Two-Phase Low-Energy N-Modular Redundancy for Hard Real-Time Multi-Core Systems

Mohammad Salehi, Alireza Ejlali, and Bashir M. Al-Hashimi, *Fellow, IEEE*

Abstract—This paper proposes an N-modular redundancy (NMR) technique with low energy-overhead for hard real-time multi-core systems. NMR is well-suited for multi-core platforms as they provide multiple processing units and low-overhead communication for voting. However, it can impose considerable energy overhead and hence its energy overhead must be controlled, which is the primary consideration of this paper. For this purpose the system operation can be divided into two phases: indispensable phase and on-demand phase. In the indispensable phase only half-plus-one copies for each task are executed. When no fault occurs during this phase, the results must be identical and hence the remaining copies are not required. Otherwise, the remaining copies must be executed in the on-demand phase to perform a complete majority voting. In this paper, for such a two-phase NMR, an energy-management technique is developed where two new concepts have been considered: *i*) Block-partitioned scheduling that enables parallel task execution during on-demand phase, thereby leaving more slack for energy saving, *ii*) Pseudo-dynamic slack, that results when a task has no faulty execution during the indispensable phase and hence the time which is reserved for its copies in the on-demand phase is reclaimed for energy saving. The energy-management technique has an off-line part that manages static and pseudo-dynamic slacks at design time and an online part that mainly manages dynamic slacks at run-time. Experimental results show that the proposed NMR technique provides up to 29% energy saving and is 6 orders of magnitude higher reliable as compared to a recent previous work.

Index Terms— Energy minimization, multi-core systems, real-time and embedded systems, reliability, scheduling.

1 Introduction

MULTI-CORE platforms have emerged to be popular and powerful computing engines for many recent embedded systems [1], [2], [3], [4], [5]. While such architectures have been employed for embedded applications that require high performance computing, we believe they also offer new considerable opportunities for designing embedded systems where hard real-time operation, high reliability in the presence of transient faults, and low energy consumption are required [6], [7], [8]. In this paper, we address the use of multi-core platforms to achieve high reliability with low energy-overhead for hard real-time embedded systems.

To achieve reliability against transient faults, we consider N modular redundancy (NMR) [9], [10], where multiple processing units execute identical copies for each task and their results are voted on to produce a single output. NMR is well-suited for multi-core platforms as they satisfy NMR requirements such as multiple processing units and low-overhead communication for voting [3]. An NMR system can mask faults while less than half of its units are faulty. Fault-tolerant real-time systems that has been considered in previous works require fault-detection mechanisms (e.g., [5], [6], [7], [8], [11]) and these works have assumed (usually implicitly) that they have perfect detection mechanisms (i.e., they can detect all faulty task executions). However, common fault-detection mechanisms are far less effective than what is required for highly reliable systems, whereas NMR does not require any specific fault-detection mechanism and uses result comparison (majority voting) for fault-detection and masking [9], [10]. Since it is very unlikely that all modules in NMR become faulty at the same time and make the same erroneous results, comparing the results can provide almost perfect fault-detection/-masking [9], [10]. Also, result comparison can be combined with hash-based detection mechanisms, e.g. Fingerprinting [31], to achieve very high detection coverage, about $1\text{-}2^{-16}$ [31]. Therefore, in our experiments in Section 5 we will assume $1\text{-}2^{-16}$ detection coverage for our system. Like all other fault-tolerance and fault-masking techniques, NMR can impose considerable energy overhead [9], [10], which is an important concern in the embedded systems where energy consumption is prominent. To reduce the energy overhead, we propose an energy-management technique that bears the major contributions of the work and is specifically developed for NMR when used for hard real-time multi-core systems (Sections 3 and 4). The main contributions of this work are:

i) Considering the dominance of the fault-free execution on faulty executions [6] [8] [38], a *two-phase NMR* is proposed that achieves minimized energy consumption in the absence of faults while guaranteeing reliability and deadline requirements.

ii) A specific type of slack time, called *pseudo dynamic slack*, is considered in this work. As explained in Section 4, this type of slack time is different from conventional slack times, i.e., static and dynamic slack [6], [8], [20].

- *Mohammad Salehi and Alireza Ejlali are with the Department of Computer Engineering, Sharif University of Technology, Tehran 14588, Iran (e-mail: mohammad_salehi@ce.sharif.edu, ejlali@sharif.edu).*
- *Bashir M. Al-Hashimi is with the School of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, U.K. (e-mail: bmah@ecs.soton.ac.uk).*

iii) An energy-management technique is proposed that exploits the pseudo-dynamic and static slacks through offline optimization (Section 4.2). This is different from previous works that have not proposed a mechanism to manage the pseudo-dynamic slack. Also, an online energy-management technique is proposed to exploit dynamic slacks at run-time (Section 4.3).

iv) A specific scheduling technique is developed, called *block-partitioned scheduling* (Section 3) that provides the ability of *in-advance parallel task execution* (Section 3) to exploit pseudo-dynamic slacks more effectively.

The remainder of this paper is organized as follows. In section 2 we review the related work. The proposed technique is presented in Section 3. Section 4 describes the energy-management method which is used for the proposed technique. The experimental results are presented in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

Some research works, e.g., [6], [7], [11], have addressed both fault tolerance and low energy-consumption in fault-tolerant real-time systems with two processors. These works have not considered multiple faults per task execution, and also they assume they have perfect fault-detection mechanisms. [13] has proposed voltage-scaling techniques to reduce the energy consumption of triple-modular redundancy (TMR). However, this work has only considered single task applications.

Many previous works in the context of multi-processor systems either propose energy reduction management techniques without considering reliability (e.g., [14], [15], [32], [33]) or consider reliability without considering energy consumption (e.g., [4], [30], [34]). [14] has considered variation in execution times to propose a scheduling algorithm based on dynamic voltage scaling (DVS) [12] for multi-processor systems. [15] has studied the energy efficiency of multi-core platforms that use multiple voltage islands. [32] has proposed a technique to minimize chip-level peak power consumption in multi-core systems running sporadic real-time tasks. [33] has proposed an adaptive task partitioning for multi-core systems running independent periodic real-time tasks. [4] has evaluated scheduling heuristics for tasks with different criticality. [30] has proposed a mapping optimization technique for mixed critical multi-core systems with different reliability requirements. [34] has proposed software transformations to increase reliability through reducing instructions vulnerabilities and the executions of critical instructions.

Recently, research works have also been focused on both energy and reliability considerations in multi-core systems. Some works, e.g. [35], [36], [37] have proposed multi-core architectures that exploit redundancy at different levels of abstraction to target low-energy consumption and reliability. [35] has proposed an adaptive multi-core architecture that selectively adjusts pipeline-level redundancy to satisfy reliability target with low energy consumption. [36] has proposed a customizable chip-level redundancy technique for multi-core systems that utilizes power efficient hardware fault-detection mechanisms along with forward recovery to reduce overheads in case of fault-free executions. [37] has considered the effects of DVS on the soft error rate and proposed a flexible dual modular redundancy (DMR) mechanism that selectively enables per-core DMR to increase reliability. However, these works require hardware modification or redesign, and hence, cannot be used by the current commercial-off-the-shelf processors, while our proposed technique is general and can be exploited by any multi-core processor that supports DVS. Some works, e.g. [5] [16], [17], [18], [38], have proposed energy-management techniques for task-level redundancy in multi-core systems. [5] and [16] have considered only one faulty execution for each task to preserve the original system reliability, while for many applications (e.g., the applications that are used in harsh environments) a high level of reliability cannot be achieved unless tolerating multiple faulty tasks [9], [10], [17], [18]. Some works have considered different application models, e.g. periodic independent real-time tasks in [17] and [38] and parallel independent applications in [18]. However, these works cannot be applied to tasks with precedence constraints (e.g., task graphs [5], [6], [7]), while we consider hard real-time applications with task precedence constraint and propose a scheduling and energy-management technique for these applications.

3 Proposed Two-Phase NMR Technique

In this paper, we consider frame-based applications [5], [6], [7] with hard timing requirements and task precedence constraints where n dependent tasks $\{T_1, T_2, T_3, \ldots, T_n\}$ are executed within each execution frame and must be completed as a whole before the end of the frame (specified by a deadline D). We also consider that the task precedence constraints (dependencies between the tasks) are depicted as a directed acyclic graph (DAG) [5], [6], [7]. For example Fig. 1a shows an example application tasks graph with six tasks where the numbers placed above the tasks is their worst-case execution time at the maximum supply voltage V_{max} and the maximum operational frequency f_{max} (denoted by W_i for each task T_i). For this type of applications we propose a two-phase NMR technique with low energy consumption running on multi-core platforms. To do this, a new scheduling technique is proposed and a new type of slack time (which is specific to the proposed two-phase NMR) is exploited to manage energy consumption. In this section we describe the two-phase operation of the system and the proposed scheduling technique and in the next section we explain the energy-management technique.

The two operation phases of the proposed NMR are:

1. *Indispensable phase*: At first the system operates in its indispensable phase where it executes a multi-core schedule containing $\lceil N/2 \rceil$ copies of each task. For each task, the results of the $\lceil N/2 \rceil$ task copies are compared. If no fault occurs, the task results must be identical and in this case it is used as the result of the system. However, when the results

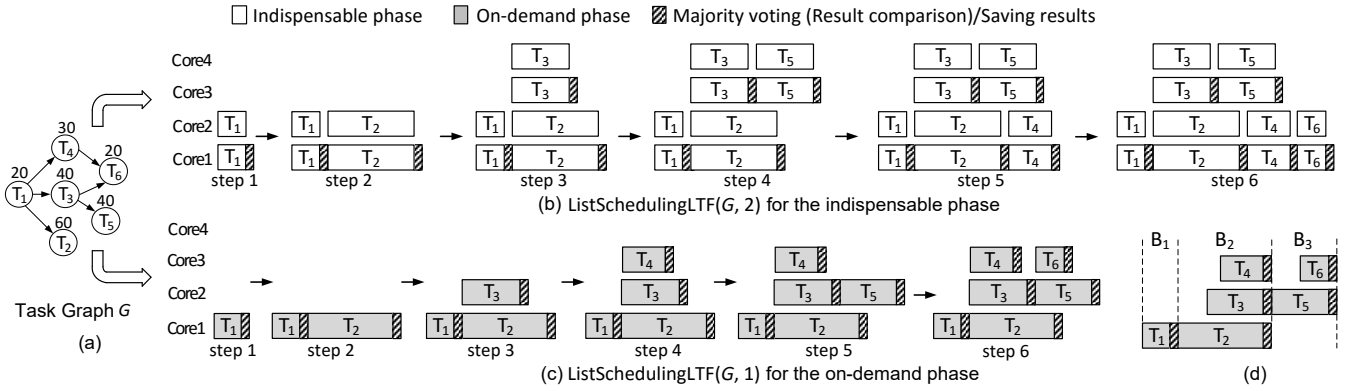

Fig. 1. Synthesizing a TMR system (i.e., NMR with N=3) on a quad-core platform. a) An example task graph, b) Creating a schedule with two copies for each task for the indispensable phase, c) Creating a schedule with one copy for each task for the on-demand phase, and d) A block-partitioned version of the on-demand phase schedule.

are not identical (when some faults have occurred during the indispensable phase), the system temporarily switches to the on-demand phase where it executes the remaining $\lfloor N/2 \rfloor$ copies of the task to perform a complete majority voting.

2. *On-demand phase*: In this phase, the system executes a part of a multi-core schedule that contains the remaining copies of the task which had faulty executions in the indispensable phase. As $\lceil N/2 \rceil$ copies of the task have already been executed in the indispensable phase, in the on-demand phase we execute the remaining $\lfloor N/2 \rfloor$ copies of the same task to obtain N results for performing a complete majority voting to mask the faults.

Therefore, each of the two operation phases of the proposed NMR technique requires its own schedule, so that we need to synthesize two schedules from the same application task graph. These two schedules are: *i*) a multi-core schedule containing $\lceil N/2 \rceil$ copies for each task for the indispensable phase, *ii*) a multi-core schedule containing $\lfloor N/2 \rfloor$ copies for each task for the on-demand phase. It is known that finding the optimal multi-core schedule to maximize parallelism (i.e., minimizing the schedule time length) is an NP-hard problem [5]. Indeed multi-core schedules are typically obtained by the *list scheduling* algorithm [19] as a simple heuristic that also provides parallelism. Similarly, in this paper we use list scheduling to synthesize the multi-core schedules of the indispensable and on-demand phases. Also, in the list scheduling, whenever several tasks can be scheduled (these are the tasks that all their predecessors are scheduled), we use the *longest task first (LTF)* policy to determine the execution order. We will discuss in Section 4 why the LTF policy is effective for our proposed technique. For example, considering a TMR system (i.e., NMR with N=3), Fig. 1 shows the step by step generation of the two multi-core schedules for a given task graph (Fig. 1a) using list scheduling with LTF policy. Fig. 1b shows the schedule with two copies of each task for the indispensable phase and Fig. 1c shows the schedule with one copy of each task for the on-demand phase.

For the schedule which is used in the on-demand phase, we require that each task can overlap (in time) with at most one other task in each of the other cores. For example, the multi-core schedule of Fig. 1c (step 6) does not satisfy this condition as in this schedule T_2 overlaps with both T_3 and T_5 on Core2 and also overlaps with both T_4 and T_6 on Core3. Indeed, we need the schedule of Fig. 1c (step 6) to be transformed to a schedule like the one in Fig. 1d that satisfies the condition as each task overlaps with at most one other task in each of the other cores. We require this condition to be satisfied because it lets us partition the multi-core schedule into time blocks, so that in each block only one single task or multiple parallel tasks exist. For example, in Fig. 1d the block B_1 only consists of the task T_1, the block B_2 consists of the parallel tasks T_2, T_3 and T_4, and the block B_3 consists of the parallel tasks T_5 and T_6. In this paper, we call such schedules, *block-partitioned (BP)* schedules. As we will show later in this section, whenever a fault occurs during the indispensable phase, we switch to the on-demand phase to execute exactly one block of the BP schedule and then we switch back to the indispensable phase to continue executing the schedule of the indispensable phase.

As a multi-core schedule which has been synthesized using the list scheduling technique with LTF policy (e.g., the schedule of Fig. 1c) may not be BP, we use a simple technique to convert ordinary schedules to BP schedules. Suppose that in a multi-core schedule a task T_A overlaps with two other tasks T_B and T_C scheduled on another core (Fig. 2a). Assuming that the task T_C comes after the task T_B, we simply shift the task T_C (and all its successor tasks) to the right until there is no overlap between T_A and T_C. As it can be seen from Fig. 2a, the amount of this shift (denoted by σ in the figure) is simply the difference between the finish time of T_A and the start time of T_C. We start from the beginning of a multi-core schedule, move to the right, and apply this technique until we obtain a BP multi-core schedule. As an example, when we apply this technique to the schedule of Fig. 1c (step 6), we obtain the BP schedule of Fig. 2b (step 3). One point that should be noted here is that block-partitioning may increase the execution time of an application and hence it may cause the application to be unschedulable. Therefore, we use the proposed energy-management technique (Section 4) when the application total execution time is less than its deadline. This implies that the energy-management technique might not be used for some applications that have

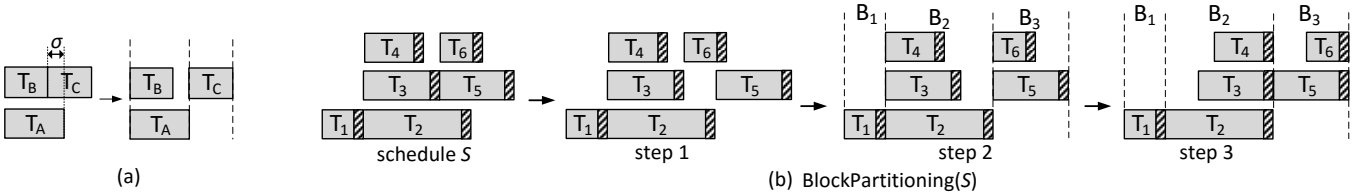

Fig. 2. Block partitioning scheme. a) A technique to convert ordinary schedules to block-partitioned (BP) schedules and b) Block-partitioning a schedule that is not BP.

tight deadlines. Similar schedulability conditions are used by other techniques, e.g. [16], [17] and [38], to define infeasible solutions.

In the following, we describe how the proposed two-phase NMR technique works by means of the example of Fig. 1 where we have a TMR system running on a quad-core platform. When no fault has occurred the system executes the schedule of the indispensable phase (Fig. 1b (step 6)) where two copies of each task T_i are executed and their results are compared. If the results are identical, it is used as the result of the system. Whenever the results of a task T_i are not identical (which indicates that some faults have occurred during the indispensable phase), we switch to the on-demand phase to execute the block of the BP schedule of the on-demand phase (Fig. 1d) that includes the same task T_i. After executing the third copy of T_i in the on-demand phase, a majority voting is done over the three results to mask the faults. Then, we switch back to the indispensable phase to continue executing this schedule from the point it was broken.

Fig. 3 shows how the proposed technique operates when some faults occur during executing the application of Fig. 1. Note that, in this paper, whenever we say a fault occurs or a task becomes faulty, we mean that the task gives an incorrect result due to some errors (e.g. one or more transient faults). Assuming that the task T_2 becomes faulty, when comparing the results of T_2, they do not match, and hence the system temporarily switches to the on-demand phase. The result mismatch may happen due to a fault during the task execution or even due to a fault that corrupts the result comparison between the two phases. In the on-demand phase as T_2 belongs to the block B_2 of the BP schedule of Fig. 1d, the block B_2 is executed (the highlighted tasks T_2 and T_4 in Fig. 3), and then a majority voting is done over the results of the three copies of

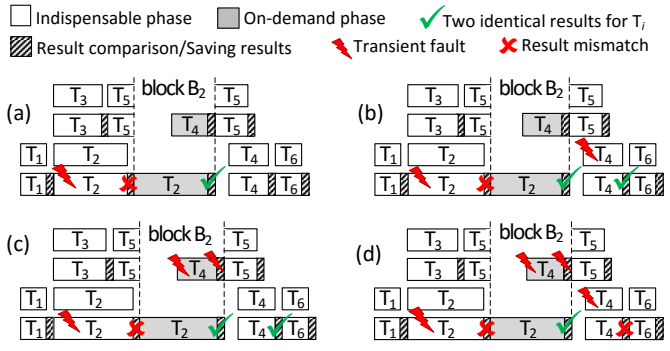

Fig. 3. Operation of the proposed technique when faults occur during the execution of the application of Fig. 1.

T_2 to mask the fault (Fig. 3a). Here, T_3 is not executed in the block B_2 during the on-demand phase as it has already finished successfully before detecting the fault in T_2 and hence it is no longer required. The important point to be noted here is that when we execute B_2 during the on-demand phase we not only execute T_2 (whose result is required for majority voting as its execution in the indispensable phase has been faulty), but also we execute T_4 in parallel with T_2 and its result is saved in memory, so that it can be used later for possible majority voting. After executing the block B_2 the system switches back to the indispensable phase and continues executing the schedule from the point it was broken. After switching back to the indispensable phase, two possible execution scenarios can be considered regarding the task T_4:

i) If a fault occurs during the execution of T_4 in the indispensable phase (Fig. 3b), when the result comparison indicates fault occurrence, the system does not need to switch to the on-demand phase as the results of three copies of T_4 are already available to be voted on (the results of two copies of T_4 are obtained in the indispensable phase and the result of another copy of T_4 already exists in the internal memory as it was executed in-advance in the previous on-demand phase).

ii) If no fault occurs during the execution of T_4 in the indispensable phase (Fig. 3a), the results of the copy of T_4 that was executed in-advance in the previous on-demand phase are no longer required and can be dropped from the memory.

One question that may arise here is "*what happens if the in-advance execution of T_4 becomes faulty?*". (Such a fault may occur during the in-advance execution of T_4 in the on-demand phase or during saving the results of the in-advance execution of T_4 between the two phases or even after the in-advance execution of T_4 in its stored results). In this case, when the system executes T_4 in the indispensable phase, if no fault occurs (Fig. 3c), we will not use the results of the in-advance execution of T_4, and hence no problem occurs. However, if the execution of T_4 in the indispensable phase also becomes faulty (Fig. 3d), the system cannot mask this second fault as the stored values of the in-advance execution are also faulty. Indeed, a TMR system can mask only one faulty execution for each task (generally speaking, an NMR system can mask $\lfloor N/2 \rfloor$ faulty executions for each task) [9], [10].

The in-advance executions of tasks (e.g., T_4 in the block B_2 in Fig. 3) in the on-demand phase are useful because:

i) Because of the use of parallel execution in the on-

Inputs:
G: application task graph
N: parameter *N* of NMR, e.g, 3 for TMR
Outputs:
S_{IND}: schedule for the indispensable phase
S_{BP}: BP schedule for the on-demand phase

1: S_{IND}=*ListSchedulingLTF*(*G*, $\lceil N/2 \rceil$); // Fig. 4b
2: S_{TMP}=*ListSchedulingLTF*(*G*, $\lfloor N/2 \rfloor$); // Fig. 4b
3: S_{BP}=*BlockPartitioning*(S_{TMP}); // Fig. 4c

(a)

function ListSchedulingLTF(*G*, *q*)
// *G*: input task graph, *q*: number of copies for each task in the
// schedule, *S*: the output schedule

1: *S* = *Null*; // Initialize *S* with an empty schedule
2: **while** *all tasks in G are not scheduled* **do**
3: T_i= *the largest unscheduled task in G whose predecessors*
-- *have all scheduled*;
4: *Add q parallel copies of* T_i *to S*;
5: **endwhile;**
6: **return** *S;*

(b)

function BlockPartitioning(*S*)
// *S*: the input multi-core schedule

1: **for** *each task* T_A *from the beginning of S* **do**
2: **if** T_A *overlaps with more than one task,* T_B *and* T_C *(where* T_C
-- *comes after* T_B *in the same core)* **then**
3: σ = *(finish time of* T_A*) – (start time of* T_C*)*;
4: *shift* T_C *and all its successors in S to the right by* σ;
5: **endif;**
6: **endfor;**
7: **for** *each block B in S* **do**
8: *shift all tasks in B to the right and place them at the end of B*;
9: **endfor;**
10: **return** *S*;

(c)

Fig. 4. The proposed scheduling technique.

demand phase, in-advance executions do not impose any time overhead. For example, it can be seen in Fig. 3 that when the system have to execute T_2 in the on-demand phase, the in-advance execution of T_4 is performed in parallel with it. Also, because of the use of LTF scheduling, tasks that come later in the schedule (e.g., T_4) can never be longer than the tasks that come earlier (e.g., T_2) which means that the in-advance execution of T_4 cannot lengthen the execution of the block B_2 in Fig. 3. Indeed, if we did not use in-advance executions, we would not have any parallel execution during the on-demand phase which implies that the use of in-advance executions helps us reserve relatively less slack time for the on-demand phase, resulting in more slack to be available for energy management.

ii) Although in-advance executions of tasks in the on-demand phase may turn out to be useless when no fault occurs later during the execution of the task in the indispensable phase, they have a negligible impact on the average energy consumption. This is because an in-advance parallel execution is performed only when a fault occurs in the indispensable phase (for example in Fig. 3 the in-advance execution of T_4 has been performed because a fault has occurred in T_2 during the indispensable phase). Note that while from a reliability point of view the consideration of faults is a must, from the average energy consumption point of view, we do not need to consider the cases where the system tolerates a fault [6], [13]. As an example, consider T_2 and T_4 in Fig. 3. Suppose that the probability of a task execution becomes faulty is 10^{-4} and the energy consumption of T_2 and T_4 are 10 mJ and 5 mJ. When no fault occurs, the system only executes T_2 in the indispensable phase and consumes 2×10=20 mJ. If a fault occurs during the execution of T_2 in the indispensable phase, the system will execute T_2 and T_4 in the on-demand phase and hence consumes (10+5)=15 mJ more energy. Therefore, the average energy consumption for the execution of T_2 and T_4 is $(1-10^{-4})\times 20+10^{-4}\times(20+15)=20.0015$ mJ which is very close to the energy consumption when no faults occur (20 mJ). This is also consistent with our experimental observations showing that the average energy consumption differs less than 0.01% from the fault-free energy consumption. This example shows that the energy overhead of the in-advance executions is negligible from the viewpoint of average energy consumption.

Fig. 4 shows the pseudo-code of the proposed scheduling method used in our technique that receives an application task graph (*G*) to make schedules for the indispensable and on-demand phases (i.e., S_{IND} and S_{BP} respectively). The pseudo-code of Fig. 4a is the main body of the scheduling technique that calls the functions presented in Figs 4b and 4c. The function of Fig. 4b (*ListScheduling-LTF(G, q)*) implements the list scheduling algorithm with the LTF policy to make a schedule *S* containing *q* copies of each task from a task graph *G*. In this function, line 1 is for the initialization purpose. In line 2, we begin a while body to apply the scheduling to all tasks. Line 3 is used to implement LTF list scheduling, as it selects the largest unscheduled task T_i whose predecessors have all scheduled. In line 4, *q* parallel copies of T_i are scheduled. Finally, line 6 returns the schedule *S*. As it can be seen from Fig. 4a, this function is required for both the indispensable and on-demand phases. For the indispensable phase we need a schedule containing $\lceil N/2 \rceil$ copies for each task, and for the on-demand phase we need a schedule containing $\lfloor N/2 \rfloor$ copies for each task. We make these two schedules in lines 1 and 2 of Fig. 4a. In line 3 of Fig. 4a we use the function of Fig. 4c (*BlockPartitioning(S)*) to convert the schedule S_{TMP} (temporary schedule obtained from line 2 of Fig. 4a) to the BP schedule S_{BP}. The function of Fig. 4c receives a multi-core schedule *S* and starts from the beginning of the schedule (line 1). In line 2 we check if each task, say T_A, overlaps with more than one task in another core in the schedule *S*, say T_B, T_C (where T_C comes after T_B on the same core). If so, through lines 3 and 4 we shift the task T_C (and all its successor tasks in *S*) to the right until there is no overlap between T_A and T_C. In

line 4, when we shift T_C to the right, we need to shift all the tasks that come after T_C on the same core and the tasks that are dependent to T_C (successors of T_C in the task graph) but are scheduled on the other cores. After removing possible overlaps in the schedule (i.e., partitioning the schedule into blocks), through lines 7 to 9 we shift all tasks in each block to the right to place them at end of the block. We will discuss in Section 4 why this is effective for our proposed technique. Finally, in line 10 the schedule S (i.e., a BP schedule) is returned.

It is noteworthy that although the proposed NMR technique needs at least $\lceil N/2 \rceil$ cores for parallel execution of each task in the indispensable and on-demand phases, if less than $\lceil N/2 \rceil$ cores are available, the proposed technique still can be used (with a slight change) but with less parallelism. Indeed the technique can be even used for a single core where for each task, at first the system executes $\lceil N/2 \rceil$ copies of the task one after another (in series) in its indispensable phase and then compares their results. If some faults occur during the indispensable phase, the system executes the remaining copies of the task (again in series) for the on-demand phase and finally the whole results are voted on to mask the faults. It should be noted that this reduced parallelism obviously takes more time and hence may not be suitable for real-time systems with tight deadlines. When more cores are available, more parallelism can be achieved that results in lower schedule length that provides higher schedulability [19]. This can also release some static slack time that can be used for energy management.

4 Energy Management

For the proposed NMR technique we have implemented a specific energy-management technique which comprises offline (Section 4.2) and online (Section 4.3) stages and exploits different types of slack time to reduce the system energy consumption through DVS [12].

Let W_{IND} and W_{BP} be the worst-case time it takes to execute the schedules S_{SC} and S_{BP} in the indispensable and on-demand phases respectively. We need not only to reserve the time W_{IND} for the indispensable phase but also to reserve the time W_{BP} for the on-demand phase. Hence, the proposed technique is feasible when $W_{IND}+W_{BP} \le D$ (D is the application deadline) and the static slack SS which is left over from the application and can be used for energy management is:

$$SS = D - \left(W_{IND} + W_{BP}\right) \quad (1)$$

where $W_{IND}+W_{BP}$ is the application total execution time.

As the amount of static slack is known at design time, offline techniques (e.g., the even slack distribution technique in [20]) can be used at design time to distribute this slack among the tasks. However, in the proposed technique, there are also two other types of slack time that are created at run-time, and hence, unlike the static slack, cannot be allocated at design time, and have to be allocated at run-time. These two types of slacks are:

- *Dynamic slack*: This slack results at run-time when a task consumes less than its worst-case execution time due to early completion [6], [8], [11]. It should be noted that the actual execution time of a task is not known at design time, and hence the dynamic slack time which is obtained from the task is also not known at design time.
- *Pseudo-dynamic slack*: Although we always reserve enough time to execute the BP schedule completely, we do not usually need to execute the tasks of this schedule at run-time. This is because when $\lceil N/2 \rceil$ copies of a task finishes successfully during the indispensable phase, this task no longer requires the additional $\lfloor N/2 \rfloor$ copies in the on-demand phase. Therefore, the task copies can be dropped from the BP schedule, thereby releasing some slack. We have called this slack pseudo-dynamic slack because, just like dynamic slacks, it is created at run-time, but unlike dynamic slacks, its amount can be calculated offline at design time.

When a task T_i executes successfully in the indispensable phase and we drop its copies from the schedule of the on-demand phase, the pseudo-dynamic slack time δ_i is released that can be exploited by DVS to reduce the energy consumption of the subsequent tasks in the indispensable phase. As the schedule of the on-demand phase is available at design time, the amount of this reclaimed slack can be calculated offline at design time. To do this, at design time, we consider dropping the tasks from the schedule of the on-demand phase one after another in the order in which they appear in the schedule of the indispensable phase and the time which is released due to dropping a task T_i is the pseudo-dynamic slack δ_i.

Fig. 5 shows in more detail how we calculate the pseudo-dynamic slack δ_i which is released after dropping T_i from the schedule of the on-demand phase. To calculate the pseudo-dynamic slack δ_i the following three cases can be considered:

1. Case I (Fig. 5a): If there is no task except T_i in the block, when T_i is dropped from the schedule the released slack δ_i will be W_i+c_i, where W_i is the worst-case execution time of T_i and c_i is the maximum time which is required for comparing the results (majority voting) or saving results.
2. Case II (Fig. 5b): If T_i is the largest task in the block (i.e., $W_i \ge \max\{W_j\}$ for all the remaining tasks in the block), after dropping T_i from the schedule the value of pseudo-dynamic slack δ_i is W_i-$\max\{W_j\}$.
3. Case III (Fig. 5c): If there exists at least one task T_j in the block larger than T_i, after removing T_i from the schedule no pseudo-dynamic slack will be released.

Considering the three cases in Fig. 5, δ_i is calculated as:

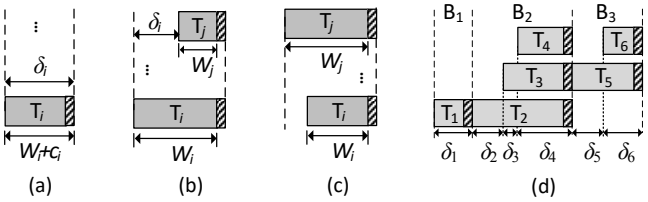

Fig. 5. Pseudo-dynamic slack (δ_i) calculation.

$$\delta_i = \begin{cases} W_i + c_i & \text{when only } T_i \text{ exists in the block (Case I)} \\ W_i - \max\{W_j\} & \text{when } W_i \geq \max\{W_j\} \text{ (Case II)} \\ 0 & \text{when } W_i < W_j \text{ for at least one task } T_j \text{ (Case III)} \end{cases} \quad (2)$$

In the following we illustrate how pseudo-dynamic slack is calculated by means of an example. Fig. 5d shows the pseudo-dynamic slack δ_i which will be released after dropping each task T_i from the BP schedule. The worst-case execution times of the example tasks are shown in Fig. 1a. The tasks are dropped from Fig. 5d in the order in which they are scheduled. In this example, without loss of generality, we assume that comparing results (majority voting) and saving results for all the tasks consume 5 time units (i.e., c_i=5 for all the tasks). For this example, it can be seen from the schedule of Fig. 5d that as the task T_1 is a single task in the block B_1 (Case I), if we drop T_1 from the schedule, the released slack will be $\delta_1=W_1+c_1=25$. After dropping T_1, if we drop T_2 from the schedule, as T_2 is the largest task in the block B_2 (Case II), the released slack will be $\delta_2=W_2-W_3=20$. After dropping T_2, as the task T_3 is the largest task in B_2, the released slack will be $\delta_3=W_3-W_4=10$ (Case II). After dropping T_3, the task T_4 will be a single task in the block B_2 and if we drop T_4 from the schedule the released slack will be $\delta_4=W_4+c_4=35$ (Case I). Similarly, we obtain: $\delta_5=20$ and $\delta_6=25$. Although the amount of the pseudo-dynamic slacks can be calculated at design time, it should be noted that this slack is not available (and hence cannot be allocated) from the beginning of the application execution and is created at run time when a task finishes successfully in the indispensable phase. This is why we call it pseudo-dynamic slack.

It is noteworthy that the proposed scheduling technique (Section 3) helps to distribute pseudo-dynamic slacks evenly among the tasks. It is known that even slack distribution results in more energy saving as compared to uneven slack distribution [6], [20]. Indeed pseudo-dynamic slack is prone to be distributed unevenly among the tasks. This is because a pseudo-dynamic slack which is obtained from a task cannot be exploited by the same task or by its previous tasks and it can only be exploited by its subsequent tasks. Therefore, those tasks that appear later in the schedule have more chance to gain larger pseudo-dynamic slacks as compared to the tasks that come earlier. This implies that when pseudo-dynamic slack becomes available sooner rather than later, it helps to distribute pseudo-dynamic slacks more evenly. To achieve this, we use two policies in our proposed scheduling technique (Section 3): *i*) we move tasks in each block of the BP multi-core schedule to the end of the block, thereby enabling the slacks to appear sooner in the block (see Fig. 5b), *ii*) we use the LTF policy. To give an insight into how the LTF policy works, consider the following example. Suppose that three tasks T_1, T_2 and T_3 with worst-case execution times W_1=6, W_2=3 and W_3=2 appear in the LTF order in the indispensable phase. Assuming that these tasks are in one block of the on-demand phase, using (2), the pseudo-dynamic slacks obtained from these tasks will be $\delta_1=W_1-W_2=3$, $\delta_2=W_2-W_3=1$ and $\delta_3=W_3=2$ (In this example we assume that c_i=0). However, if the tasks appear in the order T_3, T_1 and T_2 which is not LTF, the pseudo-dynamic slacks will be $\delta_3=0$, $\delta_1=3$ and $\delta_2=3$. Therefore, in the LTF order pseudo-dynamic slacks are available sooner and hence can be distributed among the tasks more evenly.

As we explained earlier, dynamic slack may result at run-time due to early completion of tasks [6], [8], [11]. However, as the actual execution time of a task is not known at design time, the amount of dynamic slack is also not known at design time. Hence, we provide an online energy-management technique to exploit dynamic slacks at run-time (Section 4.3). With respect to pseudo-dynamic slacks, since unlike dynamic slack the amount of pseudo-dynamic slack is known at design time, we have developed a specific offline technique to manage pseudo-dynamic slacks (Section 3.2).

4.1 Energy and Reliability Models

Power consumption of each task T_i mainly comprises dynamic power $P_{dyn}(T_i)$ and static power $P_{stat}(T_i)$. The dynamic power is determined by [6]:

$$P_{dyn}(T_i) = C_{eff} V_i^2 f_i \quad (3)$$

where C_{eff} is the effective switched capacitance, V_i and f_i are, respectively, the supply voltage and the operational frequency during the execution of T_i [6], [7]. The static power is mainly comprised of sub-threshold leakage power and can be written as:

$$P_{stat}(T_i) = I_{sub} V_i = I_0 e^{-\frac{V_{th}}{\eta V_T}} V_i \quad (4)$$

where V_i is the supply voltage, I_{sub} is the sub-threshold leakage current, I_0 depends on technology parameters and device geometries, η is a technology parameter, V_{th} is the transistors threshold voltage, and V_T is the thermal voltage [6].

When DVS is used, each task T_i is executed at a voltage V_i, which may be less than V_{max} (the maximum possible supply voltage). For each task T_i, we define the normalized supply voltage ρ_i as follows:

$$\rho_i = \frac{V_i}{V_{\max}} \quad (5)$$

When a task T_i is executed at the scaled voltage $V_i=\rho_i V_{max}$, considering an almost linear relationship between voltage and frequency [6], [7], we have: $f_i=\rho_i f_{max}$, where f_i is the operational frequency corresponding to V_i and f_{max} is the maximum possible operational frequency (corresponding to V_{max}). Therefore, when DVS is used, the actual execution time of the task is prolonged from t_i to t_i/ρ_i, and by substituting $V_i=\rho_i V_{max}$ and $f_i=\rho_i f_{max}$ in (3) and (4), the total energy which is consumed to execute the task T_i is given by [6]:

$$E(T_i) = (I_{sub}\rho_i V_{\max} + C_{eff}\rho_i^2 V_{\max}^2 \rho_i f_{\max})\frac{t_i}{\rho_i} = (P_S + \rho_i^2 P_D) t_i \quad (6)$$

where $P_S=I_{sub}V_{max}$ and $P_D=C_{eff}V^2_{max}f_{max}$ are respectively the static and dynamic powers when the system performs at the maximum voltage and frequency.

Without considering the energy consumption of the on-demand phase (which commonly has a very low

probability of being performed as faults rarely occur [6], [8]), we focus on the indispensable phase and aim at minimizing the fault-free energy consumption (like the works [5], [6], [8]). Using the energy model of (6) that gives the energy consumption of a single task, the energy which is consumed to execute a task T_i in the indispensable phase (i.e., executing $\lceil N/2 \rceil$ copies of the task and comparing the results) can be written as:

$$E_{NMR}(T_i) = \lceil N/2 \rceil (P_S + \rho_i^2 P_D) t_i + c_i \quad (7)$$

where c_i is the result comparison time. Based on (7), the energy consumption of the fault-free execution of an application with n tasks using the proposed NMR technique can be calculated as:

$$E_{app} = \sum_{i=1}^{n} E_{NMR}(T_i) = \sum_{i=1}^{n} \lceil N/2 \rceil (P_S + \rho_i^2 P_D) t_i + c_i \quad (8)$$

As it is explained in Section 3, the fault-free energy consumption is very close to the average energy consumption. Therefore, we use (8) in our offline energy management at design time (Section 4.2) to minimize the fault-free energy consumption. Also, in our experiments in Section 5 we report the fault-free energy consumption.

Transient faults are usually assumed to follow a Poisson distribution with an average rate λ [5], [6]. Considering the effects of DVS on transient fault rates, the fault rate at the scaled supply voltage $V_i=\rho_i V_{max}$ ($\rho_{min} \le \rho_i \le \rho_{max}=1$) is modeled as [5], [6]:

$$\lambda(\rho_i) = \lambda_0 10^{\frac{d(1-\rho_i)}{1-\rho_{min}}} \quad (9)$$

where $\lambda_0=\lambda(\rho_{max}=1)$ is the fault rate at the maximum voltage V_{max}, ρ_{min} is the ratio of the minimum supply voltage V_{min} to V_{max}, and the exponent value d is a technology dependent constant [5], [6]. Considering (9) (i.e., the effect of voltage scaling on transient fault rate), the probability of a task T_i being executed correctly is written as [5], [6]:

$$R_i(\rho_i) = e^{-\lambda(\rho_i)\frac{t_i}{\rho_i}} \quad (10)$$

where $\lambda(\rho_i)$ is given by (9) and t_i/ρ_i is the execution time of T_i when executed at $V_i=\rho_i V_{max}$. Conversely, the probability of failure of the task T_i (i.e., the unreliability of T_i) is denoted by [5], [6]:

$$F_i(\rho_i) = 1 - R_i(\rho_i) = 1 - e^{-\lambda(\rho_i)\frac{t_i}{\rho_i}} \quad (11)$$

To calculate the reliability of the proposed two-phase NMR technique we consider two cases: *i*) the fault-free execution where all $\lceil N/2 \rceil$ copies of each task are executed successfully in the indispensable phase and *ii*) the case where some tasks in the indispensable phase become faulty and we perform the on-demand phase. In NMR, the correct execution of at least $\lceil N/2 \rceil$ copies of each task is required for the system to be functional. In the proposed NMR technique, all the correct executions may be performed in the indispensable phase (when no fault occurs), or some of them are performed in the on-demand phase (when a fault occurs). Therefore, the reliability of the proposed system can be calculated by considering the two cases. The first case gives the reliability of T_i in the fault-free state, and the second case gives the reliability when some faults occur during the execution of T_i.

When no fault occurs, $\lceil N/2 \rceil$ copies of each task T_i are executed in the indispensable phase and the on-demand phase is not required. When we use DVS in the indispensable phase, each task T_i is executed on the scaled supply voltage $\rho_i V_{max}$. Therefore, using (10), the reliability of a task T_i in the fault-free case can be calculated as:

$$R1(T_i) = R_i(\rho_i)^{\lceil N/2 \rceil} \quad (12)$$

where $R_i(\rho_i)$ is given by (10). To calculate the reliability for the case that k ($1 \le k \le \lfloor N/2 \rfloor$) copies of each task become faulty (in NMR up to $\lfloor N/2 \rfloor$ faulty executions can be masked [9], [10]), we consider all the cases that j ($1 \le j \le k$) copies from $\lceil N/2 \rceil$ copies of T_i in the indispensable phase and $k-j$ copies from $\lfloor N/2 \rfloor$ copies of T_i in the on-demand phase become faulty. In these cases, the other $\lceil N/2 \rceil - j$ copies in the indispensable phase and $\lfloor N/2 \rfloor - (k-j)$ copies in the on-demand phase are executed correctly. Therefore, the probability of the correct execution of a task T_i when up to $\lfloor N/2 \rfloor$ executions of T_i become faulty can be calculated using (10) and (11) as:

$$R2(T_i) = \sum_{k=1}^{\lfloor N/2 \rfloor} \sum_{j=1}^{k} \underbrace{\binom{\lceil N/2 \rceil}{j} F_i(\rho_i)^j R_i(\rho_i)^{\lceil N/2 \rceil - j}}_{\text{indispensable phase}} \times \underbrace{\binom{\lfloor N/2 \rfloor}{k-j} F_i(\rho_{max})^{k-j} R_i(\rho_{max})^{\lfloor N/2 \rfloor - (k-j)}}_{\text{on-demand phase}} \quad (13)$$

where ρ_i determines the scaled voltage which is employed in the indispensable phase, $\rho_{max}=1$ is employed in the on-demand phase (as in the on-demand phase no DVS is used and tasks are executed at the maximum supply voltage V_{max}). Considering both the fault-free and faulty conditions, the reliability of a task T_i in the presence of up to $\lfloor N/2 \rfloor$ faults when executed by the proposed NMR technique, can be written as:

$$R(T_i) = R1(T_i) + R2(T_i) \quad (14)$$

The reliability of an application execution relies on the correct execution of all its tasks. Therefore, using (14), the reliability of an application with n tasks running by the proposed NMR technique can be calculated as:

$$R_{app} = \prod_{i=1}^{n} R(T_i) \quad (15)$$

4.2 Offline Energy Management

As explained in the previous sections, in the proposed NMR technique, when no fault occurs, we do not execute the on-demand phase (includes half of the copies for each task, i.e., $\lceil N/2 \rceil$), which results in considerable energy saving as compared with conventional NMR. In this section we discuss how the proposed NMR technique exploits static and pseudo-dynamic slack times to achieve even further energy reduction. For this purpose, we develop a specific technique to allocate static and pseudo-dynamic slack times to tasks offline at design time. When we allocate static and pseudo-dynamic slack times, we

assume that no dynamic slack exists, as the availability and the amount of dynamic slack times is not known at design time. Indeed, at first we minimize the expected energy consumption of the system by the offline allocation of static and pseudo-dynamic slacks assuming that no dynamic slack exists. However, at run-time we also exploit dynamic slacks through our online energy-management for further energy saving (Section 4.3).

To develop the offline slack allocation, we formulate the problem as an optimization problem. To do this, we formulate time constraints as inequalities. For the first task T_1, as the task is executed at the scaled supply voltage $\rho_1 V_{max}$, its worst-case execution time increases from W_1+c_1 to $(W_1+c_1)/\rho_1$. Considering that the only slack time which is available to T_1 is the static slack time (SS given by (1)) and no pseudo-dynamic slack is available to it (as pseudo static slack is obtained only from the previous tasks and T_1 has no previous task), T_1 cannot exploit more than the static slack SS. So we have:

$$\frac{W_1+c_1}{\rho_1}-(W_1+c_1)\le SS \tag{16}$$

It should be noted that although the whole of static slack SS is available to the first task T_1, this does not necessarily mean that it exploits all its available slack time. Indeed, each task can exploit only a part of its available slack and set aside the remaining for the subsequent tasks. During the indispensable phase, when a task T_i finishes successfully the pseudo-dynamic slack δ_i which is obtained by dropping the task T_i from the on-demand phase, is available to its subsequent tasks in the indispensable phase. Consequently, the task T_2 can exploit both the part of static slack SS left over by T_1 and the pseudo-dynamic slack δ_1 which is obtained by dropping T_1 from the on-demand phase. Hence, for the task T_2, we have:

$$\frac{W_2+c_2}{\rho_2}-(W_2+c_2)\le \underbrace{\delta_1}_{\text{Obtained from } T_1}+\underbrace{\left[SS-\left(\frac{W_1+c_1}{\rho_1}-(W_1+c_1)\right)\right]}_{\text{Static slack left over from } T_1} \tag{17}$$

Similarly, for each task T_i ($1\le i\le n$) we have:

$$\frac{W_i+c_i}{\rho_i}-(W_i+c_i)\le \underbrace{\sum_{T_j\in\Phi_i}\delta_j}_{\text{Obtained from the previous tasks}}+\underbrace{\left[SS-\sum_{T_j\in\Phi_i}\left(\frac{W_j+c_j}{\rho_j}-(W_j+c_j)\right)\right]}_{\text{Static slack left over from the previous tasks}} \tag{18}$$

where Φ_i is the set of all tasks that has been executed before starting the task T_i. The optimization problem of the offline part of energy management can be written as:

$$\text{minimze: } E_{app}=\sum_{i=1}^{n}E_{NMR}(T_i)$$

$$\text{subject to:}$$

$$c1:\frac{W_i+c_i}{\rho_i}-(W_i+c_i)\le\sum_{T_j\in\Phi_i}\delta_j+\left[SS-\sum_{T_j\in\Phi_i}\left(\frac{W_j+c_j}{\rho_j}-(W_j+c_j)\right)\right] \tag{19}$$

$$\text{for all } T_i\ (1\le i\le n)$$

$$c2: R_{app}\ge R_{demand}$$

where E_{app} is the energy consumption of an application executed using the proposed NMR technique (given by (8)), the constraint $c1$ (Inequality 18) is used to consider time constraints, i.e., to consider how much slack is available to each task (including pseudo-dynamic and static slack), and the constraint $c2$ guarantees that the system reliability does not fall below a required level R_{demand}.

The parameters, tasks worst-case execution time (W_i), result comparison time (c_i), static slack time (SS) and pseudo-dynamic slack (δ_i) are all known at design time. This implies that, this optimization problem can be solved offline at design time to determine the ρ_i values which minimize the system energy consumption. It should be noted however that we cannot assign obtained ρ_i values to the tasks at design time. Rather, we store the ρ_i values, and during the indispensable phase we assign the supply voltage $\rho_i V_{max}$ to the task T_i, whenever all its previous tasks finish successfully. In other words, the ρ_i values that we calculate using the proposed offline technique is only valid for the fault-free execution. If some faults occur during the indispensable phase, the ρ_i values will be no longer valid. This is because when a fault occurs in a task T_i during the indispensable phase, the system cannot drop it from the schedule of the on-demand phase, which means that the pseudo-dynamic slack δ_i will not be longer available. One possible solution for this problem is the offline calculation of ρ_i values for all possible fault scenarios and at run time based on how faults occur we can decide to use the proper set of ρ_i values. However, we do not use this method as the fault-free state is the most probable state and hence is the most prominent state from the viewpoint of average energy consumption [6], [8]. Therefore, in the proposed technique we use the ρ_i values that are calculated for the fault-free case. However, if a fault occurs at run-time, we temporarily do not use the ρ_i values that are calculated offline (as they are no longer valid) and from then on, we only use the proposed online management technique (Section 4.3) to allocate pseudo-dynamic slacks. From the beginning of the next frame we again use the ρ_i values that are calculated offline.

4.3 Online Energy Management

Let x_i be the slack (including the pseudo-dynamic and static slacks) that is allocated to a task T_i at design time using the offline part of our energy-management (Section 4.2). When DVS is used, the task worst-case execution time increases from W_i+c_i to $(W_i+c_i)/\rho_i$. On the other hand, as we exploit the slack x_i by DVS, we can also say that the task worst-case execution time increases from W_i+c_i to $W_i+c_i+x_i$. This implies that we have:

$$x_i=\frac{W_i+c_i}{\rho_i}-(W_i+c_i) \tag{20}$$

Indeed, after calculating the ρ_i values by solving the offline optimization problem at design time, we obtain the slack x_i (including pseudo-dynamic and static slacks) that we allocate to a task T_i using (20). At run-time, for each task T_i, the total slack time SL_i which is available to the task can be written as:

$$SL_i=x_i+DS_j \tag{21}$$

where x_i is the slack time which has been calculated offline in Section 4.2 (including both pseudo-dynamic and static slacks), and DS_j is the dynamic slack which has

been left over by the previous task (the task T_j) in the indispensable phase due to early completion at run-time. Since SL_i is the whole slack time which is available to the task T_i, the scaled supply voltage $\rho_i V_{\max}$ which is assigned to the task, must not prolong its worst-case execution time beyond the time $W_i+c_i+SL_i$, i.e., we require:

$$\frac{W_i + c_i}{\rho_i} \le W_i + c_i + SL_i \tag{22}$$

Clearly the proposed online energy management must take into account the time-constraint given by (22). Another important constraint that must be taken into account is for guaranteeing reliability. Let ε_i be the minimum value of ρ_i that does not cause the system reliability falls below the required level. Clearly we require:

$$\rho_i \ge \varepsilon_i \tag{23}$$

In the proposed online energy manager, as DVS-enabled processors usually have discrete voltage/frequency levels (Section 5), we always select the smallest value of ρ_i among the set of possible ρ_i values that satisfies both the Inequalities (22) and (23). In order to be able to check Inequalities (22) and (23) at run-time we need to have SL_i and ε_i values at run-time. To calculate the slack time SL_i (given by (21)) at run-time, note that x_i values have been calculated offline and stored to be used at run-time. Also the dynamic slack time DS_i which is obtained from the task T_i can be easily calculated at run-time as follows. When DVS is used for the first task (T_1), the actual execution time of the task is $(t_1+c_1)/\rho_1$. Since all the slack time which is available to T_1 is x_1, the maximum time which is available for executing T_1 is $W_1+c_1+x_1$ therefore the dynamic slack which is obtained from T_1 is:

$$DS_1 = (W_1 + c_1 + x_1) - \frac{t_1 + c_1}{\rho_1} \tag{24}$$

For the remaining tasks (T_i, $2 \le i \le n$), the maximum available time is $W_i+c_i+x_i+DS_j$ (where DS_j is the dynamic slack which has been left over by the task T_j which is the task that is finished just before starting the task T_i). Therefore, we can write:

$$DS_i = (W_i + c_i + x_i + DS_j) - \frac{t_i + c_i}{\rho_i} \tag{25}$$

At the end of each task T_i, we can use (25) (except for the first task that we use (24)) to calculate DS_i at run-time. It can be seen from (25) that to calculate the dynamic slack DS_i, we need to know $W_i+c_i+x_i$, DS_j, and $(t_i+c_i)/\rho_i$. The parameter $W_i+c_i+x_i$ is known at design time, and hence it can be calculated offline and stored to be used at run-time. DS_j, is the dynamic slack obtained from the task T_j (which is the task that is finished just before starting the task T_i), and is already calculated at the end of the T_j. $(t_i+c_i)/\rho_i$ is the actual execution time of the task T_i (including the result comparison time), and when the task finishes, its execution time can be easily calculated using the internal system clock (as this execution time is the difference between the start time and finish time of the task). In short, at the end of each task T_i, the dynamic slack time DS_i, can be calculated at run-time with very low overhead as its online calculation only requires a few subtraction and addition operations. The minimum possible value of ρ_i that does not cause the system reliability falls below the required level (i.e. ε_i values) can also be calculated offline at design time. To do this we can solve the optimization problem of (19), but without considering the constraint $c1$. This is because the constraint $c1$ is used to consider time constraints, but to calculate ε_i values we want to know which values of ρ_i can guarantee the required level of reliability regardless of time constraints.

5 Evaluation and Disscutions

Experiments in this paper were conducted based on the power model of the Intel PXA270 processor [21]. This processor can operate at different voltage levels in the range of 0.85-1.55V, and the corresponding frequencies vary from 13MHz to 624MHz. The energy consumption for active cores is calculated by (8) where P_S and P_D (that are respectively the static and dynamic power consumption of the system when operating at the maximum voltage and frequency) are 925mW and 260mW respectively [21]. Also, the Intel PXA270 processor has a low power sleep mode with 0.1014mW of idle power consumption. We considered that when a core is disabled or is temporarily unused, it enters the sleep mode and only consumes the idle power. We modified the tool MEET [22] to profile execution time and energy consumption while using DVS based on the power model of Intel PXA270. Like the works [5], [6], [13], [16], we performed system-level reliability simulation where the reliability was calculated by (15) and expressed in terms of *application probability of failure* PoF_{app} (i.e. $PoF_{app}=1-R_{app}$). The fault rate was modelled using (9) under the parameters $\lambda_0=10^{-6}$ faults/s and d=3 [5], [6]. Therefore, the fault rate varies between 10^{-6} faults/s and 10^{-3} faults/s, corresponding to the maximum and minimum voltage levels.

Previous research works on reliable real-time systems that do not use NMR rely on fault-detection mechanisms [5], [6], [7], [8], [11]. However, they have usually overlooked the overhead and fault coverage of detection mechanisms. Indeed, they usually do not consider any specific detection mechanism and simply assume that a detection mechanism with perfect fault coverage is part of the tasks (e.g., [5], [6], [7]). However, to provide fair comparisons, we need to include a real fault-detection mechanism in any implementation of previous works which is used in our comparisons. To do this, we considered that the previous works use fault-detection mechanisms included in their tasks (i.e., software fault-detection mechanisms). We conducted a set of experiments to investigate the energy and execution time overheads of the software fault-detection mechanisms that can be used for previous works. To consider effect of fault-detection mechanisms on energy and reliability we used two types of software fault-detection mechanisms in the implementations of previous works that were used in our comparisons:

1. *Heavy fault-detection mechanisms* (called HFD): with high fault-detection overheads but relatively high fault coverage. For this case we assumed that the system uses multiple fault-detection mechanisms based on code and data redundancy, arithmetic

TABLE 1
TIME (T) AND ENERGY (E) OVERHEADS OF HEAVY FAULT-DETECTION (HFD) AND LIGHT FAULT-DETECTION (LFD).

	No Fault-Detection		HFD		LFD		Overhead (%)			
							HFD		LFD	
Benchmark	T(ms)	E(mJ)	T(ms)	E(mJ)	T(ms)	E(mJ)	T	E	T	E
QuickSort	885	1005	1730	1937	1189	1325	95.4	92.9	34.4	31.9
BitCounts	339	385	563	656	438	484	66.0	70.2	29.2	25.8
BasicMath	960	1093	1802	2059	1282	1432	87.7	88.5	33.5	31.1
SusanSmooth	630	729	1138	1306	823	952	80.6	79.2	30.6	30.6
SusanCorners	157	180	305	342	184	204	94.5	90.7	17.2	13.9
SusanEdges	146	167	279	318	186	209	91.0	90.8	27.4	25.2

code, consistency check, and control flow checking [9], [10], [23], [24], [25], [26] to achieve high fault coverage for different fault types.

2. *Light fault-detection mechanisms* (called LFD): with relatively low fault-detection overheads and also low fault coverage. For this case we assumed that the system uses fewer mechanisms to reduce the fault-detection overhead with the cost of decreased detection coverage [26].

Table 1 shows the time and energy overheads that the software fault-detection mechanisms impose (assuming that we use the supply voltage 1.55V). To measure the overheads the applications were selected from the MiBench [27] benchmarks. It should be noted that while both time and energy overheads of software fault-detection mechanisms are lower than the overhead of modular redundancy with result comparison (majority voting), the fault coverage of software mechanisms is not sufficiently high, unlike majority voting that provides high fault masking [9], [10], [23], [24], [26]. Furthermore, these software fault-detection mechanisms are application-specific so that each task requires its specific detection mechanism [9], [10], [25], [26], while result comparison and majority voting are general and can be used for any type of tasks without requiring any hardware modification or redesign [9], [10], [25].

To evaluate the effectiveness of the proposed NMR technique (which we call it LE-NMR), we compared LE-NMR with a recent work (proposed in [5]). To provide a fair comparison, for both the implementations of LE-NMR and the system of [5], we assumed that both use the same level of task replication, i.e., when we consider an NMR with N copies for each task, we also considered that the system of [5] has N-1 backups for each task (i.e., again N copies for each task) to achieve fault tolerance. In addition, the system of [5] requires a fault-detection mechanism to determine if a backup task must be executed or not. Like most of the previous works, [5] has not addressed any fault-detection mechanism, but we considered that the tasks that are scheduled in the system of [5] use task-specific software mechanisms for fault-detection. To do this, we considered implementations of [5] where the tasks included heavy fault-detection mechanisms (called [5]-HFD) and light fault-detection mechanisms (called [5]-LFD). We also considered in our experiments an implementation of conventional NMR, called CNMR, where we do not use the two phases indispensable and on-demand. In conventional NMR, all N copies of each task are executed in parallel (assuming that enough cores are available) and the static slack time is only used for energy reduction.

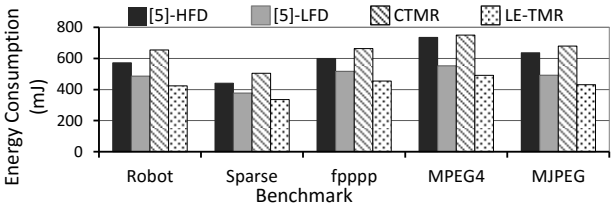

Fig. 6. Energy consumption of LE-TMR, [5]-HFD, [5]-LFD, and CTMR when running the practical applications.

It should be noted that there are various techniques to achieve low-energy fault-tolerance in real-time systems (e.g., [6], [7], [8], [11], [13], [16], [17], [18]) and it is beyond the scope of this paper to compare the proposed technique with all these various techniques. The main reason to choose the technique of [5] for the comparison is that it is a recent work with similar conditions to the proposed technique, e.g., hard real-time constraints, the use of DVS, and the frame-based application model with task precedence constraints (a set of dependent tasks with a global deadline) running on multi-core platforms. Also, it is noteworthy that for many of the previous works it is not meaningful to compare them with the proposed technique because they considerably differ from ours in application model (e.g., chain of dependent tasks in [6], single-task frame in [13], periodic tasks in [11], [17], and independent tasks in [18]).

To compare LE-NMR with [5] and conventional NMR, we used both synthetic and practical application task graphs. To do this, we used the task graph generator TGFF [28] and the Standard Task Graph set (STG) [29]. The STG benchmark suite contains both synthetic task graphs and practical real-time application task graphs including robot control, SPEC fpppp and a sparse matrix solver. We also conducted experiments on two other real-world applications: MPEG4 decoder and MJPEG encoder (their task graphs can be found in [15]).

Fig. 6 and Table 2 show, respectively, the energy consumption and probability of failure for [5]-HFD, [5]-LFD, CTMR, and the proposed LE-TMR when running the practical applications. The three following interesting observations can be made from Fig. 6 and Table 2:

1. LE-TMR not only provides more energy saving (in average 28% and up to 33%) as compared to [5]-HFD, but also has a less probability of failure, i.e., LE-TMR is more reliable.
2. Although LE-TMR provides relatively less energy saving (in average 12%) as compared to [5]-LFD, LE-TMR has a far less probability of failure (it provides much higher reliability).
3. LE-TMR provides more energy saving (in average 34%) as compared to CTMR, while provides almost the same level of reliability (Table 2).

TABLE 2
PROBABILITY OF FAILURE (POF) FOR LE-TMR, [5]-HFD, [5]-LFD, AND CTMR WHEN RUNNING THE PRACTICAL APPLICATIONS.

Application	[5]-HFD	[5]-LFD	CTMR	LE-TMR
Robot	$10^{-3.45}$	$10^{-2.4}$	$10^{-9.64}$	$10^{-9.67}$
Sparse	$10^{-3.52}$	$10^{-2.32}$	$10^{-9.52}$	$10^{-9.53}$
Fpppp	$10^{-3.64}$	$10^{-2.45}$	$10^{-9.45}$	$10^{-9.48}$
MPEG4	$10^{-3.22}$	$10^{-2.45}$	$10^{-9.42}$	$10^{-9.44}$
MJPEG	$10^{-3.31}$	$10^{-2.74}$	$10^{-9.86}$	$10^{-9.82}$

Another set of experiments were conducted in order to analyze how the parallelism degree of task graphs affects the effectiveness of our technique. To do this, synthetic task graphs were generated. It is known that for task graphs with the same number of tasks, the height of the task graph can be used to take the parallelism degree into account [39]. Based on this, in the experiment three classes of task graphs with different parallelism degrees were considered. Let n be the number of nodes (tasks) in a task graph and h be the task graph height. Clearly h can vary between 1 and n, therefore the three classes of considered task graphs are: *i*) task graphs with $1 \le h \le n/3$ (called task graphs with high parallelism degree), *ii*) task graphs with $n/3 \le h \le 2n/3$ (called task graphs with medium parallelism degree), and *iii*) task graphs with $2n/3 \le h \le n$ (called task graphs with low parallelism degree).

The tasks of the synthetic task graphs were randomly selected from the MiBench benchmarks and the time and energy overheads of the detection mechanisms for these tasks were taken from Table 1. The worst-case and actual execution times (W_i and t_i) of the tasks were generated randomly [4], [5], [6]. The worst-case execution times were uniformly distributed between 10ms and 100ms. However, as the actual execution times for each task may have different probability distributions, like works [4], [5], [6], in our experiments, we considered the uniform, normal, or exponential distributions for the actual execution time t_i and each task T_i was executed only for the duration of t_i. In the experiment, it was assumed that task graphs with 20, 50, 100, 200, 500 tasks with different parallelism degrees were executed on multi-core systems with 2, 4, 8, 16 and 32 cores. Each case (e.g., a task graph with 50 tasks on an 8-core system) was simulated for 1500 times with different parameters (i.e., tasks worst-case and actual execution times and application deadline) and the average results are reported in Figs 7 and 8. These figures show the energy consumption and probability of failure (*PoF*) for LE-TMR, [5]-HFD, [5]-LFD, and CTMR.

These observations can be made from Figs 7 and 8:

1. It can be seen from Fig. 7 that, for all the four systems, as the parallelism degree of task graphs increases, the energy consumption decreases. However, the energy consumption of LE-TMR is always less than the other three systems.
2. While the energy consumption of all the four systems decreases with the increase in the task graph parallelism degree, LE-TMR favours more energy reduction as compared to the others. For example, assuming we have 16 cores, as the task graph parallelism degree increases from low (Fig. 7a) to high (Fig. 7c), the energy consumption of LE-TMR reduces from 1698mJ to 1231mJ (28% reduction), while the energy consumption of CTMR reduces from 2132mJ to 1944mJ (9% reduction).
3. As Fig. 8 shows, LE-TMR has a far less probability of failure than the implementations of [5], even compared to the implementation of [5] that uses heavy fault-detection mechanisms ([5]-HFD). This is because of the superiority of majority voting (NMR) in covering the faults as compared to fault-detection mechanisms [9], [10], [24], [25], [26].
4. While LE-TMR provides almost the same reliability as CTMR (Fig. 8), LE-TMR consumes much less energy than CTMR (Fig. 7) mainly because of the more sophisticated energy-management technique that LE-TMR uses.

We also compared LE-NMR with N=5 and N=7 (i.e., LE-5MR and LE-7MR respectively) with [5] and the conventional NMR. The experiments demonstrate that LE-NMR completely outperform [5] from both the energy-

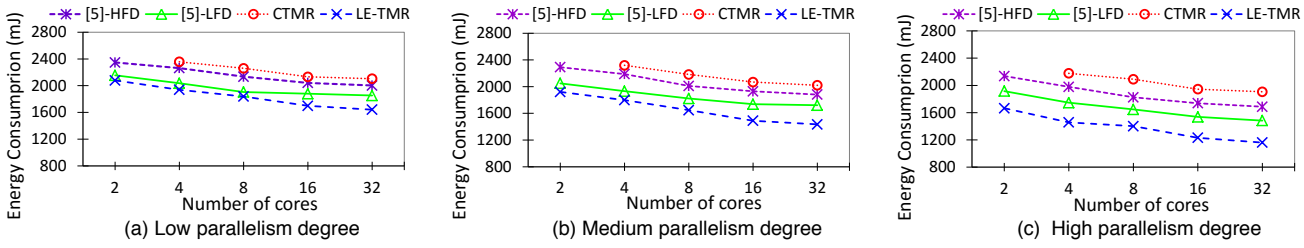

Fig. 7. Energy consumption of LE-TMR, [5]-HFD, [5]-LFD, and CTMR when running the synthetic applications.

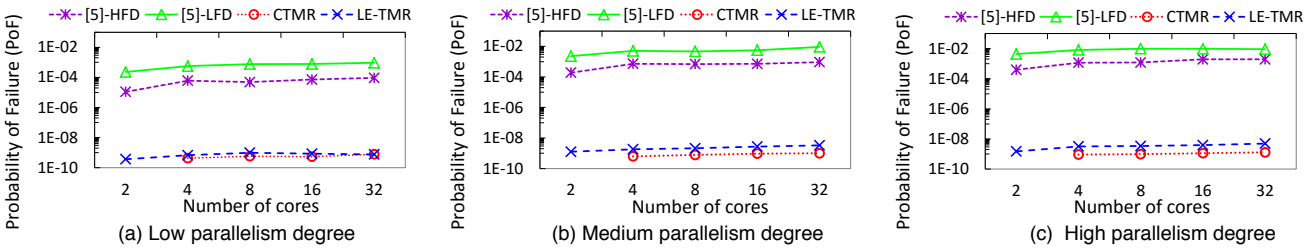

Fig. 8. Probability of failure (PoF) in logscale for LE-TMR, [5]-HFD, [5]-LFD, and CTMR when running the synthetic applications.

consumption and reliability viewpoints. LE-5MR and LE-7MR provide in average respectively 19% (up to 22%), and 17% (up to 21%), and 31% (up to 36%) energy saving as compared to the corresponding implementations of [5] and the conventional NMR. An interesting observation from the experiments is that none of the implementations of [5] can achieve high reliability (the implementations of [5] cannot achieve a probability of failure less than 10^{-3}) while LE-NMR satisfies the required reliability level of safety-critical applications as they may require probability of failure be less than 10^{-9} [6], [9], [10]. This is because the implementations of [5] use software fault-detection mechanisms while the fault coverage of these mechanisms is not sufficiently high [9], [10], [25], [26], unlike LE-NMR that uses majority voting that provides high fault masking [9], [10], [24], [25], [26].

6 Conclusion

In this paper, we described how multi-core platforms can be exploited to achieve high reliability with low energy-overhead for hard real-time systems. To do this, we proposed a low-energy NMR (we called it LE-NMR). To achieve energy saving in LE-NMR we exploit two main strategies. First, we adopt a two-phase NMR technique, where usually (when no fault occurs) only one phase is executed, resulting in a considerable energy saving compared with conventional NMR systems. Second, to achieve further energy saving, we use DVS. In developing the proposed LE-NMR technique, we have considered two new concepts: *i*) Block-partitioned scheduling and *ii*) Pseudo-dynamic slack management. To exploit available slacks in the system by DVS, we have developed an energy-management technique with offline and online parts. The offline part at design time derives and solves an optimization problem to exploit the slacks that are known at design time (i.e., static and pseudo-dynamic slacks), and to assign dynamic slacks to the tasks at run-time, the online part is used. The experimental results show that LE-NMR provides up to 34% energy saving and is 6 orders of magnitude higher reliable as compared to an implementation of a recent previous work.

Acknowledgment

Mohammad Salehi and Alireza Ejlali acknowledge Research Vice-Presidency of Sharif University of Technology for funding this work under grant no. G930827. Bashir M. Al-Hashimi acknowledges the EPSRC (UK), for funding this work in part under grant PRiME EP/K034448/1.
Experimental data used in this paper can be found at DOI:10.5258/SOTON/397799
(http://dx.doi.org/10.5258/SOTON/397799).

References

[1] J. Henkel, V. Narayanan, S. Parameswaran, and J. Teich, "Run-Time Adaption for Highly-Complex Multi-Core Systems," *Proc. Ninth IEEE/ACM/IFIP Int'l Conf. Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)*, pp. 1-8, Sept. 29 2013-Oct. 4 2013, doi: 10.1109/CODES-ISSS.2013.6659000.

[2] W.Y. Lee, "Energy-Efficient Scheduling of Periodic Real-Time Tasks on Lightly Loaded Multicore Processors," *IEEE Trans. Parall. Distr. Syst.*, vol. 23, no. 3, pp. 530-537, March 2012, doi: 10.1109/TPDS.2011.87.

[3] A. Munir, S. Ranka, and A. Gordon-Ross, "High-Performance Energy-Efficient Multicore Embedded Computing," *IEEE Trans. Parall. Distr. Syst.*, vol. 23, no. 4, pp. 684-700, April 2012, doi: 10.1109/TPDS.2011.214.

[4] H. Su, D. Zhu, and D. Mosse, "Scheduling Algorithms for Elastic Mixed-Criticality Tasks in Multicore Systems," *Proc. IEEE 19th Int'l Conf. Embed. Real-Time Computing Syst. and Applications (RTCSA'13)*, pp. 352-357, Aug. 2013, doi: 10.1109/RTCSA.2013.6732239.

[5] Y. Guo, D. Zhu, and H. Aydin, "Reliability-Aware Power Management for Parallel Real-Time Applications with Precedence Constraints," *Proc. Int'l Green Computing Conf. and Workshops (IGCC)*, pp.1-8, July 2011, doi: 10.1109/IGCC.2011.6008562.

[6] A. Ejlali, B.M. Al-Hashimi, and P. Eles, "Low-Energy Standby-Sparing for Hard Real-Time Systems," *IEEE Trans. Comput.-Aid. Des. Integr. Circuits Syst.*, vol. 31, no. 3, pp. 329-342, March 2012, doi: 10.1109/TCAD.2011.2173488.

[7] M.K. Tavana, M. Salehi, and A. Ejlali, "Feedback-Based Energy Management in a Standby-Sparing Scheme for Hard Real-Time Systems," *Proc. IEEE 32nd Real-Time Systems Symposium (RTSS'11)*, pp. 349-356, Nov. 2011-Dec. 2011, doi: 10.1109/RTSS.2011.39.

[8] R. Melhem, D. Mosse, and E. Elnozahy, "The interplay of power management and fault recovery in real-time systems," *IEEE Trans. Comput.*, vol. 53, no. 2, pp. 217-231, Feb 2004, doi: 10.1109/TC.2004.1261830.

[9] D.K. Pradhan, *Fault-tolerant Computer System Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, 1996.

[10] I. Koren, and C.M. Krishna, *Fault-Tolerant Systems*. Morgan Kaufmann, Elsevier, San Francisco, CA, 2007.

[11] M.A. Haque, H. Aydin, and D. Zhu, "Energy-Aware Standby-Sparing Technique for Periodic Real-Time Applications," *Proc. IEEE 29th Int'l Conf. Comput. Design (ICCD'11)*, pp. 190-197, Oct. 2011, doi: 10.1109/ICCD.2011.6081396.

[12] T.D. Burd, T.A. Pering, A.J. Stratakos, and R.W. Brodersen, "A dynamic voltage scaled microprocessor system," *IEEE J. Solid-State Circuits (JSSC)*, vol. 35, no. 11, pp. 1571–1580 , Nov. 2000, doi: 10.1109/4.881202.

[13] D. Zhu, R. Melhem, D. Mosse, and E. Elnozahy, "Analysis of an Energy Efficient Optimistic TMR Scheme," *Proc. Tenth Int'l Conf. Parall. and Distr. Syst. (ICPADS'04)*, pp. 559-568, July 2004, doi: 10.1109/ICPADS.2004.1316138.

[14] J. Cong and K. Gururaj, "Energy Efficient Multiprocessor Task Scheduling under Input-dependent Variation," *Proc. Design, Automation and Test in Europe Conf. and Exhibition (DATE'09)*, pp. 411-416, April 2009, doi: 10.1109/DATE.2009.5090698.

[15] X. Qi and D. Zhu, "Energy efficient block-partitioned multicore processors for parallel applications," *J. Comput. Science Tech.*, vol. 26, no. 3, pp. 418–433, May 2011, doi: 10.1007/s11390-011-1144-5.

[16] X. Qi, D. Zhu, and H. Aydin, "Global scheduling based reliability-aware power management for multiprocessor real-time systems," *J. Real-Time Syst.*, vol. 47, no. 2, pp. 109-142, March 2011, doi: 10.1007/s11241-011-9117-x.

[17] M.A. Haque, H. Aydin, and D. Zhu, "Energy-Aware Task Replication to Manage Reliability for Periodic Real-Time Applications on Multicore Platform," *Int'l Green Computing Conf. (IGCC'13)*, pp. 1-11, June 2013, doi: 10.1109/IGCC.2013.6604518.

[18] D. Zhu, R. Melhem, and D. Mosse, "Energy Efficient Redundant Configurations for Real-Time Parallel Reliable Servers," *J. Real-Time Syst.*, vol. 41, no. 3, pp. 195-221, April 2009, doi: 10.1007/s11241-009-9067-8.

[19] E.G. Coffman and R.L. Graham, "Optimal Scheduling for Two-Processor Systems," *Acta Informatica*, vol. 1, no. 3, pp. 200-213, 1972, doi: 10.1007/BF00288685.

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

[21] Intel Corp., "Intel® PXA270 Processor," Available: http://www.intel.com.

[22] M. Bazzaz, M. Salehi, and A. Ejlali, "An Accurate Instruction-Level Energy Estimation Model and Tool for Embedded Systems," *IEEE Trans. Instrum. Meas.*, vol. 62, no. 7, pp. 1927-1934, July 2013, doi: 10.1109/TIM.2013.2248288.

[23] N. Oh, P.P. Shirvani, and E.J. McCluskey, "Control-Flow Checking by Software Signatures," *IEEE Trans. Reli.*, vol. 51, no. 1, pp. 111-122, Mar 2002, doi: 10.1109/24.994926.

[24] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "Experimental Evaluation of Time-Redundant Execution for a Brake-by-wire Application," *Proc. Int'l Conf. Dependable Syst. and Networks (DSN'02)*, pp. 210-215, 2002, doi: 10.1109/DSN.2002.1028902.

[25] K.S. Yim, V. Sidea, Z. Kalbarczyk, D. Chen, and R.K.A. Iyer, "A Fault-Tolerant Programmable Voter for Software-Based N-Modular Redundancy," *Proc. IEEE Aerospace Conf.*, pp. 1-20, March 2012, doi: 10.1109/AERO.2012.6187253.

[26] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic Soft Error Reliability on the Cheap," *Proc. 15th Architectural Support for Programming Languages and Operating Syst. (ASPLOS'10)*, pp. 385-396, 2010, doi: 10.1145/1736020.1736063.

[27] M.R. Guthaus, J. S. Ringenberg, and D. Ernst, "MiBench: A free, commercially representative embedded benchmark suite," *Proc. IEEE Int'l Workshop on Workload Characterization (WWC-4)*, pp. 3-14, Dec. 2001, doi: 10.1109/WWC.2001.990739.

[28] D. Rhodes and R. Dick, "TGFF: Task Graphs for Free," *Proc. 6th Int'l Workshop on Hardware/Software Codesign (CODES/CASHE '98)*, pp. 97-101, Mar 1998, doi: 10.1109/HSC.1998.666245.

[29] T. Tobita and H. Kasahara, "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms," *J. Scheduling*, vol. 5, no. 5, pp. 379–394, Sep. 2002, doi: 10.1002/jos.116.

[30] S.-H. Kang, H. Yang, K. Sungchan, I. Bacivarov, S. Ha, and L. Thiele, "Reliability-aware mapping optimization of multi-core systems with mixed-criticality," *Proc. Design, Automation and Test in Europe Conf. and Exhibition (DATE'14)*, pp. 1-4, March 2014, doi: 10.7873/DATE.2014.340.

[31] J.C. Smolens, B.T. Gold, J. Kim, B. Falsafi, J.C. Hoe, A.G. Nowatzyk, "Fingerprinting: Bounding Soft-Error-Detection Latency and Bandwidth", *IEEE Micro*, vol. 24, no. 6, pp. 22-29, Nov./Dec. 2004, doi:10.1109/MM.2004.72.

[32] J. Lee, B. Yun, and K. G. Shin, "Reducing Peak Power Consumption in Multi-Core Systems without Violating Real-Time Constraints," *IEEE Trans. Parall. Distr. Syst.*, vol. 25, no. 4, pp. 1024-1033, April 2014, doi: 10.1109/TPDS.2013.131.

[33] S. Saha, J. S. Deogun, Y. Lu, "Adaptive energy-efficient task partitioning for heterogeneous multi-core multiprocessor real-time systems," *Int'l Conf. High Performance Computing and Simulation (HPCS)*, pp. 147-153, July 2012, doi: 10.1109/HPCSim.2012.6266904.

[34] S. Rehman, F. Kriebel, M. Shafique, J. Henkel, "Reliability-Driven Software Transformations for Unreliable Hardware," *IEEE Trans. Comput.-Aid. Des. Integr. Circuits Syst.*, vol. 33, no. 11, pp. 1597-1610, Nov. 2014, doi: 10.1109/TCAD.2014.2341894.

[35] T. Miller, N. Surapaneni, R. Teodorescu, "Flexible Error Protection for Energy Efficient Reliable Architectures," *22nd Int'l Symp. Comput. Arch. and High Performance Comput. (SBAC-PAD)*, pp. 1-8, Oct. 2010, doi: 10.1109/SBAC-PAD.2010.37.

[36] R. Jeyapaul, F. Hong, A. Rhisheekesan, A. Shrivastava, K. Lee, "UnSync-CMP: Multicore CMP Architecture for Energy-Efficient Soft-Error Reliability," *IEEE Trans. Parall. Distr. Syst.*, vol. 25, no. 1, pp. 254-263, Jan. 2014, doi: 10.1109/TPDS.2013.14.

[37] R. Vadlamani, J. Zhao, W. Burleson, and R. Tessier, "Multicore soft error rate stabilization using adaptive dual modular redundancy," *Proc. Design, Automation and Test in Europe Conf. and Exhibition (DATE'10)*, pp. 27-32, March 2010, doi: 10.1109/DATE.2010.5457242.

[38] T. Wei, P. Mishra, K. Wu, H. Liang, "Fixed-Priority Allocation and Scheduling for Energy-Efficient Fault Tolerance in Hard Real-Time Multiprocessor Systems," *IEEE Trans. Parall. Distr. Syst.*, vol. 19, no. 11, pp. 1511-1526, Nov. 2008, doi: 10.1109/TPDS.2008.127.

[39] H. Topcuoglu, S. Hariri, M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parall. Distr. Syst.*, vol. 13, no. 3, pp. 260-274, Mar 2002, doi: 10.1109/71.993206.

Mohammad Salehi received the M.S. degree in computer engineering from Sharif University of Technology, Tehran, Iran, in 2010, where he is currently working toward the Ph.D. degree in computer engineering. From 2014 to 2015, he was a visiting researcher in the Chair for Embedded Systems CES, Karlsruhe Institute of Technology (KIT), Germany. His research interests include low-power design of embedded systems, multi-/many-core systems with a focus on dependability/reliability, low power, and the tradeoff between the fault tolerance and energy efficiency in real-time systems.

Alireza Ejlali is an Associate Professor of Computer Engineering at Sharif University of Technology, Tehran, Iran. He received a Ph.D. degree in computer engineering from Sharif University of Technology in 2006. From 2005 to 2006, he was a visiting researcher in the Electronic Systems Design Group, University of Southampton, UK. In 2006 he joined Sharif University of Technology as a faculty member in the department of computer engineering and from 2011 to 2015 he was the director of Computer Architecture Group in this department. His research interests include low power design, real-time embedded systems, and fault-tolerant embedded systems.

Bashir M. Al-Hashimi (M'99–SM'01–F'09) is a Professor of computer engineering, Dean of Faculty of Sciences and Engineering, and the Director of the Pervasive Systems Center, University of Southampton, U.K. He is ARM Professor of computer engineering and the Co-Director of the ARM-ECS Research Center. His research interests include methods, algorithms, and design automation tools for low-power design and test of embedded systems.