Sustainable Computing: Informatics and Systems 6 (2015) 81–93

Contents lists available at ScienceDirect

Sustainable Computing: Informatics and Systems

journal homepage: www.elsevier.com/locate/suscom

Energy-aware standby-sparing for fixed-priority real-time task sets

Mohammad A. Haque [a], Hakan Aydin [a,*], Dakai Zhu [b]

[a] Department of Computer Science, George Mason University, 4400 University Drive, Fairfax, VA 22030, United States
[b] Department of Computer Science, University of Texas at San Antonio, San Antonio, TX 78249, United States

ARTICLE INFO

Article history:
Received 16 November 2013
Received in revised form 5 April 2014
Accepted 19 May 2014

Keywords:
Energy management
Reliability
Real-time systems
DVS

ABSTRACT

For mission- and safety-critical real-time embedded systems, energy efficiency and reliability are two important design objectives that must be often achieved simultaneously. 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 executions 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.

1. Introduction

Energy management remains as one of the important challenges for embedded system design and operation. 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 [26]. Another well-known technique is *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 [3,9]. Moreover, the timing constraints of real-time embedded applications impose strict constraints on the applicability of these techniques [19,26,28]. A few recent studies investigated how to combine DVS and DPM to maximize energy savings in the context of a single real-time application [10,36].

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 [27]. The transient faults which lead to temporary *soft errors* (or *single event upsets* (SEUs)) are known to be more frequent than the permanent faults [17,38]. Transient faults are often induced by electromagnetic interference and cosmic radiations [17]. Most importantly, the increase in component density of CMOS circuits and aggressive power management schemes significantly increase the vulnerability of systems to transient faults [14,38]. A common way to deal with transient faults is to rely on additional slack time (time redundancy) to re-invoke the faulty tasks [27,41]. 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) [27].

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 [14,38] 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 [25,35,41]. 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* [11,32,12].

* Corresponding author. Tel.: +1 703 993 3786.
E-mail addresses: mhaque4@gmu.edu (M.A. Haque), aydin@cs.gmu.edu (H. Aydin), dzhu@cs.utsa.edu (D. Zhu).

http://dx.doi.org/10.1016/j.suscom.2014.05.001
2210-5379/

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 [11,32,12]. 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 [16] 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 [6]) 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. Fixed-priority periodic real-time systems are arguably the most common in applications ranging from industrial embedded controllers to avionics and space applications [23]. 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 [8] 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 canceled 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 2, we present our workload and power models, and our assumptions. In Section 3, 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 4. Then the details of our solution are presented in Section 5. Section 6 presents our experimental evaluation. In Section 7, we discuss the research works which are closely elated to our effort in this paper. Finally, we conclude in Section 8 with a summary of our contributions.

2. Models and assumptions

2.1. Workload model

In this paper, we consider a set of periodic real-time tasks $\Psi = \{\tau_1, \ldots, \tau_N\}$. Each task τ_i has the period P_i and the worst-case execution time c_i under the maximum available CPU frequency. The jth 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 $D_{i,j} = j \cdot P_i$. 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 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 jth 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 [11,27]. On the primary processor, tasks are scheduled according to Rate Monotonic Scheduling (RMS) policy, which is known to be optimal for fixed-priority periodic workloads [24]. Throughout the paper, we use the notation $hp(\tau_i)$ to refer to the set of tasks with higher priority than a given task τ_i.

2.2. 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 [41,31]. 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.

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

Above, C_e denotes the effective switching capacitance. The reduced processor supply voltage V_{dd} has a linear relationship with the processing frequency f. Therefore, Eq. (1) can be rewritten 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 [19,38].

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 [21]. 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 [42]. Due to this transition overhead, the concept of break-even time (Δ_{crit}) is introduced in literature [3,5]. 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 [3,5,9]. 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.

2.3. 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 [38,41]. The faults occur according to Poisson distribution with a known average rate λ [35]. The average fault rate λ is dependent on the CPU frequency. In fact, λ increases exponentially with the decrease in CPU frequency [14,38]. Suppose that the average fault rate at the maximum CPU frequency is denoted by λ_0. Then the average fault rate at frequency f can be expressed as [38]:

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

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 [38]:

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

The *probability of failure* for the job J_i is 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* [27,41] on the output of the job. The result of this acceptance test is used to determine the occurrence of errors induced by transient faults.

3. 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 [11,16,12]. 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 [41].

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.

4. Dual-priority scheduling

Dual-priority scheduling [8] 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 fixed-priority (RMS) scheduling.

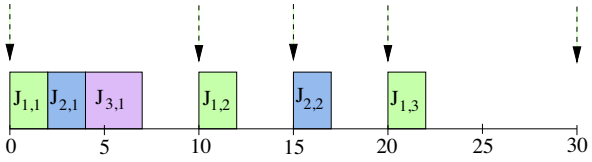


Fig. 1. A typical fixed-priority schedule.

Specifically, if S_i is the worst-case response time of the task with relative deadline D_i under RMS (which can be obtained by the exact time-demand analysis technique [1,22]), 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 [24]), then τ_i would be still able to meet its deadline.

Example 1. To illustrate the concepts, 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 $P_3 = 30$. Fig. 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 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.

In order to compute the task promotion times, one needs to compute the worst-case response time S_i of each task τ_i under rate-monotonic priorities. The well-known Time Demand Analysis (TDA) technique [1,20,22] can be used for this purpose. With TDA, to compute the worst-case response time of a task τ_i, the *critical instant* for that task is considered. It is shown in [24] that the critical instant for τ_i occurs when one of its jobs is released at the same time as all other higher priority tasks $hp(\tau_i)$. We can compute the worst-case response time of a task τ_i, iteratively as follows [20]:

$$S_i^{(m+1)} = c_i + \Sigma_{\tau_j \in hp(\tau_i)} \lceil (S_i^m / P_j) \rceil \times c_j \tag{3}$$

We start by setting $S_i^0 = c_i$ and the iteration continues until $S_i^{(m+1)} = S_i^m$. However, if S_i^m exceeds the deadline relative deadline D_i, then the task cannot be feasibly scheduled in the worst-case scenario. Otherwise, the task is feasible and its response time S_i is the last value $S_i^{m+1} = S_i^m$ obtained during the iterations.

By substituting the response time S_i from Eq. (3) in (2), the promotion times ($Y_i = D_i - S_i$) for the example task set can be computed as: $Y_1 = 8$, $Y_2 = 11$, and $Y_3 = 23$. Fig. 2 shows the delayed execution scenario according to these above mentioned principles. Despite the explicitly enforced delays, all jobs still meet their deadlines.

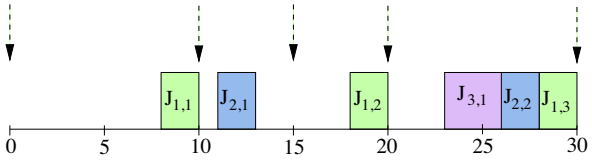


Fig. 2. Fixed-priority schedule with delayed execution.

We underline that the formula (3) will be also instrumental for our *online delayed promotion* rule that further improves the performance.

Notice that in the critical instant of task τ_i there are potentially $\lceil D_i/P_j \rceil$ instances of a higher priority task τ_j that may delay its execution before its deadline D_i. However, since those instances are separated by P_j, the response time analysis may reveal that τ_i is delayed by only a smaller number of such task instances before it completes with the response time S_i in the critical instant. We denote the maximum number of instances of τ_j that actually delay an instance of τ_i in a critical instant by $\eta_{j,i}$. Clearly:

$$\eta_{j,i} = \lceil \frac{S_i}{P_j} \rceil \tag{4}$$

5. Standby-sparing for fixed-priority scheduling

Our proposed schemes in this work are 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 will refer back to Example 1.

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. Using the aforementioned Time Demand Analysis [1], we can find that the task set remains feasible when the primary processor is slowed down to frequency $f = 0.5$. The spare executes the backups $\{B_{i,j}\}$ through the described dual-queue mechanism at the maximum frequency. Fig. 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 delayed until their promotion time and the overlaps with the main tasks on the primary are minimized.

We define the promotion time of a backup job $B_{i,j}$ as:

$$Y_{i,j} = Y_i + r_{i,j} \tag{5}$$

During the interval $[r_{i,j}, Y_{i,j})$, $B_{i,j}$ remains in the lower queue. So $B_{i,j}$ becomes eligible for execution after $t = Y_{i,j}$.

In fact, this feature enables us to cancel backup jobs when the main copy of the job completes successfully (in other words, without incurring transient faults) on the primary. Thus, we can avoid the execution of the backups by coupling it with the primary 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 (Fig. 4). This implies that the backup job $B_{1,2}$ will need to

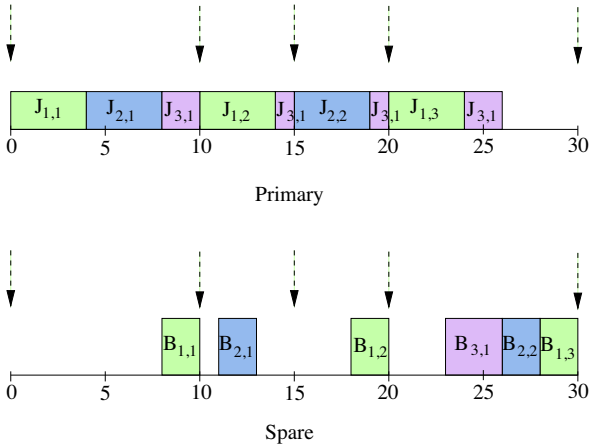


Fig. 3. Coupled schedules on the primary and spare processors.

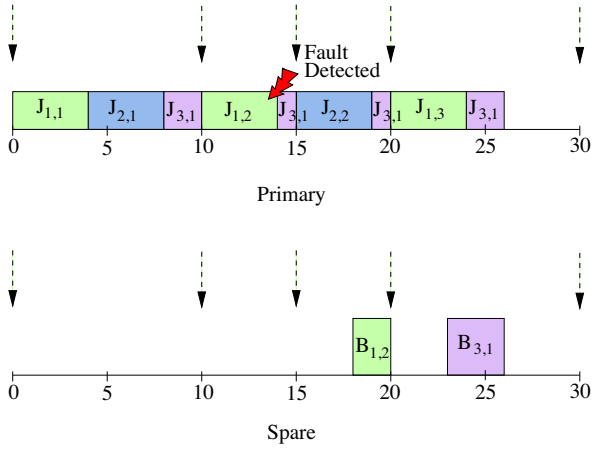


Fig. 4. Taking advantage of successful job completions.

be executed according to the pre-computed schedule on the spare. Note that, $B_{3,1}$ starts executing at its promotion time 23 as its main copy (which was preempted) did not complete yet. Assuming that all the remaining main tasks complete successfully, we obtain the schedules in Fig. 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 execution of other pending backup jobs*. The following section discusses our online delaying strategy for backup jobs on the spare.

5.1. Dynamic delaying of backup jobs

We first define the set of jobs that can benefit from the early completion of a higher priority backup job. Suppose, $B_{i,j}$ is a backup job of task τ_i which completes or is canceled early at time t after executing for $a_{i,j} < c_i$ units of time, where c_i is the worst-case execution time of τ_i. This may occur under two different scenarios. First, the backup task may be canceled due to successful completion on the primary. Second, the backup task may complete early when its actual execution time is smaller than the worst-case (c_i). Now, we claim that another lower priority backup job $B_{k,l}$ can delay its execution by $c_i - a_{i,j}$ without jeopardizing any deadline, if any of the following conditions hold.

1. **Condition 1 – ($t \geq Y_{i,j} \wedge t \geq Y_{k,l}$):** This condition requires that at the time of early completion or cancellation of $B_{i,j}$, both $B_{i,j}$ and $B_{k,l}$ have already been promoted to the upper queue (Fig. 5). As a result, now the remaining part of $B_{k,l}$ will be dispatched $c_i - a_{i,j}$ time units early. So, $B_{k,l}$ can delay its execution by $c_i - a_{i,j}$ time units.
2. **Condition 2 – ($t < Y_{i,j} \wedge Y_{k,l} \leq Y_{i,j}$):** In this scenario, shown in Fig. 6, the higher priority task instance $\tau_{i,j}$ was supposed to be promoted after the lower priority task; however it is canceled at time t before its promotion time $Y_{i,j}$. Therefore, in this scenario, $a_{i,j} = 0$. Observe that, as $B_{i,j}$ is canceled, $B_{k,l}$ will be subject to less interference during its execution. As a result, $B_{k,l}$ would complete early and we can safely delay its execution without violating its deadline.

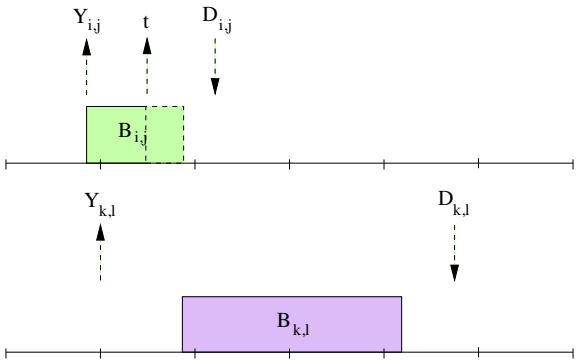


Fig. 5. Condition 1 for dynamic delaying.

We now formally define the set of jobs that are eligible for delaying due to the early completion of $B_{i,j}$ at time t as follows.

$$\Gamma_{i,j}(t) = \{B_{k,l} | \tau_i \in hp(\tau_k) \wedge B_{k,l} \text{ and} B_{i,j} \text{ satisfy Condition 1 or Condition 2}\}$$

Moroever, the fact that the promotion times are computed by the Time-Demand Analysis implies that a given backup task $B_{k,l}$ can be delayed by at most $\eta_{i,j}$ instances of τ_i, where $\eta_{i,j}$ is given by Eq. (4). Hence, $B_{k,l}$ can be delayed by considering at most $\eta_{i,j}$ instances of τ_i at run-time. The following theorem formally states our dynamic delaying scheme for backup jobs on the spare.

Theorem 1. *If a backup job $B_{i,j}$ completes or is canceled after executing for $a_{i,j}$ units of time, then any job $B_{k,l} \in \Gamma_{i,j}$ can delay its execution by $c_i - a_{i,j}$ units of time. $B_{k,l}$ will delay its execution for at most $\eta_{i,k}$ instances of τ_i.*

Proof Assume that $B_{i,j}$ completes or is canceled at time t. Consider any arbitrary job $B_{k,l} \in \Gamma_{i,j}$. We will consider each condition separately.

Condition 1: Notice that, in this scenario both $B_{i,j}$ and $B_{k,l}$ have already been promoted to the upper queue. As a result, according to fixed priority scheduling policy $B_{k,l}$ will not be executed before the completion of $B_{i,j}$. Therefore, $B_{k,l}$ can safely reclaim the slack generated by the early completion/cancellation of $B_{i,j}$ and delay its execution by $c_i - a_{i,j}$.

Condition 2: In this scenario, $B_{i,j}$ is canceled before its promotion time $Y_{i,j}$. The promotion time of $B_{k,l}$ is $Y_{k,l}$ and its absolute deadline is $D_{k,l}$. Now according to the rule for determining promotion time, $(D_{k,l} - Y_{k,l})$ is the maximum response time S_k for τ_k. The promotion times for two consecutive instances of τ_i are separated at least by P_i. Therefore, we can have at most $\eta_{i,k} = \lceil S_k/P_i \rceil$ instances of τ_i that satisfy the Condition 2 for one instance of τ_k. By assumption, for

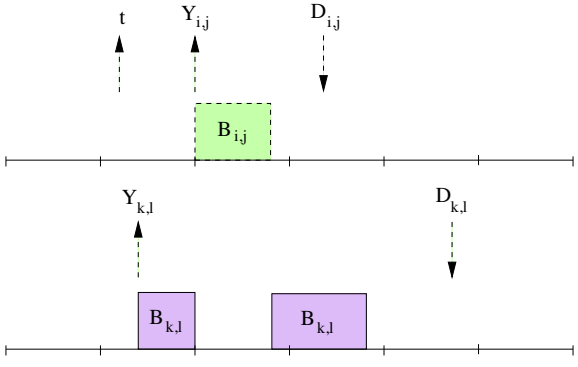


Fig. 6. Condition 2 for dynamic delaying.

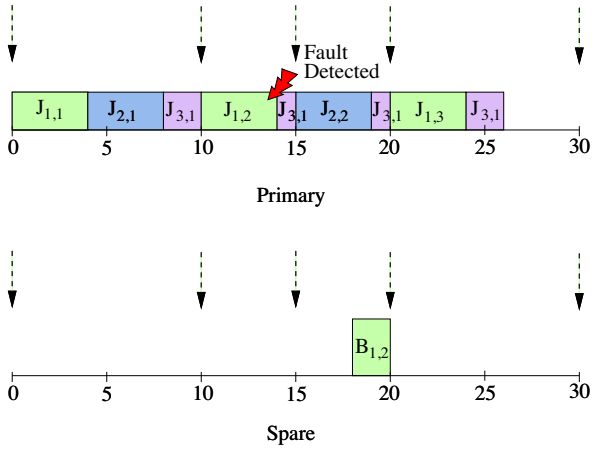


Fig. 7. Delaying promotions at run-time.

each task $\tau_m \in hp(\tau_k)$, at most $\eta_{m,k}$ instances interfere with (delay) a backup job of τ_k. So we can write:

$$Y_{k,l} + c_k + \Sigma_{B_m \in hp(B_k)} \eta_{m,k} \times c_m \leq D_{k,l}$$

This can be rewritten as,

$$Y_{k,l} + c_k + \Sigma_{B_m \in \{hp(B_k) - B_i\}} \eta_{m,k} \times c_m + (\eta_{i,k} - 1) \times c_i + c_i \leq D_{k,l} \quad (6)$$

where the last component (c_i) represents the worst-case interference due to a single high-priority instance $B_{i,j}$. Recall that, by assumption, $B_{i,j}$ is entirely canceled before starting to execute. Consequently the response time of $B_{k,l}$ decreases by an amount of c_i. Hence, delaying $B_{k,l}$ by c_i will yield a new response time,

$$Y_{k,l} + c_k + c_i + \Sigma_{B_m \in \{hp(B_k) - B_i\}} \eta_{m,k} \times c_m + (\eta_{i,k} - 1) \times c_i \quad (7)$$

which does not exceed D_k, l according to Eq. (6).

Therefore, for both conditions, $B_{k,l}$ can delay its execution by $c_i - a_{i,j}$ time units without missing any deadline. We have proved that the feasibility of $B_{k,l}$ considering only $B_{k,l}$ is delayed, while all higher priority jobs (priority higher than $B_{k,l}$) executes without delaying. Note that, if some of these jobs are also delayed due to the early completion of $B_{i,j}$, this will only improve the response time of $B_{k,l}$. As a result, $B_{k,l}$ will still remain feasible.

This completes the proof of Theorem 1. ■

Fig. 7 shows an example of delayed promotion enabled by the result in Theorem 1. The nominal promotion time of $B_{3,1}$ as computed statically is 23. As backup job $B_{2,2}$ completes at time 19, $B_{3,1}$ can be delayed until time 25. Notice that at time 24, $B_{1,3}$ is also canceled, which allows $B_{3,1}$ to be delayed for additional 2 units of time. As a result, its actual promotion time to higher queue is delayed further to 27. Therefore, we were able to avoid the execution of $B_{3,1}$ entirely as the primary copy completes successfully at time 26.

We now discuss the runtime complexity of dynamic delaying. At every backup completion time, we need to check all lower priority job instance for eligibility. As a result, it imposes $O(N)$ overhead at each job completion.

5.2. 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 [26,28,29]. The spare processor, on the other hand, uses the dual-queue mechanism and applies only DPM for energy management. The nominal promotion time of each back-up job $B_{i,j}$ is computed by adding its release time to Y_i, the pre-computed promotion time for task B_i.

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

Event - A job of τ_i, $J_{i,j}$ is released at t_0:
  Add $J_{i,j}$ to the ready queue on *primary*
  Add $B_{i,j}$ to the lower queue on *spare*
  $Y_{i,j} \leftarrow t_0 + Y_i$
  $Z_{i,j} \leftarrow Y_{i,j}$
  Set timer for promotion event at $t = Z_{i,j}$
  for every $\tau_m \in hp(\tau_i)$ **do**
    $n[i, m] \leftarrow 0$
  end for
  Dispatch the highest RM priority job on *primary*
Event - The job $J_{i,j}$ completes on *primary* at t_0:
  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**
    /* $J_{k,l}$ is the next job to be released */
    $time\_to\_next\_arrival \leftarrow r_{k,l} - t_0$
    $\Delta_p \leftarrow time\_to\_next\_arrival$
    if $\Delta_p \geq \Delta_{crit}$ **then**
      Put *primary* to sleep state for Δ_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 at t_0:
  Move $B_{i,j}$ to the upper queue on *spare*
  Dispatch the highest RM priority job on *spare*
Event - A backup job $B_{i,j}$ completes/canceled at t_0:
  if $B_{i,j}$ is not canceled **then**
    Run the acceptance test for $B_{i,j}$
  end if
  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**
    $n[k, i] \leftarrow n[k, i] + 1$
    if $n[k, i] \leq \eta_{k,i} \wedge B_{k,l}$ is in upper queue **then**
      Move $B_{k,l}$ to the lower queue
      $Z_{k,l} \leftarrow t_0 + \gamma$
      Set Promotion event at time $t = Z_{k,l}$
    else if $n[k, i] \leq \eta_{k,i} \wedge B_{k,l}$ is in lower queue **then**
      $Z_{k,l} \leftarrow Z_{k,l} + \gamma$
      Set Promotion event at time $t = Z_{k,l}$
    end if
  end for
  if the upper queue is empty **then**
    /* $B_{k,l}$ is the next job to be promoted */
    $time\_to\_next\_promotion \leftarrow Z_{k,l}$
    $\Delta_s \leftarrow \min\{\gamma, time\_to\_next\_promotion\}$
    if $\Delta_s \geq \Delta_{crit}$ **then**
      Set wake-up event at $t = t_0 + \Delta_s$
      Put *spare* to sleep state
    end if
  else
    Dispatch the highest RM priority job on *spare*
  end if
Event - the *spare* processor wakes up at t_0:
  if the upper queue is not empty **then**
    /* There are backups not yet canceled */
    Dispatch the highest RM priority job on *spare*
  else
    /* $B_{k,l}$ is the next job to be promoted */
    $time\_to\_next\_promotion \leftarrow Z_{k,l}$
    $\Delta_s \leftarrow time\_to\_next\_promotion$
    if $\Delta_s \geq \Delta_{crit}$ **then**
      Set wake-up event at $t = t_0 + \Delta_s$
      Put *spare* to sleep state
    end if
  end if

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. We have a nominal promotion time $Y_{i,j}$ with the pre-computed promotion time and an updated promotion time $Z_{i,j}$. Initially $Z_{i,j}$ is set to nominal promotion time. A timer is set accordingly to promote the backup job to the upper queue in the future. In our solution, we use a vector $n[]$ to keep track of the number of higher priority task instances whose execution times have been reclaimed by the corresponding backup job, in accordance with Theorem 1. Specifically, $n[i, m]$ represents the number of instances of task τ_m for which $B_{i,j}$ has already been delayed. The vector is initally set to 0. When an instance of τ_m is cancelled or completes without presenting its worst-case workload, then $B_{i,j}$ is delayed whenever possible. The primary processor then dispatches jobs according to RMS policy. When a job completes on the primary, we invoke the corresponding acceptance test [27] to check the sanity of the computed result. If the acceptance test does not detect any error, we cancel the corresponding backup task on 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.

Algorithm 2 gives the actions taken in response to the events on the spare processor. A job is only eligible for execution, when it is in the upper queue. At promotion time, the job is moved to the upper queue. 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/canceled, if the corresponding main task $J_{i,j}$ has not been completed yet, the execution of $J_{i,j}$ is also canceled on the primary processor. We then compute the runtime slack γ generated by $B_{i,j}$. Then, for every eligible pending job $B_{k,l}$ we increase $n[k, i]$ by 1. Notice that according to Theorem 1, $B_{k,l}$ can delay its execution for at most $\eta_{k,i}$ cancelled or prematurely-finished instances of τ_i. So if we have not delayed $B_{k,l}$ for more than $\eta_{k,i}$ instances, i.e., if $n[k, i] \leq \eta_{k,i}$, we update the promotion time according to Theorem 1. Next, if $B_{k,l}$ has already been moved to the upper queue, we can move it back to the lower queue, where it stays for γ units of time. If $B_{k,l}$ is still in the lower queue, we can delay its promotion for additional γ units of time. A new promotion event is set accordingly. If there is no backup tasks in the upper queue the spare can remain idle until the next job is promoted. The earliest time when the next job will be promoted 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 upgrade time. Otherwise, the highest-priority job is dispatched.

5.3. Dynamic reclaiming for SSFP

We now discuss our dynamic slack reclaiming strategy. We take advantage of the Generalized Dynamic Reclaiming Algorithm (GDRA) presented in [2]. GDRA was proposed for Earliest Deadline First (EDF) scheduling policy, but it can be applied for fixed priority scheduling as well, by considering the RMS priorities when ordering and reclaiming run-time slack. The algorithm starts by considering the availability of a canonical schedule S^{can}, which is the schedule generated by the static speed assignment considering every job presents its worst-case workload. Then at dispatch time the scheduler can compare against S^{can} and use the earliness of the actual schedule for further slow down.

GDRA uses a data structure called the α-queue. It is actually a priority queue of the active jobs in the system according to the canonical schedule. The high-level idea can be summarized as follows.

1. α-queue is initialized as an empty list.
2. At arrival time of $J_{i,j}$, a job is added to the α-queue according to fixed priority assignment with remaining execution time, $W_{i,j} = C_i/f_i$, where f_i is the statically computed speed for $J_{i,j}$.
3. At every time unit, the remaining execution time of the job at the head of the α-queue is reduced by one. When the remaining execution time reaches 0, the job is removed from the queue. Notice that, it is sufficient to update the queue only at arrival and completion time of jobs, as the queue is only checked at those times.
4. At dispatch time of $J_{i,j}$, the α-queue is checked to determine the earliness defined as,

$$E_{i,j}(t) = W_{i,j} + \Sigma_{k \in hp(\tau_i)} W_{k,l}$$

$E_{i,j}$ is the maximum time available for $J_{i,j}$ to complete. $J_{i,j}$ can reclaim the entire slack or part of it based on the scheduling strategy. As a result, $J_{i,j}$ can be executed at a reduced processing frequency as low as $W_{i,j}/E_{i,j}$.

At each invocation, the runtime overhead for the GDRA is $O(N)$, where N is the number of tasks in the system.

6. 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 [39,41]. 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 processing frequency to meet the deadlines, increasing the dynamic energy consumption.

We implemented several variations of our SSFP scheme to investigate the impact of slack reclaiming on the primary and dynamic delaying of the backup tasks on the spare.

1. **Static voltage scaling (Static)**: With this scheme, the speed assignment on the primary processor is performed according to *Sys-Clock* algorithm [29]. Dynamic slack reclaiming is disabled on the primary. The backup tasks are canceled on the spare as soon as their primary copy completes successfully. However, when a high-priority backup task is completed, the low-priority backup tasks are not delayed.
2. **Static with delaying (Static-D)**: Static-D scheme enables the dynamic delaying of backup tasks (when the conditions given in

Theorem 1 are satisfied) as higher priority backup tasks complete or are cancelled.

3. **Dynamic reclaiming with delaying (Dynamic-D)**: Dynamic-D combines the dynamic reclaiming on the primary processor and delaying of backup tasks on the spare. The dynamic slack reclaiming on the primary processor is conducted according to Generalized Dynamic Reclaiming Algorithm (GDRA) [2], as discussed in Section 5.3.
4. **Limited dynamic reclaiming with delaying (Dynamic*-D)**: While using reclaiming to its fullest extent on the primary may help to reduce its energy consumption, it has the potential of increasing the *overlap* with the backups due to extended execution times on the primary. With Dynamic*-D, the frequency of the tasks on the primary is not reduced below a certain threshold to avoid excessive overlaps on the secondary. Specifically, we assume that the algorithm has prior knowledge about the expected workload [2]. The expected workload is determined by considering the average case workload for every task in the system. Using this information, the scheme computes a lower bound on the speed on the primary processor according to Sys-Clock considering average workload for every task instances. At run-time the primary processor is never slowed down beyond this lower bound.
5. **Bound**: This is a yardstick scheme that we include to obtain a lower bound on the energy consumption for *any* SSFP scheme. The lower bound is obtained by considering the minimum possible energy consumption in each processor. For primary processor, Sys-Clock with full reclaiming achieves the lowest energy consumption. On the spare processor the lowest possible energy consumption occurs when it is always kept in sleep mode assuming that all tasks will complete successfully on the primary (i.e., backup tasks are never executed).

We do not include any comparison with [11,32,12], as they are limited to aperiodic, non-preemptive workloads. Similarly, the scheme in [16] 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, we generated 1000 periodic task sets. The worst-case utilizations of the tasks are generated randomly using the *UUnifast* scheme [4]. 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.

The energy 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 the SSFP variants and RAPM across different system parameters including the total utilization (U_{tot}), the ratio of best-case to worst-case execution time (BC/WC), the number of tasks and power parameters. The BC/WC ratio is used to model and assess the impact of the variability in the actual workload. Specifically, following [15,30], 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 BC/WC ratio is 0.5 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 [42]. 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 [42].

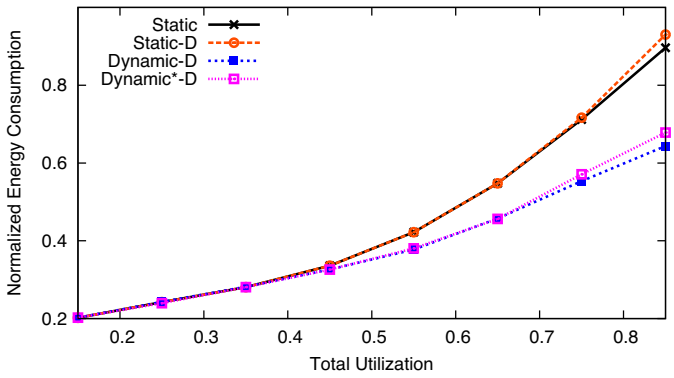


Fig. 8. Energy consumption on the primary.

6.1. Impact of system workload

We first study the impact of system workload. We vary the utilization between 15% to 85% of the CPU capacity. Notice that even though in general the feasibility for task systems with utilization >0.69 cannot be guaranteed with RMS [24], the work in [22] shows that in many cases task sets with utilization up to 0.88 can be feasibly scheduled. Consequently we include in our analysis the task sets that are schedulable for the specified utilization values. For our analysis, we record the energy consumption and the fraction of the backup tasks that are completely/partially executed on the spare processor.

We first analyze the energy consumption on the primary processor. Fig. 8 shows the energy consumption of the primary processor as we vary the system load. As a general trend, the energy consumption increases with the increased system load. Notice that, at very low utilization all SSFP schemes perform same, as the speed assigment is equal to the energy-efficient frequency. Static and Static-D do not take advantage of the dynamic slack on the primary. As a result, these schemes consume significantly higher energy compared to Dynamic-D and Dynamic*-D. Note that, especially in high utilization values, occasionally a backup copy may complete earlier than the scaled copy on the primary. In that case, our algorithms cancel the remaining part of the primary task as well to avoid unncessary execution. When delaying is applied, the executions of the backup tasks are delayed. As a result, delaying increases the primary energy consumption with Static-D slightly, compared to Static. Dynamic-D has the lowest primary energy consumption as it uses aggressive slowdown by taking advantage of the dynamic slack. Dynamic*-D has slightly higher primary energy consumption, as it does not slow down beyond a certain limit.

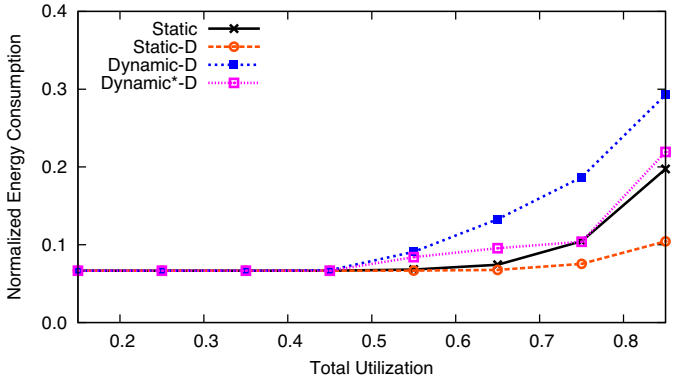


Fig. 9. Energy consumption on the spare.

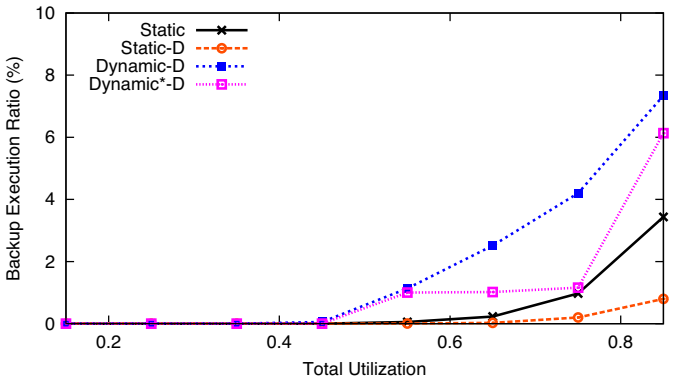


Fig. 10. Ratio of canceled backup tasks.

We now consider the energy on the spare processor. Fig. 9 shows the energy consumption on the spare processor, while Fig. 10 shows the backup execution ratio for the spare processor. The backup execution ratio is determined by computing the ratio of partially/completely executed backup jobs to the total number of backup jobs on the spare processor. Notice that up to utilization 0.45, all backup tasks are canceled as there is no overlap between the primary and the backup copy. As a result the energy consumption on the spare processor remains constant for all schemes and is due to the static energy only. With increased utilization, there is some overlap between the primary and backup copies. As a result, some backups are executed and the spare energy consumption increases for all schemes. Dynamic-D slows down the primary copies by aggressively reclaiming the dynamic slack. Therefore, the primary copies take longer to complete and there is non-trivial overlap between the primary and backup copies. As a result, Dynamic-D has the highest spare energy consumption. Dynamic*-D cautiously reclaims the dynamic slack and does not slow down beyond a limit. Therefore it consumes less energy compared to Dynamic-D. Static does not reclaim dynamic slack on the primary and reduces the overlaps with the backups. Hence, it achieves a lower spare energy consumption. Static-D consumes the lowest spare energy as it does not slow down the primary at run time and at the same time delays the execution of the backup tasks whenever possible.

Fig. 11 presents the total (primary +spare) energy consumption as a function of the utilization. At very low utilization (up to 0.35), RAPM outperforms the SSFP schemes since it avoids the static energy consumption of the spare processor. All SSFP schemes perform similarly as the speed assignment is primarily determined by the energy-efficient speed. In this interval, RAPM yields up to 3% energy savings compared to the SSFP schemes. As the utilization increases, RAPM is quickly forced to use very high frequency. So, SSFP schemes begin to outperform RAPM, despite the use of two processors. Static and Static-D perform worst among the SSFP schemes due to significantly higher primary energy consumption. But at higher utilization Static still achieves up to 10% energy savings compared to RAPM. Dynamic-D consumes the lowest primary energy, but due to high spare energy consumption can not achieve the lowest overall energy consumption. Dynamic*-D performs the best, as it can balance the primary energy and spare energy consumption by judicious reclaiming of the dynamic slack. Dynamic*-D consumes up to 15% less energy compared to RAPM and up to utilization 0.75 the energy consumption of Dynamic*-D does not exceed the energy consumption of the yardstick scheme Bound by 5%. Notice that at very high utilization (0.85) Static and Static-D perform worse than RAPM. This is because at very high utilization Static and Static-D are also forced to use the maximum frequency and they also consume static power on the spare.

Finally, we note that our current framework is based on determining the speed on the primary independently and delaying the backups as much as possible to avoid the overlaps with the main tasks. The results in Fig. 11 suggest that for a big portion of the spectrum, this approach pays off very well; i.e., when the utilization is less than 75%, the difference between the energy consumption figure of the SSFP schemes and that of the theoretical lower bound is minimal. The somewhat widening gap when the utilization is very high suggests that further investigation of schemes that re-adjust the speed by dynamically considering possible overlaps with the backups might bring additional benefits in that region. This is left as future work.

6.2. Impact of workload variability

We now consider the impact of workload variability. We set the system utilization to 0.6 and vary the BC/WC ratio from 0.1 to 1.0. $BC/WC = 1.0$ indicates the scenario where all task instance presents the worst case scenario. We record the energy consumption along with the actual number of backup tasks that are completely/partially executed on the spare.

Again, we first analyze the energy consumption on the primary (Fig. 12). Static and Static-D consume highest primary energy as they do not take advantage of the early completions on the primary. Dynamic-D consumes the lowest primary energy as it aggressively slows down the primary. Dynamic*-D performs moderately as it only reclaims up to a certain limit. When $BC/WC = 1.0$, Dynamic*-D performs similar to Static-D as the actual execution time becomes equal to the worst-case execution time, which prevents Dynamic*-D from further slowing down. Notice that, Dynamic-D consumes lower primary energy even when $BC/WC = 1.0$. This is because, due to the different scheduling orders on the primary and backup, some

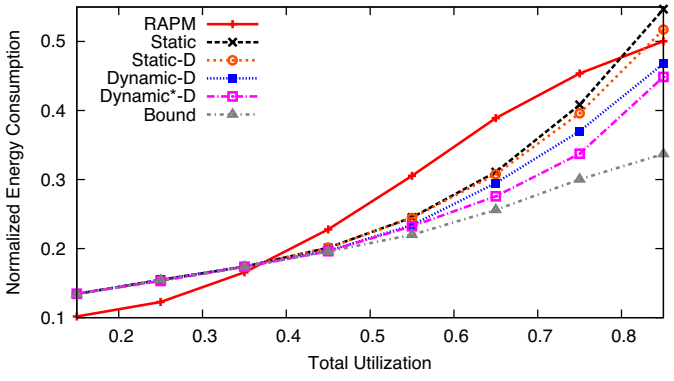


Fig. 11. Total energy consumption.

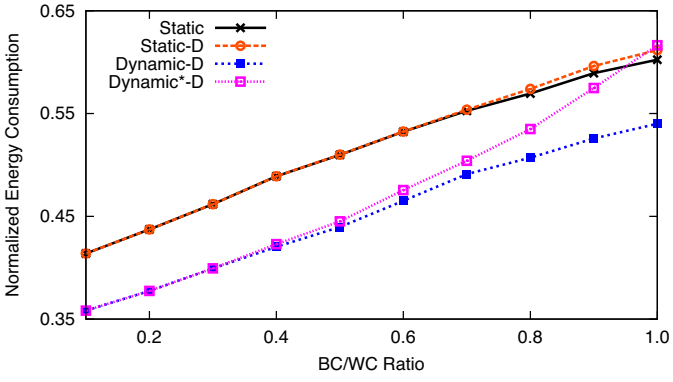


Fig. 12. Energy consumption on the primary.

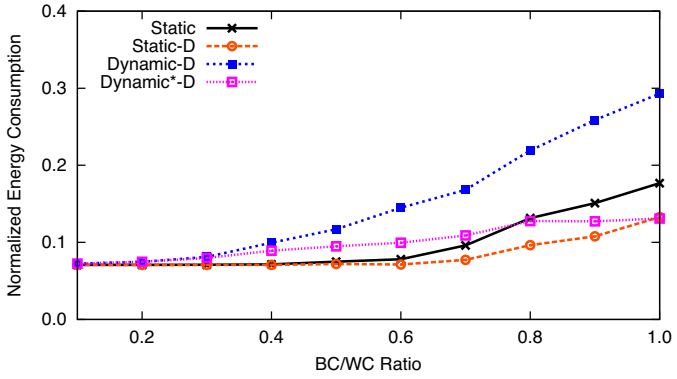


Fig. 13. Energy consumption on the spare.

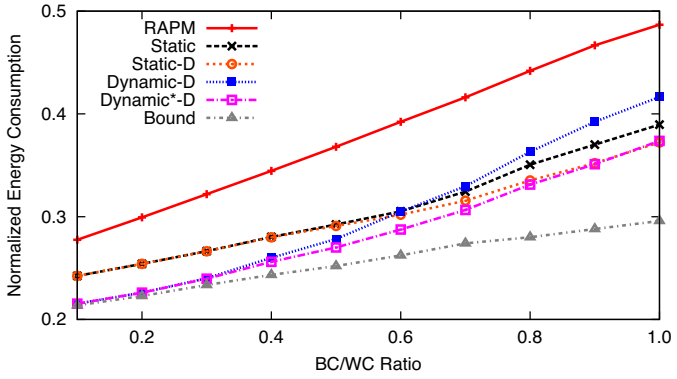


Fig. 15. Total energy consumption.

primary copies are canceled early as their backup copy completes and Dynamic-D can better exploit those case by reclaiming the dynamic slack generated in that way.

The spare energy consumption is shown in Fig. 13 along with the fraction of completely/partially executed backup tasks in Fig. 14. Notice that as we increase the BC/WC ratio, the fraction of executed backup tasks increases along with the energy consumption on the spare. Dynamic-D yields the highest energy consumption on the spare due to the significant overlap between primary and backup copies. Static consumes significantly lower spare energy as it has smaller overlap between primary and backup copy. Static-D consumes even lower spare energy as it delays the backup tasks. At lower BC/WC ratio, Dynamic*-D performs comparable to Dynamic-D as there is enough slack to reclaim and achieve primary speed closer to Dynamic-D. At higher BC/WC ratio, Dynamic*-D cannot reclaim significant slack for slow down on the primary. As a result, its energy consumption approaches that of Static-D.

Finally, Fig. 15 shows the total (primary +spare) energy consumption for varying BC/WC ratio. RAPM consumes the highest energy as it is forced to use a higher frequency and cannot take advantage of the dynamic slack. RAPM consumes up to 10% extra energy compared to the closest SSFP scheme. At low BC/WC ratio, tasks complete much earlier than the worst-case scenario and the system has ample dynamic slack. But Static and Static-D do not take advantage of the dynamic slack and they consume up to 3% higher energy compared to the dynamic schemes which consume higher energy on the primary. As the BC/WC ratio increases, dynamic slack due to early completion of primary becomes rare. Most of the dynamic slack at higher BC/WC ratio comes from the cancellation of primary copies that follow earlier completion of the corresponding backup copies on the spare. Dynamic-D uses the dynamic slack to slow down the primary tasks, but this gives a larger overlap between the primary and backup tasks. As a result, the spare energy consumption for Dynamic-D increases and despite lower primary energy consumption, Dynamic-D consumes higher energy than Static and Static-D. On the other hand, Dynamic*-D does not slow down the primary beyond a threshold determined by the expected workload. As a result, the energy consumption of Dynamic*-D does not exceed that of the ideal Bound by 8% and outperforms RAPM by up to 12%.

6.3. Impact of number of tasks

Fig. 16 shows the impact of number of tasks in the system. We increase the number of tasks from 5 to 25 in steps of 5 while keeping the total utilization fixed at 0.6. As a result, the average task size tasks get smaller. Notice that for all schemes energy consumption decreases as the average task size gets smaller. For RAPM, the energy consumption decreases as smaller tasks allow RAPM to take advantage of the dynamic slack by allocating recovery task for additional tasks. For the SSFP schemes, as tasks get smaller, the primary copies complete quickly allowing more backup tasks to be canceled. Notice that with very small number of tasks, static schemes perform better than the dynamic schemes. When tasks are larger, there is a greater chance of overlap between the primary and spare copies. By running the primary at higher speed, static schemes can reduce the overlap to obtain lower spare energy consumption, which leads to overall energy savings. As the number of tasks increases the overlap region becomes smaller. As a result, the spare energy consumption decreases for all schemes. Since Static and Static-D can not take advantage of the dynamic slack, the decrease in the primary energy consumption is smaller for them. Dynamic*-D achieves the least energy consumption.

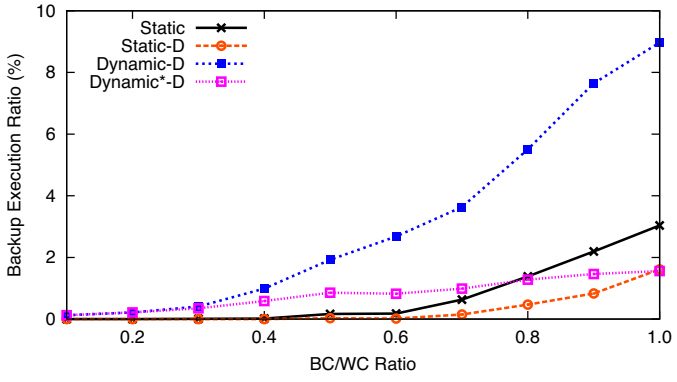


Fig. 14. Ratio of canceled backup tasks.

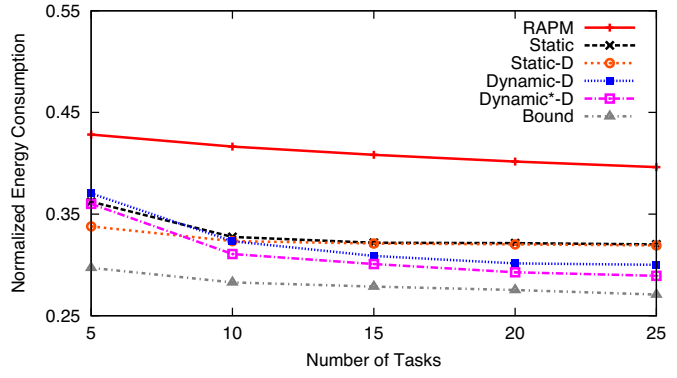


Fig. 16. Impact of the number of tasks.

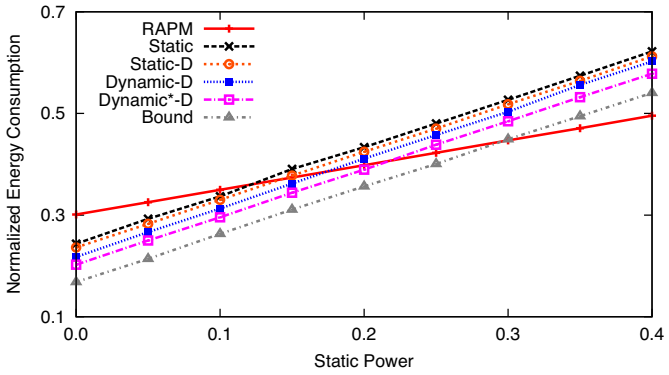


Fig. 17. Impact of static power.

6.4. Impact of static power

Fig. 17 depicts the impact of static power. For these experiments, we fix the task set utilization and BC/WC ratio to 0.75 and 0.5 respectively. As a general trend, the total energy consumption increases with the static power consumption. Initially, the SSFP schemes outperform the RAPM by taking advantage of DVS on the primary and cancellation coupled with DPM on the spare. But as we increase the static power, SSFP schemes start to suffer due to the static energy consumption of the two processors, while RAPM can save static energy as it uses only one processor. We note again, however, that RAPM cannot tolerate the permanent fault of a single processor, whereas SSFP variants offer that capability.

6.5. Impact of frequency independent power

Fig. 18 shows the impact of the frequency independent power (P_{ind}). As P_{ind} increases, the energy-efficient frequency (f_{ee}) increases. Due to the increased P_{ind}, all schemes consume more energy. Notice that at very low P_{ind} values, f_{ee} becomes effectively equivalent to f_{min}. As a result, Dynamic-D uses a very low speed, which results in significant overlap between the primary and backup copies. Consequently, the energy consumption is very high for Dynamic-D as P_{ind} approaches 0. At higher P_{ind}, f_{ee} becomes very large. In this setting, Dynamic-D and Dynamic*-D schemes become comparable as the speed assignment on the primary begins to be dominated by f_{ee}.

6.6. Impact on system reliability

We now evaluate the reliability performance of the schemes with respect to transient faults. Fig. 19 shows the *probability of failure (PoF)* trends, which is defined as 1 − *reliability* [41]. 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, we computed the *PoF* value for each task set. 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 [38]:

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

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 [37].

$$R = \Pi_{\forall_{i,j}} R_{i,j} \tag{8}$$

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 fail. So, the reliability of a job in the SSFP system is:

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

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 [39],

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

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

$$R_{i,j} = e^{-\lambda(f_{\max})c_i/f_{\max}} \tag{9}$$

With the NPM scheme, the reliability level 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 Eq. (9). After computing the individual job reliability values for all the schemes, we obtain the overall system reliability according to Eq. (8). 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.

For the set of experiments in Fig. 19, λ_0 is set to 10^{-6} and d is set to 2 [41]. The BC/WC ratio is set to 0.5 as we vary the utilization from 0.15 to 0.85. For each data point we have 1000 task sets. Each task set has 10 periodic tasks.

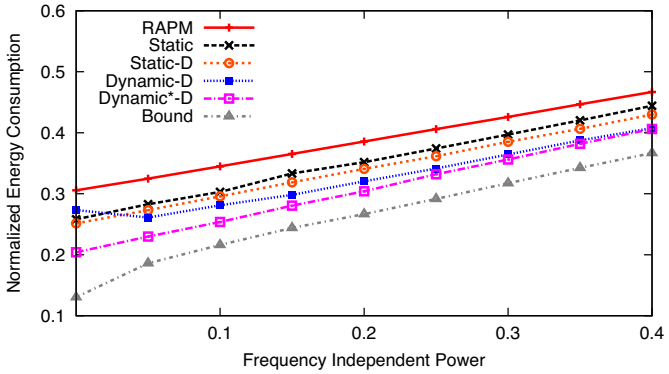


Fig. 18. Impact of frequency independent power.

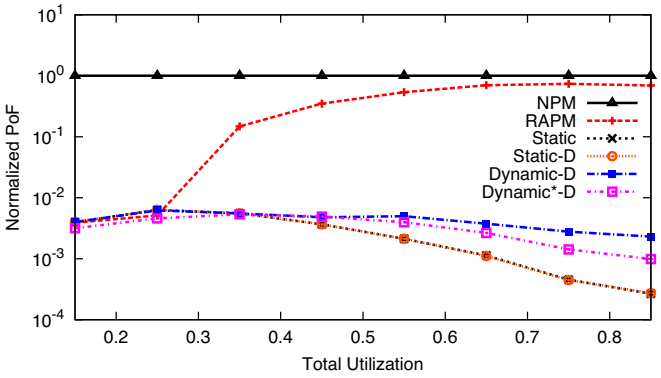


Fig. 19. Impact of total utilization on system reliability.

At very low utilization levels RAPM can reserve a recovery for every task, and achieve low *PoF* 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 schemes, 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 schemes 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. All the SSFP schemes execute the backup tasks at f_{max}. So the variation in *PoF* occurs due to the speed assignment on the primary processor. Static and Static-D have identical speed assignment for the primary. As a result they achieve same level of *PoF*. The dynamic schemes slow down the primary by using the run time slack, which leads to higher *PoF* (lower reliability) compared to the static scheme. Since, Dynamic-D uses the lowest speed for the primary processor, its *PoF* is the highest among all SSFP schemes.

7. Closely related works

The reliability- and energy-aware scheduling techniques can be broadly divided into two categories: time-redundancy and hardware-redundancy based techniques. Time-redundancy based techniques utilize the available CPU slack for both slowdown and executing recovery tasks [25,35,37,41]. In [25], the authors considered checkpointing as the fault-tolerance mechanism. By optimally modifying the number and placement of the checkpoints, the authors obtained energy savings through DVS while maintaining feasibility in the presence of faults. The study in [35] also employs DVS and checkpointing for energy-efficient fault tolerant scheduling. The number of checkpoints is adjusted dynamically based on the available slack and detected faults. In [39] and [41], the authors considered a notion of original reliability, which is the system reliability when all tasks execute at f_{max}. To preserve the original reliability, a recovery task is assigned for every task that is slowed down. The recovery tasks execute at f_{max}. The work in [39] considers the fixed priority periodic applications, while the one in [41] considers the EDF scheduling policy. In [40], the framework was extended to tasks with probabilistic execution times. In another study [37], instead of the original reliability, arbitrary reliability targets are considered. A limitation of these time-redundancy based techniques is that they cannot slow down large individual tasks (i.e., tasks with utilization greater than 50%). More importantly, these solutions are vulnerable to permanent faults since they employ only one processor.

Dabiri et al. [7] presented a reliability-aware DVS technique. A set of dependent real-time tasks is expressed as a Directed Acyclic Graph and the authors formulated an optimization problem to minimize the energy consumption while achieving an arbitrary target error rate. The scheme is limited to aperiodic tasks and is applicable only when the target error rate is higher than the original error rate at f_{max}.

Hardware redundancy techniques can achieve higher reliability targets. Also permanent faults can only be tolerated through hardware redundancy. [13] considered duplex and TMR systems where there two and three identical processors are deployed for executing copies of every job in parallel. By efficient use of DVS and leveraging the rareness of fault occurrences, the authors demonstrated that TMR can achieve higher reliability while consuming less energy than the traditional duplex system.

In [33] Unsal et al. considered a fault tolerant multiprocessor system that deploys primary and backup copies on different processors. By delaying the backup tasks, the authors aimed at achieving smaller overlap between primary and backup copies and thus minimizing the overall energy consumption compared to naive duplication. In [31], there is a primary and a backup processor which executes replicas of the same task in parallel. To compensate the reliability loss due to DVS, additional copies of the tasks are dispatched at minimum CPU frequency. Izosimov [18] coupled checkpointing and replication on heterogeneous distributed system to optimize resource consumption for time constrained applications. In [34], the authors considered a Symmetric Multiprocessor System (SMP) and proposed heuristic based technique to achieve balanced partitioning and then an optimistic fault-tolerant heuristic is applied to achieve significant energy savings in fault-free scenario.

Another hardware-redundancy based technique, called the standby-sparing system is proposed in [11,12]. The authors considered a non-preemptive, aperiodic task set running on a system with two processor called the primary and the spare. The primary processor executes the main tasks, while the spare processor is reserved for executing backup tasks. To avoid overlapping execution of the main and backup copies, backup tasks are delayed as much as possible. However, computing the exact delay for preemptive, periodic task set can not be computed trivially. As a result, the solution in [11] and [12] can not be extended to preemptive, periodic settings. A feedback-based energy-management scheme for aperiodic task is also presented in [32]. For preemptive periodic applications, the work in [16] proposes a standby sparing technique that constructs an EDL schedule for the entire hyperperiod, which is used as schedule for the spare processor. The technique is limited to EDF scheduling policy only and the requirement of constructing the entire schedule offline limits the applicability of the scheme.

8. 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.

Appendix A. Supplementary data

Supplementary data associated with this article can be found, in the online version, at http://dx.doi.org/10.1016/j.suscom.2014.05.001.

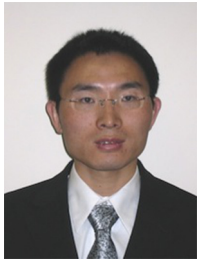
References

[1] N. Audsley, A. Burns, M. Richardson, K. Tindell, A.J. Wellings, Applying new scheduling theory to static priority pre-emptive scheduling, Softw. Eng. J. 8 (5) (1993) 284–292.
[2] H. Aydin, R. Melhem, D. Mossé, P. Mejía-Alvarez, Power-aware scheduling for periodic real-time tasks, IEEE Trans. Comput. 53 (5) (2004) 584–600.
[3] L. Benini, A. Bogliolo, G. De Micheli, A survey of design techniques for system-level dynamic power management, IEEE Trans. VLSI Syst. 8 (3) (2000) 299–316.
[4] E. Bini, G.C. Buttazo, Measuring the performance of schedulability tests, J. Real-Time Syst. 30 (1-2) (2005) 129–154.
[5] J.J. Chen, T.W. Kuo, Procrastination determination for periodic real-rime tasks in leakage-aware dynamic voltage scaling systems, in: Proceedings

of IEEE International Conference on Computer-Aided Design (ICCAD), 2007.
[6] H. Chetto, M. Chetto, Some results of the earliest deadline scheduling algorithm, IEEE Trans. Softw. Eng. 15 (1989) 1261–1269.
[7] F. Dabiri, N. Amini, M. Rofouei, M. Sarrafzadeh, Reliability-aware optimization for dvs-enabled real-time embedded systems, in: Proceedings of the 9th IEEE International Symposium on Quality Electronic Design (ISQED), 2008.
[8] R. Davis, A. Wellings, Dual priority scheduling, in: Proceedings of the 16th IEEE Real-Time Systems Symposium, 1995.
[9] V. Devadas, H. Aydin, Real-time dynamic power management through device forbidden regions, in: Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2008.
[10] V. Devadas, H. Aydin, On the interplay of voltage/frequency scaling and device power management for frame-based real-time embedded applications, IEEE Trans. Comput. 61 (1) (2012) 31–44.
[11] A. Ejlali, B.M. Al-Hashimi, P. Eles, A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems, in: Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2009.
[12] A. Ejlali, B.M. Al-Hashimi, P. Eles, Low-energy standby-sparing for hard real-time systems, IEEE Trans. Comput.-Aided Design Integr. Circuits Syst. 31 (3) (2012) 329–342.
[13] E. Elnozahy, R. Melhem, D. Mossé, Energy-efficient duplex and TMR real-time systems, in: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS), 2002.
[14] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N.S. Kim, K. Flautner, Razor: circuit-level correction of timing errors for low power operation, IEEE Micro 6 (2004) 10–20.
[15] F. Gruian, Hard real-time scheduling for low-energy using stochastic data and DVS processors, in: Proceedings of IEEE International Symposium on Low Power Electronics and Design (ISLPED), 2001.
[16] M.A. Haque, H. Aydin, D. Zhu, Energy-aware standby-sparing technique for periodic real-time applications, in: Proceedings of IEEE International Conference on Computer Design (ICCD), 2011.
[17] R.K. Iyer, D.J. Rossetti, M.C. Hsueh, Measurement and modeling of computer reliability as affected by system activity, ACM Trans. Comput. Syst. 4 (1986) 214–237.
[18] V. Izosimov, P. Pop, P. Eles, Z. Peng, Design optimization of time and cost-constrained fault-tolerant distributed embedded systems, in: Proceedings of the IEEE Conference on Design, Automation and Test in Europe, 2005.
[19] R. Jejurikar, C. Pereira, R. Gupta, Leakage aware dynamic voltage scaling for real-time embedded systems, in: Proceedings of IEEE/ACM Design Automation Conference (DAC'), 2004.
[20] M. Joseph, P. Pandya, Finding response times in a real-time system, Comput. J. 29 (5) (1986) 390–395.
[21] R. Kumar, G. Hinton, A family of 45 nm IA processors, in: Proceedings of IEEE International Conference on Solid-State Circuits (ISSCC), 2009.
[22] J. Lehoczky, L. Sha, Y. Ding, The rate monotonic scheduling algorithm: exact characterization and average case behavior, in: Proceedings of IEEE Real Time Systems Symposium, 1989.
[23] J.W.S. Liu, Real-time Systems, Prentice Hall Inc., Upper Saddle River, NJ, USA, 2000.
[24] C.L. Liu, W. James, Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, J. ACM 20 (1) (1973 Jan) 46–61.
[25] R. Melhem, D. Mossé, E. Elnozahy, The interplay of power management and fault recovery in real-time systems, IEEE Trans. Comput. 53 (2004) 217–231.
[26] P. Pillai, K.G. Shin, Real-time dynamic voltage scaling for low power embedded operating system, in: Proceedings of ACM Symposium on Operating Systems Principles, 2001.
[27] D. Pradhan, Fault Tolerant Computer System Design, Prentice Hall Inc., Upper Saddle River, NJ, USA, 1996.
[28] G. Quan, X. Hu, Energy efficient fixed-priority scheduling for real-time systems on variable voltage processors, in: Proceedings of IEEE/ACM Design Automation Conference, 2001.
[29] S. Saewong, R. Rajkumar, Practical voltage-scaling for fixed-priority RT-systems, in: Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'03), 2003.
[30] Y. Shin, K. Choi, Power conscious fixed-priority scheduling for hard real-time systems, in: Proceedings of IEEE/ACM Design Automation Conference (DAC'99), 1999.
[31] R. Sridharan, R. Mahapatra, Reliability aware power management for dual-processor real-time embedded systems, in: Proceedings of the 47th IEEE/ACM Design Automation Conference (DAC), 2010.
[32] M.K. Tavana, M. Salehi, A. Ejlali, Feedback-based energy management in a standby-sparing scheme for hard real-time systems, in: Proceedings of the 32nd IEEE Real-Time Systems Symposium, 2011.
[33] O.S. Unsal, I. Koren, C.M. Krishna, Towards energy-aware software-based fault tolerance in real-time systems, in: Proceedings of the IEEE International Symposium on Low Power Electronics and Design (ISLPED), 2002.
[34] T. Wei, P. Mishra, K. Wu, H. Liang, Fixed-priority allocation and scheduling for energy-efficient fault tolerance in hard real-time multiprocessor systems, IEEE Trans. Parallel Distrib. Syst. 19 (11) (2008) 1511–1526.
[35] Y. Zhang, K. Chakrabarty, Dynamic adaptation for fault tolerance and power management in embedded real-time systems, ACM Trans. Embed. Comput. Syst. 3 (2004 May) 336–360.
[36] B. Zhao, H. Aydin, Minimizing expected energy consumption through optimal integration of DVS and DPM, in: Proceedings of IEEE International Conference on Computer Aided Design (ICCAD), 2009.
[37] B. Zhao, H. Aydin, D. Zhu, Energy Management under general task-level reliability constraints, in: Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012.
[38] D. Zhu, R. Melhem, Mossé, The effects of energy management on reliability in real-time embedded systems, in: Proceedings of IEEE International Conference on Computer Aided Design (ICCAD), 2004.
[39] D. Zhu, X. Qi, H. Aydin, Priority-monotonic energy management for real-time systems with reliability requirements, in: Proceedings of the IEEE International Conference on Computer Design (ICCD), 2007.
[40] D. Zhu, H. Aydin, J.J. Chen, Optimistic reliability-aware energy management for real-time tasks with probabilistic execution times, in: Proceedings of the IEEE Real-Time Systems Symposium (RTSS), 2008.
[41] D. Zhu, H. Aydin, Reliability-aware energy management for periodic real-time tasks, IEEE Trans. Comput. 58 (10) (2009) 1382–1397.
[42] FreeScale Semiconductor. MPC8536E Processor Specifications. http://www.freescale.com/files/32bit/doc/data_sheet/MPC8536EEC.pdf

Mohammad A. Haque received the B.Sc. degree in Computer Science and Engineering from Bangladesh University of Engineering and Technology, Dhaka, Bangladesh in 2006 and the M.Sc. degree in Computer Science from George Mason University, Fairfax, Virginia, USA, in 2010. He is currently a PhD candidate at the Department of Computer Science at George Mason University. His area of research includes low-power computing, real-time systems, and operating systems.

Hakan Aydin is an Associate Professor of Computer Science at George Mason University, United States. He received his Ph.D. in Computer Science from the University of Pittsburgh. He has published more than sixty peer-reviewed papers and served on the program committees of several international conferences and workshops. He received the NSF CAREER Award in 2006. He served as the Technical Program Committee Chair of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS) in 2011. His current research focuses on low-power computing, real-time systems, and fault tolerance.



Dakai Zhu received the PhD degree in Computer Science from University of Pittsburgh in 2004. He is currently an Associate Professor in the Department of Computer Science at the University of Texas at San Antonio. His research interests include real-time systems, power-aware computing and fault-tolerant systems. He has served on program committees (PCs) for several major IEEE- and ACM-sponsored real-time conferences (e.g., RTSS and RTAS). He was a recipient of the US National Science Foundation (NSF) Faculty Early Career Development (CAREER) Award in 2010. He is a member of the IEEE and the IEEE Computer Society.