Technical Report: CS-TR-2013-016, UTSA; Prepared on Oct. 30, 2013

Energy-Efficient Scheduling of Primary/Backup Tasks in Multiprocessor Real-Time Systems (Extended Version)

Yifeng Guo, Dakai Zhu, *Member, IEEE,* Hakan Aydin, *Member, IEEE,* and Laurence T. Yang *Senior Member, IEEE*

Abstract—With the negative effects of the popular *Dynamic Voltage and Frequency Scaling (DVFS)* technique on transient faults being considered, the *Primary/Backup* approach has recently been exploited to save energy while preserving system reliability. In this paper, with the objectives of tolerating a single permanent fault and maintaining system reliability with respect to transient faults, we study energy-efficient dynamic-priority based scheduling algorithms for periodic Primary/Backup tasks on multiprocessor systems. Specifically, by separating primary and backup tasks on their dedicated processors, we first devise two schemes based on the idea of *Standby-Sparing (SS)*: For *Paired-SS*, processors are organized as groups of two (i.e., pairs) and the existing SS scheme is applied within each pair of processors after partitioning tasks to the pairs. In *Generalized-SS*, processors are divided into two groups (of potentially different sizes), which are denoted as *primary* and *secondary* processor groups, respectively. The main (backup) tasks are scheduled on the primary (secondary) processor group under the *partitioned-EDF* (*partitioned-EDL*) with DVFS (DPM) to save energy. Next, instead of dedicating some processors to backup tasks, we propose schemes that allocate primary and backup tasks in a mixed manner to better utilize the slack time on all processors for more energy savings. On each processor, the *Preference-Oriented Earliest Deadline (POED)* scheduler is adopted to run primary tasks at scaled frequencies *as soon as possible (ASAP)* and backup tasks at the maximum frequency *as late as possible (ALAP)* to save energy. Online power management and recovery strategies are further discussed to address the problem with multiple permanent faults. Our empirical evaluations show that, for systems with a given number of processors, there normally exists an optimal configuration of primary and secondary groups for the Generalized-SS scheme, which leads to better energy savings compared to that of the Paired-SS scheme. Moreover, the POED-based schemes normally perform more stable and achieve better energy savings compared to those of the SS-based schemes.

Index Terms—Real-Time Systems; Multiprocessor; Fault Tolerance; Primary/Backup; Energy Management; DVFS; DPM;

1 Introduction

Fault tolerance has been a traditional research topic in real-time systems as computing devices are subject to different types of faults at runtime. In general, to tolerate various faults and guarantee that real-time tasks can complete their executions successfully in time, the existing fault tolerance techniques normally adopt different forms of redundancy. For instance, as a simple and well-studied approach, *hot-standby* exploits *hardware/modular* redundancy and runs two copies of the same task *concurrently* on two processors to tolerate a single fault [32]. However, by their very nature, such redundancy-based fault-tolerance techniques demand more system resources, which can lead to excessive energy consumption (e.g., hot-standby has 100% energy overhead).

On the other hand, with the ever-increasing power density in modern computing systems, energy has been promoted as the first-class system resource and energy-aware computing has become an important research area [24]. As a common energy saving technique, *dynamic power management (DPM)* can power down (or turn off) components when they are not in use. Moreover, as a fine-grained power management technique, *dynamic voltage and frequency scaling (DVFS)* can operate computing systems at different low-performance (and thus low-power) states when the performance demand is not at the peak level by simultaneously scaling down their supply voltage and processing frequency [34].

Although both redundancy-based fault-tolerance [6], [17] and DPM/DVFS-based energy management schemes [34], [55] have been independently studied extensively, the co-management of system reliability and energy consumption has caught researchers' attention only very recently [14], [30], [37]. Note that, fault-tolerance and energy-efficiency are normally conflicting design objectives in computing systems since redundancy generally means more energy consumption [2]. Moreover, recent studies show that DVFS has a negative effect on system reliability due to significantly increased transient fault rates at low supply voltages [11], [15], [50]. With such intriguing interplay between fault-tolerance and energy-efficiency, it becomes imperative to develop effective techniques that can address both aspects while guaranteeing the timeliness of real-time tasks.

By taking the negative effects of DVFS on transient fault rates into consideration, a series of *reliability-aware power-management (RAPM)* schemes have been studied for various real-time systems based on the backward recovery technique [13], [19], [29], [33], [44], [45], [46], [47]. Basically, RAPM exploits system slack (i.e., *temporal* redundancy) for both reliability preservation and energy savings. Here, RAPM ensures that there is always a recovery task scheduled before scaling down the processing frequency of any task using the remaining slack time. By enforcing the recovery task to be

- *Y. Guo and D. Zhu are with Department of Computer Science, The University of Texas at San Antonio, San Antonio, TX 78249, USA.*
- *H. Aydin is with Department of Computer Science, George Mason University, Fairfax, VA 22030, USA.*
- *L. T. Yang is with Department of Computer Science, St. Francis Xavier University, Antigonish, NS, B2G 2W5, Canada.*

executed at the maximum frequency when the scaled task does incur transient faults, RAPM guarantees to achieve a desired system reliability level [29], [44], [47]. Although RAPM can guarantee system reliability with respect to transient faults (which were shown to be more common [26]), there is no provisions for tolerating permanent faults.

With the objective of tolerating a single permanent fault while guaranteeing system reliability with respect to transient faults, the *Standby-Sparing (SS)* schemes were recently studied for both aperiodic [12] and periodic tasks [22], [23] running on a dual-processor system based on the *Primary/Backup (PB)* fault-tolerance technique. Essentially, the SS schemes schedule primary and backup tasks *separately* on the primary and secondary processors, respectively, to tolerate one permanent fault. Note that, to improve system efficiency and reduce execution overhead (thus to save energy), the backup tasks are normally cancelled as soon as their corresponding primary tasks complete successfully [6] and should be scheduled as late as possible [37]. Hence, for energy-efficiency (and reliability preservation), the SS schemes execute primary tasks at scaled frequency as early as possible and backup tasks at the maximum frequency at their latest times on their dedicated processors, respectively [12], [22], [23].

Although the SS schemes are effective to tolerate a single permanent fault while obtaining some energy savings, the available slack time on the secondary processor is not efficiently utilized with the adopted DPM technique. Instead of dedicating one processor for backup tasks, the primary and backup tasks can be allocated in a mixed manner on both processors and all available slack time can be exploited by the DVFS technique for better energy savings [20].

Specifically, each processor will have a mixed set of primary and backup tasks where primary tasks exploit the slack time and run at scaled frequency with the DVFS technique. Moreover, the tasks are scheduled with the *preference-oriented earliest-deadline (POED)* scheduling algorithm, which can differentiate them and execute primary tasks *as soon as possible (ASAP)* while backup tasks *as late as possible (ALAP)* [18], for better energy savings. With the same idea of allocating tasks in a mixed manner, the *Secondary Execution Time Shifting (SETS)* heuristic was studied for periodic tasks running on multiprocessor systems to delay backup tasks and reduce the overlapped execution with their corresponding primary tasks and thus save energy) [37].

However, the aforementioned studies either focused on dual-processor systems [12], [20], [22], [23] or did not consider the more effective DVFS power management technique [37]. To the best of our knowledge, there is no existing work that address *how to effectively schedule periodic primary/backup tasks in multiprocessor systems to save energy with the DPM/DVFS techniques while tolerating a single permanent fault and preserving system reliability with respect to transient faults*. Considering the increasing popularity of multicore processors, we focus on such a problem and propose several *energy-efficient fault-tolerance (EEFT)* schemes. In particular, the contributions of this work are summarized as follows:

- First, we study two Standby-Sparing (SS) based EEFT schemes: *Paired-SS* organizes processors as groups of two (i.e., *pairs*) and adopts the existing SS scheme for dual-processor systems within each processor pair; *Generalized-SS* divides processors into *primary* and *secondary* processor groups (of potentially different sizes) and then schedules primary (backup) tasks on the primary (secondary) processors under the *partitioned-EDF* (*partitioned-EDL*) with DVFS (DPM) to save energy;
- Second, by allocating primary and backup tasks in a mixed manner on all processors to better utilize their slack time for more energy savings, we propose two additional EEFT scheme based on the POED scheduling algorithm; Here, once primary tasks are partitioned to *all* processors (e.g., according to the Worst-Fit-Decreasing heuristic), backup tasks can be allocated to processors following either *Cyclic* or *Mixed* approach;
- Third, the recovery strategies are further discussed for the proposed schemes to obtain smaller recovery windows and address the problem of multiple permanent faults;
- Finally, the proposed EEFT schemes are evaluated through extensive simulations and the results show their effectiveness on energy savings.

The remainder of this paper is organized as follows. Section 2 reviews the closely-related work. Section 3 presents system models and states the assumptions of this work. The Standby-Sparing based schemes are discussed in Section 4 and the POED-based schemes are investigated in Section 5. The online power management with wrapper-tasks and recovery strategies for the proposed EEFT schemes are addressed in Section 6. The evaluation results are presented and discussed in Section 7 and Section 8 concludes the paper.

2 Closely Related Work

Both fault-tolerance and energy management for computing systems have been extensively (but *independently*) studied. Based on the primary/backup fault-tolerance technique, Bertossi *et al.* studied several fixed-priority RMS scheduling algorithms for periodic real-time task running on multiprocessor systems to improve system resource utilization through backup deallocation [6]. In [5], the authors further explored the *backup phasing delay* technique to reduce the overlapped executions between the primary and backup tasks. However, the work did not consider energy management. Based on the EDF scheduling, Unsal *et al.* studied an offline Secondary Execution Time Shifting (SETS) heuristic for a set of independent periodic real-time tasks running on multiprocessor systems [37]. Here, to obtain an energy-efficient static schedule within the least common multiple (LCM) of tasks' periods, SETS iteratively delays the release time of backup tasks to reduce the overlapped executions with their corresponding primary tasks and thus to reduce system energy consumption, but without exploiting the more effective DVFS technique.

By exploiting the DVFS power management technique, Melhem *et al*. derived the optimal number of checkpoints, *uniformly* or *non-uniformly* distributed, to achieve the minimum energy consumption while tolerating a fixed number of transient faults for a duplex system (where two hardware units are used to run the same software concurrently for fault

detection) [30]. In [14], Elnozahy *et al*. studied an *Optimistic-TMR* (OTMR) scheme to reduce the energy consumption in a traditional Triple Modular Redundancy (TMR) system. OTMR allows one processing unit to run at a scaled frequency with DVFS provided that it can catch up and finish the computation before the deadline if a fault does occur on other two units. The optimal frequency settings for OTMR was explored in [51]. For independent service requests, Zhu *et al.* studied the optimal redundant configuration for server processors to tolerate a given number of transient faults [52].

Assuming that transient faults follow a Poisson distribution with a constant arrival rate, Zhang *et al.* studied an adaptive checkpointing scheme to tolerate a fixed number of transient faults during the execution of a single real-time task [40]. The adaptive checkpointing scheme was extended to a set of periodic real-time tasks on a single processor system with the EDF scheduling algorithm [42]. In [41], the authors further considered the cases where faults may occur within checkpoints. Following a similar idea and considering a fixed-priority RMS algorithm, Wei *et al.* studied an efficient online scheme to minimize energy consumption with the consideration of the run-time behaviors of tasks and fault occurrences while still satisfying the timing constraints [38]. In [39], the authors extended the study to multiprocessor real-time systems. Izosimov *et al.* studied an optimization problem for mapping a set of tasks with reliability constraints, timing constraints and precedence relations to processors and for determining appropriate fault tolerance policies (re-execution and replication) for the tasks [27].

However, despite the effectiveness of DVFS on reducing energy consumption, recent studies shown that it has a negative effect on system reliability due to the significantly increased transient fault rates at low supply voltages [15]. In particular, an exponential fault rate model with scaled voltage was proposed in [50]. Taking such negative effects of DVFS into consideration, Zhu studied a *Reliability-Aware Power Management (RAPM)* scheme that schedule a recovery task before exploiting the remaining slack time to scale down the execution of the primary task [46]. Here, to preserve system reliability with respect to transient faults, the recovery task is executed at the maximum frequency only if transient faults cause an error during the primary task's execution. Later, the RAPM scheme was extended to consider multiple tasks with a common deadline [53] as well as periodic real-time tasks [47]. For periodic tasks that have different reliability requirements, Zhu *et al.* investigated the schemes that selectively recover a subset of jobs for each task [54]. Moreover, for tasks with known statistical execution times, an optimistic RAPM was studied that deploys smaller size recovery tasks while still preserving the system reliability [48].

To address the conservativeness of RAPM that schedules an individual recovery task for any scaled task, Zhao *et al.* studied the *Shared-Recovery (SHR)* technique [45], where several scaled tasks can share one recovery task to leave more slack for DVFS to save more energy. To achieve an arbitrary system-level target reliability, SHR has been further extended to the *generalized shared recovery (GSHR)* technique where a small number of recovery tasks are shared by all the tasks [43], [44]. A similar study was also reported recently in [29]. Global scheduling based RAPM schemes for both independent [33] and dependent [19] real-time tasks running on multiprocessor systems were studied as well.

Moreover, based on the exponential fault rate model developed in [50], Ejlali *et al.* studied a number of schemes that combine the information about hardware resources and temporal redundancy to save energy and to preserve system reliability [13]. Considering dependent tasks represented by directed acyclic graphes (DAGs), Pop *et al.* proposed a novel framework by studying the energy and reliability trade-offs for distributed heterogeneous embedded systems [31]. By employing a feedback controller to track the overall miss ratio of tasks in soft real-time systems, Sridharan *et al.* [35] proposed a reliability-aware energy management algorithm to minimize the system energy consumption while still preserving the overall system reliability. Dabiri *et al.* studied the problem of assigning frequency and supply voltage to tasks for energy minimization subject to reliability as well as timing constraints [10]. For a real-time application running a dual-processor system, Aminzadeh and Ejlali performed a comparative study of different DVFS and DPM schemes to tolerate a given number of transient faults [2]. Although the above work can preserve system reliability with respect to transient faults, there is no provision for permanent faults.

To tolerate a single permanent fault while taking transient faults into consideration, Ejlali *et al.* investigated a *Standby-Sparing (SS)* scheme to save energy for dependent and aperiodic real-time tasks running on a dual-processor system [12]. SS executes primary tasks with DVFS on one processor (denoted as the *primary* processor) at their earliest times while backup tasks with DPM on another (*spare*) processor at their latest times to reduce their overlapped executions and thus to save more energy. The work was extended later with a light-weight feedback system for better energy savings [36]. With the same idea of *separating* tasks on different processors, Haque *et al.* extended standby-sparing to a more practical periodic task model based on the earliest deadline scheduling [22]. Here, for energy efficiency, primary and backup tasks are scheduled according to EDF with DVFS and EDL [8] with DPM on their dedicated processors, respectively. The fixed-priority (i.e., RM) based standby-sparing scheme was further studied in [23]. Observing the inefficient usage of slack time with DPM on the spare processor, Guo *et al.* proposed to schedule a mixed set of primary and backup copies of different tasks on both processors, where all available slack time can be utilized to scale down primary tasks with DVFS for better energy savings [20].

Different from all existing research, the generalized Standby-Sparing schemes for periodic primary/backup tasks running on multiprocessor systems have been reported in our preliminary work of this study [21]. Based on the *Preference-Oriented Earliest Deadline (POED)* algorithm [18], additional *energy-efficient fault-tolerance (EEFT)* schemes are investigated in this paper. Moreover, recovery strategies are also discussed for the problem with multiple permanent faults.

3 PRELIMINARIES AND SYSTEM MODELS

In this section, we present the system models and state our assumptions that this work is based upon. The problem description is also given at the end.

3.1 Task, System and Power Models

We consider a set of n independent periodic real-time tasks $\Gamma = \{T_1, \ldots, T_n\}$, where each task T_i is represented as a tuple (c_i, p_i). Here c_i is T_i's worst-case execution time (WCET) under the maximum available processor frequency and, p_i is its period. The tasks are assumed to have implicit deadlines. That is, the j^{th} task instance (or *job*) of T_i, denoted as $T_{i,j}$, arrives at time $(j-1) \cdot p_i$ and needs to complete its execution by its deadline at $j \cdot p_i$. Note that, a task has only one active task instance at any time. Hence, when there is no ambiguity, we use T_i to represent both the task and its current task instance. The utilization of a task T_i is defined as $u_i = \frac{c_i}{p_i}$. The system utilization of a given task set is further defined as the summation of all tasks' utilization: $U(\Gamma) = \sum_{T_i \in \Gamma} u_i$.

The tasks are to be executed on a shared-memory system with m processors, which have identical functionalities. As power management features have become common in modern processors [1], [9], we assume that all processors adopted in the system have the *dynamic voltage and frequency scaling (DVFS)* capability, which allows them to operate at one of the L discrete frequency (and voltage) levels ($F_1 < F_2 < \ldots < F_L$). With normalized frequencies being considered, the maximum frequency is assumed to be $F^{max} = F_L = 1.0$.

With the shrinking technology size, the static and leakage power increases in a faster pace when compared to that of dynamic power [28]. Hence, it becomes more important to manage power consumption at the system-level with all power consuming components being considered [3], [25]. To ease the discussion and analysis, we adopt in this work a simple system-level power model, which has been widely exploited in recent studies [29], [33], [47]. Specifically, for a system with m processors (that operate at $f_1, \ldots, f_m$, respectively), its power consumption can be expressed as:

$$P(f_1, \ldots, f_m) = P_s + \sum_{i=1}^{m} \hbar_i (P_{ind} + C_{ef} \cdot f_i^k) \qquad (1)$$

where P_s stands for the *static power*, which can be removed only by powering off the whole system. However, due to the prohibitive overhead of turning off and on the system [14] in periodic real-time execution settings, we assume that the system is in *on* state at all times and that P_s is always consumed. That is, we will focus on the energy consumption related to system active power, which is given by the second item in the above equation.

For each processor, if it is *actively* executing tasks, two types of active power will be consumed: the *frequency-independent* active power P_{ind} (which is assumed to be the same for all processors) and the *frequency-dependent* active power (which depends on the system-dependent constants C_{ef} and k, as well as the processor's frequency f_i). That is, if the i'th processor is active, we have $\hbar = 1$. Otherwise, if there is no ready task on the i'th processor, it can switch to the *sleep* state through the dynamic power management (DPM) technique and does not consume any active power (i.e., $\hbar = 0$).

Considering the fact that modern processors can switch to sleep states in a few cycles [1], [9], we assume that the time overhead for a processor to enter/exit its sleep state is negligible. Moreover, to simplify the discussion, the overhead for frequency (and voltage) changes under DVFS is also assumed to be included into tasks' WCETs or can be incorporated with the slack reservation mechanism [49].

From the above system-level power model, an *energy-efficient frequency*, $F_{ee} = \sqrt[k]{\frac{P_{ind}}{C_{ef} \cdot (k-1)}}$, can be derived, below which DVFS consumes more energy to execute a task [33]. In what follows, we assume that all available frequency levels are energy-efficient and there is $F_{ee} \leq F_1$.

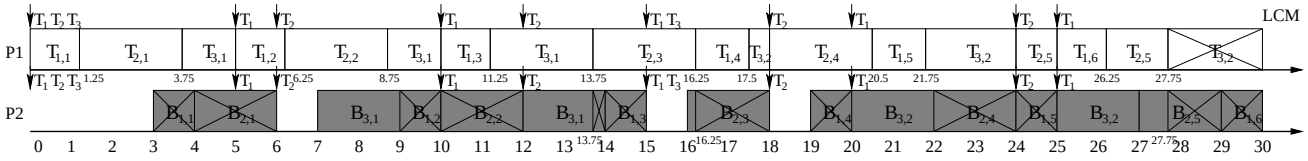
3.2 Fault and Recovery Models

During the operation of a real-time system, different faults may occur due to hardware failure, software errors or electromagnetic interferences. While *transient* faults can be tolerated with *temporal* redundancy, *permanent* faults can only be tolerated through *modular/hardware* redundancy. Historically, it has been reported that transient faults are much more common than permanent faults [26]. With the scaled technology size [15] and widely-adopted DVFS power management technique, modern computing devices are more susceptible to transient faults [11]. In particular, as supply voltage is reduced with DVFS to save energy, the rate of transient faults may increase exponentially [50]. Although the occurrence of permanent faults is very rare, a comprehensive framework should have provisions for both transient and permanent faults in a safety-critical multiprocessor real-time system.

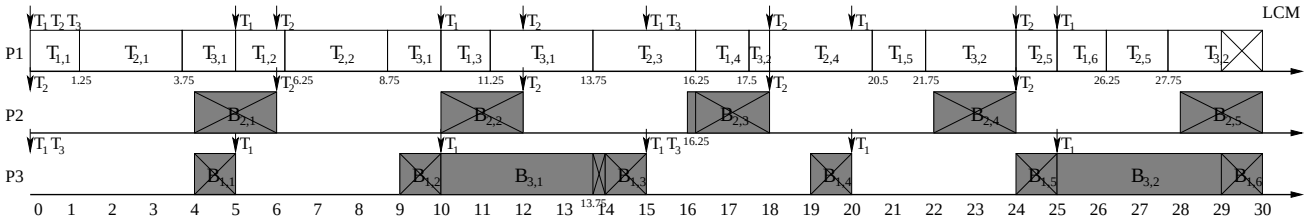
With the objective of tolerating both transient and permanent faults, we adopt the *Primary/Backup (PB)* fault-tolerance technique in this work. That is, for each task T_i, there is a periodic *backup* task B_i. To distinguish between them, we occasionally use the term *primary* (or *main*) task to refer to T_i. Here, to ensure that there is a proper backup for every task instance of T_i, we assume that B_i has the same timing parameters[1] (i.e., c_i and p_i) as T_i. Hence, in addition to the original primary task set Γ, we have a set Γ^B of backup tasks that have to be properly scheduled.

The same as in most existing fault-tolerance work, we assume that fault detection mechanisms are available in the system and the detection overhead has been incorporated into the WCETs of tasks [2]. Specifically, for soft errors caused by transient faults, they are detected at the end of a task's execution with the *sanity* (or *consistency*) checks (e.g., parity or signature checks) [32]. For permanent faults, we assume the failure-stop model and a faulty processor can be detected by other working ones at the earliest completion time of a task through the message-based mechanism [32]. Hence, we do not use the backup tasks for fault detection, which are for fault-tolerance only to replace the faulty execution whenever a transient or permanent fault is detected.

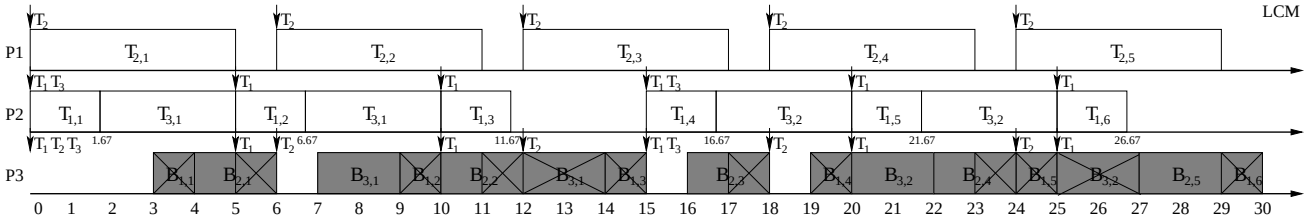
1. Note that, as long as B_i's WCET is no more than that of T_i (i.e., B_i can be either a reduced version or the replication of T_i), the proposed schemes can guarantee system reliability with respect to transient faults [12], [29], [47].



a. The schedule of the tasks under the Standby-Sparing on a dual-processor system with primary frequency at 0.8 [22];



b. The schedule of the tasks for the case of one primary processor (frequency at 0.8) with two secondary processors;



c. The schedule of the tasks for the case of two primary processors (frequencies at 0.4 and 0.6) with one secondary processor;

Fig. 1: A set of three tasks $T_1(1,5)$, $T_2(2,6)$ and $T_3(4,15)$ running on a three-processor system.

Problem Description: With DPM and DVFS techniques being exploited to save energy, the problem to be addressed in this paper is: *how to efficiently schedule the main and backup tasks on a multiprocessor system to maximize the energy savings under the conditions of a) tolerating a single permanent fault; and b) preserving system reliability with respect to transient faults (in the absence of permanent faults).*

The same as in [12], [22], the backup tasks adopted in this work have dual purposes. First, with one backup for each main task, the system is inherently robust to a single permanent fault on any processor provided that *the main and backup copies of the same task are scheduled on different processors* [6]. The second objective of the backup tasks is, in the absence of permanent faults, to preserve system reliability[2] with respect to transient faults when the execution of primary tasks is scaled down with DVFS to save energy. For such a purpose and considering the negative effects of DVFS on transient fault rates [50], backup tasks should be executed at the maximum frequency [12], [29], [47].

4 Standby-Sparing for Multiprocessor

For dual-processor systems, the central idea of Standby-Sparing (SS) is to run separately the main and backup tasks on *primary* and *secondary* processors, respectively [12], [22]. Here, the main tasks are executed at scaled frequency with DVFS on the primary processor while backup tasks run at the maximum frequency on the secondary processor but are delayed to reduce the overlapped executions with their corresponding main tasks for energy savings with DPM. Following the same idea of separating main and backup tasks on their dedicated processors, we first study two SS-based schemes for multiprocessor systems: *Paired-SS* and *Generalized-SS*, which have been reported in our preliminary work [21].

4.1 An Example: Three-Processor Systems

When there are more (i.e. > 2) processors in a system, a natural question to ask would be: *"how to configure such processors for better energy efficiency?"* We can either have additional primary processors to execute main tasks at further reduced frequencies or have more secondary processors to further delay the execution of backup tasks. Clearly, this is not a trivial problem considering the intriguing interplay between the scaled frequency of main tasks and the amount of overlapped execution with their backup tasks.

Before presenting the solution for the general problem for multiprocessor systems, we first investigate the simple case of three-processor systems. Here, we have two options for the configuration of the processors: a) one primary and two secondary processors (denoted as "X1Y2"); and b) two primary and one secondary processor (denoted as "X2Y1").

A Motivational Example: Consider a task set with three periodic real-time tasks $\Gamma = \{T_1(1,5), T_2(2,6), T_3(4,15)\}$. We can easily find that the system utilization is $U = 0.8$ and the *least common multiple (LCM)* of tasks' periods is $LCM = 30$. Suppose that the processors have four discrete (normalized) frequency levels $\{0.4, 0.6, 0.8, 1.0\}$.

Figure 1a first shows the tasks' schedule on a dual-processor system with the Standby-Sparing technique within LCM [22].

2. Higer levels of system reliability can be achieved with additional replicated copies of tasks [29], [44]. However, exploring this direction is beyond the scope of this paper and will be investigated in our future work.

Here, the primary processor executes the main tasks under *Earliest Deadline First (EDF)* at a scaled frequency of 0.8 while the secondary processor schedules the backup tasks with *Earliest Deadline Latest (EDL)* [8] for energy savings. By assuming $P_{ind} = 0.01$, $C_{ef} = 1$ and $k = 3$ in the power model, we can find the active energy consumption within LCM is $E_{SS-SPM} = 27.2$ when all tasks take their WCETs and there is no fault at run-time. Here, most executions of the backup tasks are cancelled, which are marked with an 'X'.

For the X1Y2 configuration of a three-processor system, where the extra processor is used as an additional secondary, the schedule of tasks is shown in Figure 1b. Here, the primary processor $\mathcal{P}_1$ still runs at the frequency 0.8 to execute the main tasks. However, backup tasks can be re-allocated, where backup task B_2 is allocated to processor $\mathcal{P}_2$ and B_1 and B_3 to processor $\mathcal{P}_3$, to further delay and reduce their overlapped executions. It turns out that, although the overlapped executions can be reduced slightly, the additional energy consumption from the frequency-independent active power (i.e., P_{ind}) of the extra processor overshadows such benefits and leads to the total active energy consumption of $E_{X1Y2} = 27.45$, which is slightly more than that of the traditional Standby-Sparing scheme for the dual-processor system.

However, when the extra processor is utilized as an additional primary, Figure 1c shows the schedule of tasks under the X2Y1 configuration. Here, the main task T_2 is allocated to processor $\mathcal{P}_1$ and other two tasks (T_1 and T_3) to $\mathcal{P}_2$, which can run at the scaled frequencies of 0.4 and 0.6, respectively. The total active energy consumption under this configuration can be found as $E_{X2Y1} = 23.37$, which is a 14% improvement over the traditional Standby-Sparing scheme.

4.2 Standby-Sparing Based Schemes

From the above example, we can see that different configurations of primary and secondary processors can have important effects on the energy efficiency of multiprocessor systems. Following the ideas and principles of the traditional Standby-Spring scheme [22], we propose in what follows the *Paired Standby-Sparing* (Section 4.2.1) and *Generalized Standby-Sparing* (Section 4.2.2) schemes for periodic real-time tasks running on multiprocessor systems [21].

4.2.1 Paired Standby-Sparing (P-SS)

Considering the fact that the traditional Standby-Sparing scheme was designed for dual-processor systems [22], a simple and straightforward approach is to first organize the processors in a system as groups of two (i.e., *pairs*). Then, the existing Standby-Spring scheme can be applied directly to each pair of processors after partitioning (main and backup) tasks to the processor pairs appropriately, which is thus named as the *Paired Standby-Sparing (P-SS)* scheme.

From the results in [22], we know that different system utilizations of tasks have a great impact on the energy efficiency of a dual-processor system under the Standby-Sparing scheme. The reason is that, both the scaled frequency for the main tasks on the primary processor and the delayed execution of backup tasks on the secondary processor depend heavily on system loads. When the system utilization of a given task set is high, the Standby-Sparing scheme could perform quite worse due to higher execution frequency for main tasks and the increased amount of overlapped execution between the main and backup tasks. On the other hand, once the scaled execution frequency of the main tasks reduces to the minimum (available) energy-efficient frequency, additional energy savings are rather limited with further reduced system loads [22].

Therefore, the *key* factor for the energy efficiency of a multiprocessor system under P-SS will be the *mapping* of tasks to processor pairs. However, it is well-known that the problem of finding a *feasible* mapping of a given set of periodic real-time tasks in a multiprocessor system is NP-hard. Therefore, finding the optimal mapping of (main and backup) tasks among the processor pairs in P-SS to minimize the system energy consumption is NP-hard as well. Note that, without fault-tolerance being considered, the balanced workload among processors has been shown to have the best energy efficiency for tasks running on a multiprocessor system [4]. Hence, following this intuition and considering its inherent ability to obtain a load-balanced mapping, we adopt the *Worst-Fit Decreasing (WFD)* heuristic in P-SS when mapping (main and backup) tasks to the processor pairs.

Note that, to apply the traditional Standby-Sparing within each processor pair, the main and backup of the same task (e.g., T_i and B_i) have to be mapped to the same pair of processors. Therefore, in P-SS, we can first map the main tasks in Γ to the processor pairs according to the WFD heuristic. That is, each processor pair will be allocated a subset Γ_q of the main tasks, where $1 \leq q \leq \lfloor \frac{m}{2} \rfloor$ as there are at most $\lfloor \frac{m}{2} \rfloor$ processor pairs for a system with m processors. Then, for each backup task B_i, it will be allocated to the processor pair that contains the corresponding main task T_i,

With the Standby-Sparing scheme being adopted within each processor pair, the main and backup tasks are scheduled under EDF and EDL on the primary and secondary processors, respectively [22]. Recall that backup tasks have the same timing parameters (i.e., utilizations) as their main tasks. Therefore, the resulting WFD mapping $\{\Gamma_q\}$ (and corresponding $\{\Gamma_q^B\}$) is feasible if there are $U(\Gamma_q) \leq 1$ ($1 \leq q \leq \lfloor \frac{m}{2} \rfloor$).

Once the feasible WFD mapping is obtained, the processor pairs under P-SS will operate *independently*. Although each processor pair acting as a Standby-Spring system can tolerate one permanent fault [22], it is possible for multiple permanent faults hit both processors in one pair. Hence, with each task having one backup, P-SS can only tolerate a single permanent fault in the worst case scenario. However, once the processor affected by permanent fault(s) is identified and isolated, we can re-configure the system with the remaining $(m-1)$ processors and/or re-map the tasks to tolerate additional permanent faults, which will be further discussed in Section 6.2.

4.2.2 Generalized Standby-Sparing (G-SS)

For the example system with three processors (Section 4.1), we have seen that having two primary processors to execute the main tasks while sharing one secondary processor among the backup tasks can lead to higher energy efficiency. Following this principle and generalizing the idea of Standby-Sparing,

Algorithm 1 : G-SS for a given (X, Y)-configuration

1: **Input:** task sets Γ and Γ^B; X and $Y(= m - X)$;
2: **Output:** Scaled frequencies for primary processors and EDL schedules for secondary processors;
3: Find the (X, Y) WFD partitions of Γ and Γ^B:
4: $\Pi(X) = \{\Gamma_1, \cdots, \Gamma_X\}$ and $\Pi^B(Y) = \{\Gamma_1^B, \cdots, \Gamma_Y^B\}$;
5: **if** $(\forall i, U(\Gamma_i) \leq 1$ and $\forall j, U(\Gamma_j^B) \leq 1)$ **then**
6: //Suppose the first X processors are primary processors
7: **for** (each primary processor $\mathcal{P}_x$: $x = 1 \rightarrow X$) **do**
8: $f_x = \min\{F_i | F_i \geq U(\Gamma_x), i = 1, \ldots, L\}$;
9: **end for**
10: **for** (each secondary processor $\mathcal{P}_y$: $y = 1 \rightarrow Y$) **do**
11: Generate the offline EDL schedule for tasks in Γ_y^B;
12: **end for**
13: **end if**

Algorithm 2 : Find the optimal configuration for G-SS

1: **Input:** task sets Γ and Γ^B; m (number of processors);
2: **Output:** the optimal processor configuration (i.e., X^{opt}) for G-SS to minimize energy consumption;
3: $X^{min} = \lceil U(\Gamma) \rceil$; $X^{max} = m - X^{min}$;
4: $E^{min} = \infty$; $X^{opt} = -1$; //initialization
5: **for** $(X = X^{min} \rightarrow X^{max})$ **do**
6: $Y = m - X$; //number of secondary processors
7: **if** (Γ is schedulable under G-SS with X/Y) **then**
8: Get $E_{G-SS}(X, Y)$ from emulation in LCM;
9: **if** $(E^{min} > E_{G-SS}(X, Y))$ **then**
10: $X^{opt} = X$;
11: **end if**
12: **end if**
13: **end for**

we propose the *Generalized Standby-Sparing (G-SS)* scheme, which organizes the m processors of the system into two groups: the *primary* group of X processors and the *secondary* group of Y processors, where $m = X + Y$. Then, the main and backup tasks are *separately* scheduled on the processors in the primary and secondary groups, respectively.

Considering the fact that the EDF/EDL schedulers are exploited in the P-SS scheme and the simplicity of partitioned scheduling, we adopt the *partitioned-EDF* and *partitioned-EDL* for G-SS to schedule the main and backup tasks, respectively. Hence, for a given (X, Y)-configuration of the processors, Algorithm 1 summarizes the major steps of G-SS.

First, the main and backup tasks are partitioned among the X primary and Y secondary processors, respectively (lines 2 and 3). Again, to obtain the mappings with balanced-workload for better energy savings, the WFD heuristic is adopted [4]. Then, the schedulabilities of the resulting WFD mappings for both the main and backup tasks under the EDF and EDL schedulers on the primary and secondary processors, respectively, are examined (line 4).

If any processor is overloaded with the resulting mappings $\Pi(X)$ and $\Pi^B(Y)$, we say that the (X, Y)-configuration is *not feasible*. Otherwise, to save energy, the scaled frequency for each primary processor to execute its main tasks is determined (lines 6 and 7); in addition, assuming that backup tasks run at the maximum frequency, the EDL schedule for each secondary processor is generated offline (lines 9 and 10).

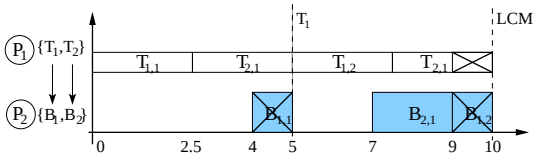
With all backup tasks running on different processors from their main tasks, G-SS is able to tolerate a single permanent fault. Moreover, the system reliability with respect to transient faults can also be preserved since all backup tasks are assumed to run at F^{max}. Note that, the same as in the traditional Standby-Sparing scheme [22], if a main (or backup) task completes successfully on one processor at runtime, the related processor will be notified to cancel the execution of the corresponding backup (or main) task for energy savings.

It is clear that different configurations of the processors in G-SS have a great impact on the energy efficiency of a multiprocessor system. For the special case that has the same number of primary and secondary processors (i.e., $X = Y$), we can find that G-SS will be effectively reduced to P-SS since they adopt the same WFD mapping heuristic and the backup tasks have the same timing parameters as their corresponding main tasks. However, for the configurations that have different numbers of primary and secondary processors (i.e., $X \neq Y$), it is very likely that the backup tasks are mapped to different secondary processors in G-SS even if their main tasks are mapped to the same primary processor. This is quite different from P-SS, where each primary processor has a *dedicated* secondary processor and the same subset of main and backup tasks are always executed on these two processors, respectively. Due to such implications, it is quite difficult to identify the overlapped execution regions between the main and backup tasks in the EDF and EDL schedules on different processors, which makes it impossible to find the optimal configuration of processors for G-SS to minimize energy consumption analytically.

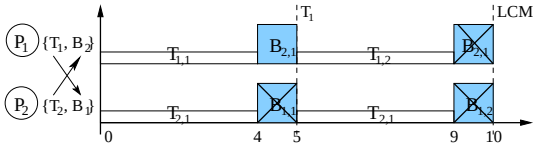
4.2.3 Optimal Processor Configuration for G-SS

For a given task set Γ running on a m-processor system, we present an iterative algorithm to find out the optimal processor configuration for G-SS to minimize system energy consumption, where the major steps are shown in Algorithm 2. Note that, with the system utilization of $U(\Gamma)$, the minimum number of required primary processors for the tasks to be schedulable under partitioned-EDF can be obtained as $X^{min} = \lceil U(\Gamma) \rceil$. X^{min} also gives the minimum number of required secondary processors. Thus, the maximum number of primary processors can be found accordingly $X^{max} = m - X^{min}$ (line 3).

For each possible (X, Y)-configuration of the processors, the schedulability of the given task set Γ under G-SS can be checked using Algorithm 1 (lines 5 to 7). If Γ is schedulable, the system energy consumption under G-SS can be obtained from the emulated execution of the tasks within LCM (line 8). During such emulations, we assume that tasks take their WCETs and no fault occurs. Finally, searching through all feasible configurations of the processors, the optimal (X^{opt}, Y^{opt})-configuration with the lowest system energy consumption can be found out (lines 9 and 10). As shown in Section 7.1, such an optimal configuration for G-SS normally has more primary processors and can lead to better energy efficiency when compared to that of the P-SS scheme.



a. Schedule for the Standby-Sparing scheme [22]



b. Schedule for mixed allocations of Primary/Backup

Fig. 2: An example of two tasks $T_1 = (1, 5)$ and $T_2 = (2, 10)$ running on a dual-processor system.

5 Schemes with Mixed Primary/Backup

The separation of main and backup tasks on their dedicated processors simplifies the scheduling algorithm on each processor for the SS-based schemes. However, since backup tasks need to run at the maximum frequency for reliability preservation, the available slack time on secondary processors can only be used to idle processors with the DPM technique to save energy. As illustrated in the following example of a dual-processor system, better energy savings can be obtained if the main and backup tasks are allocated in a mixed manner on both processors [20], which can more efficiently utilize all available slack time with the DVFS technique.

5.1 Inefficient Slack Usage in Standby-Sparing

Consider a dual-processor system with two periodic tasks $T_1 = (1, 5)$ and $T_2 = (2, 10)$. The schedule within the LCM of tasks' periods under the Standby-Sparing scheme is shown in Figure 2a. Here, the main tasks T_1 and T_2 are executed at the scaled frequency of 0.4 on the primary processor under EDF, while the backup tasks B_1 and B_2 are scheduled on the secondary processor under EDL [22]. Clearly, as B_1 and B_2 are required to run at the maximum frequency for reliability preservation, the slack time on the secondary processor can only be exploited by DPM to idle the processor.

However, it is well-known that slack time can be more efficiently utilized by the DVFS technique [34], [55]. Therefore, instead of dedicating one processor for backup tasks, we can allocate the main and backup tasks to both processors in a *mixed* manner as shown in Figure 2b. Here, T_1 and B_2 are allocated to the first processor while T_2 and B_1 to the second processor. Hence, each processor can utilize its slack time for its main task and it turns out that, with DVFS, both T_1 and T_2 can be executed at the scaled frequency of 0.25.

Suppose that tasks take their WCETs and no fault occurs during tasks' execution. When tasks are executed according to the schedule within the LCM in Figure 2b, most executions of backup tasks will be cancelled (marked with 'X'). Hence, when compared to the case of the Standby-Sparing schedule as shown in Figure 2a, about 20% more energy savings can be obtained under the new scheme with mixed allocations of main and backup tasks on both processors.

However, we should point out that it is not trivial to obtain such a schedule in Figure 2b, which is neither an EDF nor EDL schedule. From the figure, we can see that, to obtain more energy savings, the main tasks on each processor are executed at their earliest times while the backup tasks are delayed as much as possible (without causing any deadline miss). To efficiently generate such schedules, in what follows, we first review the basic ideas of the *preference-oriented earliest deadline (POED)* scheduling algorithm [18], which forms the **foundation** of the novel energy-efficient fault-tolerance schemes with mixed allocations of main and backup tasks.

5.1.1 A Preference-Oriented Scheduling Algorithm

Basically, POED is a dynamic-priority based scheduler to schedule a set of periodic real-time tasks on a single processor system. However, different from the conventional earliest-deadline schedulers, such as EDF and EDL [8] (which treat *all* tasks *uniformly* and schedule them at their earliest and latest times, respectively), POED can distinguish different *execution preferences* of tasks, which can be either *as soon as possible (ASAP)* or *as late as possible (ALAP)* [18].

To incorporate such execution preferences of tasks, POED follows two principles when making scheduling decisions [18]. First, even if an ASAP task has a later deadline than that of an ALAP task, the ASAP task should be executed before the ALAP task if it is possible to do so without causing any deadline miss; Second, the execution of ALAP tasks should be delayed as much as possible given that it does not cause any deadline miss for both current and future tasks.

Given these two principles, at any scheduling event (such as the arrival or completion of a task, or a timer interrupt), the basic steps of the POED scheduler can be summarized as follows. For cases where the ready task with the highest priority (i.e., earliest deadline) has ASAP preference, POED will execute the task normally as in EDF. However, in case an ALAP task has the earliest deadline, POED will focus on a *look-ahead interval* from the invocation time to the earliest deadline of an ASAP task. All (current and future arrival) tasks within this interval will be considered to see whether it is safe to delay the ALAP task's execution and if yes, for how long can it be delayed. We have shown that POED can guarantee to meet all tasks' deadlines when scheduling them according to their preferences. In particular, we have the following theorem regarding to the schedulability of a task set under POED. Interested readers can refer to [18] for the detailed analysis.

Theorem 1 (POED Schedulability [18]). *For a set Γ of periodic tasks with either ASAP or ALAP preferences, no task will miss its deadline under POED if there is $U(\Gamma) \leq 1$.*

Therefore, with the POED scheduler, the main tasks (i.e., T_1 and T_2) in the above example will have ASAP preference while the backup tasks (i.e., B_1 and B_2) have ALAP preference on their respective processors. Moreover, with the scaled frequency for the main tasks being 0.25, the *inflated* system utilization is exactly 1 on both processors. Hence, from Theorem 1, the mixed sets of main and backup tasks on both processors can be successfully scheduled under POED, which results in the schedule as shown in Figure 2b.

Algorithm 3 : Major steps of POED-based EEFT schemes

1: **Input:** task sets Γ and Γ^B; number of processors m;
2: **Step 1:** Allocate main tasks in Γ to m processors;
3: Suppose the WFD partition is $\Pi = \{\Gamma_1, \ldots, \Gamma_m\}$;
4: **Step 2:** Allocate backup tasks in Γ^B to all processors;
5: Suppose backup partition is $\Pi^B = \{\Gamma_1^B, \ldots, \Gamma_m^B\}$;
6: **Step 3:** Calculate scaled frequencies for main tasks;
7: **for** $(i : 1 \rightarrow m)$ **do**
8: $f_i = \min\{F_x | F_x \geq \frac{U(\Gamma_i)}{1-U(\Gamma_i^B)}, x = 1, \ldots L\}$;
9: Assign f_i to the main tasks in Γ_i;
10: Assign $f^{max} = F_L$ to the backup tasks in Γ_i^B;
11: **Step 4:** Execute tasks on each processor under POED;
12: **for** $(i : 1 \rightarrow m)$ **do**
13: Assign ASAP preference to main tasks in Γ_i;
14: Assign ALAP preference to backup tasks in Γ_i^B;
15: Execute Γ_i and Γ_i^B on $\mathcal{P}_i$ under POED;

5.2 POED-Based EEFT Schemes

From the above example, we can see that, significant energy savings can be obtained when a mixed set of main and backup tasks are allocated to each processor and scheduled under POED. The reasons come from two aspects: First, the slack time on all processors can be efficiently exploited by their main tasks with the DVFS technique. Second, with the POED scheduler, most executions of backup tasks can be effectively cancelled at runtime as such executions are delayed as much as possible while the corresponding main tasks are executed (on another processor) at their earliest times. By generalizing these ideas, the major steps of the POED-based energy-efficient fault-tolerance (EEFT) schemes for multiprocessor systems can be summarized in Algorithm 3.

The first step is to allocate main tasks in Γ (line 2). Without the need to dedicate processors for backup tasks, all processors in the system are accessible to the main tasks. Again, we assume that the WFD heuristic is adopted to balance the workload of main tasks among the processors (line 3).

After that, the second step is to allocate the backup tasks in Γ^B to all processors (lines 4 and 5). Recall that, to tolerate a single permanent fault, a main task T_i and its backup task B_i have to be allocated to different processors [32]. Following this principle, we consider in this work two approaches when allocating backup tasks.

Cyclic Backup Allocation: First, considering that the WFD partition obtained in the first step has relatively balanced workload of main tasks among the processors, a simple approach is the *Cyclic Allocation* of the backup tasks. That is, for the main tasks allocated to processor $\mathcal{P}_i$, the corresponding backup tasks will be mapped to the next neighbor processor $\mathcal{P}_{i+1}$ and so on $(i = 1, \ldots, m-1)$. For the main tasks on the last processor $\mathcal{P}_m$, their backup tasks are allocated to the first processor $\mathcal{P}_1$, forming a cyclic chain allocation of backup tasks (and the scheme is denoted as *POED-Cyclic*).

The cyclic allocation is easy to implement and can simplify the messages among processors at runtime when no permanent fault occurs. Here, the backup task of a main task can always be found on its next neighbor processor and vice versa. However, as discussed later, once a processor fails, the recovery steps can be quite complicated to re-establish such a cyclic allocation of backup tasks, which may require all tasks to be re-mapped among the remaining processors and have a rather long recovery window.

Mixed Backup Allocation: To avoid such cyclic dependency between processors, the second approach is to *scatter* backup tasks among all processors. Specifically, by considering one processor $\mathcal{P}_i$ $(i = 1, \ldots, m)$ at a time, the corresponding backup tasks of its main tasks are allocated to all other processors. Again, for the purpose of load-balancing, the WFD mapping heuristic is adopted. At the end, each processor will be allocated a completely mixed set of main and backup tasks and thus the scheme is denoted as *POED-Mix*.

After backup tasks are allocated, each processor $\mathcal{P}_i$ will have a subset Γ_i of main tasks and a subset Γ_i^B of backup tasks. Suppose that, for every processor, its allocated main and backup tasks are schedulable under POED. That is, there are $U(\Gamma_i) + U(\Gamma_i^B) \leq 1$ $(i = 1, \ldots, m)$. As the third step, the spare capacity (i.e., *static slack*) in the amount of $(1 - U(\Gamma_i) - U(\Gamma_i^B))$ on each processor $\mathcal{P}_i$ is exploited and the scaled frequency for the main tasks on that processor is calculated accordingly (lines 7 and 8). Then, the scaled frequency and the maximum frequency are assigned to the main and backup tasks, respectively (lines 9 and 10).

As mentioned previously, to cancel as much execution of backup tasks as possible at runtime, they should be delayed to the maximum extent and are given the ALAP preference while the main tasks have the ASAP preference on each processor (lines 13 and 14). Moreover, the *inflated* system utilization on each processor, which takes the scaled frequency of main tasks into consideration, is ensured to be no more than 1. Therefore, after frequency assignment for the (main and backup) tasks, they are guaranteed to be schedulable on each processor under POED (from Theorem 1). Hence, the last step is to execute the tasks on each processor under POED, which is actually the online phase of the POED-based schemes (line 15).

The same as in the Standby-Sparing scheme, once a main task successfully completes its execution, it will notify the processor that has its backup task and cancel its execution. As the evaluation results shown in Section 7, compared to that of the SS-based schemes, the POED-based schemes are more effective to cancel the execution of backup tasks due to their delayed execution as well as early execution of main task under the POED scheduler. Thus, better energy savings can be obtained under the POED-based schemes.

With backup tasks being forced to run at the maximum frequency, the system reliability with respect to transient faults can be preserved (in the absence of permanent faults). Moreover, since both the POED-Cyclic and POED-Mix schemes schedule any main task and its backup task on different processors, it guarantees to tolerate a single permanent fault on any processor at runtime. The recovery strategies for the POED-based schemes to handle additional permanent faults are further discussed in the next section.

6 RUNTIME MANAGEMENT

As the two major runtime issues, in what follows, we briefly address the *online power management* and *recovery strategies* for the proposed energy-efficient fault-tolerance schemes.

6.1 Online Power Management with Wrapper-Tasks

In both SS-based and POED-based schemes, the scaled frequencies for main tasks are obtained with the assumption that all tasks take their WCETs. However, it is well-known that real-time tasks normally take only a small fraction of their WCETs at runtime [16]. Hence, significant amount of dynamic slack can be expected, which should be exploited to further scale down main tasks for more energy savings.

Many online techniques have been studied for real-time tasks to exploit such dynamic slack at runtime [34], [55]. In this work, we study a *generic* online power management scheme, which can be applied to both the SS-based and POED-based EEFT schemes, based on the *wrapper-task* technique [47]. Essentially, a wrapper-task represents a piece of dynamic slack with two parameters (c, d), where the size c denotes the amount of slack and the deadline d equals to that of the task giving rise to this slack. At runtime, wrapper-tasks are kept in a separate wrapper-task queue with the increasing order of their deadlines. Under the earliest-deadline based scheduling, a task can safely reclaim any slack that has a deadline being no later than that of the task [47].

Considering the partitioned scheduling adopted in both the SS-based and POED-based schemes, each processor will manage its own slack time (i.e., wrapper-tasks) independently. Specifically, since backup tasks are executed at the maximum frequency on the secondary processors according to the offline generated EDL schedules in the SS-based schemes, only primary processors need to manage dynamic slack with the wrapper-task technique and, if possible, to further scale down the execution of their main tasks at runtime.

For the POED-based schemes, since each processor has a mixed set of main and backup tasks, all processors need to manage dynamic slack, which may come from the early completion of main tasks or cancellation of the backup tasks. However, on each processor, only the main tasks may reclaim (and actually utilize) the dynamic slack for further scaled execution. Un-used slack time will compete for the processor with other active tasks based on their priorities (i.e., deadlines). When the slack (i.e., wrapper-task) has the earliest deadline, it can either be *pushed forward* (i.e., lent to an active main task and returned with a later deadline) or consumed to idle the processor and further delay the execution of backup tasks. Detailed steps for such slack management with wrapper-tasks and its application under the POED scheduler can be found in [18], [47], which are omitted due to space limitation.

6.2 Recovery Strategies: Multiple Permanent Faults

By scheduling main tasks and their backup tasks on different processors, both the SS-based and POED-based schemes can tolerate a single permanent fault. Once a processor fails due to permanent faults and is detected, recovery strategies are needed to re-configure the system for it to tolerate additional permanent faults. The duration of recovery operations is referred to as *recovery window*, which denotes the time interval from the detection of a faulty processor to the time instance after which an additional permanent fault can be tolerated [32]. Here, to preserve system reliability with respect to transient faults, the main tasks whose backup tasks are on the faulty processor are assumed to run at the maximum frequency during the recovery process. Hence, in general, it is desired to have a smaller recovery window.

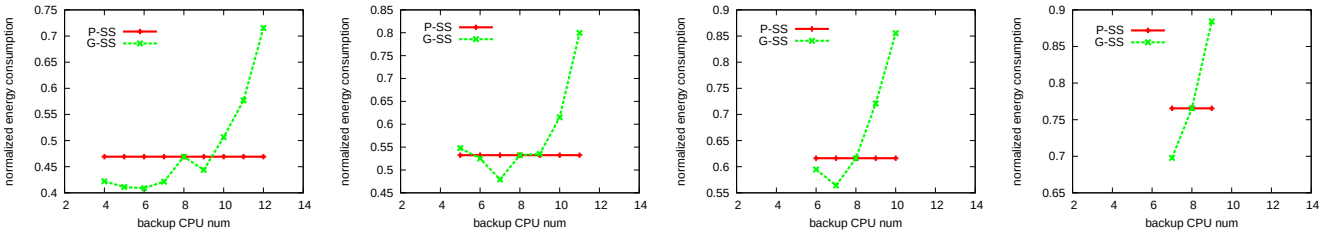
Considering the periodicity of the tasks, a simple recovery strategy that can be applied to all the proposed EEFT schemes would be to re-map the (main and backup) tasks to the remaining processors at the beginning of the next LCM of all tasks in Γ (denoted as *Global-LCM*). Clearly, such a simple strategy can have an extremely large recovery window (with the size of Global-LCM). Instead of re-allocating the affected tasks on the faulty processor *all at once* at the Global-LCM, we may consider the LCM of the tasks on a particular processor (pair), which is denoted as *Local-LCM*, and gradually migrate the affected tasks to other processors (or pairs) *one at a time*.

Suppose that a subset of affected (main and/or backup) tasks can be feasibly re-mapped to a working processor (pair) that has its existing set of (main and/or backup) tasks. At the time of the *new* Local-LCM (which considers both existing and to-be-mapped affected tasks), it is safe to re-calculate the scaled frequency for the main tasks or re-generate the EDL schedule for the backup tasks on that processor (pair). Therefore, based on such a Local-LCM principle, recovery strategies with smaller recovery windows can be devised by considering the specific features of different EEFT schemes.

Note that, Paired-SS can only deploy an *even* number of processors in a system. Therefore, if m is an odd number initially and there is one leftover processor, it may replace the faulty processor once it obtains the correct states of all tasks from another processor in the pair, which can lead to a much smaller recovery window. Another special case is POED-Cyclic, extra steps are needed to re-map the tasks on all remaining processors to obtain the balanced workload and re-establish the chain of backup tasks on these processors. The detailed discussions on the recovery strategies of the proposed EEFT schemes can be found in Appendix A.

7 EVALUATIONS AND DISCUSSIONS

In this section, we evaluate the performance of the proposed SS-based and POED-based EEFT schemes for multiprocessor real-time systems through extensive simulations. For such purposes, we developed a discrete event simulator using C++. From our previous studies [22], [20], we know that, in addition to the guarantee of tolerating a single permanent fault, the Standby-Sparing and the POED-based schemes can preserve the original system reliability with respect to transient faults by enforcing backup tasks run at the maximum frequency. Since the schemes studied in this paper follow the same design principle for fault tolerance, the reliability goals (in terms of tolerating both permanent and transient faults) can be ensured as well. Therefore, in what follows, we focus on evaluating the energy efficiency of the proposed schemes.



a. System utilization $U = 3.0$; b. System utilization $U = 4.0$; c. System utilization $U = 5.0$; d. System utilization $U = 6.0$;

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

Considering the fact that most modern processors have a few frequency levels [1], [9], we assume that there are seven frequency levels, which are normalized as $\{0.4, 0.5, 0.6, 0, 7, 0.8, 0.9, 1.0\}$ in the evaluations. Moreover, for the parameters in the power model, we assume that $P_{ind} = 0.01$, $C_{ef} = 1$ and $k = 3$, where similar parameters have been used in previous studies [22], [55]. Moreover, we consider a system with up to 16 processors.

The utilizations of tasks are generated according to the *UUniFast* scheme proposed in [7], where the average task utilization is set as $u^{ave} = 0.1$ and $u^{ave} = 0.05$, respectively. For each task set, we generate enough number of tasks so that the system utilization reaches a given target value. That is, for a given system utilization U, the average number of tasks in a set will be $\frac{U}{u^{ave}}$. The periods of tasks are uniformly distributed in the range of $[10, 100]$ and the WCET of a task is set according to its utilization and period. Each data point in the figures corresponds to the average result of 100 task sets.

7.1 Optimal Configuration for G-SS vs. P-SS

First, we illustrate the variations in energy consumption under different primary and secondary processor configurations in the G-SS scheme for a 16-processor system and compare them against that of the P-SS scheme. Here, we assume that all tasks run at their statically assigned frequencies and take their WCETs at run-time. Moreover, it is assumed that no fault occurs during the execution of tasks and backup (main) copies of tasks are cancelled under both schemes once their corresponding main (backup) copies complete successfully[3]. We show the normalized energy consumption, where the one under the *basic P-SS* scheme with both primary and secondary processors running at the maximum frequency is used as the baseline.

For a 16-processor system, the upper-bound of the total main task system utilization schedulable under the proposed schemes would be 8 (since a similar processor capacity should be reserved for backup tasks). For the cases of system utilization $U = 3.0, 4.0, 5.0$ and 6.0, Figure 3 shows the results for the G-SS scheme with varying numbers (Y) of secondary processors as well as that of the P-SS scheme for comparison. Here, the average task utilization is set as $u^{ave} = 0.1$.

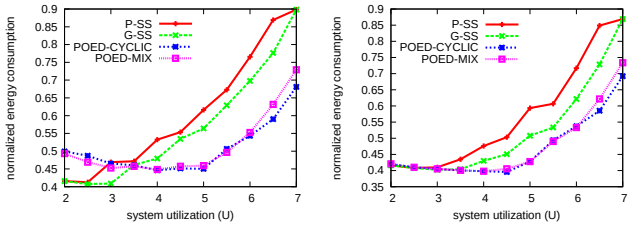
3. Note that, due to independent scheduling of tasks' main and backup copies under EDF and EDL, respectively, it is possible for a task's backup copy finishes earlier than its main copy in the SS scheme [22].

Not surprisingly, for different processor configurations (i.e., as the number of secondary processors varies) in the G-SS scheme, the system energy efficiency can have rather large differences (from 30% to 45%). As in our example in Section 4.1, for a given system utilization, the optimal processor configuration that can lead to the best system energy efficiency normally has more primary processors (i.e., smaller values of Y). On the other hand, since the backup copies of tasks have to be executed at the maximum frequency for reliability preservation [22], the spare capacity on secondary processors is normally wasted, which leads to inferior performance for G-SS when more processors are used as secondaries.

From the results, we can also see that, with a judicious selection of the processor configurations (i.e., the values of X and Y), G-SS can outperform P-SS with up to 7% more energy savings. In the remaining evaluations, for any given task set, we assume that the G-SS scheme always adopts the optimal processor configuration for better energy efficiency.

7.2 Performance with Offline Scaled Frequencies

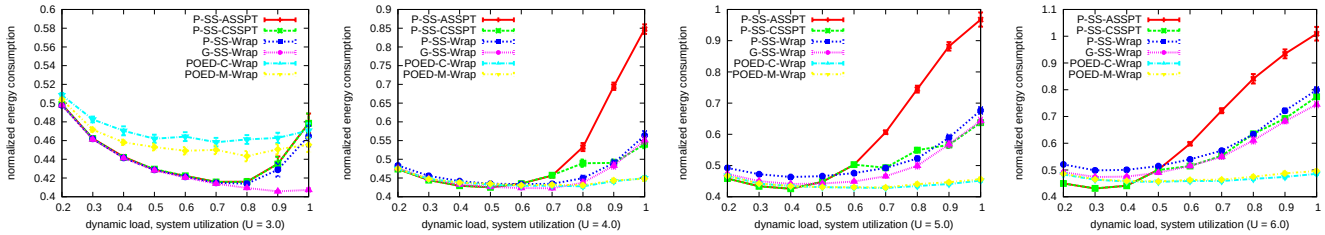
Without considering the online slack reclamation (which will be evaluated next), Figure 4 shows the performance of both SS-based and POED-based schemes with offline determined scaled frequency under varying system utilizations. Again, tasks are assumed to take their WCETs and no fault occurs during the execution. First, the results show that, compared to the P-SS scheme, the G-SS scheme with optimal processor configuration can always perform better in terms of obtaining more energy savings under different system utilizations.



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

Fig. 4: Performance of the SS-based and POED-based schemes with offline determined scaled frequencies for main tasks.

However, the performance difference between P-SS and G-SS diminishes at very low or high system utilizations. The reason is that, at low system utilizations (i.e., $U \leq 2.5$),



a. system utilization $U = 3.0$; b. system utilization $U = 4.0$; c. system utilization $U = 5.0$; d. system utilization $U = 6.0$;

Fig. 5: Performance of both SS-based and POED-based schemes with online techniques under different system loads.

all main tasks can be executed at 0.4 (the lowest available frequency) while most backup tasks can be cancelled under both schemes. At high system utilizations (e.g., $U = 7.0$), there is only one feasible processor configuration (i.e., $X = 8$) for the G-SS scheme, which makes G-SS to act exactly the same as P-SS due to the same WFD heuristic when partitioning main and backup copies of tasks.

For the POED-based schemes, POED-Cyclic and POED-Mix have very close performance on energy savings even though they have different backup partitions. However, in most cases, POED-based schemes can outperform P-SS and G-SS with up to 20% more energy savings. The reason is that, with mixed allocation of main and backup tasks on the processors, POED-based schemes can better utilize the available slack to slow down main tasks and reduce the overlapped execution with their corresponding backup tasks.

Interestingly, for the case of $u^{ave} = 0.1$ (i.e., relatively large tasks), Figure 4a shows that POED-based scheme may perform worse compared to that of the SS-based schemes when system utilization is very low ($U \leq 3.0$). This comes from the fewer number of available tasks, which cause unbalanced partitions of tasks among the processors under the POED-based schemes. For smaller tasks (i.e., $u^{ave} = 0.05$) where there are more tasks for the same system utilization, Figure 4b shows that the POED-based schemes perform no worse than the SS-based schemes.

7.3 Performance of Online Schemes

In this section, by varying the ratio of average over worst case execution times of tasks, we further evaluate the performance of the SS-based and POED-based schemes with different online techniques. For comparison, we also implemented both ASSPT and CSSPT techniques [22] for the P-SS scheme, which are denoted as "P-SS-ASSPT" and "P-SS-CSSPT", respectively. For the online scheme based on wrapper-tasks, it can be applied to the primary processors under both P-SS and G-SS, which are denoted as "P-SS-Wrap" and "G-SS-Wrap", respectively. The POED-based schemes enhanced with the online wrapper-tasks based technique are further denoted as "POED-C-Wrap" and "POED-M-Wrap", respectively.

Here, we set $u^{ave} = 0.1$. To emulate the dynamic execution behaviors of tasks, we use a system wide average-to-worst case execution time ratio α. For each task T_i, its average-to-worst case execution time ratio α_i is generated randomly around α. Then, at run-time, the actual execution time for each instance of task T_i is randomly generated around $\alpha_i \cdot c_i$, where c_i is task T_i's WCET. Essentially, α indicates the amount of dynamic slack that will be available at runtime where lower values indicate more slack.

Figures 5 show the performance of the schemes with varying α (average-to-worst case execution times of tasks) under different system utilizations (i.e., $U = 3.0, 4.0, 5.0$ and 6.0, respectively). Again, when the system utilization is low (i.e., $U = 3.0$), the main tasks can be executed at the lowest frequency of 0.4 and most backup tasks are cancelled, which leads to very close (within 6% difference) normalized energy consumptions for P-SS and G-SS with different online techniques.

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

When the system utilization gets higher (i.e., $U = 4.0, 5.0$ and $U = 6.0$), we can see that the ASSPT technique can cause dramatical performance degradation for P-SS as the dynamic load of tasks increases (i.e., with higher values of α). The results are in line with what have been reported in [22]. The reason comes from the aggressive slack usage under the ASSPT technique, which executes the main tasks at very low frequency at the beginning of the schedule. Such scaled executions force remaining main tasks to run at much higher frequencies and cause more overlapped executions with their backup tasks on the secondary processors.

To address the above mentioned problem, based on the static and dynamic loads of tasks, the CSSPT scheme statically determines a lower bound for the scaled frequency for executing tasks' main copies when reclaiming slack at run-time [22]. With such a scaled frequency bound, CSSPT can effectively prevent the aggressive usage of slack time in the early part of the schedule. Therefore, when compared to ASSPT, P-SS performs much better with the CSSPT online technique, especially for tasks with higher dynamic loads.

For the wrapper-task based online technique, we can see that its performance is pretty stable under different dynamic loads of tasks. Although it performs (slightly) worse than that of ASSPT and CSSPT for the P-SS scheme at low dynamic

loads (i.e., $\alpha \leq 0.5$), its performance is very close to that of CSSPT at higher dynamic loads of tasks. However, different from CSSPT, the wrapper-task based online technique does not requires the pre-knowledge of tasks' average-case workloads.

Moreover, as a generic online technique, wrapper-tasks can also be applied to the primary processors in the G-SS scheme, which is shown to have a stable performance as well. Although the performance gain of applying the wrapper-task technique on G-SS is rather limited (within 5%) when compared to that of P-SS, we can see that G-SS-Wrap always performs better than that of P-SS-CSSPT at higher dynamic loads of tasks.

For the POED-based schemes, when the system utilization is low, the main tasks can be executed at the lowest frequency 0.4 and most backup tasks can be cancelled. However, the same as before, due to the unbalanced workload among the processors at very low system utilization (i.e., $U = 3.0$), the POED-based schemes can have slightly inferior performance compare with SS-based schemes.

Moreover, as system utilization increases (i.e., for the cases of $U = 4.0, 5.0$ and $U = 6.0$), both POED-Cyclic and POED-Mix with wrapper-task based online technique can achieve much better and more stable energy savings comparing with the SS-based schemes. Again, this comes from the fact that with more workload in the system, both POED-based schemes can utilize the available system resource (CPU time) more efficiently. Specifically, by mixing the main and backup tasks on all processors, with the wrapper-task based online technique, all available (static and dynamic) slack time can be exploited to slow down the execution of main tasks and/or delay the execution of backup tasks, which results in much reduced overlapped executions.

8 Conclusions

In this paper, with the objectives of tolerating a single permanent fault while preserving system reliability with respect to transient faults, we study energy-efficient fault-tolerance (EEFT) schemes for periodic tasks running on multiprocessor systems. Specifically, based on the idea of Standby-Sparing (SS) technique, we first propose the *Paired-SS* scheme, where processors are organized as groups of two (i.e., pairs) with the traditional Standby-Sparing being applied directly to each processor pair after partitioning tasks. Then, we propose a *Generalized-SS* technique that partitions processors into two groups, which are for *primary* and *secondary* processors, respectively. The *main* and *backup* tasks are executed on the primary and secondary processor groups under the *partitioned-EDF* and *partitioned-EDL* scheduling, respectively, to reduce their overlapped executions and thus to obtain more energy savings. Moreover, based on the preference-oriented earliest deadline (POED) scheduling algorithm, we further study two POED-based schemes (i.e., *POED-Cyclic* and *POED-Mix*), which allocates the main and backup tasks in a mixed manner on all processors to better utilize the available slack time. An online power management with wrapper-tasks is also studied, which can be applied to both SS-based and POED-based schemes for more energy savings. The recovery strategies for the proposed EEFT schemes are also addressed to handle multiple permanent faults.

The proposed EEFT schemes are evaluated through extensive simulations. The results show that, for systems with a given number of processors, there normally exists an *optimal* configuration of primary and secondary processors for the Generalized-SS scheme, which can have better energy savings when compared to that of the Paired-SS scheme. Moreover, POED-based schemes can outperform SS-based schemes regarding to energy savings, especially for systems with modest to high system loads.

References

[1] AMD. Amd opteron quad-core processors; http://www.amd.com/us/products/embedded/processors/, 2009.

[2] S. Aminzadeh and A. Ejlali. A comparative study of system-level energy management methods for fault-tolerant hard real-time systems. *IEEE Trans. Comput.*, 60(9):1288–1299, Sep. 2011.

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

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

[5] A. A. Bertossi, L. V. Mancini, and A. Menapace. Scheduling hard-real-time tasks with backup phasing delay. In *Proc. of the IEEE Int'l Symp. on Distributed Simulation and Real-Time Applications*, 2006.

[6] A. A. Bertossi, L. V. Mancini, and F. Rossini. Fault-tolerant rate-monotonic first-fit scheduling in hard-real-time systems. *IEEE Trans. on Parallel and Distributed Systems*, 10(9):934 –945, 1999.

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

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

[9] Intel Corp. Intel embedded quad-core xeon; http://www.intel.com/products/embedded/processors.htm, 2009.

[10] F. Dabiri, N. Amini, M. Rofouei, and M. Sarrafzadeh. Reliability-aware optimization for dvs-enabled real-time embedded systems. In *Proc. of the Int'l Symp. on Quality Electronic Design*, pages 780–783, 2008.

[11] V. Degalahal, L. Li, V. Narayanan, M. Kandemir, and M. J. Irwin. Soft errors issues in low-power caches. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 13(10):1157–1166, Oct. 2005.

[12] A. Ejlali, B. M. Al-Hashimi, and P. Eles. A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems. In *Proc. of the IEEE/ACM Int'l Conf. on Hardware/Software Codesign and System Synthesis*, pages 193–202, 2009.

[13] A. Ejlali, M. T. Schmitz, B. M. Al-Hashimi, S. G. Miremadi, and P. Rosinger. Energy efficient seu-tolerance in dvs-enabled real-time systems through information redundancy. In *Proc. of the Int'l Symp. on Low Power and Electronics and Design*, pages 281–286, 2005.

[14] E. (Mootaz) Elnozahy, R. Melhem, and D. Mossé. Energy-efficient duplex and tmr real-time systems. In *Proc. of The 23^{rd} IEEE Real-Time Systems Symposium (RTSS)*, pages 256–265, 2002.

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

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

[17] S. Ghosh, R. Melhem, and D. Mossé. Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems. *Parallel and Distributed Systems, IEEE Transactions on*, 8(3):272 –284, March 1997.

[18] Y. Guo, H. Su, D. Zhu, and H. Aydin. Preference-oriented scheduling framework for periodic real-time tasks. Technical Report CS-TR-2013-007, Dept. of Computer Science, Univ. of Texas at San Antonio, 2013.

[19] Y. Guo, D. Zhu, and H. Aydin. Reliability-aware power management for parallel real-time applications with precedence constraints. In *Proc. of the Int'l Green Computing Conference (IGCC)*, pages 1 –8, July 2011.

[20] Y. Guo, D. Zhu, and H. Aydin. Efficient power management schemes for dual-processor fault-tolerant systems. In *Proc. of the First Workshop on Highly-Reliable Power-Efficient Embedded Designs (HARSH), in conjunction with HPCA*, Feb. 2013.

[21] Y. Guo, D. Zhu, and H. Aydin. Generalized standby-sparing techniques for energy-efficient fault tolerance in multiprocessor real-time systems. In *Proc. of the IEEE Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Aug. 2013.

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

[23] M.A. Haque, H. Aydin, and D. Zhu. Energy management of standby-sparing systems for fixed-priority real-time workloads. In *Proc. Of the Second Int'l Green Computing Conference (IGCC)*, Jun. 2013.

[24] http://public.itrs.net. International technology roadmap for semiconductors. 2008. S. R. Corporation.

[25] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. In *Proc. of the 14th Annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 37–46, 2003.

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

[27] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Design optimization of time- and cost-constrained fault-tolerant distributed embedded systems. In *Proc. of Design, Automation and Test in Europe*, pages 864–869, 2005.

[28] N.-S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J.-S. Hu, M.-J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, Dec. 2003.

[29] Z. Li, L. Wang, S. Li, S. Ren, and G. Quan. Reliability guaranteed energy-aware frame-based task set execution strategy for hard real-time systems. *Journal of Software and System*, 2013.

[30] R. Melhem, D. Mossé, and E. (Mootaz) Elnozahy. The interplay of power management and fault recovery in real-time systems. *IEEE Trans. on Computers*, 53(2):217–231, 2004.

[31] P. Pop, K.H. Poulsen, V. Izosimov, and P. Eles. Scheduling and voltage scaling for energy/reliability trade-offs in fault-tolerant time-triggered embedded systems. In *Proc. of the IEEE/ACM Int'l Conf. on Hardware/software codesign and System Synthesis*, 2007.

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

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

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

[35] R. Sridharan, N. Gupta, and R. Mahapatra. Feedback-controlled reliability-aware power management for real-time embedded systems. In *Proc. of the Design Automation Conference*, pages 185–190, 2008.

[36] 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 IEEE Real-Time Systems Symposium*, 2011.

[37] O. S. Unsal, I. Koren, and C. M. Krishna. Towards energy-aware software-based fault tolerance in real-time systems. In *Proc. of the Int'l Symp. on Low Power Electronics and Design*, pages 124–129, 2002.

[38] T. Wei, P. Mishra, K. Wu, and H. Liang. Online task-scheduling for fault-tolerant low-energy real-time systems. In *Proc. of IEEE/ACM Int'l Conf. on Computer-Aided Design*, pages 522–527, 2006.

[39] T. Wei, P. Mishra, K. Wu, and H. Liang. Fixed-priority allocation and scheduling for energy-efficient fault tolerance in hard real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 19:1511–1526, 2008.

[40] Y. Zhang and K. Chakrabarty. Energy-aware adaptive checkpointing in embedded real-time systems. In *Proc. of Design, Automation and Test in Europe (DATE)*, pages 918 – 923, 2003.

[41] Y. Zhang and K. Chakrabarty. Task feasibility analysis and dynamic voltage scaling in fault-tolerant real-time embedded systems. In *Proc. of Design, Automation and Test in Europe Conference(DATE)*, pages 1170 – 1175, 2004.

[42] Y. Zhang, K. Chakrabarty, and V. Swaminathan. Energy-aware fault tolerance in fixed-priority real-time embedded systems. In *Proc. of the IEEE/ACM int'l Conf. on Computer-Aided Design (ICCAD)*, 2003.

[43] B. Zhao, H. Aydin, and D. Zhu. Generalized reliability-oriented energy management for real-time embedded applications. In *Proc. of the 48th Design Automation Conference (DAC)*, Jun. 2011.

[44] B. Zhao, H. Aydin, and D. Zhu. Energy management under general task-level reliability constraints. In *Proc. of the 18^{th} IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 285–294, Apr. 2012.

[45] B. Zhao, H. Aydin, and D. Zhu. Shared recovery for energy efficiency and reliability enhancements in real-time applications with precedence constraints. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 18(2-23), 2013.

[46] D. Zhu. Reliability-aware dynamic energy management in dependable embedded real-time systems. *ACM Trans. on Embedded Computing Systems*, 10(2):26.1–26.27, 2010.

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

[48] D. Zhu, H. Aydin, and J.-J. Chen. Optimistic reliability aware energy management for real-time tasks with probabilistic execution times. In *Proc. of the IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2008.

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

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

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

[52] D. Zhu, D. Mossé, and R. Melhem. Energy efficient redundant configurations for real-time parallel reliable servers. *Journal of Real-Time Systems*, 41(3):195–221, Apr. 2009.

[53] 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.

[54] D. Zhu, X. Qi, and H. Aydin. Energy management for periodic real-time tasks with variable assurance requirements. In *Proc. of the IEEE Int'l Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

[55] S. Zhuravlev, J.-C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of energy-cognizant scheduling techniques. *IEEE Trans. Parallel Distrib. Syst.*, 24(7):1447–1464, 2013.

Yifeng Guo is currently a PhD Candidate in the Department of Computer Science at the University of Texas at San Antonio. His research interests include real-time systems, power management 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 is in the general area of real-time systems. 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.

Hakan Aydin received the PhD degree in computer science from the University of Pittsburgh in 2001. He is currently an associate professor in the Computer Science Department at George Mason University. He was a recipient of the US National Science Foundation (NSF) Faculty Early Career Development (CAREER) Award in 2006. His research interests include real-time systems, low-power computing, and fault tolerance. He is a member of the IEEE.

Laurence T. Yang is with the Department of Computer Science, St. Francis Xavier University, Antigonish, NS, Canada and holds adjunct position at Huazhong University of Science and Technology, China. His research interests include high performance computing and networking, embedded systems, ubiquitous/pervasive computing, and intelligence. His research is supported by National Sciences and Engineering Research Council, Canada and Canada Foundation for Innovation.

APPENDIX
A. RECOVERY STRATEGIES

In previous chapters, with the goals of preserving system reliability with respect to transient faults and tolerating *one* permanent fault, we introduced SS and POED based schemes for fault-tolerant multiprocessor systems, with emphasis on energy efficiency. The detailed experimental results have illustrated the energy savings. Although having a fault during the execution of an application has rather low probability, the strategies on how the system should react when faults (transient and/or permanent) happen on those systems have not been discussed yet. Note that efficient recovery strategies are necessary to recover the system from fault(s) and keep functional with continuous tolerance to future faults. The problem will be addressed in this chapter.

.1 Recover from Transient Faults

From previous studies [12], [22], [20], we know that both SS and POED based schemes can preserve the original system reliability with respect to transient faults by enforcing tasks' backup copies to be executed at the maximum frequency. Since the proposed schemes follow the same design principle for fault tolerance, the reliability goal for tolerating transient faults can still be ensured. When a transient fault happens on one copy (main or backup) of a task, the sanity (or consistency) check executed afterwards will detect the soft error. Therefore, the other copy of it gets fully executed to tolerate such a transient fault.

.2 Recover from Permanent Faults

To facilitate the discussion of recovery mechanisms for permanent faults, we have defined *Mean Time to Failure* (MTTF) and *Time to Second Failure* (TTSF) in Section 3. MTTF represents the frequency of the occurrence of permanent faults on the system, whereas TTSF is seen to be the recovery window after that the system is ready to tolerate another permanent fault. The design of the recovery mechanisms should have the following properties to achieve the satisfactory capability of tolerating more than one future permanent faults and maintaining energy efficiency:

- small TTSF for quick recovery
- low overhead with limited number of task migrations
- keep balanced load for energy efficiency after recovery

In order to recover from a permanent fault and provide the ability to tolerate the next one as soon as possible when there is no spare processor in the system, all tasks on the failed processor should be re-allocated to other functional ones. Note that it is always feasible to do task re-partition for the whole task set (main and backup) to the remaining functional processors at the beginning of the next LCM of the task set (we denote it *Global LCM* in this chapter) for all SS and POED based schemes, as long as no processor with workload exceeds its capacity (with utilization larger than 1). However, the problem is that the corresponding TTSF could be as long as the Global LCM, which might be too large. Therefore, we discuss below some improvement we could achieve for each scheme, respectively.

.2.1 P-SS

Since P-SS can only use an even number of processors in the system with its paired nature, the failure of one processor in a pair causes the system to abandon the other one after TTSF (although it can be treated as a spare once another processor failed in the future). Moreover, because of the static EDL scheme applied to the spare processor in each pair, it is prohibited to migrate tasks to other pairs between their corresponding LCMs. Therefore, the basic idea of this improvement is to migrate all tasks in the failed processor pair to other functional ones at their corresponding *Local LCMs*. Here, the Local LCM is defined as the LCM of all tasks' periods allocated to that pair/processor, which include the existing tasks and future migrated tasks. It can also be understood as the time point at which all tasks (existing and future migrated tasks) will arrive.

Algorithm 4 Basic Steps for the P-SS Recovery

1: **Input:** number of processors m, main tasks for each pair $\Psi_1^T, \ldots, \Psi_{\frac{m}{2}}^T$;
2: //processor P_k in pair i ($1 \leq i \leq \frac{m}{2}$) is detected to be failed at time t_0, and there is no extra spare processor;
3: **if** (P_k is a spare processor) **then**
4: corresponding primary processor execute in f_{max} from t_0;
5: **end if**
6: scheduler emulate the process of partition tasks in pair i (Ψ_i^T) to $\frac{m}{2} - 1$ functional pairs according to a given (e.g., WFD) heuristic, getting $\Psi_{i,1}^T, \ldots, \Psi_{i,j}^T, \ldots, \Psi_{i,\frac{m}{2}}^T$ ($1 \leq j \leq \frac{m}{2}$, $j \neq i$);
7: **for** ($(1 \leq j \leq \frac{m}{2})$ && $(j \neq i)$) **do**
8: get Local LCM (LCM_j) for pair j with tasks $\Psi_j^T \cup \Psi_{i,j}^T$;
9: $\Psi_{i,j}^T$ migrate to pair j at t_j, with $t_j - t_0 \leq LCM_j$;
10: **end for**

The main recovery steps for the P-SS scheme when a permanent fault happens are summarized in Algorithm 4. Suppose the main copies of tasks allocated to each pair is represented as $\Psi_1^T, \ldots, \Psi_{\frac{m}{2}}^T$, in which Ψ_i^T ($1 \leq i \leq \frac{m}{2}$) is allocated to pair i. If a permanent fault is detected at time t_0, the system will switch to the *emergency operation mode*, during which the system can't tolerate another permanent fault. Specifically, if the primary processor in pair i is detected to be failed, tasks in the spare processor should keep executing with no change. On the other hand, if the spare processor fails in pair i, all tasks in the primary processor should start executing at the maximum frequency f_{max} to keep preserving the system's original reliability with respect to transient faults (line 3 and 4). The functional processor in pair i should keep operating until all tasks in its pair get migrated to other functional ones.

Meanwhile, the scheduler should emulate the process of partition tasks in the failed pair (i.e., Ψ_i^T) to other $\frac{m}{2} - 1$ functional ones. Again, the WFD heuristic is applied to get balanced partition for both feasibility and energy savings (line 5). Moreover, for each functional processor pair (i.e., pair j, $1 \leq j \leq \frac{m}{2}$, $j \neq i$), an updated Local LCM is calculated

with all tasks' periods from its own local task set (Ψ_j^T) *and* the future migrated task set ($\Psi_{i,j}^T$) (line 7). Note that $\Psi_{i,j}^T$ is designated as a group of main tasks in pair i that will migrate to pair j. This updated Local LCM serves as the upper bound of the transition window, within which tasks in $\Psi_{i,j}^T$ can migrate to pair j safely (line 8) . Finally, the P-SS scheme (Section 4.2) can be re-applied to the corresponding pair after the migration. Once all tasks have been migrated from pair i, the system could return to the *normal operation mode* and have the ability to tolerate another permanent fault in the future. Note that for this recovery algorithm, the size of the recovery window is bounded by the maximum of all updated Local LCMs, which could be much smaller than the Global LCM in most cases, and the corresponding TSSF should be extremely reduced.

We show through an example in Figure 6 to further illustrate the idea of P-SS recovery. Suppose we have a multiprocessor system with 6 processors. During its execution, a permanent fault is detected at time t_0 on processor P_2. Under this circumstance, the system will switch to the emergency operation mode. If P_2 is a spare processor, P_1 needs to schedule all tasks in the maximum frequency from t_0. Moreover, $\Psi_{1,2}^T$ and $\Psi_{1,3}^T$ is generated using WFD heuristic for pair 2 and 3, respectively. And the corresponding Local LCMs (i.e., LCM_2 and LCM_3) are obtained based on each pair's local and future migrated tasks from the pair 1. The migration can only happen at the time point when all local and future migrated tasks arrive together (i.e., t_1 and t_2), at which the EDL algorithm can generate the static schedule for the spare processor in each functional pair. And the transition windows (i.e., $t_1 - t_0$ and $t_2 - t_0$) will be bounded by LCM_2 and LCM_3, respectively. The recovery step is finished at the instance when the last migration happens at t_2, and the system will switch back to the normal operation mode.

.2.2 G-SS

To reduce the size of the recovery window for quick recovery, the same idea of using Local LCMs for migration points can be applied to G-SS as well. However, since G-SS organizes the processors into primary and secondary groups with main and backup copies of tasks executing in the corresponding group separately, different situations need to be considered when either one of the primary or secondary processors failed.

The main steps for G-SS recovery when a permanent fault occurs are summarized in Algorithm 5. Note that the main (backup) copies of tasks allocated to X primary (Y secondary) processors are represented as $\Pi_M = \{\Psi_1^T, \cdots, \Psi_X^T\}$ ($\Pi_B = \{\Psi_1^B, \cdots, \Psi_Y^B\}$). Moreover, $\Psi_{k,j}^T$ ($\Psi_{k,j}^B$) is designated as a group of main (backup) tasks in processor P_k that will migrate to P_j.

Specifically, if a permanent fault is detected at time t_0, the system will switch to the *emergency operation mode*. If the fault happens on one of the X primary processors (i.e., P_k, $1 \leq k \leq X$), it won't affect the execution of the backup tasks in Y secondary processors. The scheduler should emulate the process of partition tasks in P_k to other $X - 1$ functional ones, and WFD heuristic is adopted to again balance workload among processors (line 4). Moreover, for each functional primary processor (i.e., P_j, $1 \leq j \leq X$, $j \neq k$), an updated Local LCM is calculated with all tasks' periods from its own local task set (Ψ_j^T) *and* the future migrated task set ($\Psi_{k,j}^T$) (line 6). Again, this updated Local LCM serves as the upper bound of the transition window, within which tasks in $\Psi_{k,j}^T$ can migrate to P_j (line 7). Finally, all tasks in P_j can be scheduled using EDF after the migration. Once all tasks have been migrated from the failed processor P_k, the system could return to the *normal operation mode* and have the ability to tolerate another permanent fault in the future.

The same steps can be applied to the recovery of the failure occurs in one of the Y spare processors (line 9 to 13). Except that when the failure is detected at time t_0, the corresponding main tasks dispersed in X primary processors with backup copies on the failed processor (i.e., P_k) should be executed in f_{max} to still preserve the system's original reliability with respect to transient faults. They can return to the scaled frequencies when their corresponding backup tasks (i.e., Ψ_k^B) migrate to other spare processors.

We show through examples on how G-SS recovery steps are applied when a primary or spare processor failed, respectively. As illustrated in Figure 7a, suppose we have a multiprocessor system with 6 processors, in which P_1, P_2, P_3 and P_4 are primary processors, while P_5 and P_6 are spares. Assume that a permanent fault is detected at time t_0 on P_1, the system will switch to the emergency operation mode. The scheduler generates partitions of Ψ_1^T for the rest of functional primary

Algorithm 5 Basic Steps for the G-SS Recovery

1: **Input:** task set Ψ, X and $Y = (m - X)$, with WFD partitions of Ψ: $\Pi_M = \{\Psi_1^T, \cdots, \Psi_X^T\}$ and $\Pi_B = \{\Psi_1^B, \cdots, \Psi_Y^B\}$;
2: //processor P_k is detected to be failed at time t_0;
3: **if** (P_k is a primary processor) **then**
4: scheduler emulate the process of partition tasks in P_k (Ψ_k^T) to $X - 1$ functional ones according to a given (e.g., WFD) heuristic, getting $\Psi_{k,1}^T, \ldots, \Psi_{k,j}^T, \ldots, \Psi_{k,X}^T$ ($1 \leq j \leq X$, $j \neq k$);
5: **for** (($1 \leq j \leq X$) && ($j \neq k$)) **do**
6: get Local LCM (LCM_j) for P_j with tasks $\Psi_j^T \cup \Psi_{k,j}^T$;
7: $\Psi_{k,j}^T$ migrate to P_j at t_j, with $t_j - t_0 \leq LCM_j$;
8: **end for**
9: **else**
10: main tasks with backup copies in P_k start executing in f_{max} from t_0;
11: scheduler emulate the process of partition tasks in P_k (Ψ_k^B) to $Y - 1$ functional ones according to a given (e.g., WFD) heuristic, getting $\Psi_{k,1}^B, \ldots, \Psi_{k,j}^B, \ldots, \Psi_{k,Y}^B$ ($1 \leq j \leq Y$, $j \neq k$);
12: **for** (($1 \leq j \leq Y$) && ($j \neq k$)) **do**
13: get Local LCM (LCM_j) for P_j with tasks $\Psi_j^B \cup \Psi_{k,j}^B$;
14: $\Psi_{k,j}^B$ migrate to P_j at t_j, with $t_j - t_0 \leq LCM_j$;
15: **end for**
16: **end if**

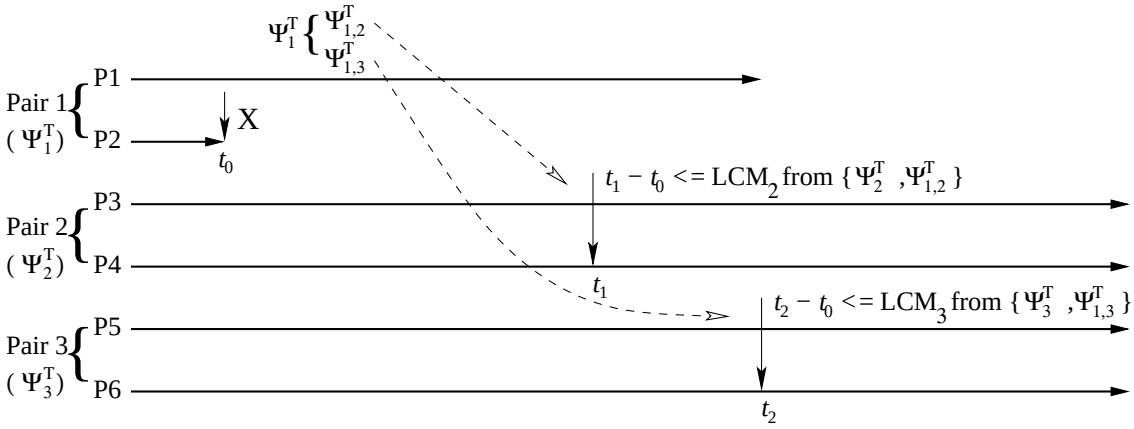
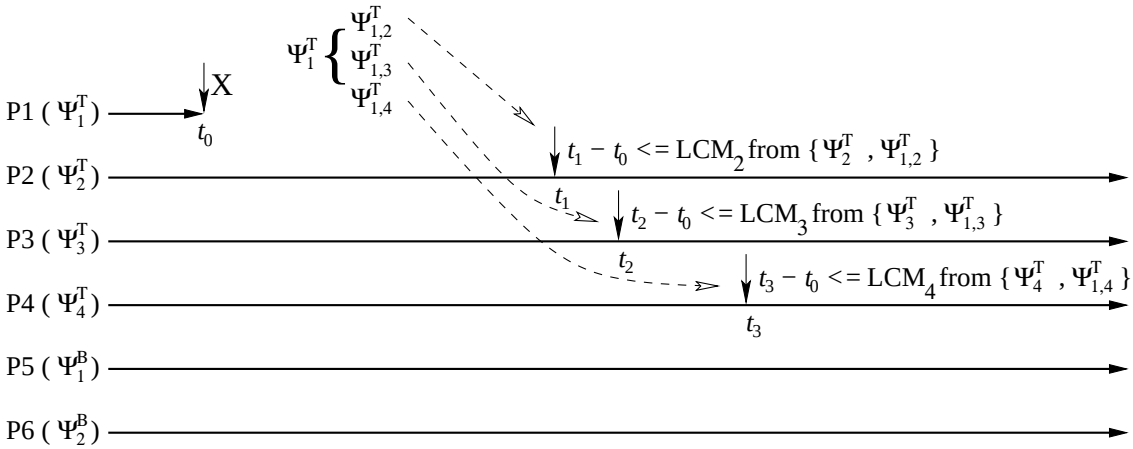
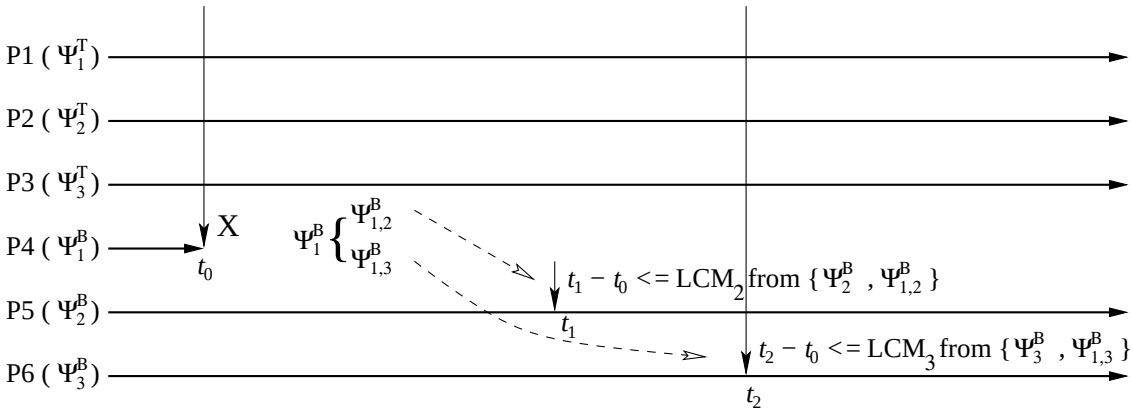


Fig. 6: Example of recovery steps for P-SS



a. a primary processor fails



b. a spare processor fails

Fig. 7: Example of recovery steps for G-SS

processors using WFD heuristic and gets $\Psi^T_{1,2}$, $\Psi^T_{1,3}$ and $\Psi^T_{1,4}$. In the meantime, the corresponding Local LCMs (i.e., LCM_2, LCM_3 and LCM_4) are obtained with their local tasks and future migrated tasks from P_1. Again, the migration can only happen at the time point when all local and future migrated tasks arrive together (i.e., t_1, t_2 and t_3). And the transition windows (i.e., $t_1 - t_0$, $t_2 - t_0$ and $t_3 - t_0$) are bounded by LCM_2, LCM_3 and LCM_4, respectively. In Figure 7a, after t_3, the system will finish the recovery and switch back to the normal operation mode.

Similarly, in Figure 7b, we still have a multiprocessor system with 6 processors, in which P_1, P_2 and P_3 are primary processors, while P_4, P_5 and P_6 are spares. In this case, a permanent fault is detected on P_4 at t_0. Therefore, the system will switch to the emergency operation mode and all main tasks with backup copies allocated in P_4 should start execute using f_{max}. The rest of the recovery steps are the same as before. However, when the system finishes the recovery and switches back to the normal operation mode at t_2, those main tasks with execution frequency switched to f_{max} during emergency operation mode could switch back to their original scaled frequencies for energy savings.

.2.3 POED-Cyclic

As discussed in Section 5, the allocation constraint from POED-Cyclic makes it difficult to improve the TTSF from Global LCM.

.2.4 POED-Mix

Algorithm 6 Basic Steps for the POED-Mix Recovery

1: **Input:** number of processors m, task set Ψ, with tasks' main and backup allocation: $\Pi_M = \{\Psi^T_1, \cdots, \Psi^T_m\}$ and $\Pi_B = \{\Psi^B_1, \cdots, \Psi^B_m\}$;
2: //processor P_k is detected to be failed at time t_0;
3: **for** $((1 \leq j \leq m)$ && $(j \neq k))$ **do**
4: main tasks in P_j with backups in P_k start executing in f_{max};
5: for main tasks in P_k, convert their backups in P_j ($\Psi^B_{j,k}$) to be main copies ($\Psi^T_{j,k}$), but still execute in f_{max} ;
6: generate the partition for $\Psi^B_{j,k}$ to all processors except for P_k and P_j, getting $\Psi^B_{j,k,l}$, $1 \leq l \leq m$, $l \neq k$, $l \neq j$;
7: for backup tasks in P_k which is for main copies in P_j ($\Psi^B_{k,j}$), generate the partition to all processors except for P_k and P_j, getting $\Psi^B_{k,j,u}$, $1 \leq u \leq m$, $u \neq k$, $u \neq j$;
8: **end for**
9: **for** $((1 \leq j \leq m)$ && $(j \neq k))$ **do**
10: //denote tasks that will migrate to P_j as Ψ^{mig}_j;
11: get Local LCM (LCM_j) for P_j with tasks $\Psi^T_j \cup \Psi^B_j \cup \Psi^{mig}_j$;
12: Ψ^{mig}_j migrate to P_j at t_j, with $t_j - t_0 \leq LCM_j$;
13: **end for**

As discussed in Section 5, the mixed nature of POED-Mix scheme can benefit the permanent fault recovery. The main steps are summarized in Algorithm 6. Note that the main (backup) copies of tasks allocated to m processors are represented as $\Pi_M = \{\Psi^T_1, \cdots, \Psi^T_m\}$ ($\Pi_B = \{\Psi^B_1, \cdots, \Psi^B_m\}$). Moreover, different from the denotation in Section B.1 and B.2, $\Psi^T_{i,j}$ ($\Psi^B_{i,j}$) is designated as a group of main (backup) tasks in processor P_i whose backup (main) copies are in P_j. And the task set in $\Psi^T_{i,j}$ ($\Psi^B_{i,j}$) that will migrate to processor P_k is denoted as $\Psi^T_{i,j,k}$ ($\Psi^B_{i,j,k}$).

Specifically, once a processor (i.e., P_k) is detected to be failed at time t_0, the system will switch to the *emergency operation mode*. For all backup tasks allocated to it (i.e., Ψ^B_k), the corresponding main copies on other processors should start executing using f_{max} to preserve the system's original reliability with respect to transient faults (line 4). Their execution can be scaled down again once the system returns to the normal operation mode. Then, for all main tasks allocated to the failed processor (i.e., Ψ^T_k), the corresponding backup copies have already been evenly distributed among remaining processors. We can convert these backup copies as their new main tasks (line 5). However, they still need to execute in f_{max}, again to preserve the transient fault rate. And they can be slowed down as normal main tasks after the recovery. Moreover, the corresponding backups for Ψ^T_k still need to be mixed allocated to all processors except for P_k and itself. The partition is generated using WFD heuristic (line 6).

The same strategy applies to the backup copies on P_k by migrating them to other processors (line 7). After figuring out the tasks that will be migrated to each functional processors, their corresponding Local LCMs can be generated as well using all tasks' periods from its local (i.e., Ψ^T_j and Ψ^B_j) and future migrated tasks (i.e., Ψ^{mig}_j) (line 10). Here, Ψ^{mig}_j represents all tasks that will migrate to processor P_j, which can be formally represented as $\{\Psi^B_{k,x,j} \cup \Psi^B_{x,k,j}, 1 \leq x \leq m, x \neq k, x \neq j\}$. Again, these updated Local LCMs are the upper bounds of the transition windows for each processor for task migration. And the actual task migration happens within the transition window when both local and migrated tasks arrive at the same time (line 11). After the final migration point, the system can switch back to the *normal operation mode* to provide tolerance to another permanent fault in the future.

The idea of Algorithm 6 is further illustrated in Figure 8. Suppose a permanent fault is detected in processor P_1 at t_0 in a multiprocessor system with 4 processors. For all main tasks in P_2, P_3 and P_4 with backups in P_1, they should start executing in f_{max} to preserve the transient fault rate. For main tasks in P_1 (i.e., Ψ^T_1), the corresponding backup copies can be found in other processors as $\Psi^B_{2,1}$, $\Psi^B_{3,1}$ and $\Psi^B_{4,1}$. They are converted to main tasks but still execute in f_{max}. The partition for their corresponding backups are generated using WFD heuristic (i.e., $\Psi^B_{2,1,3}$ for P_3 and $\Psi^B_{2,1,4}$ for P_4, $\Psi^B_{3,1,2}$ for P_2 and $\Psi^B_{3,1,4}$ for P_4, $\Psi^B_{4,1,2}$ for P_2 and $\Psi^B_{4,1,3}$ for P_3). For backup copies on P_1 (i.e., Ψ^B_1), their migration destinations are generated as well (i.e., $\Psi^B_{1,2,3}$ for P_3 and $\Psi^B_{1,2,4}$ for P_4, $\Psi^B_{1,3,2}$ for P_2 and $\Psi^B_{1,3,4}$ for P_4, $\Psi^B_{1,4,2}$ for P_2 and $\Psi^B_{1,4,3}$) for P_3). After figuring out the tasks for migration, the Local LCM for each functional processor is obtained as LCM_2, LCM_3

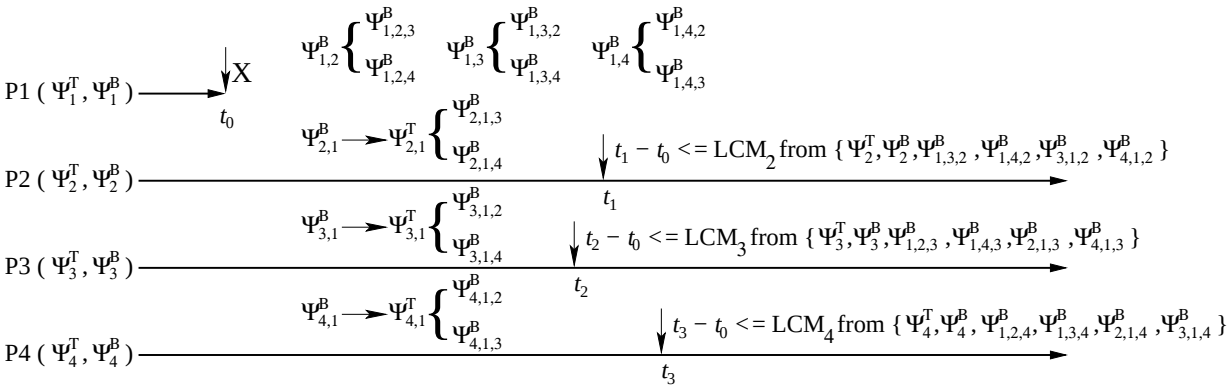


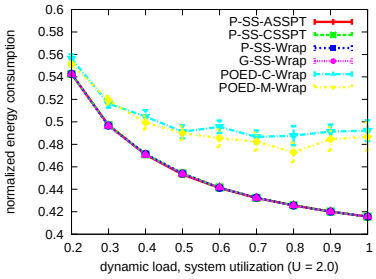
Fig. 8: Example of recovery steps for POED-Mix

and LCM_4, respectively; and the actual migrations happen at t_1, t_2 and t_3. In the example, the system will finish the recovery and switch back to the normal operation mode at t_3.

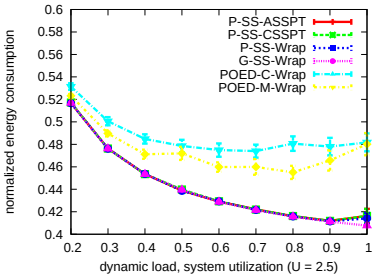
We can see that after applying failure recovery strategies above for each SS and POED based schemes, the system can be quickly recovered to the status that can provide tolerance to future permanent faults again. Moreover, the system's original reliability with respect to transient fault can still be preserved. Although the worst case TTSF could be as long as the Global LCM, the idea of using Local LCMs for migration points can extremely reduce TTSF in most cases.

APPENDIX
B. MORE EVALUATION RESULTS

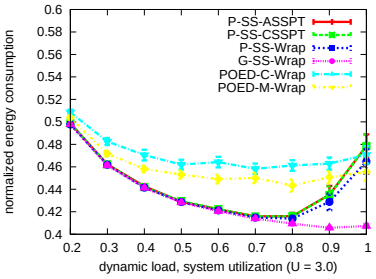
This section illustrates all available evaluation results for SS and POED based schemes in online execution (Figure 9 and Figure 10).



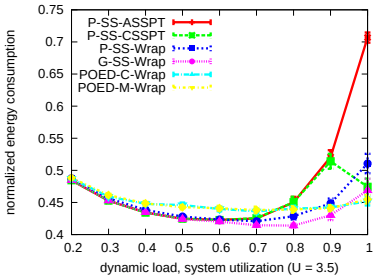
a. system utilization $U = 2.0$;



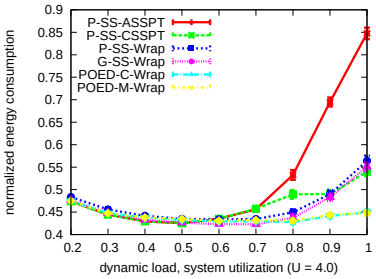
b. system utilization $U = 2.5$;



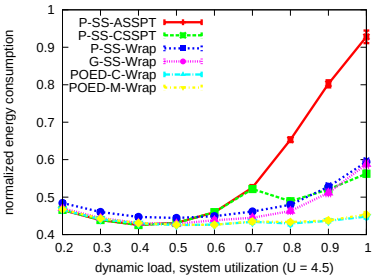
c. system utilization $U = 3.0$;



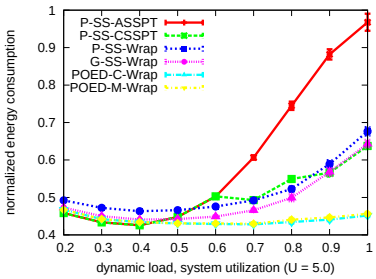
d. system utilization $U = 3.5$;



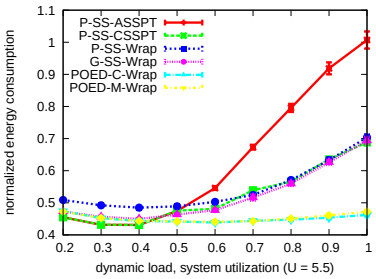
e. system utilization $U = 4.0$;



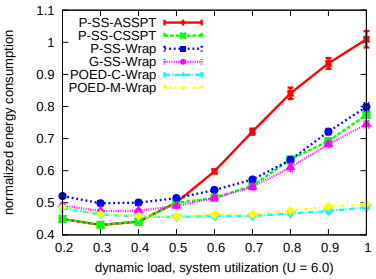
f. system utilization $U = 4.5$;



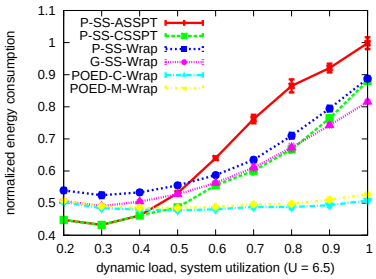
g. system utilization $U = 5.0$;



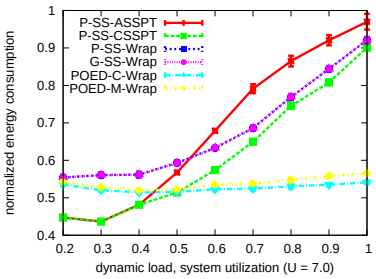
h. system utilization $U = 5.5$;



i. system utilization $U = 6.0$;

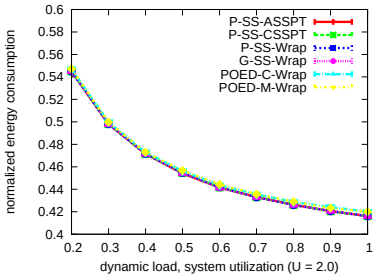


j. system utilization $U = 6.5$;

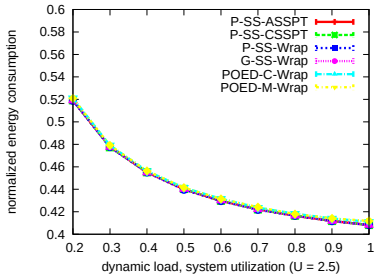


k. system utilization $U = 7.0$;

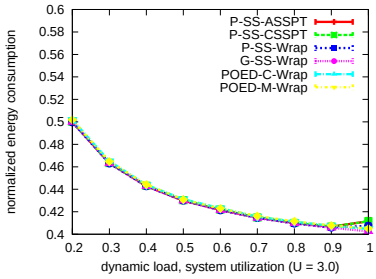
Fig. 9: Performance of P-SS, G-SS, POED-Cyclic and POED-Mix with online techniques under different system loads. ($u^{ave} = 0.1$)



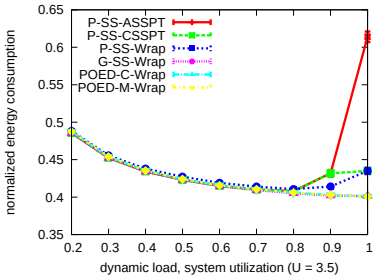
a. system utilization $U = 2.0$;



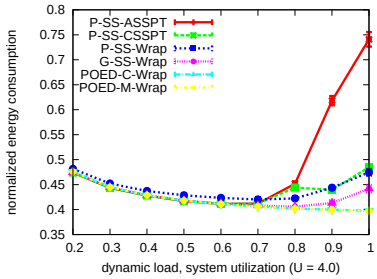
b. system utilization $U = 2.5$;



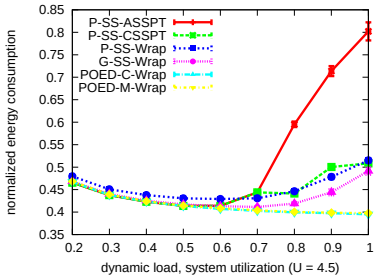
c. system utilization $U = 3.0$;



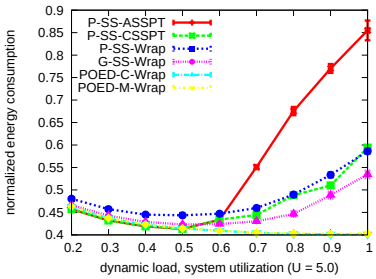
d. system utilization $U = 3.5$;



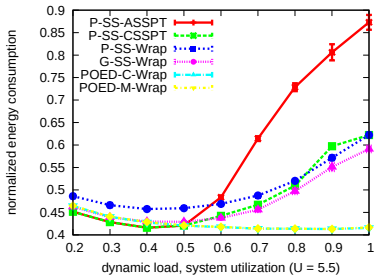
e. system utilization $U = 4.0$;



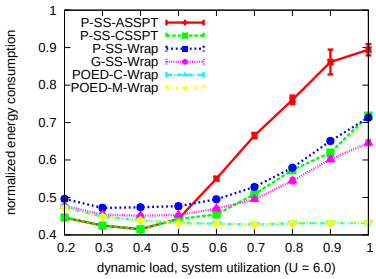
f. system utilization $U = 4.5$;



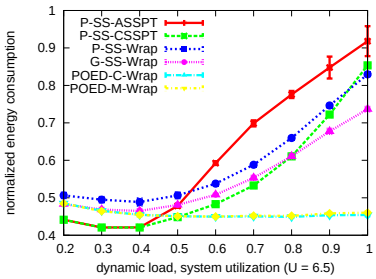
g. system utilization $U = 5.0$;



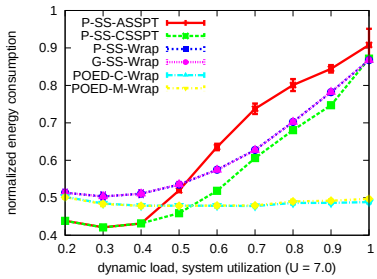
h. system utilization $U = 5.5$;



i. system utilization $U = 6.0$;



j. system utilization $U = 6.5$;



k. system utilization $U = 7.0$;

Fig. 10: Performance of P-SS, G-SS, POED-Cyclic and POED-Mix with online techniques under different system loads. ($u^{ave} = 0.05$)