions, to the framework in seamless fashion. Specifically, when a primary job J_{ij} completes early on the primary, we can treat the remaining time slots for the corresponding backup B_{ij} in the EDL schedule as de facto idle intervals after that point. This increases slow down opportunities for future jobs on the primary. Returning to our example in Fig. 6, if B_{kl} had been cancelled prior to time t, we could have used the time allocated for B_{kl} to further slow down J_{ij}. However, note that no job is slowed down below the energy-efficient frequency threshold. Also, the spare processor does not employ any slack reclaiming technique and follows the pre-computed EDL schedule.

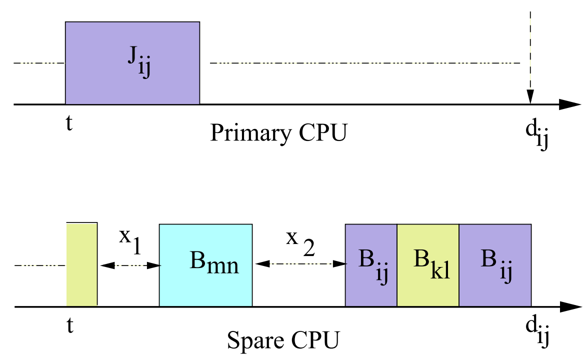


Fig. 6. Determining the frequency on the primary at job dispatch time

We now discuss the steps of our algorithm. It is assumed that during the pre-processing phase, an *Initialization* function, which computes the EDL schedule to be used by the spare and the vector of idle intervals, α, is invoked. The spare then executes all the (non-cancelled) backup jobs at f_{max} according to the static EDL schedule. At run-time, whenever a job is released, it is dispatched immediately if the primary processor is idle. Otherwise, a job is dispatched only if it has a higher priority than the currently executing job according to EDF policy. In that case, the current job is preempted. During preemption we update the minimum additional time required to complete the job (w_k) in the worst-case (under maximum frequency). The *Dispatch* procedure first checks for available static and dynamic slack to reclaim. This is achieved by computing the Δ_{ψ}^{EDL} value through the α vector. Every time a job is dispatched or resumed after preemption at time t, it can use the idle intervals between t and its deadline for slowdown. If no such idle interval exists, the job runs at f_{max} on the primary. At completion time of a primary job, we initiate the corresponding acceptance test. If no error is detected, we can cancel the backup on the spare immediately. In case of an error, we can continue running the backup as scheduled. If a primary job completes early, the dynamic slack is added as idle interval to the α vector. The *Update_alpha* procedure

Algorithm Standby-Sparing for Periodic Tasks (SSPT)

Event - J_{ij} is released at time t:
$w_i \leftarrow c_i$ /* remaining execution time of J_{ij} at f_{max}*/
if the processor is idle **then**
Dispatch(J_{ij}, t)
else /* J_{kl} is running */
if $priority(J_{ij}) > priority(J_{kl})$ **then**
/* preemption case */
$w_k \leftarrow w_k - (\Gamma_k \times f_k)$
/* Γ_k : execution time of J_{kl} in current dispatch */
Dispatch(J_{ij}, t)
end if

Event - J_{ij} completes at time t:
Run the acceptance test
if no error is detected **then**
Cancel the backup B_{ij} on the spare processor
Update_alpha(J_{ij}, t)
end if

Function *Dispatch(J_{ij}, t)*
$slack \leftarrow \Delta_{\psi}^{EDL}(t,\ deadline(J_{ij}))$
$f_i \leftarrow$ *Set_Speed(w_i, slack)*
$Dispatch\ J_{ij}$ on the primary processor at frequency f_i
end Function

performs this operation. If the backup B_{ij} is scheduled to run for b units of time at time t' in the EDL schedule, and it is subsequently cancelled, we add $(\alpha_{t'} = b)$ to the α vector. The updated α vector is later used to determine the total idle time, $\Delta_{\psi}^{EDL}(t_1,\ t_2)$ for any interval $[t_1,\ t_2]$.

The *Dispatch* procedure invokes a *Set_Speed* procedure that takes into account the remaining execution time requirement of task T_i under maximum frequency (namely, w_i) andavailable slack (denoted by the variable *slack* in the algorithm) to determine the frequency assignment for the job. The *Set_Speed* procedure can use different heuristics to determine the *exact* amount of slack to allocate to the job at dispatch time. In this work, we focus on two efficient heuristics for slack distribution.

Aggressive SSPT: We can allow a job to utilize the entire available slack and slow down as much as possible. This is based on the assumption that the job is also likely complete early, leaving sufficient slack for the following jobs. Hence, the frequency for ASSPT is determined as:

$$f_i = \max\{f_{ee}, \frac{w_i}{w_i + slack}\} \tag{7}$$

Conservative SSPT: In many cases, it is possible to have information about the *average-case* workload of the applications, in addition to the worst-case. In this scheme, a job is not allowed to run at a frequency lower than the average-case total utilization, U_{avg} of the task set. Note that $f = U_{avg}$ corresponds to the optimal frequency under the average-case behavior. In other words, CSSPT tries to achieve a balanced slack distribution among all jobs. The frequency assignment for CSSPT is given as:

$$f_i = \max\{f_{ee}, U_{avg}, \frac{w_i}{w_i + slack}\} \tag{8}$$

V. Experimental Evaluation

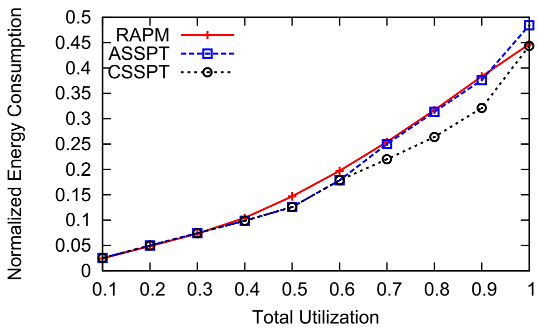
In this section, we present experimental results to demonstrate the performance of the SSPT framework. Our evaluation involves comparison of the two SSPT variants, namely ASSPT and CSSPT, against the RAPM technique proposed for periodic task sets in [13]. Notice that a direct comparison against the original energy-aware standby-sparing study [15] is not possible; because [15] can deal neither with periodic task sets nor preemptive scheduling, as explained in Section III. Therefore, we limit our comparison to RAPM only.

Note that in terms of resilience and fault tolerance, SSPT has clear advantages over RAPM. First, due to the inherent hardware redundancy, SSPT can potentially continue to execute the task set even if one of the processors is subject to a *permanent* fault. Second, SSPT deploys a separate backup task for every periodic task in the application, while RAPM schedules recoveries only for scaled tasks. Since the advantages of SSPT over RAPM on this dimension are clear, this section evaluates the energy consumption figures of the algorithms.

We designed a discrete event simulator using C++ to compare the energy performance of the schemes. We evaluated the effect of the system load (given by the task set utilization U_{tot}) and the workload variability, on the system's total energy consumption. For each data point, we generated 1000 synthetic task sets each with 10 tasks. The utilization of each task is generated using the *UUnifast* scheme proposed in [23]. The task periods are generated randomly in the range $[10ms, 100ms]$.

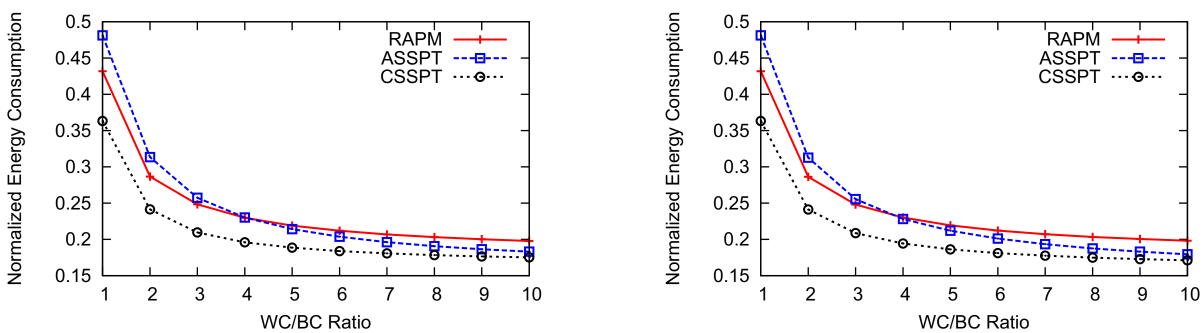
To model the workload variability, we generated the actual workload of each job according to probability distributions at run-time. Specifically, the actual execution time of each job (under f_{max}) varies between a worst-case WC and a best-case BC value. We varied the ratio $\frac{WC}{BC}$ from 1 to 10 to investigate the impact of the workload variability. Clearly, the higher this ratio, the more the actual workload deviates from the worst-case. While generating the actual workload of each job, we considered *uniform* and *normal* distributions. In case of the *normal* distribution, the mean is set to $(WC+BC)/2$, and the standard deviation is set to $((WC-BC)/6)$. The latter choice ensures that 99.7% of the actual execution times lie within the worst-case and best-case execution time range for the job [3] (values beyond the $[BC, WC]$ range are not considered).

The frequency-dependent power consumption is assumed to be a cubic function of the CPU frequency and equal to unity at f_{max}. The frequency-independent power P_{ind}, and the static power P_s are set to 10% and 5% of the maximum



(a) Uniform Distribution, $\frac{WC}{BC} = 5$ (b) Normal Distribution, $\frac{WC}{BC} = 5$

Fig. 7. Impact of System Utilization



(a) Uniform Distribution, $U_{tot} = 0.7$ (b) Normal Distribution, $U_{tot} = 0.7$

Fig. 8. Impact of Workload Variability

frequency dependent power, respectively. The energy figures are normalized with respect to *No Power Management (NPM)* scheme, which executes all the tasks on both CPUs at the maximum frequency f_{max}.

We first examine the impact of the system utilization when $\frac{WC}{BC} = 5$ for uniform (Figure 7(a)) and normal (Figure 7(b)) distributions. In general, with increasing utilization, the slow-down opportunities with DVS are less due to the feasibility requirement, and the energy consumptions of all schemes increase. CSSPT and ASSPT outperform RAPM for most of the spectrum. The difference between ASSPT and RAPM diminishes once the utilization exceeds 0.7; but CSSPT exhibits superior performance and energy gains up to 15%. This is because, CSSPT distributes the slack in a fairer fashion among jobs, in contrast to ASSPT that slows down each job as much as possible without considering other ready or future jobs. ASSPT also has typically more overlap between the primary and the spare, as jobs on the primary run at a lower speed and take longer to complete. Note that energy savings are marginally higher when the actual workload follows a normal distribution as shown in Figure 7(b).

Figures 8(a) and 8(b) show the impact of workload variability on the performance of the algorithms when the actual execution time follows a uniform and normal distribution, respectively. For this set of experiments, the total utilization of the system is set to 0.7. The results show that, both CSSPT and ASSPT outperform RAPM for most of the execution scenarios. When the WC/BC ratio is set to 1, the performance of each scheme is primarily determined by the available static slack in the system. In that case CSSPT performs best due to its balanced approach in determining the slow-down factor, and RAPM performs better than ASSPT which aggressively slows down each dispatched job at the expense of using high frequency for other jobs. As the WC/BC ratio increases, there is more dynamic slack for the algorithms to reclaim and the energy consumption decreases correspondingly. Note that the relative performance of RAPM deteriorates at large WC/BC ratios, as it requires a separate recovery job for each scaled job. CSSPT shows consistently superior performance for the entire spectrum. Note that even with large WC/BC ratios, the average execution time of each job is bounded by WC/2 and the energy consumption figures of all schemes show only marginal decrease after a certain point.

VI. Conclusions

In this paper, we explored a hardware redundancy technique for periodic real-time tasks based on standby-sparing technique. The main contribution of this research effort is an energy-efficient scheduling algorithm for preemptive periodic real-time tasks running on a standby-sparing system. The framework uses the EDF algorithm on the primary CPU and the EDL algorithm on the spare CPU. This allows executing the primary copies as soon as possible, while the backups are delayed on the spare CPU. An advantage of this framework is that often the execution of the backups can be canceled upon the early and successful completion of the primary copies. The proposed scheme uses the idle intervals in the EDL schedule for efficiently sharing the slack among tasks. This allows us to avoid the complex dynamic slack management techniques. Simulation results underline potential for energy savings compared to RAPM for most scenarios, while provisioning for permanent faults and still preserving the original reliability in terms of tolerance to transient faults. The proposed algorithm is particularly efficient for low-to-modest workload scenarios. To the best of our knowledge, this is the first attempt for reliability- and energy-aware scheduling of preemptive periodic real-time tasks on a standby-sparing system.

Acknowledgments: This work was supported by US National Science Foundation awards CNS-1016855, CNS-1016974, and CAREER awards CNS-0546244 and CNS-0953005.

REFERENCES

[1] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced cpu energy," in *Mobile Computing*, ser. The International Series in Engineering and Computer Science. Springer US, vol. 353, pp. 449–471.

[2] M. T. Schmitz, B. M. Al-Hashimi, and P. Eles, *System-Level Design Techniques for Energy-Efficient Embedded Systems*. Norwell, MA, USA: Kluwer Academic Publishers, 2004.

[3] H. Aydin and R. Melhem, "Power-aware scheduling for periodic real-time tasks," *IEEE Trans. on Computers*, vol. 53, no. 5, pp. 584 – 600, May 2004.

[4] D. Pradhan, *Fault Tolerant Computer System Design*. Prentice Hall, 1996.

[5] X. Castillo, S. R. McConnel, and D. P. Siewiorek, "Derivation and calibration of a transient error reliability model," *IEEE Trans. Comput.*, vol. 31, pp. 658–671, July 1982.

[6] 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.*, vol. 4, pp. 214–237, August 1986.

[7] 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," *Micro IEEE*, vol. 6, pp. 10–20, November-December 2004.

[8] H. Aydin, "Exact fault-sensitive feasibility analysis of real-time tasks," *IEEE Transactions on Computers*, vol. 56, no. 10, pp. 1372 – 1386, 2007.

[9] D. Zhu, R. Melhem, and Mossé, "The effects of energy management on reliability in real-time embedded systems," in *Proceedings of Int'l Conf. Computer Aided Design*, 2004, pp. 35–40.

[10] R. Melhem, D. Mossé, and E. Elnozahy, "The interplay of power management and fault recovery in real-time systems," *IEEE Transactions on Computers*, vol. 53, pp. 217–231, 2004.

[11] Y. Zhang and K. Chakrabarty, "Dynamic adaptation for fault tolerance and power management in embedded real-time systems," *ACM Trans. Embed. Comput. Syst.*, vol. 3, pp. 336–360, May 2004.

[12] B. Zhao, H. Aydin, and D. Zhu, "Enhanced reliability-aware power management through shared recovery technique," in *Proc. International Conference on Computer Aided Design (ICCAD)*, 2009.

[13] D. Zhu and H. Aydin, "Reliability-aware energy management for periodic real-time tasks," *IEEE Trans. on Computers*, vol. 58, no. 10, pp. 1382 – 1397, October 2009.

[14] Q. Xi, D. Zhu, and H. Aydin, "Global scheduling based reliability-aware power management for multiprocessor real-time systems," *Journal of Real-Time Systems*, vol. 47, no. 2, pp. 109 – 142, March 2011.

[15] 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 *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ISSS'09)*.

[16] R. Ernst and W. Ye, "Embedded program timing analysis based on path clustering and architecture classification," in *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, ser. ICCAD '97, 1997, pp. 598–604.

[17] P. Pillai and K. G. Shin, "Real-time dynamic voltage scaling for low-power embedded operating systems," in *Proceedings of the 18th ACM symposium on Operating systems principles (SOSP'01)*.

[18] G. C. Buttazo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2005.

[19] H. Chetto and M. Chetto, "Some results of the earliest deadline scheduling algorithm," *IEEE Transactions on Software Engineering*, vol. 15, pp. 1261–1269, 1989.

[20] X. Fan, C. Ellis, and A. Lebeck, "The synergy between power-aware memory systems and processor voltage scaling," in *Power - Aware Computer Systems*, ser. Lecture Notes in Computer Science, B. Falsafi and T. Vijaykumar, Eds. Springer Berlin / Heidelberg, 2005, vol. 3164, pp. 151–166.

[21] R. Jejurikar and R. Gupta, "Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems," in *Proceedings of the Low Power Electronics and Design, ISLPED*, 2004.

[22] J. Zhuo and C. Chakrabarti, "System-level energy-efficient dynamic task scheduling," in *Proceedings of the 42nd Design Automation Conference*, 2005.

[23] E. Bini and G. C. Buttazzo, "Biasing effects in schedulability measures," *European Conference on Real-time Systems*, 2004.