Generalized Standby-Sparing Techniques for Energy-Efficient Fault Tolerance in Multiprocessor Real-Time Systems

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

Hakan Aydin
George Mason University
Fairfax, VA 22030
aydin@cs.gmu.edu

***Abstract*—The *Standby-Sparing (SS)* technique has been previously explored to improve energy efficiency while providing fault tolerance in dual-processor real-time systems. In this paper, by considering both transient and permanent faults, we develop energy-efficient fault tolerance techniques for real-time systems deploying an arbitrary number of identical processors. First, we study the *Paired-SS* technique, where processors are organized as groups of two (i.e., pairs) and SS is applied within each pair of processors directly after partitioning tasks to the pairs. Then, we propose a *Generalized-SS* technique that partitions processors into two groups containing *primary* and *secondary* processors, respectively. The *main* and *backup* copies of tasks are executed on the primary and secondary processors under the *partitioned-EDF* and *partitioned-EDL* scheduling policies, respectively. The objective is to reduce the overlapped executions of the main and backup copies in order to improve energy savings. Our experimental evaluations show that, for a given system with fixed number of processors, typically there exists a configuration of primary and secondary processors under the Generalized-SS technique that can lead to better energy savings when compared to the Paired-SS technique.**

I. Introduction

The recent progress in the multiprocessor/multicore systems has important implications for real-time embedded system design and operation. From vehicle navigation to space applications as well as industrial control systems, the trend is to deploy real-time embedded systems with multiple processors: Systems with 4-8 processors are common [1] and it is expected that *many-core* systems with dozens of processing cores will be available in near future [2]. For such systems, in addition to general *temporal predictability* requirement common for all real-time systems, two additional operational objectives are seen as critical: *energy efficiency* and *fault tolerance*.

There is a large body of literature (see Section VI for a partial list of related works) on multiprocessor energy management. Essentially, the solutions are based on two well-known techniques. With *Dynamic Power Management (DPM)*, the system components are put into idle/sleep states when they are not in use [19]. Another widely studied technique is *Dynamic Voltage Scaling (DVS)*, where the CPU supply voltage and frequency are simultaneously reduced to slow down the execution to save energy [21].

In addition to energy efficiency, providing *fault tolerance* features to achieve *high reliability* figures is also important, especially for safety-critical real-time applications that operate in harsh environments [8]. Computer systems may be subject to different types of faults at run-time due to various effects, such as hardware defects, electromagnetic interference, and cosmic ray radiations [17]. *Permanent faults* can bring a system component (e.g., the processor) to halt and cannot be recovered from without some form of hardware redundancy. *Transient faults* are often induced by electromagnetic interference or cosmic ray radiations and disappear after a short time interval. When they occur, transient faults may result in *soft errors* (e.g., incorrect computation) and there may be a need to re-execute the affected task (or, an alternative recovery task) to obtain the correct result. While transient faults are more common than permanent faults [17], a comprehensive framework should have provisions for both transient and permanent faults in a safety-critical real-time system.

An intriguing dimension of the problem is that energy efficiency and fault tolerance are typically *conflicting* objectives, due to the fact that tolerating permanent/transient faults often requires *extra* resources with high energy consumption potential. In addition, existing research shows that the probability of having transient faults increases drastically when systems are operated at lower supply voltages in the context of the popular DVS technique [10], [27].

Recently, a *dual-processor* framework has been proposed to provide tolerance to transient and permanent faults while managing energy in real-time systems. The technique, called *Standby-Sparing (SS)*, deploys two processors. For each task, a main copy is executed on the first *(primary)* processor, while a backup copy is executed on the second *(spare)* processor. Due to its nature, the framework naturally tolerates a single permanent fault on any processor. In terms of energy management, the key point is to use DVS on the primary processor and to *delay* the backup tasks on the spare processor as much as possible – with the objective of canceling a backup task as soon as the main task completes successfully (i.e., without encountering a soft error) to avoid unnecessary energy consumption. However, should the primary task fail, the backup runs to completion at the maximum speed on the spare processor. Moreover, the system's original reliability (in terms of resilience with respect to transient faults) is also preserved since a full re-execution at maximum speed is guaranteed in case of a transient fault [9]. The problem is then to statically allocate backup tasks on the spare processor to minimize the overlap between primary and backup copies of the same task, to reduce the energy consumption.

978-1-4799-0850-9/13/$31.00 ©2013 IEEE

However, the original proposal in [9] considers only *nonpreemptive* scheduling of *aperiodic* tasks. In [14], Haque *et al.* extended the Standby-Sparing technique for *preemptive periodic* real-time applications. In that work, the *earliest-deadline-first (EDF)* scheduling policy is applied on the primary processor while the backup tasks are executed on the spare processor according to the *earliest-deadline-late (EDL)* scheduling policy [7]. By delaying the backup tasks as much as possible to obtain idle intervals as early as possible, the joint use of EDF and EDL on the primary and spare processors provides better opportunities to reduce the overlapped execution of primary and backup copies of the tasks at run-time and thus reduces the system energy consumption.

The existing SS-based solutions ([9], [14]), while effective, are limited to dual-processor systems. Considering that the number of available cores grows quickly in existing systems ([1], [2]), our objective in this paper *is to explore how the standby-sparing technique can be applied to systems with more than two processors.* Since the SS-based solutions are suitable for addressing both transient and permanent faults while managing energy, there may be significant value in investigating the design options for applying the SS-based solutions to general multiprocessor configurations.

Specifically, we investigate two main design options, namely, the *Paired-SS* and the *Generalized-SS* techniques, and their impact on energy savings. In Paired-SS, processors are organized as groups of two (i.e., *pairs*) and, after partitioning tasks to the pairs, the conventional SS technique is applied within each pair of processors directly. In contrast, *Generalized-SS* partitions processors into two groups of potentially different size, designated for *primary* and *secondary* processors, respectively. Then, following the principle of reducing the overlapped executions of tasks and thus to obtain more energy savings as in SS, the *main* and *backup* copies of tasks are executed on the primary and secondary processors with DVS and DPM techniques and under the *partitioned-EDF* and *partitioned-EDL* scheduling, respectively.

By scheduling a separate backup task for each task in the workload at the maximum frequency, our solutions have the desirable feature of tolerating a single permanent fault on any processor while preserving the original system reliability (in terms of resilience to transient faults) [25]. In addition, we propose an online scheme based on the *wrapper-task based slack management* framework [25] to improve energy savings at run-time. The online component is applicable to both Paired-SS and Generalized-SS schemes.

We evaluate the energy savings of the proposed techniques through extensive simulations. The results show that, for a system with a given number of processors, typically there exists a configuration of primary and secondary processors for Generalized-SS outperforming Paired-SS in terms of energy savings. Moreover, the performance of the wrapper-task based online energy management scheme is comparable to that of the existing SS-based CSSPT scheme (that needs statistical execution information of tasks [14]) and can be much better than the ASSPT scheme (which is very aggressive [14]) at high system loads.

The remainder of this paper is organized as follows. Section II presents task and system models and states our assumptions. As the motivational example, an instance of the problem with three processors is discussed in Section III. Both the Paired and Generalized Standby-Sparing techniques are presented in Section IV. Section V presents and discusses the evaluation results. In Section VI, we present an overview of the closely-related work and Section VII concludes the paper.

II. System Models and Problem Statement

A. Task Model

We consider a set of n periodic real-time tasks $\Gamma = \{T_1, \ldots, T_n\}$. Each task T_i is represented as a tuple (c_i, p_i), where c_i is its worst-case execution time (WCET) under the maximum available CPU frequency and p_i is its period. The utilization of a task T_i is defined as $u_i = \frac{c_i}{p_i}$. The system utilization U of a given task set is the summation of all task utilizations ($U = \sum_{T_i \in \Gamma} u_i$).

The tasks have implicit deadlines. That is, the j^{th} task instance (or, job) of T_i, denoted by $T_{i,j}$, arrives at time $(j-1) \cdot p_i$ and must complete its execution by its deadline at $j \cdot p_i$. Since a task has only one active task instance at any time, when there is no ambiguity, we use T_i to represent both the task and its current task instance.

B. Processor and Power Models

The tasks are executed on a multi-processor system with m *identical* processors. The processors are assumed to have DPM and DVS capabilities, which are common features in modern processors. With increasing importance of leakage power and emphasis on the need for considering all system components in power management [3], [15], [16], we adopt a system-level power model used in the previous energy-efficient fault-tolerant real-time system research [25]. Specifically, the total power consumption is expressed as:

$$P = P_s + m \cdot (P_{ind} + C_{ef} \cdot f^k) \quad (1)$$

Above, P_s stands for *static power*, which can be removed only by powering off the whole system. Due to the prohibitive overhead of turning off and on the system in periodic real-time execution settings, we assume that the system is in *on* state at all times and that P_s is always consumed.

Hence, we focus on the energy consumption related to active power, which is given by the last two items in the above equation. P_{ind} is the per-processor *frequency-independent active power*. The *frequency-dependent active power* depends on the system-dependent constants C_{ef} and k, as well as the processing frequency f, which can be managed through the DVS technique [6]. We assume that the processors can only operate at one of the L different discrete frequency levels $\{f_1, f_2, \ldots, f_L\}$, where $f_L = f_{max}$ is the maximum frequency. Moreover, the time overhead for frequency (and supply voltage) changes is assumed to be incorporated into the WCETs of tasks [26].

Given the frequency-independent component in the active power, normally there exists an *energy-efficient frequency* below which the DVS technique cannot save energy [18]. In addition, when there is no active task on a processor, we further assume that both components of the active power can be efficiently removed by putting the idle processor to sleep states through the DPM technique.

C. Fault and Recovery Models

For fault tolerance and system reliability, we assume that each task T_i has a *backup* task B_i, which has the same timing parameters (i.e., c_i and p_i) as task T_i. To distinguish with the backup tasks, we occasionally use the term *main tasks* to refer to the tasks in Γ. As with main tasks, the j^{th} task instance (or, job) of B_i is denoted by $B_{i,j}$.

There are *two objectives* for deploying the backup tasks. First, they are utilized to tolerate *permanent* faults. Individual processors may be subject to such faults due to circuit wear-out or manufacturing defects [17]. For this purpose, a given main task and its backup task must be scheduled on different processors. Note that, a number of schemes are readily available from the literature to detect the permanent faults at run-time [17].

With only one backup task for each main task, we know that only a single permanent fault on any processor can be tolerated at a given time. Once a permanent fault is detected, the system will switch to the *emergency operation mode* where the related processors, which have at least one task whose corresponding backup (or main) task is on the faulty processor, have to execute their tasks at the maximum processor frequency. This is necessary to maintain system reliability (in terms of resilience to transient faults) [25]. However, unless an appropriate recovery mechanism re-maps the tasks and re-configures the system with the remaining processors, the system may not be able to tolerate any additional permanent fault. The investigation of such additional recovery mechanisms is beyond the scope of this paper and we focus on tolerating a single permanent fault in this work.

The second objective of backup tasks is to recuperate the reliability loss due to increased transient faults when the DVS technique is exploited for main tasks to save energy [27]. For such a purpose, the backup tasks have to be executed at the maximum processing frequency, should they are needed at runtime. This guarantees that the *original reliability* of a task is preserved with respect to transient faults [24]. A task's original reliability is defined as the probability of finishing the execution of the task successfully when no voltage scaling is applied.

To detect the existence of soft errors triggered by transient faults, we assume that *sanity* (or *consistency*) checks are performed at the end of execution of every task instance [17]. If there is no error, the backup task running (or, scheduled to run) on another processor is canceled to save energy. Moreover, the overhead for fault detection is also assumed to be incorporated into tasks' WCETs.

D. Problem Statement

The main problem that we address in this paper can be stated as follows: *For a given set of n periodic tasks to be executed on m identical processors, how to schedule the tasks (and their backup tasks) on the processors to: a) tolerate a single permanent fault; b) maintain system original reliability with respect to transient faults; and c) minimize system energy consumption.*

For dual-processor systems, the *Standby-Sparing (SS)* technique has been studied to address this problem [14]. The central idea of SS is to run the main and backup copies of tasks on separate primary and secondary processors, respectively. The tasks' main copies are executed at scaled frequency with DVS under EDF, while the backup copies run at the maximum frequency but are delayed using EDL to reduce overlapped executions with the corresponding main copies for energy savings [14]. When there are more than two processors available in the system, a natural question to ask would be: *"how to configure such processors for higher energy efficiency?"* We can either have more primary processors to execute main copies of tasks to further slow down their executions or have more secondary processors for additional delayed execution of the backup copies.

Focusing on the SS technique and partitioned scheduling, we investigate the following dimensions to improve energy savings:

- how the available processors should be partitioned as *primary* or *secondary* processor groups,
- how the main and backup tasks should be allocated to the available primary and secondary processor groups, and,
- how the DVS mechanism can be used on the primary processors to reduce the overlap between the main and backup tasks to save energy.

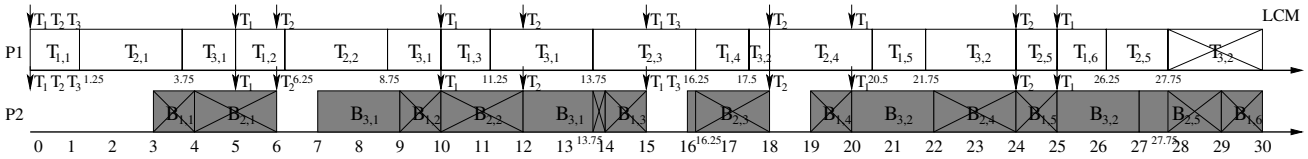
Clearly, this is not a trivial problem considering the intriguing interplay between the scaled frequency of tasks' main copies and the amount of overlapped execution with their backup copies. In fact, even without the fault tolerance and system reliability dimensions, finding the optimal partitioning of tasks to processors and their scaled frequencies with DVS to minimize energy consumption has shown to be NP-hard in the strong sense [4].

III. An Example: Three-Processor Systems

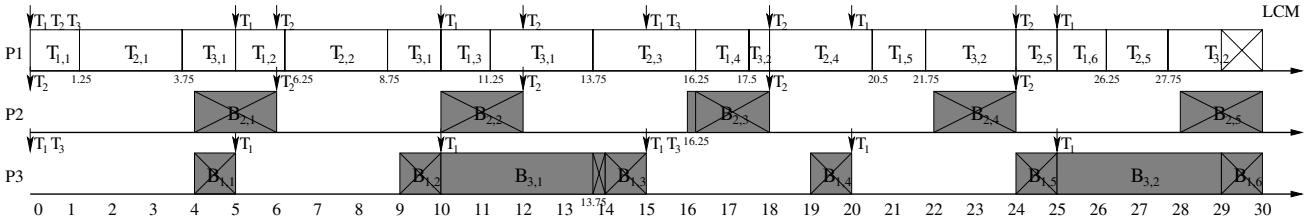
Before presenting our solutions for the general problem, we first consider a simple case of a 3-processor system and illustrate the intriguing dimensions of the problem. For such a 3-processor system, we have two options: a) one primary and two secondary processors (denoted as "X1Y2"), and, b) two primary and one secondary processor (denoted as "X2Y1").

A. A Motivational Example

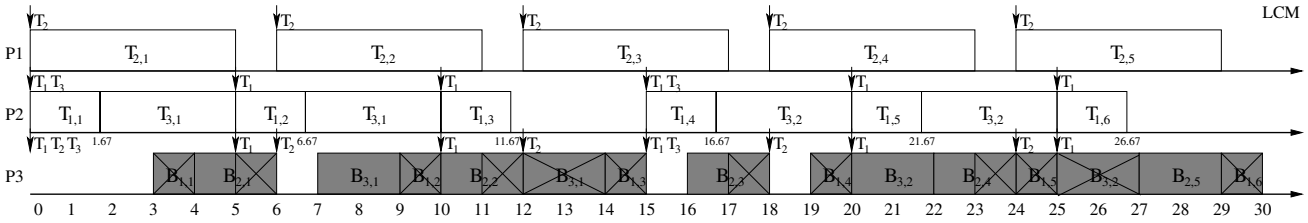
Consider a task set with three periodic real-time tasks $\Gamma = \{T_1(1,5), T_2(2,6), T_3(4,15)\}$. We can easily find that the system utilization is $U = 0.8$ and the *least common multiple (LCM)* of task periods is $LCM = 30$. Suppose that



a. The schedule of the tasks under Standby-Sparing on a dual-processor system with primary frequency at 0.8 [14]



b. The schedule with one primary processor (using frequency 0.8) and two secondary processors



c. The schedule with two primary processors (using frequencies 0.4 and 0.6) and one secondary processor

Fig. 1: Scheduling alternatives for three tasks $T_1(1,5)$, $T_2(2,6)$ and $T_3(4,15)$ running on a 3-processor system

the processors have four discrete (normalized) frequency levels $\{0.4, 0.6, 0.8, 1.0\}$.

Figure 1a shows the schedule within LCM on a dual-processor system with the Standby-Sparing technique [14]. The primary processor executes the main tasks under *Earliest-Deadline-First (EDF)* at the scaled frequency of 0.8 while the secondary processor schedules their backup copies with *Earliest-Deadline-Late (EDL)* policy [7] for energy savings. Suppose that $P_s = 0.02$, $C_{ef} = 1$, and $k = 3$ in the power model. Further, assume that tasks take their WCETs, and there are no faults at run-time. Then, we can find the overall energy consumption for the tasks within LCM as $E_{SS-SPM} = 27.2$, where the majority of backup tasks (marked by 'X') are cancelled.

For the X1Y2 configuration of the three-processor system, where the extra processor is used as an additional secondary (with increased $P_s = 0.03$), the schedule of tasks is shown in Figure 1b. The primary processor P_1 still runs at the frequency 0.8, executing the main copies of tasks. However, the backup copies of tasks can be re-allocated. Suppose that task T_2's backup runs on processor P_2 and tasks T_1 and T_3's backups run on processor P_3. This can further delay the execution of backup tasks and reduce the overlaps with their main tasks. It turns out that, although the overlapped executions can be reduced slightly, the additional energy consumption from the static power of the extra processor overshadows such benefits and leads to the overall energy consumption of $E_{X1Y2} = 27.45$. This is slightly than that of the dual-processor system under the traditional SS scheme.

When the extra processor is utilized as an additional primary (the X2Y1 configuration), Figure 1c shows the schedule of tasks. The main task T_2 is allocated to processor P_1. The other tasks (T_1 and T_3) are allocated to P_2, running at the scaled frequencies of 0.4 and 0.6, respectively. The overall energy consumption under this configuration can be found as $E_{X2Y1} = 23.37$, which represents a 14% improvement over the traditional SS scheme on a dual-processor system.

B. Energy Efficiency of Different Configurations

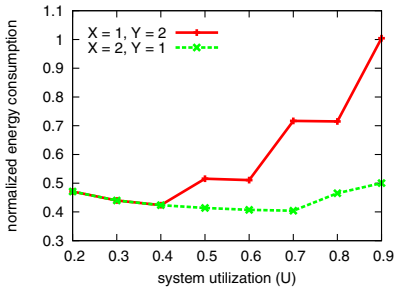


Fig. 2: Energy consumption of X1Y2 and X2Y1 configurations

For task sets with different system utilizations, Figure 2 further shows the energy consumption of the two configurations for a three-processor system, using the same power parameters as in Section III-A. The Y-axis represents the energy consumption under two system configurations, normalized to the energy consumed by the *SS-NPM* scheme where we deploy the traditional SS on a dual-processor system and without any voltage scaling. We assume that no fault occurs during

execution and each data point shows the average result of 100 synthetic task sets and each set has 20 tasks (Section V gives further details on task set generation methodology).

First, at low system loads (i.e., $U \leq 0.4$), the main copies of tasks are executed at the lowest available frequency 0.4 and almost all backup copies are cancelled under both configurations, giving the same energy performance. However, at high system loads, having two primary processors (X2Y1) allows the system to execute tasks' main copies at low frequencies compared to the case with one primary and two secondary processors (X1Y2), improving energy savings. When $U = 0.9$, such differences can be up to 50% since tasks' main copies under the X1Y2 configuration need to run at the maximum frequency due to the limitations of discrete frequencies.

IV. Standby-Sparing for Multiprocessors

From the above discussion, we can see that the different configurations of primary and secondary processors can have important effects on the energy efficiency of multiprocessor systems. In this section, following the ideas and principles of the traditional SS scheme for periodic real-time tasks [14] and focusing on partitioned scheduling, we discuss the details of the *Paired Standby-Sparing (P-SS)* and *Generalized Standby-Sparing (G-SS)* schemes in Sections IV-A and IV-B, respectively. Moreover, based on the wrapper-task slack management mechanism [25], we also develop a generic online energy management technique that can be applied to both P-SS and G-SS in Section IV-C.

A. Paired Standby-Sparing

Considering the fact that the traditional SS scheme was designed for dual-processor systems [14], a straightforward approach is to first configure processors as groups of two (i.e., *pairs*). Then, after partitioning tasks to processor pairs appropriately, the SS scheme can be applied directly within each pair of processors. We call this scheme *Paired Standby-Sparing (P-SS)*.

From the results in [14], we know that different system utilizations may have a great impact on the energy efficiency of a dual-processor system under the SS scheme. This is because both the scaled frequency on the primary processor to execute tasks' main copies and the delayed execution of tasks' backup copies depend heavily on system utilizations. The SS scheme may perform quite poorly when the system utilization gets higher, due to higher execution frequency of tasks' main copies and the increased amount of overlapped executions between tasks' main and backup copies [14]. On the other hand, once the scaled execution frequency of tasks' main copies reduces to the minimum (available) energy-efficient frequency, additional energy savings are rather limited with further reduced system utilizations [14].

Therefore, the key factor that determines the energy consumption under P-SS is the mapping of tasks to processor pairs. Intuitively, for higher energy efficiency, the mapping with balanced workload among processor pairs sould be preferred. Unfortunately, it is well-known that finding the optimal task-to-processor mapping with the most balanced workloads is NP-hard. Hence, in this work, we adopt the *Worst-Fit-Decreasing (WFD)* heuristic when mapping tasks to processor pairs in P-SS, given its inherent ability to produce relatively balanced workloads [4].

For a system with m processors, although there are $\lfloor \frac{m}{2} \rfloor$ processor pairs, they are operated under the SS scheme *independently*. Therefore, with one backup copy for each task running on its secondary processor in the pair, the system under P-SS can still tolerate only a single permanent fault at any time, which is the case as well for the G-SS scheme (as will be discussed in Section IV-B). However, once the processor affected by a permanent fault is identified and isolated, we can re-configure the system by re-grouping the remaining $(m-1)$ processors and/or re-mapping the tasks to tolerate additional permanent faults, which is left as a future work.

B. Generalized Standby-Sparing

In Section III, for systems with three processors, we showed through an example that having two primary processors for the execution of tasks' main copies while sharing one secondary processor for the backup copies of all tasks may lead to higher energy efficiency. Following this principle and generalizing the idea of Standby-Sparing [14] for m-processor systems, the *Generalized Standby-Sparing (G-SS)* scheme organizes the processors into two groups: a *primary group* of X processors and a *secondary group* of Y processors (where $m = X + Y$). Then, the main copies of tasks are executed under the *partitioned-EDF* on the primary processor group at scaled frequencies with the DVS technique, while the secondary processor group adopts the *partitioned-EDL* to execute the backup copies at the maximum frequency and as late as possible, to save more energy.

Note that, for a given (X, Y)-configuration of processors, finding the feasible partitioning of main and backup tasks to the primary and secondary processors, respectively, is NP-hard. Therefore, the problem of finding the optimal partitioning of tasks to processors for minimizing system energy consumption under G-SS is NP-hard as well. As in P-SS, we again use the WFD heuristic to map the main and backup tasks to primary and secondary processors for a given (X, Y) configuration, by considering its ability to produce the relatively balanced partitionings.

Algorithm 1 summarizes the main steps of G-SS for a given (X, Y) system configuration. First, the main and backup copies of tasks are partitioned to X primary and Y secondary processors, respectively, by using the WFD heuristic (line 2). Then, the schedulabilities of the resulting WFD mappings on all primary and secondary processors under EDF and EDL, respectively, are examined (line 3).

If tasks are schedulable with the resulting WFD mappings for the given processor configuration, the main copies of tasks on each primary processor are executed under EDF at the appropriate scaled frequency while tasks' backup copies are executed on each secondary processor under EDL (lines 4

Algorithm 1 Main steps of G-SS for a X/Y configuration

1: **Input:** task set Γ, X and $Y = (m - X)$;
2: Find X and Y WFD partitions of Γ:
$\Pi_M = \{\Gamma_1^T, \cdots, \Gamma_X^T\}$ and $\Pi_B = \{\Gamma_1^B, \cdots, \Gamma_Y^B\}$;
3: **if** $(\forall i, U(\Gamma_i^T) \leq 1$ and $\forall j, U(\Gamma_j^B) \leq 1)$ **then**
4: On X primary processors: run main copies of tasks under EDF at scaled frequencies;
5: On Y secondary processors: run backup copies of tasks under EDL;
6: **end if**

and 5). Here, the scaled frequency on the i^{th} primary processor can be determined as the lowest discrete frequency level f_x such that $f_{x-1} < U(\Gamma_i^T) \cdot f_{max} \leq f_x$.

Note that, as in the conventional SS scheme [14], when the main (or backup) copy of a task completes successfully on one processor at run-time, the processor that has the backup (or main) copy of the task will be notified to cancel the corresponding execution for energy savings, which is not shown in the algorithm for brevity.

Energy-Efficient Processor Configuration for G-SS: It is clear that different configurations of primary and secondary processors in G-SS have great impact on the energy efficiency of the system under consideration. Depending on the configuration of primary and secondary processors (i.e., depending on the (X, Y) configuration), it is very likely that the backup copies of two main tasks running on the same primary processor will be allocated to different secondray processors. This is quite different from P-SS where each primary processor has a *dedicated* secondary processor and the main and backup copies of the same subset of tasks are executed on these two processors, respectively.

Due to such complications, it is quite difficult to identify the overlapped execution regions between tasks' main and backup copies in the EDF and EDL schedules on different processors, which makes it infeasible to find the optimal configuration of processors in G-SS for energy minimization analytically. In this section, for a given task set Γ running on a m-processor system, we present an iterative approach to find out the best processor configuration(s) for the G-SS scheme to minimize system energy consumption. Again, we use the WFD task-to-processor mapping heuristic. The main steps of such an iterative scheme are summarized in Algorithm 2.

In Algorithm 2, the minimum and maximum numbers of primary processors needed are first determined (line 3). Then, for each possible processor configuration (i.e., a pair of X and Y), we check the schedulability of the given task set Γ under G-SS assuming the WFD task-to-processor mapping (by invoking Algorithm 1 at line 6). If Γ is schedulable, the system energy consumption under G-SS can be obtained from the emulated execution of the tasks in LCM (line 7). During such emulations, we assume that tasks take their WCETs and no faults occur. Finally, considering all feasible pairs of X and Y, the efficient configuration with the lowest fault-free system

Algorithm 2 Finding the optimal X/Y Configuration

1: **Input:** task set Γ and m (number of processors);
2: $E^{min} = \infty$; $X^{opt} = -1$; //initialization
3: $X^{min} = \lceil U(\Gamma) \rceil$; $X^{max} = m - X^{min}$;
4: **for** $(X = X^{min} \rightarrow X^{max})$ **do**
5: $Y = m - X$; //number of secondary processors
6: **if** (Γ is schedulable under G-SS with X/Y) **then**
7: Get $E_{G-SS}(X, Y)$ by emulation in LCM;
8: **if** $(E^{min} > E_{G-SS}(X, Y))$ **then**
9: $X^{opt} = X$;
10: **end if**
11: **end if**
12: **end for**

energy consumption can be found out (lines 8 and 9). The variations in the system energy consumption with different processor configurations are further illustrated in Section V-A.

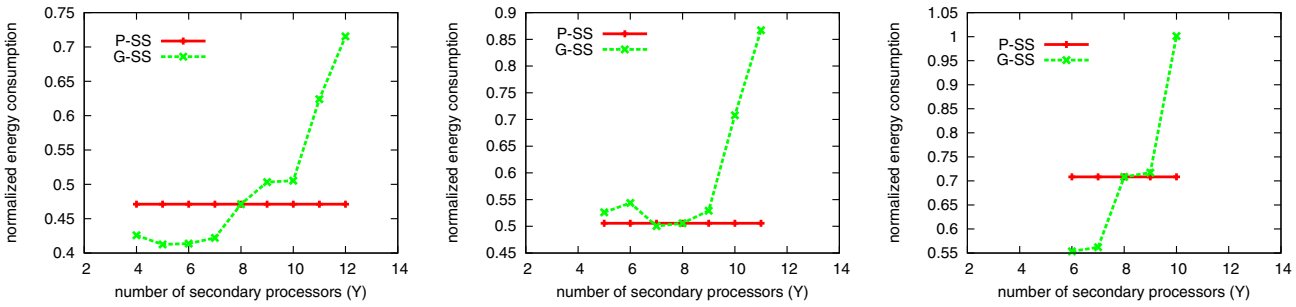
C. Online Scheme with Wrapper-Tasks

As shown in Section V, the statically determined scaled frequencies for the main copies of tasks in P-SS and G-SS can lead to energy savings. Considering the fact that real-time tasks normally only take a small fraction of their WCETs [11], large amount of dynamic slack can be expected at run-time and should be exploited for better energy efficiency. In [14], two online schemes, namely *ASSPT* and *CSSPT*, were proposed. These two schemes reclaim dynamic slack in *aggressive* and *conservative* manner, respectively, to further slow down the main copies of tasks at run-time to save more energy.

Although ASSPT and CSSPT can be applied directly to each processor pair in the P-SS scheme, it is not obvious how they can be extended to the G-SS scheme. Observing that a task's backup copy normally starts later than its main copy, ASSPT and CSSPT determine the amount of available slack for a task's main copy based on the start time of its backup copy in the EDL schedule of the secondary processor [14]. However, such relations may not hold under the G-SS scheme, especially for the configurations with different numbers of primary and secondary processors that can have completely different subsets of tasks.

Therefore, we propose a *generic* online scheme based on the *wrapper-task* slack management mechanism [25], which can be applied to main tasks running on primary processors for both the P-SS and G-SS schemes at run-time. Essentially, a wrapper-task represents a piece of slack with two parameters (c, d), where the size c denotes the amount of the slack and the deadline d equals to that of the task creating this slack [25].

In our online scheme, the available slack times (i.e., wrapper-tasks) are managed *separately* on each primary processor. At run-time, the main copy of a task reclaims the available slack on its own processor, independently of the backup execution of the task. Since the main tasks are executed under the EDF scheduling on primary processors, from [25], we know that a main task can safely reclaim any dynamic



a. System utilization $U = 3.0$; b. System utilization $U = 4.0$; c. System utilization $U = 5.0$;

Fig. 3: The effects of different configurations of processors in G-SS for a 16-CPU system under different system loads.

slack that has an earlier deadline than that of the task to further reduce its processing frequency and thus to save more energy. The detailed steps on how to manage and utilize the dynamic slack at run-time with wrapper-tasks can be found in [25].

V. Evaluation and Discussion

In this section, we evaluate the performance of the proposed P-SS and G-SS schemes for multiprocessor real-time systems through extensive simulations. To this aim, we developed a discrete event simulator in C++. From previous studies [9], [14], we know that, in addition to tolerating a single permanent fault, the Standby-Sparing scheme can preserve the original system reliability with respect to transient faults by executing the backup tasks at the maximum frequency. Since the proposed P-SS and G-SS schemes follow the same design principle for fault tolerance (see the discussions in Section IV), the reliability goals (of tolerating both permanent and transient faults) can be ensured as well. Therefore, in what follows, we focus on evaluating the energy efficiency of the proposed schemes.

As modern processors have a few frequency levels, we assume that there are four frequency levels, which are normalized as $\{0.4, 0.6, 0.8, 1.0\}$ in the evaluations. Moreover, we focus on 16-processor systems, and we assume that $P_s = 0.16$, $C_{ef} = 1$ and $k = 3$, as in [14].

The utilizations of tasks are generated according to the *UUniFast* scheme proposed in [5]. We used two different average task utilization values $u^{ave} = 0.1$ and $u^{ave} = 0.05$, in different experiments. For each task set, we generate enough number of tasks so that the system utilization reaches a given target value. That is, for a given system utilization U, the average number of tasks in a task set is $\lfloor \frac{U}{u^{ave}} \rfloor$. The periods of tasks are uniformly distributed in the range of $[10, 100]$ and the WCET of a task is set according to its utilization and period. Each data point in the figures corresponds to the average result of 100 task sets.

A. Energy Efficiencies for Different Configurations in G-SS

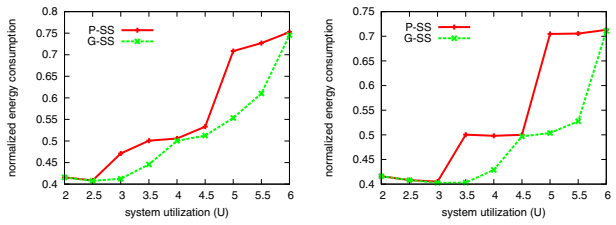
First, we illustrate the variations in energy consumption under different primary and secondary processor configurations in the G-SS scheme for 16-processor systems and compare them against that of the P-SS scheme. Here, without considering online slack reclamation (which will be evaluated next), we assume that all tasks run at their statically assigned scaled frequencies and take their WCETs at run time. Moreover, it is assumed that no fault occurs during the execution of tasks and the backup (main) copies of tasks are cancelled under both schemes once their coresponding main (backup) copies complete successfully[1]. We show the energy consumption figures as normalized with respect to the P-SS scheme when both primary and secondary processors execute all the tasks at the maximum frequency.

For 16-processor systems, the upper-bound on the total main task system utilization schedulable under the proposed schemes would be 8 (since a similar processor capacity should be reserved for backup tasks). For the cases of system utilization $U = 3.0, 4.0$, and 5.0, Figure 3 shows the results for the G-SS scheme with varying numbers (Y) of secondary processors as well as that of the P-SS scheme for comparison. In these experiments, the average task utilization is set to $u^{ave} = 0.1$.

Not surprisingly, for different processor configurations (i.e., as the number of secondary processors varies) in the G-SS scheme, the system energy efficiency can exhibit significant differences ranging from 30% to 45%. As in the example discussed in Section III, for a given system utilization, the processor configuration that can lead to the best system energy efficiency normally has more primary processors than secondary processors (i.e., smaller values of Y). On the other hand, since the backup copies of tasks have to be executed at the maximum frequency for reliability preservation [14], the spare capacity on secondary processors is normally wasted, which leads to inferior performance for G-SS when more processors are used as secondaries. From the results, we can see that with a judicious selection of the processor configurations (i.e., the X and Y values), G-SS can outperform P-SS by 15% additional energy savings when $U = 5.0$.

With varying system utilizations, the performance of the G-SS scheme with the most energy-efficient processor con-

[1]Note that, due to independent scheduling of tasks' main and backup copies under EDF and EDL, respectively, it is possible for a task's backup copy to finish earlier than its main copy in the SS scheme [14].



a. $u^{ave} = 0.1$ b. $u^{ave} = 0.05$

Fig. 4: Energy consumptions of P-SS and G-SS vs. U

figuration is shown in Figure 4a. The results indicate that, compared to the P-SS scheme, the G-SS with its most energy-efficient processor configuration gives better energy savings. However, the exact performance difference diminishes at very low and high system utilizations. This is because, at low system utilizations ($U \leq 2.5$), the tasks' main copies can be executed at $f = 0.4$ (the lowest available frequency) while most backup copies of tasks can be cancelled under both P-SS and G-SS schemes.

At high system utilizations (for example, $U = 6.0$), there are only a few feasible processor configurations (e.g., $X = 7, 8$ and 9) in most cases for the G-SS scheme. Further, the most energy-efficient configuration turns out to be $X = 8$, which makes G-SS to act exactly the same as P-SS since we use the same WFD heuristic when partitioning main and backup copies of tasks. The close performance between G-SS and P-SS at the middle range of system utilizations comes from the limitation of discrete frequencies, which becomes clearer for systems with smaller tasks (i.e., $u^{ave} = 0.05$) as shown in Figure 4b.

B. Performance of Online Schemes

In this section, by varying the ratio of average over worst case execution times of tasks, we further evaluate the performance of P-SS and G-SS with different online techniques. Specifically, we implement both ASSPT and CSSPT techniques [14] and apply them to processor pairs under P-SS, which are denoted as "P-SS-ASSPT" and "P-SS-CSSPT", respectively. Moreover, the wrapper-task based online scheme is also implemented and applied on the primary processors under P-SS and G-SS, which are denoted as "P-SS-Wrap" and "G-SS-Wrap", respectively. For G-SS and a given task set with certain total utilization, we assume that the most energy-efficient processor configuration (obtained from Algorithm 2) is adopted.

Since we focus on evaluating the energy efficiency of the proposed schemes, we again assume that no fault occurs during tasks' execution. Again, the execution of backup (or main) copies of tasks are cancelled once the corresponding main (or backup) copies complete under all schemes. Moreover, the normalized energy consumptions are reported, where the baseline corresponds to the P-SS scheme when both primary and secondary processors running at the maximum frequency.

In these experiments, we set $u^{ave} = 0.1$. To emulate the dynamic execution behaviors of tasks, we use a system wide average-to-worst case execution time ratio α. For each task T_i, its average-to-worst case execution time ratio α_i is generated randomly around α. Then, at run-time, the actual execution time for each instance of task T_i is randomly generated with the mean $\alpha_i \cdot c_i$, where c_i is task T_i's WCET. Essentially, α as an indicator of the expected amount of dynamic slack, lower values suggesting greater dynamic slack.

Figures 5abc show the performance of the schemes with varying α (average-to-worst case execution times of tasks) under three different system utilizations $U = 3.0, 4.0$, and 5.0, respectively. Again, when the system utilization is low (i.e., $U = 3.0$), the main copies of tasks can be executed at the lowest frequency 0.4 and most backup copies are cancelled, leading to very similar (within 6% difference) energy consumption figures for P-SS and G-SS with all online techniques.

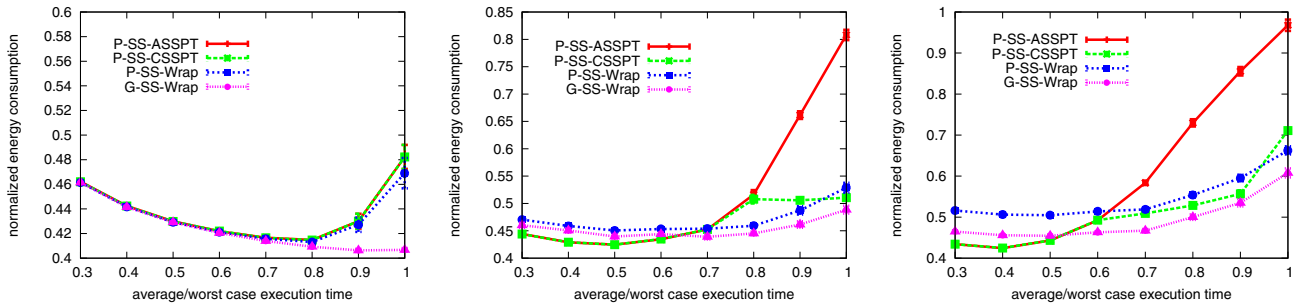
When $\alpha = 1$, there is no dynamic slack at run-time. However, due to the limitation of discrete frequencies, there will be some spare capacity on each primary processor, which can be exploited by the wrapper-task based scheme and some additional energy savings can be obtained when compared to that of ASSPT and CSSPT. Therefore, with the limited benefits of the online techniques at $\alpha = 1$, G-SS outperforms P-SS slightly, which is consistent with the prior results.

When the system utilization gets higher (i.e., $U = 4.0$ and $U = 5.0$), we can see that the ASSPT technique can cause drastic performance degration for P-SS as tasks' dynamic loads increase (with higher values of α). These results are consistent with those in [14]. The reason comes from the aggressive slack usage under the ASSPT technique, which executes the main copies of tasks at very low frequency at the beginning of the schedule. Such scaled executions force remaining tasks' main copies to run at much higher frequency and cause more overlapped executions with tasks' backup copies on the seconary processors.

To address the above mentioned problem, based on the static and dynamic loads of tasks, the CSSPT scheme statically determines a lower bound for the scaled frequency for executing tasks' main copies when reclaiming slack at run-time [14]. With such a scaled frequency bound, CSSPT can effectively prevent the aggressive usage of slack time in the early part of the schedule. Therefore, when compared to ASSPT, P-SS performs much better with the CSSPT online technique, especially for tasks with higher dynamic loads.

For our proposed wrapper-task based online technique, we can see that its performance is pretty stable under different dynamic loads of tasks. Although it performs (slightly) worse than that of ASSPT and CSSPT for the P-SS scheme at low dynamic loads (i.e., $\alpha \leq 0.6$), its performance is very close to that of CSSPT at higher dynamic loads of tasks. However, unlike CSSPT, the wrapper-task technique does not require the pre-knowledge of tasks' average-case workloads.

Moreover, as a generic online technique, wrapper-tasks can also be applied to the primary processors in the G-SS scheme, which exhibits a stable performance as well. Although the



a. system utilization $U = 3.0$; b. system utilization $U = 4.0$; c. system utilization $U = 5.0$;

Fig. 5: Performance of P-SS and G-SS with online techniques under different system loads.

performance gain of applying the wrapper-task technique on G-SS is rather limited (within 5%) when compared to that of P-SS, we can see that G-SS-Wrap always performs better than that of P-SS-CSSPT at higher dynamic loads of tasks.

VI. Closely Related Work

By incorporating the negative effects of DVS on system reliability, the *Reliability-Aware Power Management (RAPM)* framework has been studied, exploiting time-redundancy for both reliability preservation and energy savings [12], [18], [25]. Here, the main idea is to schedule a separate recovery job for every job that has been selected to scale down to preserve the system's original reliability, which is assumed to be satisfactory. The recovery job is executed at the maximum speed only when its scaled primary job incurs transient faults. The shared-recovery based RAPM scheme has also been studied where several tasks with scaled frequencies may share one recovery task for reliability preservation [22]. More recently, considering such negative effects of DVS on system reliability, the recovery allowance based scheme that can achieve arbitrary reliability levels for each periodic task is further studied [23].

Although the RAPM schemes can preserve the system reliability with respect to transient faults, there are no provisions for permanent faults. Moreover, due to the separate allocation of the recovery job on the same processor with its primary job, large tasks with utilization greater than 50% cannot be managed under RAPM, which limits its applicability.

Instead of dedicating a spare processor for backup tasks as in [9], [14], the designs in [13] and [20] treat both processors equally and schedule a mixed set of primary and backup copies of different tasks on each of them. Specifically, based the Preference-Oriented Earliest Deadline (POED) scheduling algorithm, the authors of [13] developed energy-efficient fault tolerance (EEFT) schemes. Here, on each processor, primary tasks are executed as soon as possible while backup tasks are delayed as much as possible under POED. That is, the slack time on both processors can be efficiently utilized to minimize the overlapped execution of the primary and backup copies of the same task for better energy savings. However, the work only considered dual-processor systems. In our future work, we will investigate how to extend the EEFT scheme to multiprocessor systems and compare it against with our proposed Standby-Sparing based schemes.

VII. Conclusions

The *Standby-Sparing (SS)* technique has been studied as an effective technique to address both energy and reliability issues in dual-processor systems [9], [14]. However, no existing work has analyzed applicability of this framework to systems with more than two processors. In this paper, with the focus on multiprocessor systems, we studied the SS-based energy efficient fault tolerance techniques that can address both transient and permanent faults. Specifically, we first proposed the *Paired-SS* scheme, where processors are organized as groups of two (i.e., pairs) and SS is applied directly within each processor pair after partitioning tasks. Then, we proposed a *Generalized-SS* technique that partitions processors into two groups, which are for *primary* and *secondary* processors, respectively. The *main* (*backup*) copies of tasks are executed on primary (secondary) processor group under the *partitioned-EDF* (*partitioned-EDL*) scheduling. An online energy management scheme with wrapper-tasks is also studied.

Our empirical evaluations show that, for systems with a given number of processors, there normally exists an energy-efficient configuration of primary and secondary processors for the Generalized-SS technique that can lead to better energy savings when compared to that of the Paired-SS technique. Moreover the wrapper-task based online scheme has a pretty stable performance for both P-SS and G-SS schemes.

Acknowledgments

This work was supported by US National Science Foundation awards CNS-0855247, CNS-1016855, CNS-1016974, and CAREER Award CNS-0953005.

References

[1] Intel corporation. single-chip cloud computer: Project. http://www.intel.com/content/www/us/en/research/intel-labs-single-chip-cloud-computer.html, 2013. [Online; accessed 10-June-2013].

[2] Tilera, tile-gx processor family. http://www.tilera.com/products/processors/, 2013. [Online; accessed 10-June-2013].

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

[4] H. Aydin and Qi Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proc. of the Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2003.

[5] E. Bini and G.C. Buttazzo. Biasing effects in schedulability measures. In *Proc. of the Euromicro Conf. on Real-Time Systems*, 2004.

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

[7] H. Chetto and M. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Trans. Softw. Eng.*, 15:1261–1269, 1989.

[8] D. Clark, C. Brennan, and M. Jaunich. Survey of nasa mars exploration missions software fault protection architecture. available at http://mkjaunich.typepad.com/files/ssw689-surveyofnasamarsexpnmsn-final.pdf, 2010. [Online; accessed 26-March-2013].

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

[10] 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.

[11] 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.

[12] Y. Guo, D. Zhu, and H. Aydin. Reliability-aware power management for parallel real-time applications with precedence constraints. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1 –8, july 2011.

[13] Y. Guo, D. Zhu, and H. Aydin. Efficient power management schemes for dual-processor fault-tolerant systems. In *First Workshop on Highly-Reliable Power-Efficient Embedded Designs*, Feb. 2013.

[14] M. Haque, H. Aydin, and D. Zhu. Energy-aware standby-sparing technique for periodic real-time applications. In *Proc. of the IEEE International Conference on Computer Design (ICCD)*, 2011.

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

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

[17] D. K. Pradhan, editor. *Fault-tolerant computer system design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

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

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

[20] 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 Symp. on Low Power Electronics and Design*, pages 124–129, 2002.

[21] 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.

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

[23] B. Zhao, H. Aydin, and D. Zhu. Energy management under general task-level reliability constraints. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Apr. 2012.

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

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

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

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