Microprocessors and Microsystems 38 (2014) 88–97

Contents lists available at ScienceDirect

Microprocessors and Microsystems

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

DFTS: A dynamic fault-tolerant scheduling for real-time tasks in multicore processors

Mohammad H. Mottaghi [a], Hamid R. Zarandi [a,b,*]

[a] Department of Computer Engineering and Information Technology, Amirkabir University of Technology (Tehran Polytechnic), Iran
[b] School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran

ARTICLE INFO

Article history:
Received 26 May 2013
Revised 10 November 2013
Accepted 30 November 2013
Available online 8 December 2013

Keywords:
Real-time systems
Dynamic scheduling
Fault tolerance
Multicore processors

ABSTRACT

This paper presents a dynamic scheduling for real-time tasks in multicore processors to tolerate single and multiple transient faults. The scheduling is performed based on three important issues: (1) current released tasks, (2) current available processor cores, and (3) consideration of the number of faults and their occurrences. Using tasks utilization along with a defined criticality threshold in the proposed scheduling method, current ready tasks are divided into critical- and noncritical ones. Based on whether a task is critical or noncritical, an appropriate fault-tolerance policy is exploited. Moreover, scheduling decisions are made to fulfill two key goals: (1) increasing scheduling feasibility and (2) decreasing the total tasks execution time. Several simulation experiments are carried out to compare the proposed method with two well-known methods, called checkpointing with rollback recovery and hardware replication. Experimental results reveal that in the presence of multiple transient faults, the feasibility rate of the proposed method is considerably higher than the other well-known fault-tolerance methods. Moreover, the average timing overhead of this method is lower than the traditional methods.

1. Introduction

Real-time systems have been extensively used in many human applications like sensor networks, satellites, unmanned vehicles, and personal mobile equipment [1]. Time constraints, energy efficiency, and throughput are emerged as important criteria in the design process of such systems [1,2].

Multicore architectures, which integrate several processing units (known as cores) into a single chip, play an important role in the development of real-time systems [1,2]. The reason is that multicore architectures present several advantages compared to single core architectures such as: (1) higher throughput with the same clock frequency [1], (2) linearity of power consumption over the throughput for multicore architectures [1], (3) efficient utilization of processor cores [3], and (4) high performance per cost [4]. ARM MPCore [5] and IBM Cell [6] are two examples of synthesizable multicore processors employed in the real-time embedded applications [7].

Multicore processors can be classified as homogenous or heterogeneous [2,8]. The former is composed of several processing cores, which are similar regarding their internal architecture. The latter is composed of processing cores with different internal architectures. As stated in [8], most of the existing multicore processors are homogeneous. If multicore technology continues growing, the capability of utilizing intra-task parallelism and performing computation-intensive real-time tasks will also be increased [2].

The main concern of multicore processors is how to manage tasks in order to utilize the processing cores effectively [9]. The main role of schedulers in an operating system is to keep all processing cores busy during execution of real-time tasks to improve total execution time. This will be complex if executing tasks are logically correlated to each other, and they need to use shared resources. To avoid any contentions in the shared resources, schedulers should be aware of multicore architectures, shared resources topology, resource requirements of tasks, and inter-relationships between the tasks [10].

Due to lowering voltages [11], high-energy particle strikes [12], voltage fluctuations [11], shrinking technology features such as increasing the die size [11], transistor density growth [13], and increasing number of core instances [13], multicore architectures are vulnerable to single/multiple transient faults. To use these architectures in safety-critical applications, correct functionality and meeting the timing constrains are essential even in the presence of faults. Therefore, improving the fault-tolerant property of these systems is critical. However, it is important to consider timing constraints of the tasks when we use fault-tolerance mechanisms. This means that we should apply error detection and recovery mechanism in a manner of no task deadlines are missed [12].

* Corresponding author at: Department of Computer Engineering and Information Technology, Amirkabir University of Technology (Tehran Polytechnic), Iran. Tel.: +98 2164542702.
E-mail addresses: mh_mottaghi@aut.ac.ir (M.H. Mottaghi), h_zarandi@aut.ac.ir (H.R. Zarandi).

0141-9331/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.micpro.2013.11.013

An efficient method to improve fault tolerance in any system is to employ redundancy [14,15]. Redundancies exploited in the real-time systems can be classified into two categories [14,16]: (1) hardware-based redundancy and (2) time-based redundancy. Hardware-based redundancy methods such as task replication [11], stand-by-sparing [17] (hot-, warm- and cold ones), DMR (Dual Modular Redundancy) [16], and TMR (Triple Modular Redundancy) [16] attempt to tolerate transient faults in the real-time systems by copy-executions of each original task on another separated hardware. Although these methods are effective to tackle spatial multiple faults which are not correlated in two distinct processing units, they impose significant overheads in terms of hardware cost and power consumption.

Time-based redundancy methods such as check pointing with rollback recovery [18] and re-execution [15] try to cope with transient faults by serial copy-executions in the same hardware of original task. Therefore, these methods do not impose any hardware cost overhead and they are cost-effective compared to the hardware-based redundancy. However, due to extra executions on the same hardware, these methods are not effective to tolerate permanent faults or even transient faults whose durations are very long. Moreover, serial executions may cause the system fail to meet timing constraints.

Based on the previously mentioned discussion, it is necessary to have a method to tolerate multiple transient/permanent faults with low overhead in terms of hardware cost, power consumption and total execution time. Several research works optimize hardware cost, power consumption and time overhead: (1) some of related works optimize number of checkpoints, e.g., [14] and (2) some of them tune time intervals between checkpoints, e.g., [18]. Although these researches focus on reducing the execution time overhead, they are not successful in all cases since decision methods are made statically and it is not feasible to find a schedule due to time redundancy overhead, (3) decreasing hardware cost and power consumption in the hardware-based methods are addressed in [1,17]. Several methods benefit from low-power techniques like DPM (Dynamic Power Management), DVS (Dynamic Voltage Scaling) [14,17]; however, utilizing DPM and DVS decreases reliability of the systems [17].

In this paper, a dynamic method is proposed to schedule real-time tasks in multicore processors. We proposed a parameter namely "task criticality" which is based on two other parameters: (1) "utilization" and (2) time of resource allocation to the tasks. These two parameters play key roles in the proposed method since there are many cases where two previous methods provide worse feasibility rate compared to the proposed method. Here, the main contributions are summarized as follows:

- Tolerating both single and multiple transient faults, which may occur temporally or spatially.
- Enhancing scheduling feasibility compared to conventional methods.
- Decreasing the total task execution time, and therefore, reducing the time overhead of the proposed method over conventional methods.
- Utilizing hardware resources efficiently so that both hardware constraint and timing constraint will be met.

The proposed method tries to optimize the utilization of hardware-based redundancy and time-based redundancy in the multicore processors. The method assigns a criticality property to each task, and decides which fault-tolerant method will be a good candidate to tolerate maximum number of expected faults. In this paper, task replication and checkpointing with rollback recovery are utilized as hardware- and time-based redundancies, respectively.

Two main reasons for selection of task replication are as follows: (1) it is fault-tolerant like other hardware-based redundancy methods and (2) task replication does not require any comparator hardware, which is highlighted in other hardware-based redundancy methods [16]. Since no comparator hardware is used in multicore processors, task replication plays a key role in their architecture. The main reason to select checkpointing with rollback recovery is that checkpointing methods increases the probability of a task to be completed on time [18].

In order to evaluate the proposed method, many task sets are generated, scheduled, and mapped on a model of one quad-core processor. The experimental results show that, the scheduling feasibility rate of the proposed algorithm is higher than other fault-tolerant scheduling methods, i.e. replication and checkpointing with rollback recovery. Moreover, total execution time overhead of the proposed method is lower than the ones in replication and checkpointing with rollback recovery.

The rest of this paper is organized as follows. Section 2 describes the application model, hardware model, and fault model which are being used in this paper. In Section 3, a comparison between static and dynamic scheduling algorithms is carried out. Section 4 provides a brief description of fault-tolerance policies that we used in the paper. More details about the proposed method are presented in Section 5. The experimental results are described in Section 6, and finally, Section 7 concludes the paper.

2. System model

Suppose a set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ consists of n sporadic and non-preemptive independent real-time tasks. Non-preemptive tasks cannot be interrupted during their execution by other tasks; therefore, they will be executed until completion [19]. In order to handling system interrupts two schemes can be adopted: (1) based on previous related works like [20], handling system interrupts can be seen as a real-time task with a very early deadline in the system, so based on EDF scheduling, it will be scheduled as the highest priority task in our ready-list. However, it will not preempt other tasks executing in the system. (2) System interrupts are considered as preemptive tasks. In this scheme, the effects of interrupts can be modeled in our proposed method as a task which impose hardware and time overheads to the systems tasks and cores, and therefore, can be considered in the proposed scheduling method. Each task τ_i is modeled by a tuple (C_i, T_i, D_i) where C_i is the worst-case execution time of the task in a fault-free condition, T_i is the minimum inter-arrival time or period of the task, and D_i is the relative deadline of the task. Relative deadline is defined as the deadline time relative to the release time. Moreover, it is assumed that tasks have implicit deadlines, i.e., $T_i = D_i$ (all task deadlines are equal to their periods [19]). In the sporadic task model, each job of a task may arrive at any time instance once a minimum inter-arrival time has elapsed since the arrival of the previous job of the same task [19]. The release time (when τ_i enters the ready list) is denoted by R_i. The absolute deadline, AD_i of each task is also defined as the sum of R_i and D_i. Another parameter used in this paper is task utilization, U_i $(0 \leqslant U_i \leqslant 1)$ which is defined as the division of worst-case execution time by relative deadline,

$$U_i = \frac{C_i}{D_i} \tag{1}$$

Total utilization of an application denoted by U can be extracted by the summation of all task utilizations of the application,

$$U = \sum_{i=1}^{n} U_i \tag{2}$$

where n is the number of tasks in the application. When U_i is close to "1", the scheduler will not be capable of postponing the task execution, so that the task would be critical. According to this statement, we classify arriving tasks into two categories: (1) *critical* tasks and (2) *noncritical* tasks. Furthermore, in the presence of faults, time-consuming fault-tolerant methods such as checkpointing with rollback recovery will not be suitable for critical tasks. To decide on whether the task is *critical* or not, a threshold θ is used as follows:

$$Class(\tau_i) = \begin{cases} Noncritical & U_i < \theta \\ Critical & U_i \geqslant \theta \end{cases}$$

where $0 \leqslant \theta \leqslant 1$. The criticality threshold will be computed for each job of task entering to the ready list of scheduler as soon as idle resources are available. Section 5 describes how to compute this threshold.

2.1. Hardware model

In this paper, multicore platforms are considered as a set of M identical and homogeneous processing cores, $\boldsymbol{P} = \{P_1, P_2, \ldots, P_M\}$. Every task can be executed on each core of the processor. In multicore architectures, parallel tasks in an application can potentially be executed simultaneously. However, the number of such tasks may exceed the number of available processing cores. Therefore, task scheduling and core mapping is required to assign the parallel tasks to the available cores. It is assumed that an external scheduler globally monitors the released tasks in the system as well as the idle time of each processing core. The scheduler also decides on other factors, such as fault-tolerant policy assignment, fault-tolerant scheduling for the tasks in the ready list and mapping the tasks on the available processing cores.

2.2. Fault model

As most of failures in digital systems are due to transient faults [21], single and multiple transient faults are addressed in this paper. According to the numerous research works [14,21], Poisson distribution function can be a good estimation of fault occurrence in the time domain in hardware systems. The fault-arrival rate (λ) can be determined as a factor depending on environmental- and operating conditions. In harsh environments, λ (fault arrival rate) is in the range of 10^{-2} to 10^{2} per hour for old technologies [14]. For new technologies, due to low voltages, shrinking technology features such as increasing die size, transistor density growth, and increasing number of core instances, fault arrival rate will be higher [13]. For a given fault arrival rate, λ and a task execution interval, t the average number of expected faults is λt [14]. This paper uses the k-fault-tolerant model [18], which determines the maximum number of k faults to be tolerated for each task. Authors in [14] show that the value of k should be taken as a small multiple of λt, e.g. $2\lambda t \leqslant k \leqslant 3\lambda t$.

Similar to the previous works in the fault-tolerant real-time systems, it is assumed that a given transient fault affects only the task running on a specific core rather than the other cores [11,12] as transient faults are momentary, and the result of a task cannot be committed and propagated to other tasks unless it passes sanity checks and consistency checks [12]. Faults occur during task execution, checkpoint saving and fault-recovery states. As processing cores are homogeneous, fault distribution and fault arrival rate for each core are assumed to be the same for all cores.

2.3. Fault detection

Fault detection plays a key role in designing fault-tolerant systems. There are several methods for fault detection, presented at different abstraction levels. These methods can be classified in hardware and software parts of a system [22], some of which are employed in embedded systems, such as: (1) sanity and consistency check to verify the correctness of results at user level [12], (2) memory range violation and illegal opcode detection at OS level, (3) control-flow error detection, (4) CRC codes, (5) acceptance tests at software-level, and (6) hardware duplication with comparison at hardware-level [16]. It is worth mentioning that each of these methods is proposed to detect some special types of faults, hence a perfect method which is capable of detecting any arbitrary types of faults has not been presented [22].

In this paper, it is assumed that faults are detected at the end of each task or at checkpoint times using sanity and consistency checks [12]. In addition, other available hardware-based fault detection methods like division by zero, illegal opcode and traps are useful to find errors happened in program execution. However, any extra hardware dedicated for the proposed method is not suggested in this paper. Moreover, fault detection latency is assumed to be close to zero, while detection overhead is considered based on a fraction of worst-case execution time for each task [15].

3. Task scheduling in multicore processors

The problem of scheduling real-time tasks in a multicore processor is a NP-hard problem, so a number of heuristic solutions have been proposed to overcome its complication [1,2,19]. These solutions are mainly divided into two major categories: (1) the first approach called partitioned scheduling algorithms where all executions of a particular task take place in the same processor. (2) In the other approaches called "global scheduling algorithms", tasks are allowed to be executed on different processors [1].

In this paper, granularity of program code is assumed to "task". In other words, different tasks of the same program are modeled as two independent tasks. Besides, scheduler and task manager are two sub-modules in an OS (Operation System). It means that when a scheduler schedules tasks, task manager will run tasks based on the provided scheduling. All required operations in task manager will be done by kernel-level instructions [23,27,15]. Therefore, these actions are transparent from user or program code and no modifications are needed in the program code. So, it is not concerned in the proposed scheduling method.

3.1. Dynamic scheduling in the proposed method

Static scheduling methods can pre-compute an optimal schedule for an application providing that certain information is accessible during design-time [23]. This information includes execution time and memory referencing behavior of tasks.

Dynamic scheduling does not require the mentioned information during design-time, but instead it handles tasks at run-time. These methods put all tasks in a ready list, which is a shared queue, and schedule the highest priority task on any available processor [24]. As we study sporadic tasks with non-specific release time, using dynamic scheduling methods is inevitable [25].

3.2. Non-preemptive EDF scheduling

There are several dynamic priority scheduling algorithms [19] such as Earliest-Deadline-First (EDF), Least-Laxity (LL), Least-Slack-Time-First (LST), and Minimum-Laxity-First (MLF), where in each one tasks are prioritized and scheduled according to spe-

cific properties. However, it can be proven that EDF is an optimal algorithm for single-processor systems with a set of preemptive independent tasks [24]. EDF scheduling assigns higher execution priorities to the tasks with earlier absolute deadlines. The optimality of EDF breaks down on multiprocessor systems [26], especially when the tasks are non-preemptive. There is no online scheduling algorithm for sporadic tasks where the tasks have no common deadline [24].

In [27], it is shown that there might be task sets with the total utilization slightly greater than (and arbitrarily close to) "1", that cannot be scheduled for M processing cores. It can be shown that the sufficient, but not the necessary condition to have a feasible scheduling for a number of preemptive tasks with the total utilization, U on a set of M single-core processors is [19]:

$$U \leqslant \frac{M^2}{2M-1} \tag{3}$$

We can use this boundary for multicore processors with M cores; however, this boundary is not sufficient in the case of non-preemptive task sets. Fig. 1 shows the EDF scheduling algorithm used in this paper for non-preemptive tasks.

Whenever a task is released, it will be added to *ReadyList*, and its absolute deadline (*AD*) will be calculated. When there is only one idle core, the system checks whether the task with the earliest deadline can be scheduled on this idle core or not. If the task can be scheduled, it will be assigned to the core, and the core will be assumed busy during the execution. Otherwise, missing the task deadline means that scheduling of this application on the given architecture is not feasible. For simplification, hardware resources are reserved through the worst-case execution time of each task. For efficiency consideration, an event-driver scheduler takes care of the actual execution time of each task.

An example of a task set composed of six tasks is shown in Table 1. Fig. 2 shows the EDF scheduling of this task set on a quad-core processor. As it can be seen in this figure, after scheduling the first released task τ_2 on P_1, τ_1 is scheduled on P_2 and τ_3 is scheduled on P_3. Next, the competition is between τ_4 and τ_5 where both are released at the same time, but the absolute deadline of τ_5 is earlier than τ_4, so τ_5 is scheduled first. The other tasks will be scheduled on idle cores in the same manner.

4. Fault-tolerance policies in real-time systems

Hardware- and time redundancy are the most well-known fault-tolerant methods in time-constrained embedded system design [11,17]. Hardware redundancy methods such as task replication [21] and stand-by-sparing [17] impose considerable cost, especially in harsh environments [11]. In the case of soft real-time systems and systems with flexible slack times, low-cost methods like re-execution [15] and recovery with checkpointing [18,21] are more appropriable to be utilized. In hard real-time systems with tight deadlines, using time redundancy methods has the risk of missing deadlines. On the other hand, checkpointing increases the probability of on-time task completion compared to re-execution methods [18]. However, checkpointing and re-execution methods increase the total execution time.

As already stated, checkpointing and replication can be done by task manager module located in operating systems using kernel-level instructions [23,27,15]. Because task manager has access to each task data workspace and is also aware of current running instruction. Therefore task manager can checkpoint the workspace. In addition, checkpointing can be done using some inserted system calls in a task by developers. In this case, developers should place some checkpoint instructions but these instructions are statically inserted and are not activate in normal execution. At runtime, task manager can activate checkpoint instructions using some provided configurations. In embedded systems, which numbers and types of tasks are so limited compared to desktop systems, this capability is feasible to implement based on some predefined instruction templates.

```
ALGORITHM Dynamic_EDF_Scheduling
{
1.    ReadyList = { }
2.    Feasibility = TRUE
3.    FOR ALL τi , 1 ≤ i ≤ n
4.        IF Ri = ThisTime THEN
5.            ADi = ThisTime + Ci
6.            ReadyList = ReadyList + {τi}

7.    WHILE ReadyList ≠ { }
8.        SORT ReadyList ASCENDING BY ADi
9.        IF EXISTS any idle core THEN
10.           τi = GET_TOP_OF (ReadyList)
11.           IF (ThisTime + Ci) < ADi THEN
12.               SCHEDULE τi on an idle core
13.               TAG the core as busy during Ci
14.               ReadyList = ReadyList - {τi}
              ELSE
15.               Feasibility = FALSE
}
```

Fig. 1. Online EDF scheduling algorithm for non-preemptive tasks.

Table 1
Example of a task set with the task utilization parameter.

Task	R_i	C_i	D_i	AD_i	U_i
τ_1	3	20	54	57	0.37
τ_2	0	10	34	34	0.29
τ_3	6	16	60	66	0.27
τ_4	12	40	82	94	0.49
τ_5	12	18	44	56	0.41
τ_6	16	14	47	63	0.30

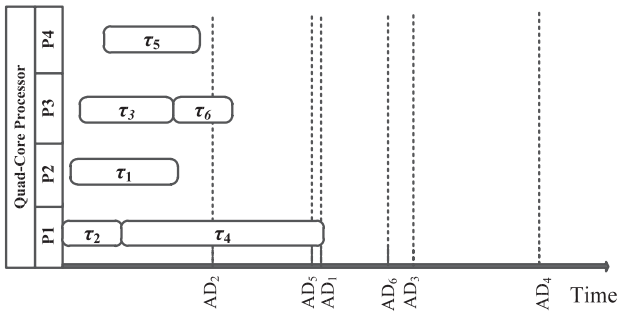


Fig. 2. EDF task scheduling of a task set in Table 1 on a quad-core processor.

4.1. Checkpointing optimization

There are trade-offs between applying frequent- and infrequent checkpoints for tasks in a system. Frequent checkpointing decreases re-execution time in the presence of faults, while task execution time is increased. On the other hand, infrequent checkpointing has lower time overhead in the absence of faults, whereas the amount of re-execution will be increased if a fault is detected [11].

Same as [1], let us assume that the time overheads of getting/saving a checkpoint (*Cs*) and recovering from a checkpoint (*Cr*) are constant for each task and are proportional to the worst-case execution time of each task. As described in [18] we have

$$Cs = \varphi \times C \quad (4)$$

$$Cr = \mu \times C \quad (5)$$

where φ and μ are constant factors. Similar to the related works in [18,21], equidistant check pointing, which benefits from equality in checkpointing intervals throughout the execution time tasks, is used in this paper. Hence, $\Delta = C/m + 1$ is the checkpoint interval for an isolated task regarding m checkpoints. We denote the jth execution part of the task τ_i by $\tau_{i(j)}$. In [14], it is shown that the optimal number of checkpoints considering k faults through the task can be given by:

$$m = \left\| \sqrt{\frac{k \times C}{Cs}} - 1 \right\| \quad (6)$$

Considering Eq. (6), the worst-case response time of a task using checkpointing with rollback recovery (Cc) is given by:

$$Cc = (C + m \times Cs) + k \times (Cs + Cr) + \frac{k \times C}{m + 1} \quad (7)$$

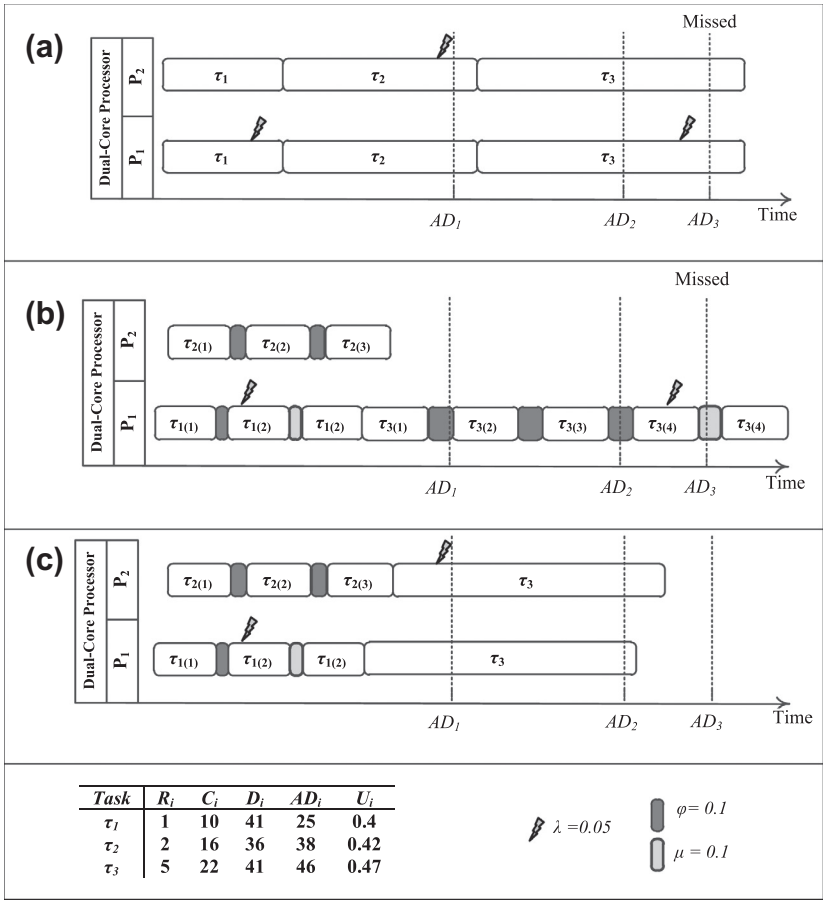
where the term $(C + m \times Cs)$ is the execution time of τ_i using checkpointing without any faults, and $(Cs + Cr) + C/m + 1$ is the cost of fault recovery for single fault, which is multiplied by k to recover maximum of k faults (see also [18]).

4.3. Hardware replication

Although checkpointing with rollback recovery methods have the advantage of reducing time overhead by re-executing only one part of the task in the presence of faults, they cannot utilize available spare resources (i.e. other processing cores in multicore systems) in order to reduce the schedule length. If the rate of fault occurrence on one processing core is high, a task needs more time to recover from faults, which means that the task is likely to miss its deadline. Hardware replication methods have the ability of parallel execution of the redundant copies of original tasks on the other processing cores. Hardware replication methods are generally classified into two distinct categories: (1) active replication [21] in which all the task replicas are executed simultaneously and (2) passive replication [17] in which the backup replicas are executed only if a fault occurs. In this work, a hybrid method composed of these two categories is used; the replica of a task may be executed with a delay from the execution of the primary execution. Since in the dynamic assignment of fault-tolerance policies, there may be some cases where sufficient resources are not available to schedule and map two replicas of one task in the same time. Hence, the scheduler may postpone the execution of the second replica until at least one separate processing core becomes idle.

5. Dynamic fault-tolerant scheduling

In the proposed method called *dynamic fault-tolerant scheduling* (*DFTS*), a hybrid method composed of time- and hardware-based redundancies is used; therefore, based on: (1) available hardware resources, (2) task utilization, and (3) expected number of faults for each task, the scheduler selects an appropriate fault-tolerant method.



Task	R_i	C_i	D_i	AD_i	U_i
τ_1	1	10	41	25	0.4
τ_2	2	16	36	38	0.42
τ_3	5	22	41	46	0.47

Fig. 3. Fault-tolerant scheduling for a set of three tasks: (a) replication-only, (b) checkpoint-only cause deadline miss where all tasks meet their deadlines in (c) by the combination of two methods.

5.1. Policy assignment using task utilization

Task utilization is used to indicate deadline tightness of each task. When the worst-case execution time of a task is considerably greater than its relative deadline, i.e. critical task, the scheduler has no flexibility to defer the execution of the task. Furthermore, there is not enough time to re-execute the task or even rollback the task to the last saved checkpoints in the presence of faults. In this case, hardware replication is the only solution to guarantee that at least one execution unit will perform correctly.

For *noncritical tasks*, there is some flexibility to use time redundancy methods. Hence, checkpointing with rollback recovery method can be applied to reduce the cost of hardware replication. It is worth mentioning that the task utilization and the time at which the scheduler assigns resources to a task are the parameters that should be considered during policy assignment. There may be some cases that a task with low utilization, where there are no idle resources to be allocated for the task; therefore, the task execution should be deferred. Consequently, the role of criticality threshold will be prominent.

Note that θ is not constant and may vary from one task to another and even to another job of the same task. For example, when the first job of a task enters the ready list, the scheduler applies checkpointing to the task; however, for the second job of the task,

ASSUMPTIONS:
Application $\Gamma = \{\tau_1, \tau_2, ..., \tau_n\}, \{C, T, D\}$
Multicore processors with M processing cores
Maximum expected faults ***k*** *for each task*

```
ALGORITHM Dynamic_Fault_Tolerant_Scheduling
{
1.   ReadyList ={ }
2.   Feasibility = TRUE
3.   FOR ALL τi , 1 ≤ i ≤ n
4.        IF Ri = ThisTime THEN
5.             ADi =ThisTime +Ci
6.             Ui = Ci / Di
7.             ReadyList = ReadyList + {τi}

8.   WHILE ReadyList ≠ { }
     {
9.        SORT ReadyList ASCENDING BY ADi
10.       τi = GET_TOP_OF(ReadyList)
11.       IF EXISTS any idle core THEN
          {
12.            RAi = ThisTime
13.            Delayi = RAi - Ri
14.            IF ExecutionCount [τi] = 1 THEN          /* The first execution of the task */
               {
15.                 IF (ThisTime + Ci) > ADi THEN
16.                      Feasibility = FALSE
                    ELSE
17.                      CALCULATE θ For τi
18.                      IF Ui < θ THEN                  /* Checkpointing */
19.                           SCHEDULE τi on an idle core
20.                           APPLY checkpointing for τi
21.                           TAG the core as busy within Cci
22.                           ReadyList = ReadyList - {τi}
                         ELSE /* Ui ≥ θ */              /* Replication */
23.                           SCHEDULE the τi on an idle core
24.                           TAG the core as busy within Ci
25.                           ReadyList = ReadyList - {τi}
26.                           ExecutionCount [τi] ++
27.                           IF EXISTS another idle core THEN
28.                                SCHEDULE τi on the idle core
29.                                TAG the core as busy within Ci
                              ELSE                        /* Putting its replica copy in the ReadyList */
30.                                ReadyList = ReadyList + {τi}
               }
31.            ELSE IF ExecutionCount [τi] > 1 THEN     /* The second task execution task (replica) */
               {
32.                 IF The primary copy of this task is finished as non-faulty THEN   /* The original copy is fault-free */
33.                      ReadyList = ReadyList - {τi}
                    ELSE
34.                      IF (ThisTime + Ci) < ADi THEN   /* The second copy is schedulable */
35.                           SCHEDULE the τi on an idle core
36.                           TAG the core as busy within Ci
37.                           ReadyList = ReadyList - {τi}
                         ELSE
38.                           Feasibility = FALSE
               }
          }
39.       IF Both primary and backup copy of at least one critical task are faulty THEN
40.            Feasibility = FALSE
     }
```

Fig. 4. Dynamic fault-tolerant scheduling algorithm (DFTS).

the scheduler replicates the task since the execution of the second job has been deferred due to the absence of idle resources. So this task becomes critical.

5.2. Motivational example

Fig. 3 shows an application consists of three tasks, which are scheduled and mapped on a dual-core processor. The checkpoint saving and recovery overhead is set to 10% of the worst-case execution time of each task, and the fault rate for each core is 0.05. Regarding these values, the optimal number of checkpoints is 1, 2 and 3 for τ_1, τ_2 and τ_3, respectively. In this figure, the task τ_3 has more utilization compared to the other two tasks. On the other hand, using EDF leads to τ_1, τ_2 and τ_3 having the highest priority to be scheduled in the system, respectively. As depicted in Fig. 3(a and b), applying only hardware replication or checkpointing with rollback recovery policies for all tasks will cause the task τ_3 miss its deadline.

Using DFTS in Fig. 3(c), the criticality threshold for τ_1 and τ_2 are 0.55 and 0.57 respectively. So these two tasks are not critical and checkpointing with rollback recovery can be applied. For τ_3, the criticality threshold is calculated to be 0.32, which means that this task is critical; consequently, it is replicated on P_1 and P_2. In this case, all the tasks have met their deadlines and all the occurred faults have been tolerated.

5.3. Fault-tolerant scheduling based on task criticality

Fig. 4 shows the proposed algorithm for dynamic fault-tolerant scheduling. This algorithm selects a suitable fault-tolerance policy assignment for each task when there are available resources. No task will be scheduled until all other tasks with higher priorities are scheduled, and previous faults in the system are tolerated as well. The scheduler calculates criticality threshold (θ) for the task at top of the *ReadyList* as soon as there is an idle core available in the system. The time when an idle core is assigned to each task is defined as *Resource Allocation* time (RA_i). As discussed before, the difference between the time when a task enters the *ReadyList* and its *Resource Allocation* time is a determinant factor to compute the criticality threshold of the task. We define another parameter, $Delay_i$ for each task, by:

$$Delay_i = RA_i - R_i \tag{8}$$

where $Delay_i$ indicates the wasted time between the time when a current job of task τ_i entered the *ReadyList* and the time when the scheduler assigns hardware resources to this task and applies fault-tolerance policies. Increasing *Delay* parameter for a task may result in a *noncritical* task become *critical*. In order to tolerate expected faults for each task, the scheduler applies checkpointing to *noncritical* tasks and hardware replication to *critical* tasks. In order to compute criticality threshold, it is important to note that the maximum response time of a task in the case of checkpointing with rollback recovery (Cc_i) should be less than the remaining time to its deadline. So, we have

$$Cc_i < D_i - Delay_i \tag{9}$$

Using Eq. (7), this inequality can be represented as:

$$C_i + m \times Cs + k(Cs + Cr) + \frac{K \times C_i}{m+1} < D_i - Delay_i \tag{10}$$

By replacing the values of *Cs* and *Cr* from Eqs. (4) and (5), we have

$$C_i + m \times C_i \times \varphi + k \times C_i(\varphi + \mu) + \frac{k \times C_i}{m+1} < D_i - Delay_i \tag{11}$$

$$U_i\left(1 + m \times \varphi + k(\varphi + \mu) + \frac{k}{m+1}\right) < \left(1 - \frac{Delay_i}{D_i}\right) \tag{12}$$

and by rearranging the inequality, we have

$$U_i < \frac{1 - \frac{Delay_i}{D_i}}{1 + m \times \varphi + k(\varphi + \mu) + \frac{k}{m+1}} = \theta. \tag{13}$$

This inequality implies that by increasing $Delay_i$, checkpointing cost, and the expected number of faults for each task, the criticality threshold will be decreased, and therefore, the task will become *critical*.

If a task is *noncritical*, the checkpointing needs to be applied. Hence, the maximum response time of a task (Cc_i) in the case of checkpointing with rollback recovery will be calculated and its core will be reserved during Cc_i. For the *critical* tasks, the scheduler checks whether at least two idle cores are available and then, schedules two copies of them. If only one idle core is available, the scheduler adds the second copy to the *ReadyList* and waits until a processing core becomes idle.

If the first copy of a task is critical, consequently, the replicated copy will be critical since the second copy has less or equal time than the first copy to be recovered from faults. However, in this paper, it is assumed that only two copies of a critical task will be executed. This is managed by *ExecutionCount* in the proposed algorithm which indicates the number of task copies scheduled in the system. The default value of this parameter is "1" for every task. For the second copy, *ExecutionCount* becomes "2", and therefore, no other copies of the task will be added to the *ReadyList*.

The scheduling of an application on a target platform can be *infeasible* if one of the following three cases occurs: (1) the scheduler cannot find any idle core for the primary copy of a critical task before its deadline, (2) the primary copy is faulty and the backup copy of a critical task misses its deadline, and (3) both primary and backup copies of a critical task are faulty due to multiple fault occurrence. In these cases, the task misses the deadline, and therefore, the other upcoming tasks cannot be scheduled on the platform.

6. Experimental results

In order to evaluate the proposed method, a time-driven task scheduler simulator written in C++ has been used. The number of cores in the multicore processors, the fault-arrival rate, and the parameters of checkpoint cost and checkpoint recovery for tasks are given to the simulator as inputs. The software block diagram of the system simulator used in this paper is depicted in Fig. 5. During each round of execution, a given application as a set of non-scheduled tasks is being scheduled on a multicore processor. The simulator has the capability to schedule tasks in both Non-Fault-Tolerant (*NFT*) and fault-tolerant scheduling methods. Here, three fault-tolerant methods are used: (1) checkpointing with roll back recovery, (2) hardware replication, and (3) the proposed method, i.e. *DFTS*. The *NFT* is a basic application scheduling based on EDF

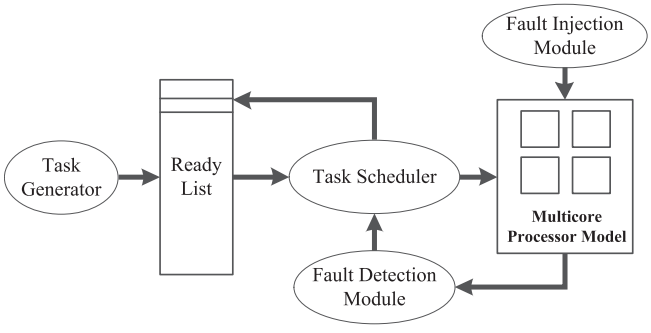


Fig. 5. The block diagram of system simulator.

(similar to the example shown in Fig. 2). *NFT* is utilized to find the best total execution time for a given application on the target architecture without any fault-tolerance time overhead.

From the implementation point of view, fault injection procedure is developed using "*Fault Injection Module*" which runs in parallel with task scheduler. According to the specified fault rate, the *Fault Injection Module* selects one of the processing cores randomly and generates the fault-occurrence instants with Poisson distribution for each core separately. At these times, the status of the selected core will become faulty, and therefore, the task running on the core will fail to complete its execution. "*Fault Detection Module*" is also needed to run in parallel with scheduler in order to detect whether the primary and backup or both are faulty. In the case of both are faulty, this module will send a failure scheduling message to the scheduler. *Fault Detection Module* will start at the end of each task scheduling time when a task is ready to run.

In our experiments, a set of 50 different applications, each of which consists of 10 tasks is generated. The worst-case execution times of each task are randomly generated using uniform distribution function within 10–100 ms. Other task parameters such as release time, and deadline are also randomly generated.

We are able to derive the feasibility analysis for each task set scheduled on a given architecture over the maximum execution time of the simulator results. The experiments are performed using Intel® core™ i7-2670QM processors with 4 GB of RAM. The first evaluation of the proposed algorithm is the feasibility analysis of *DFTS* over well-known fault-tolerance policies such as checkpointing with rollback recovery (*CH*) and hardware replication (*RP*). Fig. 6 shows the feasibility rate of *DFTS* versus checkpointing and replication methods with variations of fault-arrival rates (λ). As it can be seen in this figure, the feasibility rate of *DFTS* is considerably higher than the other methods for different fault rates. However, by doubling the λ, the feasibility rate of all methods will be decreased. The feasibility rate of *DFTS* decreases from 79% to 4% within varying λ from 0.005 to 0.04. The reason is that by growing λ, the maximum expected faults (k) for each task will increase. This will result in the criticality threshold being decreased; hence, the probability of fault occurrence on replicated tasks will increase as well.

In Fig. 7, the effect of checkpointing parameters such as checkpoint saving ratio (φ) and checkpoint recovery ratio (μ) on the feasibility rate of *DFTS* over *CH* and *RP* with the constant fault rate of $\lambda = 0.01$ is shown. As depicted in this figure, doubling the checkpoint saving cost decreases the feasibility rate of *DFTS* from 60% to 37% for constant checkpoint recovery cost. This is due to the double effect of checkpoint saving time in the worst-case response time of task using checkpointing. In contrast, for constant checkpoint saving cost, doubling the checkpoint recovery cost does not have a significant impact on the feasibility rate.

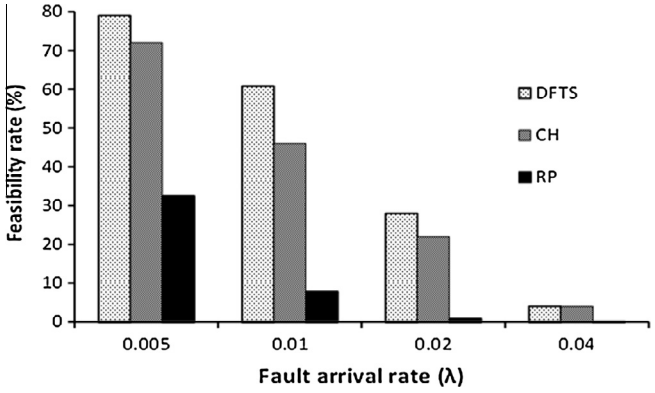


Fig. 6. Feasibility rate of *DFTS* compared to simple checkpointing and replication for quad-core processors regarding doubled fault arrival rates.

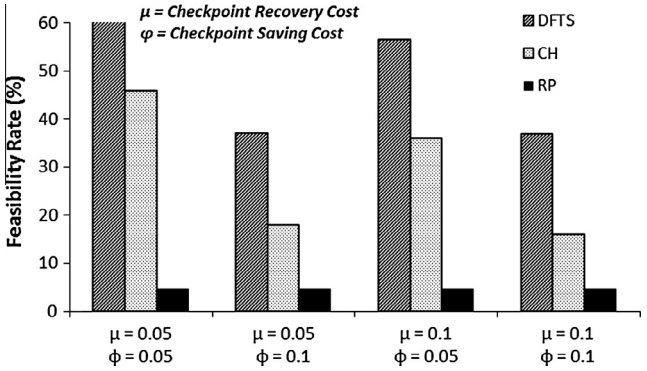


Fig. 7. Feasibility rate of *DFTS* compared to simple checkpointing, and replication by varying the amount of checkpoint saving and recovery cost with constant fault rate ($\lambda = 0.01$).

The effect of increasing number of processing cores on the feasibility rate of *DFTS* compared to the other methods is also depicted in Table 2. For processors with two and four cores, 50 applications with 10 tasks are considered and for processors with 8, 12, 14, and 16 cores, 50 applications with 20 tasks are considered. The reason of having different number of tasks is when hardware resources increases, number of tasks should also be increased to get better comparison between methods. Otherwise, all methods' feasibility rate may be close to 100% when hardware resources are so many to use. Based on achieved results, none of the three fault-tolerant methods can achieve high feasibility rate in the case of dual-core processors. Moreover, it can be seen that the effect of doubling the number of processors from 2 to 4 on feasibility rate is considerably higher than increasing from 4 to 8. For example, the feasibility rate of *DFTS* increases from 47% for 8-core processors to 73% for 12-core processors. It can be concluded that replication method has the worst feasibility rate compared to the others since when replicas are running, resources are busy to be used by the others. Moreover, checkpointing with rollback recovery does not have good feasibility rate since it is applied on each task regardless of other tasks that may be critical to run dynamically.

Another parameter studied in this paper in order to evaluate the performance of the proposed *DFTS* algorithm is time overhead imposed on the system compared to the *NFT*. Suppose T_{DFTS} and T_{NFT} be the maximum scheduled length (total execution) obtained by *DFTS* and *NFT*, respectively. Then, the fault tolerance overhead is defined as $\frac{T_{DFTS}-T_{NFT}}{T_{NFT}} \times 100$. In Table 3, the minimum, maximum and average timing overhead of *DFTS* and *CH* are depicted for different parameters which are checkpoint saving and recovery cost, and constant fault rate ($\lambda = 0.01$). The average overhead of *DFTS* is always lower than *CH* for all the variations of checkpointing parameters. This is due to the combination of checkpointing with hardware replication methods which result in decreasing the schedule length of *DFTS*. Similar to Fig. 7, the effect of increasing checkpoint recovery cost is more than the effect of checkpoint saving cost on the scheduling timing overhead. The minimum timing overhead of *DFTS* is considerably lower than *CH* in all conditions. However, there are special cases where 53% overhead has been imposed on the system by *DFTS*.

The effect of increasing fault rate on the fault-tolerant timing overhead of *DFTS*, *CH* and *RP* is shown in Table 4. When the fault rate is low, the minimum timing overhead of *DFTS* is at least 8% and the average overhead of *CH* is lower than *DFTS*. The average

Table 2
Feasibility rate of DFTS compared to simple checkpointing and replication for different number of processing cores.

Fault-tolerant method	10-Task applications		20-Task applications			
	2 Cores	4 Cores	8 Cores	12 Cores	16 Cores	32 Cores
DFTS	5.07	63.24	47.82	73.02	73.54	74.56
CH	2.00	46.00	28.00	36.00	38.00	42.00
RP	0.44	7.84	0.84	1.04	1.08	2.47

Table 3
Timing overhead of DFTS compared to CH considering various checkpoint parameters (constant fault rate $\lambda = 0.01$).

μ	Φ	*Fault-tolerant method*	Timing overhead		
			Minimum (%)	Maximum (%)	Average (%)
0.05	0.05	*DFTS*	22.01	48.78	37.52
		CH	31.41	42.16	37.69
	0.1	*DFTS*	27.06	51.47	42.55
		CH	38.95	48.0	43.90
0.1	0.05	*DFTS*	23.46	49.51	40.22
		CH	33.84	42.86	40.37
	0.1	*DFTS*	28.32	52.99	42.53
		CH	40.51	50.17	45.49

Table 4
Timing overhead of DFTS compared to CH and RP considering various fault rates (constant checkpointing cost $\mu = \varphi = 0.05$).

λ	*Fault-tolerant method*	Timing overhead		
		Minimum (%)	Maximum (%)	Average (%)
0.005	*DFTS*	8.15	50.99	33.09
	CH	8.15	50.99	31.63
	RP	29.69	51.01	40.88
0.01	*DFTS*	22.01	48.78	37.52
	CH	31.41	42.16	37.69
	RP	29.69	51.01	41.13

time overhead of *DFTS* increases less than *CH* when the fault rate increases. However, the time overhead of *RP* does not change for various fault rates.

7. Conclusions and future works

In this paper, a dynamic fault-tolerant scheduling algorithm called *DFTS* has been proposed. This algorithm uses task utilization to dynamically select the type of fault recovery method in order to tolerate the maximum number of multiple spatial and temporal faults. Each task is categorized into *critical* or *noncritical* based on the task utilization and the time at which scheduler allocates resources to the task. *Noncritical* tasks are scheduled on a single core, and checkpointing with rollback recovery will be applied to them. *Critical* tasks will be replicated on separated cores to increase the probability of on time completion of the task in the presence of faults. Experimental results on several applications running on multi-core processors show that fault-tolerant scheduling feasibility rate of *DFTS* is higher than conventional methods for different fault rates and checkpoint costs. Moreover, the maximum fault-tolerance overhead is lower than the checkpointing with rollback recovery method, whereas the maximum overhead of *DFTS* is always lower than 53% for various checkpoint parameters.

Concerning to optimality of the proposed method, it should be mentioned that the proposed method is a heuristic method utilizing EDF. Each heuristic method is not necessarily optimal. However, experimental results reveal that the proposed method has better fault-tolerance property compared to the other related works.

As a future work, a real implementation of the proposed method on a typical embedded system is offered. Authors try to use today's embedded system platforms like ARM-based embedded boards, to show the method efficiency practically. ARM processors are so rich in terms of hardware and software-based fault detection mechanisms, so they are good candidates for implementation purposes.

Acknowledgement

The authors genuinely thank Mrs. Mahroo Zand-Rahimi, for her good comments of editing the paper. Also, authors would like to appreciate the valuable comments given by the reviewers which improved this paper very well.

References

[1] E. Seo, J. Jeong, S. Park, J. Lee, Energy efficient scheduling of real-time tasks on multi-core processors, IEEE Trans. Parallel Distrib. Syst. 19 (11) (2008) 1540–1552.
[2] A. Saifullah, K. Agrawal, C. Lu, C. Gill, Multi-core real-time scheduling for generalized parallel task models, in: 32nd IEEE Real-Time Systems Symposium (RTSS), 2011, pp. 217–226.
[3] J. Yan, W. Zhang, WCET analysis for multi-core processors with shared L2 instruction caches, in: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2008, pp. 80–89.
[4] F. Kong, W. Yi, Q. Deng, Energy-efficient scheduling of real-time tasks on cluster-based multi-cores, in: Design Automation and Test in Europe, 2011, pp. 1–6.
[5] ARM11MPCore Processor. <http://www.arm.com/products/processors/classic/arm11/arm11-mpcore.php>.
[6] IBM Cell Project. <http://domino.research.ibm.com/comm/research.nsf/pages/r.arch.innovation.html>.

[7] V. Suhendra, T. Mitra, Exploring locking & partitioning for predictable shared caches on multi-cores, in: Design Automation Conference (DAC), 2008, pp. 300–3303.
[8] J. Chen, L.K. John, Efficient program scheduling for heterogeneous multi-core processors, in: Design Automation Conference (DAC), 2009, pp. 927–930.
[9] M. Fan, G. Quan, Harmonic semi-partitioned scheduling for fixed-priority real-time tasks on multi-core platform, in: Design Automation and Test in Europe, 2012, pp. 503–508.
[10] S. Siddha, V. Pallipadi, A. Mallick, Process scheduling challenges in the era of multi-core processors, Intel® Technol. J. 11 (4) (2007) 61–370.
[11] P. Eles, V. Izosimov, P. Pop, Z. Peng, Synthesis of fault-tolerant embedded systems, in: Desing Automation and Test in Europe, 2008, pp. 1117–1122.
[12] H. Aydin, Exact fault-sensitive feasibility analysis of real-time tasks, IEEE Trans. Comput. 56 (10) (2007) 1372–1386.
[13] M.D. Powell, A. Biswas, S. Gupta, S.S. Mukherjee, Architectural core salvaging in a multi-core processor for hard-error tolerance, in: Annual International Symposium on Computer Architecture (ISCA), 2009, pp. 93–104.
[14] Y. Zhang, K. Chakrabarty, A unified approach for fault tolerance and dynamic power management in fixed-priority real-time embedded systems, IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst. 25 (1) (2006) 111–125.
[15] V. Izosimov, P. Pop, P. Eles, Z. Peng, Scheduling of fault-tolerant embedded systems with soft and hard timing constraints, in: Desing Automation and Test in Europe, 2008, pp. 915–920.
[16] E. (Mootaz) Elnozahy, R. Melhem, D. Mossé, Energy-efficient duplex and TMR real-time systems, in: IEEE Real-Time Systems Symposium (RTSS), 2002, pp. 256–266.
[17] A. Ejlali, B.M. Al-Hasihmi, P. Eles, A standby-sparing technique with low energy-overhead for fault-tolerant hard real-time systems, in: 7th International Conference on Hardware/Software Codesign and System Synthesis (CODES/ISSS), 2009, pp. 193–202.
[18] Y. Zhang, K. Chakrabarty, Fault recovery based on checkpointing for hard real-time embedded systems, in: 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT), 2003, pp. 320–327.
[19] R.I. Davis, A. Burns, A survey of hard real-time scheduling for multiprocessor systems, ACM Comput. Surv. 43 (4) (2011) (Article 35).
[20] T. Facchinetti, G.C. Buttazzo, M. Marinoni, G. Guidi, Non-preemptive interrupt scheduling for safe reuse of legacy drivers in real-time systems, in: 17th Euromicro Conference on Real-Time Systems (ECRTS), 2005, pp. 98–105.
[21] P. Pop, V. Izosimov, P. Eles, Z. Peng, Design optimization of time and cost-constrained fault-tolerant embedded systems with checkpointing and replication, IEEE Trans. Very Large Scale Integr. Syst. 17 (3) (2009) 389–402.
[22] F. Liberato, R. Melhem, D. Mosse, Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems, IEEE Trans. Comput. 49 (9) (2000) 906–914.
[23] B. Hamidzedeh, D.J. Lilja, Dynamic scheduling strategies for shared-memory multiprocessors, in: 16th International Conference on Distributed Computing Systems, 1996, pp. 208–215.
[24] J.W.S. Liu, Real-Time Systems, Prentice Hall, 2000.
[25] S. Ghosh, R. Melhem, D. Mossé, Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems, IEEE Trans. Parallel Distrib. Syst. 8 (3) (1997) 272–284.
[26] T.P. Baker, An analysis of EDF schedulability on a multiprocessor, IEEE Trans. Parallel Distrib. Syst. 16 (8) (2005) 760–768.
[27] K. Lakshmanan, S. Kato, R. Rajkumar, Scheduling parallel real-time tasks on multi-core processors, in: 31st IEEE Real-Time Systems Symposium (RTSS), 2010, pp. 259–268.

Mohammad H. Mottaghi received his B.Sc. and M.Sc. degrees in computer engineering from the Amirkabir University of Technology (Tehran Polytechnic), Iran, in 2010 and 2012, respectively. He was a member of Design and Analysis of Dependable Systems (DADS) Lab. in Amirkabir University of Technology (Tehran Polytechnic), Iran. He is currently pursuing the Ph.D. degree in computer engineering department. His research interests are design optimization and scheduling of fault-tolerant real-time embedded systems.

Hamid R. Zarandi received his B.Sc., M.Sc., and Ph.D. degrees all in department of computer engineering at Sharif University of Technology (SUT), Tehran, Iran, in 2000, 2002, and 2007, respectively. He is currently an assistant professor in computer engineering and information technology department at Amirkabir University of Technology (AUT) (Tehran Polytechnic), since 2007. His research interests include dependability evaluation using fault injection techniques, fault-tolerant computing, dependable computer architecture, high performance computing, and fault-tolerant embedded and real-time systems, on which he has published more than 90 referred conference and journal papers. Dr. Zarandi established the "Design and Analysis of Dependable Systems (DADS)" laboratory at Amirkabir University in 2007, and has chaired the laboratory since then. He is a member of the IEEE Computer Society, and the Computer Society of Iran (CSI).