Energy Management of Standby-Sparing Systems for Fixed-Priority Real-Time Workloads

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*—Energy management and reliability are two important design objectives for real-time embedded systems. Recently, the standby-sparing scheme that uses a primary processor and a spare processor has been exploited to provide fault tolerance while keeping the energy consumption under control through DVS and DPM techniques. In this paper, we consider the standby-sparing technique for fixed-priority periodic real-time tasks. We propose a dual-queue mechanism through which the execution of backup tasks are maximally delayed, as well as online algorithms to manage energy consumption. Our experimental results show that the proposed scheme provides energy savings over time-redundancy based techniques while offering reliability improvements.**

I. Introduction

Energy efficiency is a major design dimension for many embedded systems. Several energy management techniques have been widely studied in recent literature. *Dynamic Voltage Scaling* (DVS) reduces the energy consumption by switching CPU frequency and voltage to low levels [20]. Another well-known technique is the *Dynamic Power Management (DPM)*, through which the system is put to sleep/low-power states when it is idle. A key challenge in DPM is to guarantee that the energy saved in the low-power state is not offset by the time/energy overhead involved in power state transitions [2], [7]. Moreover, the timing constraints of real-time embedded applications impose strict constraints on the applicability of these techniques [15], [20], [22]. A few recent studies investigated how to combine DVS and DPM to maximize energy savings in the context of a single real-time application [8], [29].

Another increasingly important design objective is *reliability*. In fact, computer systems are vulnerable to faults which often manifest as runtime *errors*. Faults are generally classified as *transient* or *permanent* faults [21]. The transient faults which lead to temporary *soft errors* (or *single event upsets (SEUs))* are known to be more frequent than the permanent faults [14], [30]. Transient faults are often induced by electromagnetic interference and cosmic radiations [14]. Most importantly, the increase in component density of CMOS circuits and aggressive power management schemes significantly increase the vulnerability of systems to transient faults [11], [30]. A common way to deal with transient faults is to rely on additional slack time (time redundancy) to re-invoke the faulty tasks [21], [32]. Permanent faults, on the other hand, are caused by hardware failures, including manufacturing defects and circuit wear out. This may lead to the unavailability of the system for extended time periods until it is repaired or replaced. In real-time embedded systems that need to be operational continuously, permanent faults can only be dealt with extra processors (hardware redundancy) [21].

Given the importance of both design dimensions, the research community has recently started to explore the *co-management of energy and reliability* more aggressively. For example, the popular DVS technique tends to increase the rate of transient faults [11], [30] at low voltage/frequency levels. Consequently, several studies focused on mitigating the reliability degradation due to DVS by provisioning for extra *recovery* tasks that are invoked at maximum frequency if errors are detected in the scaled tasks [19], [27], [28], [32]. These time-redundancy based techniques are applicable to, and explored mostly on, the single-processor settings.

The increasing availability of multicore/multiprocessor systems is also making the deployment of additional processor units in the co-management of energy and reliability more appealing. With extra processors (hardware redundancy), the system can sustain permanent faults of some processors. A particularly interesting framework, in that regard, is the *standby-sparing systems* [9], [10], [26].

In a standby-sparing solution, a dual-processor system that consists of a *primary* and a *spare* processor is deployed. Associated with each task executed on the primary processor, a separate *backup* task is scheduled on the spare processor. The system inherently tolerates the permanent fault of any of the processors. Moreover, the primary processor uses both DVS and DPM: the aim is to minimize the energy consumption of the real-time workload. The spare processor uses only DPM; the objective is to keep the spare in idle/sleep states during long intervals by invoking the backups as late as possible. Consequently, several solutions are suggested in the literature to minimize the *overlap* between the two copies of the same task on both processors, in order to be able to cancel the high-cost backup execution on the spare when the copy on the primary completes successfully [9], [10], [26]. These solutions are limited to aperiodic, non-preemptive workloads, while a significant portion of the applications on real-time embedded systems are preemptive and periodic in nature. Moreover, they rely on the existence of the entire static feasible schedule so that it can be manipulated to obtain the schedule of the backup tasks. The work in [13] provides a solution for periodic preemptive workloads scheduled with the *Earliest Deadline First (EDF)* policy on the primary. However, the solution is computationally expensive (it generates beforehand the entire schedule for the hyperperiod using the *Earliest Deadline Late (EDL)* algorithm [5]) and is not applicable to fixed-priority systems that are more frequently deployed.

In this paper, we consider a dual-processor standby-sparing system that executes a fixed-priority periodic real-time workload. As in existing standby-sparing solutions, the primary processor uses both DVS and DPM, while the spare relies only on DPM for energy management. To address the problem of maximally delaying the backup tasks on the spare, we propose an efficient *dual-queue* mechanism. Our solution is inspired by the *Dual-Priority Scheduling* framework [6] which was originally proposed to improve the responsiveness of soft real-time tasks in a system with a hard real-time workload. We show how this solution can be coupled with a DVS-enabled low-energy schedule on the primary to achieve energy savings. Moreover, we provide a *delayed promotion* rule that dynamically postpones the execution of the pending backups when the earlier backup tasks are cancelled at runtime. The experimental results suggest that while offering definitive advantages on the reliability side and an ability to withstand permanent faults, our *standby-sparing fixed-priority (SSFP)* algorithm provides also non-trivial energy gains over the traditional time-redundancy solutions for medium-to-high load conditions. In addition, our framework has low computational complexity and run-time overhead, making it appealing for periodic and preemptive execution settings.

The rest of the paper is organized as follows. In Section II, we present our workload and power models, and our assumptions. In Section III, we elaborate on the features of the standby-sparing solutions in joint management of reliability and energy. We review the principles of Dual-Priority Scheduling in Section IV. Then the details of our solution are presented in Section V. Section VI presents our experimental evaluation and finally we conclude in Section VII with a summary of our contributions.

II. Models and Assumptions

A. Workload Model

We consider a set of periodic real-time tasks $\Psi = \{\tau_1, ..., \tau_n\}$. Each task τ_i has the period P_i and the worst-case execution time c_i under the maximum available CPU frequency. The j^{th} job of task τ_i (namely, $J_{i,j}$) arrives at time $r_{i,j} = (j-1) \cdot P_i$ and must complete by its deadline $j \cdot P_i$. Hence, the relative deadline D_i of job $J_{i,j}$ is equal to the period P_i. The *utilization* of task τ_i is defined as $\frac{c_i}{P_i}$ and the *total utilization* U_{tot} is the sum of individual task utilizations.

For reliability and fault tolerance purposes, we associate with each task τ_i a *backup* task B_i having the same timing parameters as τ_i. The j^{th} instance (job) of B_i is denoted by $B_{i,j}$. To distinguish with the backup tasks, we occasionally use the term *main task* to refer to a task in Ψ. The aggregate workload that consists of the main and backup tasks are executed on a dual-processor standby-sparing system with one *primary* and one *spare* processor [9], [21]. On each processor, tasks are scheduled according to Rate Monotonic Scheduling (RMS) policy, which is known to be optimal for fixed-priority periodic workloads [18].

B. Power Model

Each processor has the capability of operating in three different power modes. The tasks are executed in the *active* state of the processor. When the processor is not executing tasks, it can be in *idle* or *sleep* states. We now describe the power characteristics of each of these states.

1) **Active:** We model the power consumption in the active mode following recent works on energy and reliability management [25], [28], [32]. The power consumption of the system consists of static and dynamic power components. The *static power* P_s is dominated by the leakage current of the system. The *dynamic power* P_d includes a frequency-independent power component P_{ind} driven by the modules such as memory and I/O subsystem in the active state, and a frequency-dependent power component which depends on the supply voltage and frequency of the system.

$$P_{active} = P_s + P_{ind} + C_e V_{dd}^2 f \qquad (1)$$

Above, C_e denotes the effective switching capacitance. The processor supply voltage V_{dd} has a linear relationship with frequency f. Therefore, Equation (1) can be re-written as,

$$P_{active} = P_s + P_{ind} + C_e f^3$$

When the voltage/frequency scaling is applied through the DVS technique, the processor frequency can be adjusted within a range between a minimum CPU frequency f_{min} and a maximum CPU frequency f_{max}. All frequency values in the paper are normalized with respect to f_{max} (i.e. $f_{max} = 1.0$). Note that the existence of P_s and P_{ind} implies the existence of a threshold frequency, called *critical speed* or *energy-efficient frequency*, below which DVS ceases to be effective [15], [30].

2) **Idle:** The processor can switch to idle power state when it is not executing any task. In this state, the processor consumes low dynamic power, P_0. The overhead for transitioning to idle state and back is not significant; for example, the processor can return to active state within 10 ns in recent processor designs [16]. Hence, the power consumption in idle state is given by:

$$P_{idle} = P_s + P_0$$

3) **Sleep:** Sleep state is the lowest power state for the processor. In this state, power components other than the static power P_s become negligible. Ideally, we would like to put the processor to sleep state whenever the system is idle. However, putting a processor to sleep state involves significant time and energy overhead [33]. Due to this transition overhead, the concept of break-even time (Δ_{crit}) is introduced in literature [2], [4]. If the processor is put to sleep state for at least as long as Δ_{crit} time, the energy savings in the sleep state can amortize the transition overhead. This is the key idea behind the Dynamic Power Management (DPM) scheme [2], [4], [7]. In DPM scheme, when the idle interval is expected to exceed Δ_{crit}, the processor is transitioned to sleep state for energy savings. Therefore, it is beneficial to have longer idle intervals to take advantage of the DPM technique. If the idle interval is expected to be relatively short, the processor switches to *idle* state instead.

C. Fault Model

The system that we consider may be subject to both permanent and transient faults. Our system, by taking advantage of the hardware redundancy provided by the *primary* and *spare* processor, can tolerate at most one permanent fault. We consider only the permanent fault of processing units. The permanent faults of other components in the system (e.g. main memory) are not considered in this work.

We consider a transient fault model similar to [30], [32]. The faults occur according to Poisson distribution with a known average rate λ [27]. The average fault rate λ is dependent on the CPU frequency. In fact, λ increases exponentially with the decrease in CPU frequency [11], [30]. Suppose, the average fault rate at the maximum CPU frequency is denoted by λ_0. Then, the average fault rate for a frequency f can be expressed as [30]:

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

The exponent d (typically a constant > 0) represents the sensitivity of the system to voltage scaling. With higher values of d, the reliability of the system degrades rapidly with system voltage.

The *reliability* of a job is defined as the probability of executing the task successfully in the presence of potential transient faults. The reliability of a single job J_i running at frequency f_i can be expressed as [30]:

$$R_i(f_i) = e^{-\lambda(f_i)\frac{c_i}{f_i}}$$

The *probability of failure* for the job J_i is then given by:

$$PoF(f_i) = 1 - R_i(f_i)$$

At the completion of a job in any processor, the system initiates an *acceptance test* [21], [32] considering the output of the job. The result of this acceptance test is used to determine the occurrence of errors induced by transient faults.

III. Standby-Sparing Systems

Standby-sparing system solutions have been recently explored to enhance the reliability of real-time embedded systems, by exploiting increasingly available dual-processor settings [9], [10], [13]. Here, the dual-processor system is configured as a *primary* and a *spare* processor. The primary processor has both DVS and DPM capability, and executes the main tasks of the workload. The backup tasks, each associated with a main task, are scheduled on the spare processor. The spare processor does not employ voltage/frequency scaling; hence it can delay the execution of the backup tasks as much as possible, and execute them at the maximum processing speed before their deadlines when needed.

At the completion of each job, the acceptance test is performed to determine the existence of an error induced by a transient fault. If no error is detected, the copy running (or, scheduled to run) on the other processor is cancelled; otherwise that copy is executed according to the schedule on its own processor.

Standby-sparing systems have the following features:

- The primary processor can use both DVS and DPM as needed and in tandem to reduce the energy consumption by employing sophisticated system-level energy management solutions. On the other hand, by *delaying the backup tasks as much as possible and cancelling them when the main copy completes successfully,* the extra energy overhead due to the second (spare) processor is significantly reduced, thanks to the use of DPM.
- By scheduling the main and backup copies of all the jobs on separate processors, the system can tolerate the *permanent* fault of a single processor: the functional processor can finish the workload even if the faulty processor remains unavailable.
- In terms of robustness with respect to *transient faults*, by scheduling a backup copy of each job at the maximum frequency (if needed), the reliability loss due to the application of DVS on the primary processor is fully mitigated [32].

Despite these promises, the main technical challenge in standby-sparing systems is how to *delay the backup tasks* on the spare processor *while still guaranteeing their deadlines with low computational overhead.* Notice that if both the main and backup copies of a given job are scheduled *concurrently* on two processors, the power consumption significantly increases due to high-power profile of the spare processor. Consequently, a key issue is to *minimize the concurrent executions of the main and backup tasks as much as possible*. For periodic workloads scheduled by preemptive scheduling policies (such as RMS), reaching these objectives with low overhead is particularly challenging. Our solution to this problem is based on *dual-priority scheduling* framework, which is described next.

IV. Dual-Priority Scheduling

Dual-priority scheduling [6] was originally proposed to improve the response time of soft (or, non-real-time) tasks (SRTs) in a system that also executes periodic hard real-time (HRT) tasks according to the RMS policy. Specifically, the scheme uses three ready queues, denoted as *lower, middle,* and *upper* queues. The names of the queues reflect their execution priorities: the scheduler first executes jobs in the upper queue. Jobs in the middle and lower queue are executed, and in that order, only if the upper queue is empty.

SRTs always execute in the middle queue. An HRT instance, on the other hand, is first put to the lower queue upon its release. However, after a certain time interval, the HRT instance is *promoted* to the upper queue and is eligible for urgent service. The jobs in the upper queue are executed according to rate-monotonic priorities.

The main objective of the scheme is to offer relatively fast service to SRT instances as long as the timeliness of the HRT instances is not compromised. The key problem in dual-priority scheduling is to determine the *promotion time* for HRTs, to make sure that they will eventually make their deadlines in the upper queue, using RMS. The promotion time is computed based on the worst-case response time of the task under RMS. Specifically, if S_i is the worst-case response time of the task with relative deadline D_i under RMS (which can be computed through well-known analysis techniques [1]), then the promotion time for τ_i, after its release time is computed as:

$$Y_i \;=\; D_i - S_i \tag{2}$$

The above result follows from the fact that if all other high priority HRTs were to be promoted simultaneously to the upper queue at time Y_i (which would maximize the response time of τ_i under RMS [18]), then τ_i would be still able to meet its deadline.

V. Standby-Sparing for Fixed-Priority Scheduling

Our proposal in this work is based on the observation that the dual-priority mechanism provides a powerful basis to manage the execution of the backup copies on the spare processor with low offline and online computational overhead. To illustrate the main components of our solution, we first present a running example. Consider three periodic tasks τ_1, τ_2 and τ_3. The worst-case execution time and periods of the tasks are given as $c_1 = 2, P_1 = 10, c_2 = 2, P_2 = 15, c_3 = 3$ and $P_3 = 30$. Figure 1 shows the schedule for this task set during the hyperperiod (the least common multiple of all the task periods), when executed according to the rate-monotonic priorities. In the figure, the arrows indicate the arrival times of jobs of periodic tasks.

Now consider a *dual queue* mechanism to delay the execution of the tasks. Specifically, at arrival, jobs are put to the lower queue and after the corresponding promotion time they are promoted to the upper queue. The promotion times are computed statically for each task before execution. There is

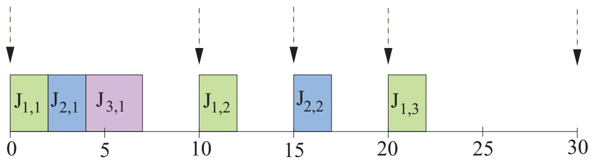

Fig. 1. A typical fixed-priority schedule

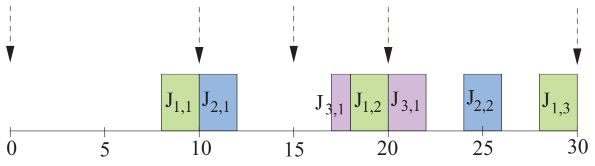

Fig. 2. Fixed-priority schedule with delayed execution

no middle queue and jobs are executed only when they are in the upper queue based on RM priorities. This will essentially delay the execution of each backup job while still meeting its deadline. For the computation of the promotion times, we note that there are multiple techniques to compute the worst-case response time of the tasks [1]. For example, the time-demand analysis technique (TDA) [17] can be used to compute the exact worst-case response time of each task. However, it is of pseudo-polynomial time complexity and it may involve non-trivial overhead as the ratio of the maximum period to the minimum period gets larger. Instead, in this paper, we adopt the conservative but fast (linear-time) technique used in [20] to compute an upper bound on the response time of task τ_i:

$$S_i \;=\; \Sigma_{\tau_j \in\; hp(\tau_i)} \lceil (P_j/P_i) \rceil \times c_j \;+\; c_i \tag{3}$$

where, $hp(\tau_i)$ is the set of tasks with priority higher than τ_i.

By substituting Equation (3) in (2), the promotion times ($Y_i = D_i - S_i$) for the example task set can be computed as: $Y_1 = 8, Y_2 = 9$, and $Y_3 = 17$. Figure 2 shows the delayed execution scenario according to these above mentioned principles. Despite the explicitly enforced delays, all jobs still meet their deadlines. We underline that the formula (3) will be also instrumental for our *online delayed promotion* rule that further improves the performance.

Now, consider a dual-processor standby-sparing scheme where the DVS-enabled primary executes the main tasks according to RMS. The total utilization of the task set is 0.433. The Liu-Layland utilization bound [18] for task sets with 3 tasks is 0.88. So, we can safely slow down the primary processor by using the frequency $f = 0.5$. The spare executes the backups $\{B_{i,j}\}$ through the described dual-queue mechanism at the maximum frequency. Figure 3 shows the corresponding schedules for the primary and spare processors. It is notable that, thanks to the dual-queue mechanism, the backups on the spare are significantly delayed and the overlaps with the main tasks on the primary are minimized.

In fact, this feature enables us to cancel backup jobs when the main copy of the job completes successfully (i.e. without incurring transient faults) on the primary. Thus, we can avoid the execution of the backups by coupling it with the primary

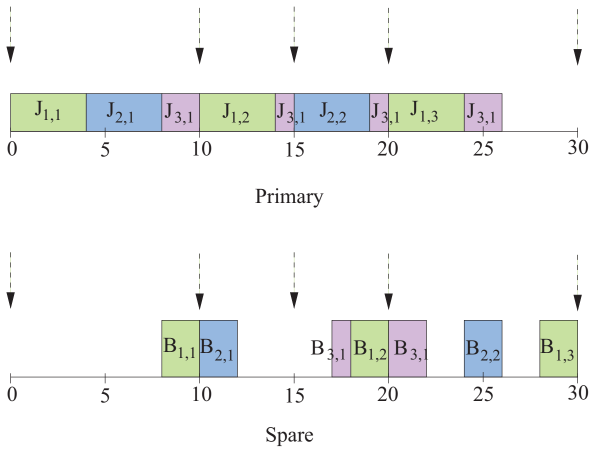

Fig. 3. Coupled schedules on the primary and spare processors

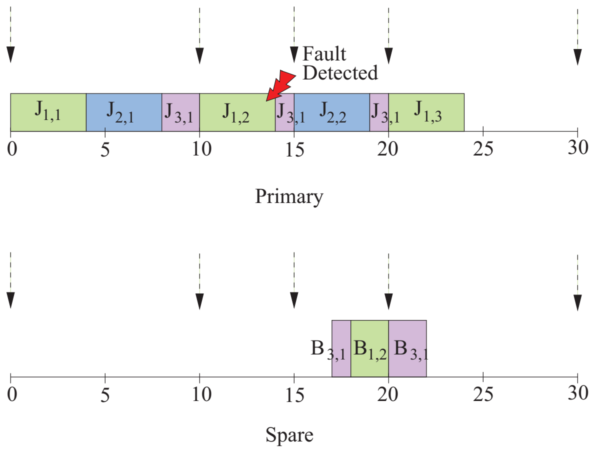

Fig. 4. Taking advantage of successful job completions

schedule. For example, assume that $J_{1,1}$ and $J_{2,1}$ complete successfully on the primary; then $B_{1,1}$ and $B_{2,1}$ will be completely cancelled. $J_{3,1}$ will be preempted by $J_{1,2}$, which is assumed to be subject to a transient fault (Figure 4). This implies that the backup job $B_{1,2}$ will need to be executed according to the pre-computed schedule on the spare. Note that, $B_{3,1}$ starts executing at its promotion time 17 as its main copy (which was preempted) did not complete yet. However, when the backup copy $B_{3,1}$ completes at time 22, the remaining fraction of the main job $J_{3,1}$ can be also cancelled. Assuming that all the remaining main tasks complete successfully, we obtain the schedules in Figure 4.

Further online optimizations are also possible. In fact, main tasks on the primary processor typically complete successfully as faults are relatively rare. In addition, the worst-case execution time is often a pessimistic estimate of the actual execution time. This also increases the chance of early completion or entire cancellation of the backups on the spare processor. Whenever a backup is cancelled or completes early, *the runtime slack can be used to further delay the promotion time of other pending backup jobs.*

Suppose that a backup job $B_{i,j}$ has been cancelled after executing $a_{i,j}$ units of time from its allocated c_i units of CPU time. Note that $a_{i,j} = 0$ if $B_{i,j}$ is cancelled entirely. All pending backup jobs released before $B_{i,j}$ and with a priority lower (hence, periods larger) than $B_{i,j}$ will benefit from the reduced interference of $B_{i,j}$.

We use the notation $\Gamma_{i,j}$ to denote the set of backup jobs that benefit from (i.e. that can be promoted later with) the early completion of $B_{i,j}$. Formally:

$$\Gamma_{i,j} = \{B_{k,l} | B_{k,l} \text{ is in lower queue } \wedge (P_k > P_i) \wedge (r_{k,l} \leq r_{i,j})\}$$

The following theorem gives the amount by which the promotion time of a job $B_{k,l} \in \Gamma_{i,j}$ can be delayed without missing its deadline.

***Theorem** 1:* If a backup job $B_{i,j}$ completes or is cancelled after executing for $a_{i,j}$ units of time, the promotion time for any job $B_{k,l} \in \Gamma_{i,j}$ can be delayed by $c_i - a_{i,j}$ units of time.

Proof: Let us consider any arbitrary job $B_{k,l} \in \Gamma_{i,j}$. The release time of $B_{k,l}$ is $r_{k,l}$ and its absolute deadline is $D_{k,l}$. Assume that the earlier promotion time was t_0 which is delayed to t_1. Now according to the rule of determining promotion time, $(D_{k,l} - t_0)$ is the maximum response time for $B_{k,l}$. According to the definition of $\Gamma_{i,j}$, B_i has priority higher than B_k and $B_{i,j}$ is released after $B_{k,l}$. Therefore $B_{i,j}$ is supposed to interfere with the execution of $B_{k,l}$ and it is considered in the response time of $B_{k,l}$. Suppose that the set of tasks with priority higher than B_k is denoted by $hp(B_k)$ and for each task $B_m \in hp(B_k)$, n_m instances interfere with $B_{k,l}$.

$$D_{k,l} - t_0 = c_k + \Sigma_{B_m \in hp(B_k)} n_m \times c_m$$

This can be rewritten as,

$$D_{k,l} - t_0 = c_k + \Sigma_{B_m \in \{hp(B_k) - B_i\}} n_m \times c_m + (n_i - 1) \times c_i + c_i$$

where the last component (c_i) represents the worst-case interference due to $B_{i,j}$. Recall that, by assumption, $B_{i,j}$ completes after executing $a_{i,j} \leq c_i$ time units. Consequently the response time of $B_{k,l}$ decreases by an amount of $c_i - a_{i,j}$ and its promotion time t_0 can be delayed to $t_1 = t_0 + c_i - a_{i,j}$ without compromising its deadline. ∎

Figure 5 shows an example of delayed promotion enabled by the result in Theorem 1. As backup tasks $B_{1,1}$, $B_{2,1}$ and $B_{2,2}$ are cancelled, the promotion time for the backup task $B_{3,1}$ can be delayed to 23 beyond the originally computed instant of 17. As a result, the chances of cancelling $B_{3,1}$ would be higher in an actual execution, in particular if the main job $J_{3,1}$ completes earlier than the worst-case.

Algorithm SSFP. We are now ready to present the full details of our algorithm SSFP. The primary processor schedules tasks without any delay and uses both DVS and DPM. Its supply voltage/frequency can be selected according to various algorithms proposed in literature [20], [22], [23]. The spare processor, on the other hand, uses the dual-queue mechanism and applies only DPM for energy management. The promotion time of each back-up job $B_{i,j}$ is computed by adding its

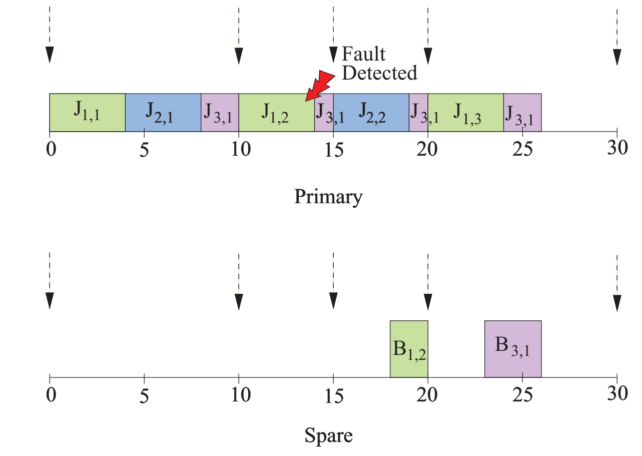

Fig. 5. Delaying promotions at run-time

Algorithm 1 Standby-Sparing for Fixed-Priority (SSFP) (Events on the primary processor)

```
Event - A job of $\tau_i$, $J_{i,j}$ is released:
    Add $J_{i,j}$ to the ready queue on primary
    Add $B_{i,j}$ to the lower queue on spare
    $Y_{i,j} \leftarrow Y_i$
    Set timer for promotion event at $t = time + Y_{i,j}$
    Dispatch the highest RM priority job on primary

Event - The job $J_{i,j}$ completes on primary:
    Run the acceptance test for $J_{i,j}$
    if no error is detected and $B_{i,j}$ is not completed yet then
        Cancel $B_{i,j}$ on spare
    end if
    if ready queue of primary is empty then
        $time\_to\_next\_arrival \leftarrow$ time to earliest
                                               release time
        $\Delta_p \leftarrow time\_to\_next\_arrival$
        if $\Delta_p \geq \Delta_{crit}$ then
            Put primary to sleep state for $\Delta_p$ units of time
        end if
    else    /* jobs are available for execution */
        Dispatch the highest RM priority job on primary
    end if
```

Algorithm 2 Standby-Sparing for Fixed-Priority (SSFP) (Events on the spare processor)

```
Event - A backup job $B_{i,j}$ is promoted:
    Add $B_{i,j}$ to the upper queue on spare
    Dispatch the highest RM priority job on spare

Event - A backup job $B_{i,j}$ completes:
    Run the acceptance test for $B_{i,j}$
    if $J_{i,j}$ is not completed yet then
        Cancel $J_{i,j}$ on primary
    end if
    $\gamma \leftarrow c_i - a_i$
    for every $B_{k,l}$ in $\Gamma_{i,j}$ do
        $Y_{k,l} = Y_{k,l} + \gamma$
        Set new promotion event
    end for
    if $B_{i,j}$ is the current active job then
    /* Check if the spare can 'sleep' in the slack time of $B_{i,j}$*/
    $time\_to\_next\_promotion \leftarrow$ time to earliest
                                            promotion event
        $\Delta_s \leftarrow \min\{\gamma,\ time\_to\_next\_promotion\}$
        if $\Delta_s \geq \Delta_{crit}$ then
            Set wake-up event at $t = time + \Delta_s$
            Put spare to sleep state
        end if
    end if

Event - the spare processor wakes up:
    end for
    if the upper queue is not empty then
        /* There are backups not yet cancelled */
        Dispatch the highest RM priority job on spare
    else
        $\Delta_s \leftarrow time\_to\_next\_promotion$
        if $\Delta_s \geq \Delta_{crit}$ then
            Set wake-up event at $t = time + \Delta_s$
            Put spare to sleep state
        end if
    end if
```

release time to Y_i, the pre-computed promotion time for task B_i.

The algorithm is invoked at every job release, completion, and cancellation time. The detailed pseudo-code is presented in Algorithms 1 and 2. Algorithm 1 shows the events on the primary processor and the corresponding actions. At job arrival, the main job is added to the ready queue. A corresponding backup job is also added to the lower queue on the spare. With the pre-computed promotion time, a timer is set accordingly to promote the backup job to the upper queue in the future. The primary processor then dispatches jobs according to RMS policy. When a job completes on the primary, we invoke the corresponding acceptance test [21] to check the sanity of the computed result. If the acceptance test does not detect any error, we cancel the corresponding backup task in the spare processor. Then, the primary processor continues to execute the next job in the ready queue. However, if there is no ready task available for execution, the processor will remain idle. The primary processor will start executing jobs again when the next job arrives. Considering task period values, we can compute the earliest arrival times among all future jobs in linear time. The time to the earliest arrival time is denoted by the *time_to_next_arrival* in the pseudo code. If the idle time exceeds the break-even time Δ_{crit}, the primary processor is put to sleep until next arrival.

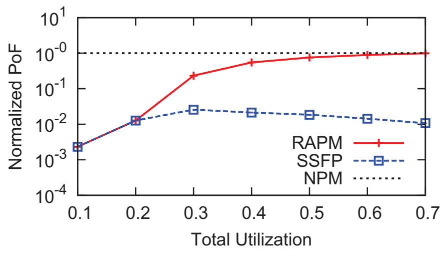

Fig. 6. Impact on Probability of Failure

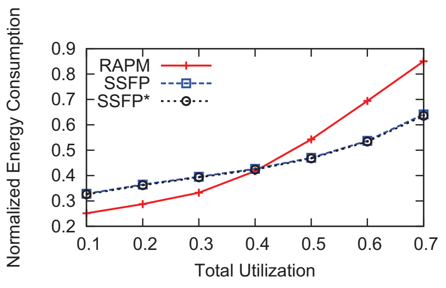

Fig. 7. Effect of System Utilization

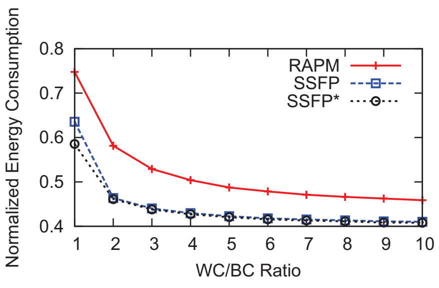

Fig. 8. Effect of Workload Variability

Algorithm 2 gives the actions taken in response to the events on the spare processor. Whenever a job is promoted to the upper queue it is eligible for execution. The spare processor then dispatches the job at the highest RMS priority level, without any voltage scaling. When a backup job $B_{i,j}$ is completed/cancelled, if the corresponding main task $J_{i,j}$ has not been completed yet, the execution of $J_{i,j}$ is also cancelled on the primary processor. We then compute the runtime slack γ generated by $B_{i,j}$ and delay the promotion times of the eligible pending tasks γ units of time according to Theorem 1. If $B_{i,j}$ is the current active task on the spare processor and no additional task is scheduled to be promoted within γ units of time, the spare processor can remain idle for γ units of time. However, if a job is promoted before that, the spare will have to resume execution to avoid any deadline violation. The earliest promotion time among all instances of the jobs can be also computed in linear time. The time to earliest promotion event is denoted by the variable *time_to_next_promotion* in the pseudo-code. If the idle interval is greater than Δ_{crit}, the spare is put to sleep and the corresponding wake-up event is scheduled. As the wake-up timer expires, the wake-up event handler in the spare is initiated. At wake-up, the spare inspects the upper queue. If the upper queue is empty, an attempt is made to switch to sleep state by considering the next promotion time. Otherwise, the highest-priority job is dispatched.

VI. Evaluations

To evaluate the performance of our scheme experimentally, we constructed a discrete-event simulator in C. We compare our scheme against the state-of-the-art time redundancy-based energy and reliability management technique RAPM [31], [32]. RAPM selects a subset of the main tasks for slowdown through DVS and schedules a separate recovery task for each of those tasks to mitigate the reliability loss due to voltage scaling. One advantage of RAPM is that both the main and recovery tasks can be executed on the same processor. Hence, unlike standby-sparing systems, it requires only one processor and avoids the potential energy overhead of the spare processor. However, this is also a shortcoming in terms of inability to tolerate possible permanent fault of the processor. In addition, due to the limited computational power, the workload may need to be executed at high frequency to meet the deadlines.

We also implemented a clairvoyant version of our SSFP scheme, called SSFP*. SSFP* has a priori information about the actual execution time of the tasks and hence can make optimal DPM decisions in terms of putting the processor to *sleep* state when it is idle. Unlike SSFP, SSFP* is not practical; but it is included as a yardstick algorithm. We do not include any comparison with [9], [10], [26], as they are limited to aperiodic, non-preemptive workloads. Similarly, the scheme in [13] is not included as it targets dynamic-priority EDF-based periodic systems and requires constructing the full schedule for the hyperperiod in advance.

In our simulations, for each data point, we conducted 1000 experiments and computed the average. The results are normalized with respect to the scheme which executes the main tasks at the maximum frequency without any power or reliability management. We call this scheme *NPM (No Power Management)*. We evaluated the performance of SSFP and RAPM across different system parameters including the total utilization (U_{tot}), the ratio of worst-case to best-case execution time (WC/BC), the state transition overhead and the number of tasks. The worst-case utilizations of the tasks are generated randomly using the *UUnifast* scheme [3]. The periods are generated randomly between 10 and 100 ms. Given the worst-case utilization of a task, its worst-case execution time (WC) is computed as the product of its period and worst-case utilization. Following [12], [24], the actual execution time of a task instance is then obtained randomly according to the normal distribution with mean $(WC + BC)/2$ and variance $(WC + BC)/6$ to ensure that 99.7% of the actual execution times lies within the $[BC, WC]$ range of the task. The default value for the WC/BC ratio is 5 and the number of tasks in each task set is 15 unless otherwise specified.

The energy parameters are computed based on the Freescale MPC8536 processor [33]. The default value for static power and frequency-independent power consumption are set to 5% and 15% of the maximum frequency-dependent power consumption, respectively. The energy-efficient state transition time Δ_{crit} is set to 1500 μs [33]. On the primary, we use the *Cycle-Conserving DVS* algorithm proposed for RMS in [20].

We first evaluate the reliability performance of the schemes with respect to transient faults. Figure 6 shows the *probability of failure (PoF)* trends, which is defined as $1 -$ *reliability* [32]. Clearly, the lower *PoF*, the higher the reliability. Using the analytical formulations of system reliability as a function of the processing frequency and the number of backups (the details of the reliability evaluation can be found in Appendix), we computed the *PoF* value for each task set. The results are

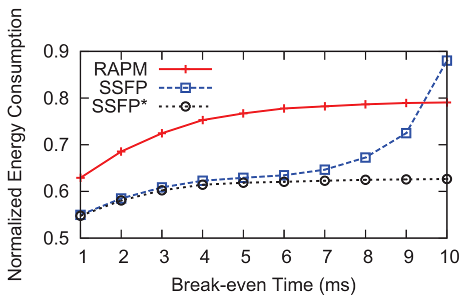

Fig. 9. Impact of Break-even Time

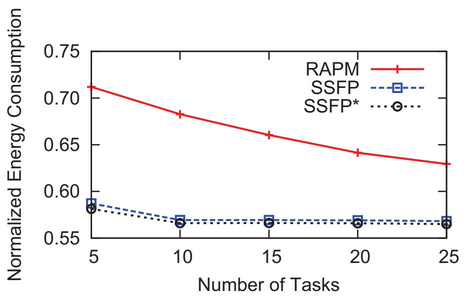

Fig. 10. Impact of Number of Tasks

normalized with respect to that of NPM scheme, which also represents the original reliability of the system in the absence of power or reliability management. At very low utilization levels RAPM can reserve a recovery for every task, and achieve low *PoF* figures comparable to SSFP. As the system load (utilization) increases, RAPM is unable to assign recovery tasks to some main tasks and its *PoF* increases (reliability degrades), approaching that of NPM. SSFP, on the other hand, can maintain a high system reliability as it always allocates a backup task for every main task. We observe that SSFP can offer to up to 100 times lower *PoF* numbers compared to RAPM. Also note that SSFP can effectively tolerate the permanent fault of any single processor.

Next we evaluate the impact of the system load, as we increase the total utilization from 0.1 to 0.69 (the asymptotic schedulability bound for RMS [18]), in Figure 7. We observe that at low utilization values, RAPM consumes less energy than the proposed SSFP schemes due to the static power consumed by the spare processor. However, as utilization increases, SSFP outperforms RAPM and can achieve up to 25% additional energy savings when U_{tot} exceeds 0.4. The main reason is that RAPM is forced to run at high frequency at high utilization values, consuming excessive energy. On the other hand, SSFP can still adopt relatively low execution frequencies on the primary, by dynamically delaying and, in many cases, cancelling the backup tasks on the spare. SSFP* can offer marginally better energy performance in these settings thanks to the exact prediction of the idle intervals.

Figure 8 shows the impact of workload variability. For these experiments, we set the total utilization to 0.5 and vary the WC/BC ratio from 1 to 10. As this ratio increases, the actual workload increasingly deviates from the worst-case, and the energy consumption decreases for all schemes. We observe that SSFP outperforms RAPM in the entire input spectrum. When $WC/BC = 1$, the algorithms differ in their handling of static slack. SSFP* outperforms SSFP by taking advantage of the backup cancellation information to initiate some additional sleep intervals. As the WC/BC ratio increases, we have increasing dynamic slack. The early completions allow SSFP to further delay and cancel backup tasks.

We also explore the effect of state transition overhead when applying DPM to switch the processor to sleep states at run-time. As the state transition overhead increases, the system needs to stay in the sleep state longer to compensate the high overhead. Figure 9 shows the impact of varying Δ_{crit}. As Δ_{crit} increases, all schemes consume more energy due to longer enforced waits in idle state and less transitions to low-power sleep state. When Δ_{crit} approaches the minimum task period in the system, the energy consumption for SSFP increases sharply. The reason is that, at this point, SSFP can very rarely put the spare processor to sleep due to the timing constraints, consuming high energy. This is in contrast to SSFP* that can predict the actual length of the sleep interval and thus can take advantage of the backup cancellation to put the spare to sleep state. Figure 10 shows the impact of changing the number of tasks in the system. The performance of SSFP is not significantly affected by the number of tasks. However, with large number of small tasks, RAPM can make better use of the dynamic slack generated at run-time. Therefore its energy consumption drops with increased number of tasks.

VII. Conclusions

In this paper, we considered the problem of joint energy and reliability management for fixed-priority periodic real-time tasks. By using a dual-processor standby-sparing system and a dual-queue mechanism, we proposed the algorithm SSFP that delays the backup tasks on the spare as much as possible. When compared to the time-redundancy techniques experimentally, our solution is seen to save more energy at medium to high load values despite deploying the additional spare processor, while offering clear advantages in terms of reliability. To the best of our knowledge, this is the first work for energy-efficient scheduling of fixed-priority periodic tasks on a standby-sparing system.

Acknowledgements

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

References

[1] T. A. AlEnawy and H. Aydin. Energy-Aware Task Allocation for Rate Monotonic Scheduling. In *Proc. of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2005.

[2] L. Benini, A. Bogliolo and G. De Micheli. A Survey of Design Techniques for System-Level Dynamic Power Management. *IEEE Trans. on VLSI Systems*, vol. 8, no. 3, pp. 299 - 316, 2000.

[3] E. Bini, and G. C. Buttazo. Measuring the Performance of Schedulability Tests. *Journal of Real-Time Systems*, vol. 30, no. 1-2, pp. 129 - 154, 2005.

[4] J. J. Chen and T. W. Kuo. Procrastination Determination for Periodic Real-Rime Tasks in Leakage-Aware Dynamic Voltage Scaling Systems. In *Proc. of IEEE International Conference on Computer-Aided Design (ICCAD)*, 2007.

[5] H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Trans. on Software Engineering*, vol. 15, pp. 1261 - 1269, 1989.

[6] R. Davis and A. Wellings. Dual Priority Scheduling, In *Proc. of the 16th IEEE Real-Time Systems Symposium*, 1995.

[7] V. Devadas and H. Aydin. Real-Time Dynamic Power Management through Device Forbidden Regions. In *Proc. of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2008.

[8] V. Devadas and H. Aydin. On the Interplay of Voltage/Frequency Scaling and Device Power Management for Frame-based Real-Time Embedded Applications. *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 31 - 44, 2012.

[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 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2009.

[10] A. Ejlali, B. M. Al-Hashimi, and P. Eles. Low-Energy Standby-Sparing for Hard Real-Time Systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 3, pp. 329 - 342, 2012.

[11] 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*, vol. 6, pp. 10 - 20, 2004.

[12] F. Gruian. Hard Real-Time Scheduling for Low-Energy Using Stochastic Sata and DVS Processors. In *Proc. of IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2001.

[13] M. A. Haque, H. Aydin and D. Zhu. Energy-Aware Standby-Sparing Technique for Periodic Real-Time Applications. In *Proc. of IEEE International Conference on Computer Design (ICCD*, 2011.

[14] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh. Measurement and Modeling of Computer Reliability as Affected by System Activity. *ACM Trans. on Computer System*, vol. 4, pp. 214 - 237, 1986.

[15] R. Jejurikar , C. Pereira and R. Gupta. Leakage Aware Dynamic Voltage Scaling for Real-Time Embedded Systems. In *Proc. of IEEE/ACM Design Automation Conference (DAC'*, 2004.

[16] R. Kumar, and G. Hinton. A Family of 45nm IA Processors. In *Proc. of IEEE International Conference on Solid-State Circuits (ISSCC)*, 2009.

[17] J. Lehoczky, L. Sha and Y. Ding. The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior. In *Proc. of IEEE Real Time Systems Symposium*, 1989.

[18] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of ACM*, vol. 20, no. 1, pp. 46 - 61, Jan 1973.

[19] R. Melhem, D. Mossé, and E. Elnozahy. The Interplay of Power Management and Fault Recovery in Real-Time Systems. *IEEE Trans. on Computers*, vol. 53, pp. 217 - 231, 2004.

[20] P. Pillai, and K. G. Shin. Real-Time Dynamic Voltage Scaling for Low Power Embedded Operating System. *In Proc. of ACM Symposium on Operating Systems Principles*, 2001.

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

[22] G. Quan and X. Hu. Energy Efficient Fixed-Priority Scheduling for Real-Time Systems on Variable Voltage Processors. In *Proc. of IEEE/ACM Design Automation Conference*, 2001.

[23] S. Saewong, and R. Rajkumar. Practical Voltage-Scaling for FIxed-Priority RT-Systems. In *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03)*, 2003.

[24] Y. Shin and K. Choi. Power Conscious Fixed-Priority Scheduling for Hard Real-Time Systems. In *Proc. of IEEE/ACM Design Automation Conference (DAC'99)*, 1999.

[25] R. Sridharan and R. Mahapatra. Reliability Aware Power Management for Dual-Processor Real-Time Embedded Systems. In *Proc. of the 47th IEEE/ACM Design Automation Conference (DAC)*, 2010.

[26] M. K. Tavana, M. Salehi and A. Ejlali. Feedback-Based Energy Management in a Standby-Sparing Scheme for Hard Real-Time Systems. In *Proc. of the 32nd IEEE Real-Time Systems Symposium*, 2011.

[27] Y. Zhang and K. Chakrabarty. Dynamic Adaptation for Fault Tolerance and Power Management in Embedded Real-Time Systems. *ACM Trans. on Embedded Computer System*, vol. 3, pp. 336 - 360, May 2004.

[28] B. Zhao, H. Aydin, and D. Zhu. Enhanced Reliability-Aware Power Management Through Shared Recovery Technique, In *Proc. of IEEE International Conference on Computer Aided Design (ICCAD)*, 2009.

[29] B. Zhao and H. Aydin. Minimizing Expected Energy Consumption through Optimal Integration of DVS and DPM. In *Proc. of IEEE International Conference on Computer Aided Design (ICCAD)*, 2009.

[30] D. Zhu, R. Melhem, and Mossé. The Effects of Energy Management on Reliability in Real-Time Embedded Systems. In *Proc. of IEEE International Conference on Computer Aided Design (ICCAD)*, 2004.

[31] D. Zhu, X. Qi, and H. Aydin. Priority-Monotonic Energy Management for Real-Time Systems with Reliability Requirements. In *Proc. of the IEEE International Conference on Computer Design (ICCD)*, 2007.

[32] 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, 2009.

[33] FreeScale Semiconductor. MPC8536E Processor Specifications. http://www.freescale.com/files/32bit/doc/data_sheet/MPC8536EEC.pdf.

APPENDIX

Details of Reliability Evaluation

In this section, we provide the details of the reliability evaluation methodology we followed in Section VI. As discussed in Section II-C, the fault rate at frequency f is expressed as:

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

For the set of experiments in Figure 6, λ_0 is set to 10^{-6} and d is set to 2 [32]. The *reliability* of a job is defined as the probability of executing the task successfully in the presence of potential transient faults. The reliability of a single job $J_{i,j}$ running at frequency $f_{i,j}$ can then be expressed as [30]:

$$R_{i,j} = e^{-\lambda(f_{i,j})\frac{c_i}{f_{i,j}}}$$

The *system reliability* is the probability of executing all jobs correctly even in the presence of transient faults. The system reliability can be computed by evaluating the product of reliability figures over all the jobs [28].

$$R = \Pi_{\forall_{i,j}} R_{i,j} \tag{4}$$

We now discuss how we obtained overall reliability figures for SSFP, RAPM, and NPM. The SSFP scheme allocates a backup job for every main job in the system. The main job executes at a lower frequency according to DVS policy, while the backup job is executed at f_{max}. Therefore, a job will fail only if both the main job and the corresponding backup job fails. So, the reliability of a job in the SSFP system is:

$$R_{i,j} = 1 - [(1 - e^{-\lambda(f_{i,j})\frac{c_i}{f_{i,j}}})(1 - e^{-\lambda(f_{max})\frac{c_i}{f_{max}}})]$$

RAPM on the other hand selects a subset of task for slowdown. A recovery task is allocated for those tasks only. Moreover, the recovery task is executed only if the main task fails. The tasks that are not selected execute at f_{max}. So, if a task τ_i is chosen for slow down and hence is assigned a recovery task, the reliability for a job of τ_i can be computed as [31],

$$R_{i,j} = e^{-\lambda(f_{i,j})\frac{c_i}{f_{i,j}}} + (1 - e^{-\lambda(f_{i,j})\frac{c_i}{f_{i,j}}})e^{-\lambda(f_{max})\frac{c_i}{f_{max}}}$$

On the other hand, if τ_i is not assigned a replica, the reliability of one its jobs will be

$$R_{i,j} = e^{-\lambda(f_{max})\frac{c_i}{f_{max}}} \tag{5}$$

With the NPM scheme, the reliability level of of the system is equal to the *original* reliability level in the absence of voltage scaling and backup scheduling. NPM executes only the main tasks at the maximum frequency. Hence, the reliability of a job in NPM follows Equation (5). After computing the individual job reliability values for all the schemes, we obtain the overall system reliability according to Equation (4). The probability of failure *(PoF)* is obtained by subtracting the total reliability from 1. The probability of failure values are presented in normalized form with respect to the NPM scheme.