IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 7, 2020, accepted May 18, 2020, date of publication June 3, 2020, date of current version June 12, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2999092

A Novel and Adaptive Transient Fault-Tolerant Algorithm Considering Timing Constraint on Heterogeneous Systems

JING LIU, ZIQI ZHU, AND CHUNHUA DENG

College of Computer Science and Technology, Wuhan University of Science and Technology, Wuhan 430081, China
Hubei Province Key Laboratory of Intelligent Information Processing and Real-time Industrial System, Wuhan 430081, China

Corresponding author: Ziqi Zhu (zhuzq@wust.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61602350 and Grant 61702382, in part by the Hubei Natural Science Foundation under Grant 2018CFB195, in part by the Hubei Education Department Foundation under Grant Q20181104, and in part by the Key Laboratory of Image Processing and Intelligent Control, Ministry of Education Foundation, under Grant IPIC2018-04.

ABSTRACT Due to high performance and low power consumption, heterogeneous processors are widely used in many real-time systems. In these systems, if tasks are not completed before deadline, it will cause disastrous consequences, and thus it is important to provide fault-tolerance. This paper proposes a novel, adaptive and transient fault-tolerant scheduling algorithm to solve the fault-tolerant problem in heterogeneous real-time systems, aiming to improve system reliability within a given deadline. Since task replication is efficient in improving system reliability, the proposed algorithm supports multiple replicas for each primary task and allows the primary tasks and their replicas to be scheduled on the same processor to increase reliability and lower latency. Also, the algorithm can dynamically adjust the number of replicas for each task to accommodate the deadline and ensure higher reliability. Simulated results show that the proposed algorithm can achieve higher reliability in comparison with existing and related fault-tolerant algorithms. To be specific, the proposed algorithm can obtain the reliability of 89.37% whereas the two existing algorithms DB-FTSA and FTSA obtain the reliability of 47.05% and 84.75% for the benchmark of sixty tasks, respectively, to be detailed in Fig. 4 in experiment.

INDEX TERMS Reliability, heterogeneous system, task scheduling, fault-tolerant, deadline.

I. INTRODUCTION

A. BACKGROUND

In the past decades, society has witnessed continually improved performance of computing systems, which successfully caters to the demands in both industry and academia [1]. Although the performance of computing systems has been improved, its scale and complexity have also been increased, which has triggered increased failures [2], [3]. According to the 54th edition of the TOP500 in November 2019, Oak Ridge National Laboratoryąŕs Summit system holds top honors with an High Performance Linpack (HPL) result of 148.6 petaflops and contains 2,414,592 cores. Heterogeneous systems are widely applied to many security-related real-time systems, due to the high performance and low cost, as their scale and complexity become larger and larger, the possible failures increase. Therefore, it is no longer reasonable to ignore the fact that an application running on a very large system can crash. Especially, higher reliability is an indispensable design goal for real-time systems that are safe-critical. In these systems, the correct behavior depends not only on computing results but also on the issue when results are yielded [4]. No matter how good the computation result is, as long as the completion time exceeds the timing constraint, it is invalid. What is worse, missing deadline may cause serious catastrophe. Fault-tolerant scheduling is an effective way to achieve fault tolerance. Different scheduling schemes may result in different reliability and completion time. Therefore, designing efficient fault-tolerant scheduling techniques has become essential for achieving better reliability within timing constraint and thus far has attracted great interest amongst researchers.

Faults may happen when computing systems are running [5]. They can be primarily classified into two categories:

The associate editor coordinating the review of this manuscript and approving it for publication was Xiao-Sheng Si.

permanent faults and transient faults [6]. Once permanent faults occur, they exist all the time in the system. The failed component cannot perform any task, and the unfinished task must be migrated to other non-faulty components [7]. Transient faults are short-lived and occur primarily due to temporary malfunctioning of the components or external interferences such as electrostatic discharge, electrical power drops or overheating [8], [9]. In computing systems, there are all kinds of faults, but a majority of them are transient [10]. Thus, this paper is centered on analyzing transient faults.

There are several methods to solve transient failure problems. One commonly used method is based on the task duplication technique which allows multiple replicas for each task. Task duplication has two forms. One is the active form, where replicas of a task can be assigned to multiple processors, and executed in parallel to reduce failures [11]. Only when a task or one of its replicas is successfully performed, the task is regarded to have been finished. The other is the passive form [12]. A task and its replicas are classified as the primary task and the backup tasks, where the original task is called the primary task and all its replicas are deemed as the backup tasks. When a primary task fails, it needs to select a backup task to restore the pre-failure state, resulting in a longer recovery time. This form also makes an assumption that a fault detection mechanism is used to detect processor crashes. Nevertheless, there is no need for the active form to detect and handle failures [13], thereby saving time. In this paper, we take the active form to cope with transient failures. Unfortunately, most of the time, increasing the reliability causes an increase in the completion time [14]. Therefore, how to achieve higher reliability within a given time period is a big challenge.

B. RELATED WORK

Fault-tolerant scheduling under different situations has been widely studied in previous literatures.

Some aim to maximize the system reliability. Luo *et al.* [15] develop a real-time fault-tolerant algorithm RDFTAHS to schedule preemptive periodic tasks on heterogeneous distributed systems to boost system reliability. Yan *et al.* [16] focus on the fault tolerance when task runtime is uncertain. They propose an efficient fault-tolerant scheduling algorithm DEFT for real-time tasks in the cloud, aiming to achieve both fault tolerance and resource utilization efficiency, whilst not taking hosts' communication time into account. Latiff *et al.* [17] propose a DCLCA technique for dynamic clustering fault tolerance aware intelligent scheduling using the LCA optimization algorithm, not considering timing constraint. Zhang *et al.* [18] devise a novel algorithm RMEC which incorporates task priority establishment, frequency selection, and processor assignment to maximize the system reliability with energy constraint. They do not consider timing constraint, either. Mottaghi and Zarandi [19] present a dynamic scheduling algorithm called DFTS for real-time tasks in multicore processors to tolerate single and multiple transient faults. However, DFTS is only used for independent tasks and identical processing cores. Wei *et al.* [20] propose a fault-tolerant algorithm to deal with the transient failures which allows at most one backup task and cannot make full use of deadline.

Some others focus on studying energy-efficient fault-tolerance. Guo *et al.* [11], [21] develop energy-efficient fault-tolerance (EEFT) techniques to schedule periodic tasks running on systems with identical functionalities. Zhao *et al.* [22] propose the SHR-DAG algorithm for scheduling a set of frame-based real-time tasks with individual deadlines on a single-processor system to minimize energy consumption while preserving the system reliability. Xie *et al.* [23] study the energy-efficient fault-tolerant scheduling problem on heterogeneous distributed embedded systems. They want to reduce the energy consumption while satisfying the reliability goal and do not consider timing constraint. Chatterjee *et al.* [24] study fault-tolerant dynamic task mapping and scheduling problem for Network-on-Chip-based multicore platform.

Apart from the works above, others have their different focuses. Nair *et al.* [25] and Devaraj *et al.* [26] study the fault-tolerant problem on independent tasks. Studies [27]–[30] tackle the fault-tolerant problem in homogeneous multicore systems. Benoit *et al.* [31] propose an efficient fault-tolerant scheduling algorithm named FTSA for heterogeneous systems based on an active replication scheme to minimize the latency given a fixed number of failures. They do not consider communication time. Zhao *et al.* [32] design a fault-tolerant scheduling algorithm called MaxRe for heterogeneous systems to satisfy users' reliability requirements with minimum resources. Nor do they consider communication time. Samal *et al.* [33] propose a hybrid GA for primary-backup fault-tolerant scheduling of hard real-time tasks on multiprocessor systems with identical processors to maximize system utilization and efficiency. Kurt *et al.* [34] present a fault-tolerant dynamic task graph scheduling algorithm that recovers from faults without global coordination to minimize the slowdown of the application in the presence of soft errors. Timing constraint is not taken into consideration, either.

In this paper, we focus on transient failures and applications represented by DAGs considering communication time, aiming to improve system reliability by using redundancy to tolerate faults within timing constraint.

C. OUR CONTRIBUTION

In this paper, we propose a novel, adaptive and transient fault-tolerant scheduling algorithm to solve the fault-tolerant problem with timing constraint in heterogeneous system, aiming to improve system reliability. Generally, the more replicas, the more likely to obtain higher reliability. This further generate smaller communication overhead between tasks executed on the same processor than that on different processor. Based on this, our proposed algorithm allows multiple replicas for each primary task. The primary task and its replicas can be assigned to one processor. After determining

the assignment of all primary tasks and their replicas under the given deadline, the algorithm calculates the maximum reliability that the system can achieve. We conduct a series of simulated experiments including randomly generated graphs with various characteristics and real-word applications to evaluate the proposed algorithm.

The main contributions of this paper are summarized as follows:

- We propose a novel, adaptive and transient fault-tolerant scheduling algorithm to solve the transient fault-tolerant problem. The algorithm can dynamically determine the number of replicas based on the given deadline, and obtain as much reliability as possible.
- We propose a task assignment algorithm to assign each task to its best fit processor.
- Simulated results show that the proposed algorithm can always keep a higher reliability compared with existing and related fault-tolerant algorithms within the given deadline.

The remainder of this paper is organized as follows. Section II defines models and the problem in discussion. Section III shows a motivational example to illustrate the importance of proper fault-tolerant scheduling. Section IV proposes a novel, adaptive and transient fault-tolerant scheduling algorithm. A comparison of the proposed algorithm with existing and related algorithms is conducted in Section V to evaluate the performance of the proposed algorithm, before Section VI concludes this paper.

II. MODELS AND PROBLEM DEFINITION

In this section, we first describe the system model, task model, and fault model. Then, we offer the definition of the problem discussed in this paper.

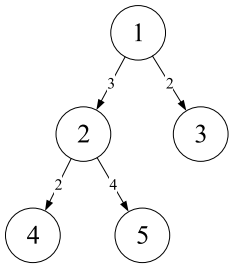
A. SYSTEM MODEL

The system targeted in this study is composed of a set of heterogeneous processors labeled as $p_1, p_2, \ldots, p_M$, where M is the number of total processors. Denote $P = \{p_1, p_2, \ldots, p_M\}$ to be the set of processors. These processors are connected to each other with communication links and can communicate with each other. Different links between processors have different data transmission rate. The transmission rate of a link (namely bandwidth) is measured in bits per second [35], [36]. Suppose communication bandwidth between any two processors is symmetrical, and we use notation B_{ij} to represent the communication bandwidth from processor p_i to processor p_j. As a result, we have $B_{ij} = B_{ji}$. We also assume that all interprocessor communications are performed without contention.

B. TASK MODEL

An application is generally modeled by a weighted directed acyclic graph (DAG) $G =< V, E >$. V is the set of nodes and contains N nodes $v_1, v_2, \ldots, v_N$. Each node represents a task. In this paper, we use the terms ''node'' and ''task'' interchangeably. $E \subseteq V \times V$ is the set of edges corresponding to precedence relations between tasks. For instance, the directed edge (v_i, v_j) connecting node v_i to node v_j indicates that node v_j cannot start executing until task v_i has finished execution. The computational heterogeneity of tasks means the difference of execution time that each task is executed on each available processor in a system. Denote ET_{ij} as the execution time of task v_i on processor p_j. When the redundancy technique is taken to provide transient fault-tolerance, a task may have multiple replicas. We use v_i and $v_i^{b_j}$ to label the primary task and its jth replica, and they have the same execution time when executed on the same processor. In a DAG, a task may need the output generated by other tasks as its input, then data transfer happens. Let $data(v_i, v_j)$ be the data volume transferred from task v_i to task v_j. If two tasks are mapped to the same processor, the communication time will be zero. Otherwise, if v_i is mapped to processor p_m and v_j is mapped to processor p_k, then the communication time from task v_i to task v_j will be computed by $data(v_i, v_j)/B_{mk}$. We assume that all the tasks have a shared deadline and tasks are performed in a non-preemptive means.

If there is an edge (v_i, v_j) from node v_i to node v_j, then v_i can be said to be the predecessor (parent) of v_j and v_j as the successor (child) of v_i. A task v_i may have multiple predecessors or multiple successors, and we use $pre(v_i)$ to denote the set of all predecessors of task v_i. Similarly, we use $succ(v_i)$ to denote the set of all successors of task v_i. Fig.1 gives an example of a DAG.



(a)

Task	P1	P2	P3
1	8	3	6
2	7	5	4
3	6	10	6
4	7	6	8
5	4	5	7

(b)

FIGURE 1. An example of DAG. (a) A tree. (b) Values of execution time for tasks executed on different processors.

In a DAG, a task without any predecessor is called an *entry* task, and a task without any successor is called an *exit* task. A DAG may have multiple entry tasks and multiple exit tasks. Without loss of generality, suppose there is only one entry task and one exit task in a DAG. The DAG is finished only when its exit task is finished. Therefore, the finished time (namely makespan) is obtained by

$$makespan = FT(v_{exit}), \tag{1}$$

where $FT(v_{exit})$ is the finished time of task v_{exit}.

C. FAULT MODEL

At run-time, failures may occur due to various reasons, such as hardware failures, electromagnetic interferences as

well as the effects of cosmic ray radiations. Based on the common exponential distribution assumption in the reliability research [37] for every processor, the arrival of failures follows a poisson distribution with the λ value representing the expected number of occurrence of failures in unit time. Different processors have different λ values. Denote $\lambda_1, \lambda_2, \ldots, \lambda_M$ to be the λ values for processors $p_1, p_2, \ldots, p_M$, respectively, and $\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_M\}$. The failure distribution in t unit times for processor p_j can be defined as

$$f(k, \lambda_j, t) = \frac{\lambda_j^k e^{-\lambda_j t}}{k!}, \tag{2}$$

where k is the number of actual failures in t unit times.

The reliability of task v_i executed on processor p_j is the possibility of its successful execution. That is

$$R_{ij} = f(0, \lambda_j, ET_{ij}) = \frac{\lambda_j^0 e^{-\lambda_j ET_{ij}}}{0!} = e^{-\lambda_j ET_{ij}}. \tag{3}$$

task v_i is successfully executed if it or at least one of its replicas is successfully finished. So the possibility for successful execution of v_i is computed by

$$R_i = 1 - \prod_{i' \in C_i} (1 - R_{i'p(i')}), \tag{4}$$

where C_i is the set of indexes for v_i and its replicas, and $p(i')$ is the index of the processor that the $v_{i'}$ is assigned to.

Therefore, the reliability of a system with N tasks is calculated by

$$R = \prod_{i=1}^{N} R_i. \tag{5}$$

D. PROBLEM

We address the problem as follows: Given a system composed of a set of M heterogenous and connected processors $P = \{p_1, p_2, \ldots, p_M\}$, and an application represented by a DAG $G =< V, E >$ with a shared deadline D, the goal is to find a fault-tolerant scheduling scheme which can maximize the system reliability as well as satisfy task dependencies and meet the give deadline. The problem can be mathematically described as follows:

$$\begin{aligned} &\max\ R = \prod_{i=1}^{N} R_i, \\ &\text{s.t. } makespan \leq D. \end{aligned} \tag{6}$$

Since the problem is NP-hard [31], a heuristic algorithm is proposed to cope with it. Redundancy is an efficient technique to improve the system reliability. Thus, the heuristic takes active backup strategy. It first calculates out the most replicas that the system can tolerate within the given deadline. Then it computes the final system reliability. Different schedules will lead to different makespan and system reliability. Different number of replicas will also cause different makespan and system reliability. This will be illustrated in-depth in the following section.

III. A MOTIVATIONAL EXAMPLE

An example is initially presented here to illustrate that different fault-tolerant scheduling schemes have a significant impact on the makespan of an application and the system reliability when it is executed on a heterogeneous system. Suppose that the application is shown in Fig. 1(a) and the execution times of all its five tasks executed on a system with three heterogeneous processors are shown in Fig. 1(b). For simplicity, we assume that the value of the communication bandwidth between any two processors is 1.

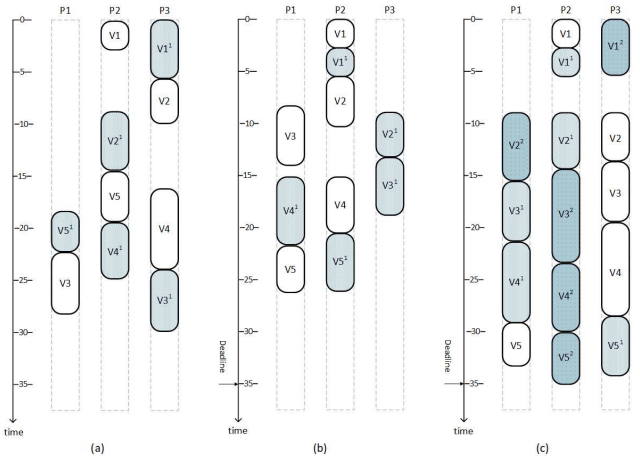


FIGURE 2. A motivational example with deadline $D = 35$. (a) The first scheme with makespan 30 and the reliability 97.54%; (b) The second scheme with makespan 26 and the reliability 97.67%; (c) The third scheme with makespan 35 and the reliability 99.76%.

Fig. 2 displays three different scheduling schemes when the given deadline is 35 time units. In some schemes, each task has one replica, whereas in other schemes, each task may have multiple replicas. Denote vi^j to be the jth replica of task vi, $1 \leq i \leq 5$, $j \geq 1$. In the first scheme shown in Fig. 2(a), every task has one replica, and the primary task as well as its replica are assigned to different processors: $v1, v2, v3, v4, v5$ are separately assigned to processors $p2, p3, p1, p3, p2$ while $v1^1, v2^1, v3^1, v4^1, v5^1$ are separately assigned to processors $p3, p2, p3, p2, p1$. The makespan is 30. In the second scheme shown in Fig. 2(b), although each task still has one replica, the primary task and its replica could be assigned to the same processor, different from the first scheme. The makespan is 26. In the third scheme shown in Fig. 2(c), each task has multiple replicas. The makespan is 35. Assume that processors $p1, p2, p3$ have λ values: $\Lambda = \{0.015, 0.014, 0.013\}$, and we obtain the reliability calculated by (5) for the three scheduling schemes in Fig. 2 from left to right are 97.54%, 97.67%, and 99.76%, respectively.

Different scheduling schemes generate different reliability and different makespan. Usually, more replicas means higher reliability. Compared with the first two schemes, the third scheme has more replicas and the highest reliability. This brings us to the question of whether more replicas represents a better scenario.

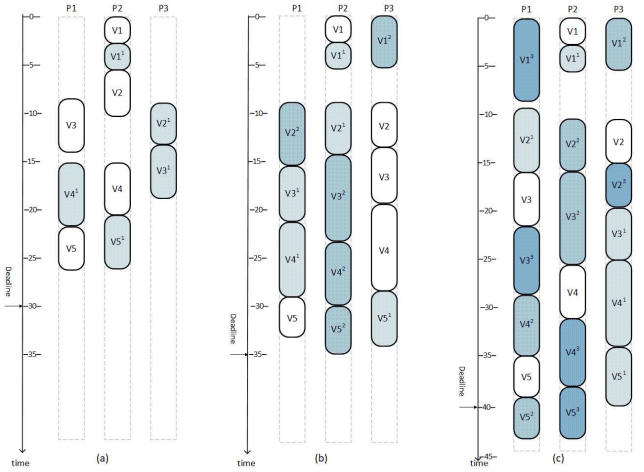


FIGURE 3. Three cases of different number of replicas generated by the third scheme above. (a) One replica with makespan 26 and the reliability 97.67%; (b) Two replicas with makespan 35 and the reliability 99.76%; (c) Three replicas with makespan 43 and the reliability 99.98%.

Fig. 3 shows what will happen when the number of replicas increases from one to three using the third scheme above. In Fig. 3(a), each task has one replica, the makespan is 26 time units, and the reliability is 97.67%. In Fig. 3(b), each task has two replicas, the makespan is 35 time units, and the reliability is 99.76%. In Fig. 3(c), each task has three replicas; the makespan is 43 time units and the reliability is 99.98%, missing the given deadline 35 time units. From Fig. 3, we can get to know: (1) when the number of replicas increases, the reliability and makespan also increase; (2) when the number of replicas increase to some value, and the reliability is high enough (i.e., larger than 99%), it will not cause significant increase in reliability if more replicas are considered.

Although more replicas can provide higher reliability, it consumes more resources like CPU, time, and memory, etc. As a consequence, it is necessary and important to design efficient fault-tolerant scheduling method to improve the system reliability with various constraints.

Notations used in this paper are summarized in Table 1.

IV. THE PROPOSED ALGORITHM

In this section, we propose a novel, adaptive and transient fault-tolerant scheduling algorithm (AFTSA) which is the third scheme mentioned in Section III to solve our stated problem. For the convenience of reading, we first explain some concepts and formulas, and then present the proposed algorithm in detail.

A. CONCEPTS AND FORMULAS

$EST(v_i, p_k)$, the earliest execution start time of task v_i on processor p_k, is computed by

$$EST(v_i, p_k) = \begin{cases} 0 & \text{if } v_i = v_{entry} \\ \max\{RT(v_i, p_k), avail(p_k)\} & \text{if } v_i \neq v_{entry}, \end{cases} \tag{7}$$

where $avail(p_k)$ is the time once processor p_k finishes its last assigned task and becomes idle.

$RT(v_i, p_k)$ is the time when v_i receives all the data from its predecessors and is ready to be executed on p_k. It is computed by

$$\begin{aligned} &RT(v_i, p_k) \\ &= \max_{v_x \in pre(v_i)} \{\max\{AFT(v_x, pro(v_x)) \\ &\quad + c_{xi}, \max_{v_x{}^{b_j} \in Backup(v_x)} \{BAFT(v_x^{b_j}, pro(v_x^{b_j})) + c_{xji})\}\}\}, \end{aligned} \tag{8}$$

where $pro(v_x)$ is the processor to which the primary task v_x is assigned and $pro(v_x{}^{b_j})$ is the processor to which the *jth* replica $v_x{}^{b_j}$ is assigned, respectively. $c_{xi}(c_{xji})$ is the communication time from tasks v_i to $v_x(v_x^{b_j})$.

$EFT(v_i, p_k)$, the earliest execution finish time of task v_i on processor p_k, is computed by

$$EFT(v_i, p_k) = EST(v_i, p_k) + ET_{ik}. \tag{9}$$

$AFT(v_x, pro(v_x))$, the actual finish time of primary task v_x on its assigned processor $pro(v_x)$, is computed by

$$AFT(v_x, p_k) = \min_{p_j \in P}\{EFT(v_x, p_j)\}, \tag{10}$$

where p_k is the processor that v_x has the minimum earliest execution finish time among all processors. Let $p_k = pro(v_x)$.

$BAFT(v_x{}^{b_j}, pro(v_x{}^{b_j}))$, the actual finish time of the jth replica of v_x on processor $pro(v_x{}^{b_j})$, is computed by

$$BAFT(v_x{}^{b_j}, p(v_x{}^{b_j})) = \min_{p_s \in P}\{EFT(v_x^{b_j}, p_s)\}. \tag{11}$$

$FT(v_i)$, the finish time of v_i, if v_i has no replicas, then we have

$$FT(v_i) = AFT(v_i, pro(v_i)). \tag{12}$$

Otherwise, the finish time of v_i is the time when both itself and its replicas have finished execution. Thus, the finish time of v_i is computed by

$$FT(v_i) = \max\{BAFT(v_i{}^{b_j}, pro(v_i{}^{b_j})\}, \quad \forall v_i{}^{b_j} \in Backup(v_i), \tag{13}$$

where $Backup(v_i)$ is the set of replicas of task v_i.

$rank_u(v_i)$, the uprank of task v_i, is used to determine the scheduling order of v_i and computed as following

$$rank_u(v_i) = \overline{ET_i} + \max_{v_j \in succ(v_i)} (\overline{c_{ij}} + rank_u(v_j)), \tag{14}$$

where $\overline{ET_i}$ is the average execution time of task v_i on all processors and $\overline{c_{ij}}$ is the average time consumed when task v_i communicates with task v_j. We have

$$\overline{ET_i} = \frac{\sum_{j=1}^{M} ET_{ij}}{M}, \tag{15}$$

$$\overline{c_{ij}} = \frac{data(v_i, v_j)}{\overline{B}}, \tag{16}$$

TABLE 1. Definitions of the notations.

$Notation$	**Definition**
N	the number of tasks in the given application
M	the number of processors in the given system
V	the set of tasks in the given application
E	the set of directed edges representing the constraint among tasks in V
P	the set of processors in the given system
D	the given deadline
v_i	the task v_i
${v_i}^{b_j}$	the jth backup task of v_i
(v_i, v_j)	the edge from task v_i to v_j
Λ	the set of the expected numbers of failures in unit time for all processors
λ_i	the expected numbers of failures in unit time for processor p_i
B_{ij}	the communication bandwidth from processor p_i to processor p_j
$\overline{B}$	the average communication bandwidth
$\overline{c_{ij}}$	the average communication time from task v_i to task v_j
ET_{ij}	the execution time of task v_i on processor p_j
$\overline{ET_{ij}}$	the average execution time of task v_i on all processors
$data(v_i, v_j)$	the data volume transferred from task v_i to task v_j
$pre(v_i)$	the set of predecessors of v_i
$succ(v_i)$	the set of successors of v_i
$pro(v_i)$	the processor that task v_i is assigned to
$EST(v_i, p_k)$	the earliest execution start time of task v_i on processor p_k
$EFT(v_i, p_k)$	the earliest execution finish time of task v_i on processor p_k
$RT(v_i, p_k)$	the time when v_i has received all data from its predecessors when it is allocated to p_k
$avail(p_k)$	the instant that processor p_k becomes idle
$AFT(v_i, p_k)$	the actual finish time of the primary task v_i on processor p_k
$BAFT({v_i}^{b_j}, p_k)$	the actual finish time of the jth backup task of v_i on processor p_k
$FT(v_i)$	the final finish time of task v_i
$rank_u(v_i)$	the uprank of task v_i
R_{ij}	the reliability of task v_i executed on processor p_j
R_i	the reliability of task v_i
R	the system reliability
$makespan$	the completion time of the given application
$Backup(v_i)$	the set of replicas of task v_i

where $\overline{B}$ is the average data transmission rate among processors. It can be calculated by

$$\overline{B} = \frac{\sum_{m=1}^{M} \sum_{k=1}^{M} B_{mk}}{M^2}. \tag{17}$$

A larger value of uprank value implies a higher priority. For example, if $rank_u(v_1) > rank_u(v_2)$, then the priority of v_1 will be higher than that of v_2 and it will be scheduled earlier than v_2. If $rank_u(v_1) = rank_u(v_2)$, then tie is broken by firstly executing the task of smaller index.

B. ALGORITHM DESCRIPTION

The basic idea of the proposed AFTSA is as follows. First, it calculates all tasks' $rank_u$ values and stores them in the descending order of their $rank_u$ values. Next, it calculates the makespan without backup tasks (replicas) by calling a scheduling function. Then, it calculates the maximum number of replicas for each primary task that the system can tolerate within the given deadline. Finally, it computes the maximum reliability that the system can achieve. AFTSA is advantageous in that it makes good use of the given time to dynamically generate the maximum number of replicas for each task to obtain higher reliability in limited time, allows the primary task and its replicas to be assigned to the same processor, and assigns the primary task and its replicas to processors of the minimum earliest execution finish time among all processors. We now describe the proposed algorithm in detail.

Algorithm 1 outlines AFTSA. The Input is a DAG $G = <V, E>$, a deadline D, and M processors $p_1, p_2, \ldots, p_M$. The Output is a schedule scheme for all tasks of G that achieves the maximum system reliability. Firstly, the proposed algorithm calculates the $rank_u$ values for all tasks by Equation (14), stores these tasks according to the non-increasing order of $rank_u(v_i)$ in a list *list*, as well as initializes the parameters *makespan*, *CN*, and *Fmakespan* to be zero (lines 1-3). Secondly, it invokes the function shown in Algorithm 2 to get the makespan without considering replicas (line 4). Thirdly, it uses a while loop to calculate out the most replicas for each task that the system can tolerate within the given deadline D (lines 5-15). The inner while loop from line 7 to line 14 checks if the system can tolerate *CN* replicas or not for each task until the makespan exceeds D or all tasks have been copied. In order to reduce overhead, the maximum

Algorithm 1 AFTSA

Input: A DAG $G = \{V, E\}$, a common deadline D, $P = \{p_1, p_2, \ldots, p_M\}$;
Output: A schedule scheme for all tasks of G that achieves maximum system reliability;
1: compute $rank_u(v_i)$ for $\forall v_i \in V$ by (14);
2: store all tasks by a non-increasing order of $rank_u(v_i)$ in a list *list*;
3: $makespan \leftarrow 0$, $CN \leftarrow 0$, $Fmakespan \leftarrow 0$;
4: $makespan \leftarrow TaskAssignment(list, \emptyset, P, CN)$;
5: **while** $makespan <= D \&\& CN < 3$ **do**
6: $k \leftarrow 0$, $clist \leftarrow \emptyset$, $CN += 1$;
7: **while** $makespan <= D \&\& k < N$ **do**
8: $Fmakespan \leftarrow makespan$;
9: **for** $j = 0$ to $CN - 1$ **do**
10: $clist \leftarrow clist \cup \{list[k]\}$;
11: **end for**
12: $makespan \leftarrow TaskAssignment(list, clist, P, CN)$;
13: $k++$;
14: **end while**
15: **end while**
16: **if** $makespan > D$ **then**
17: $clist \leftarrow clist - \{list[k-1]\}$;
18: **end if**
19: $makespan \leftarrow Fmakespan$.
20: compute the system reliability R by (5).

Algorithm 2 TaskAssignment(list, clist, P, CN)

Input: *list*, *clist*, *P*, *CN*;
Output: makespan;
1: **while** $list != \emptyset$ **do**
2: $v_i \leftarrow$ the task with maximum $rank_u(v_i)$ in *list*;
3: **for** $j = 1$ to M **do**
4: calculate $EFT(v_i, p_j)$ by (9);
5: **end for**
6: schedule v_i on processor $pro(v_i)$ which makes v_i have the minimum *EFT*;
7: $AFT(v_i, pro(v_i)) \leftarrow EFT(v_i, pro(v_i))$ by (10);
8: **if** $makespan < AFT(v_i, pro(v_i))$ **then**
9: $makespan \leftarrow AFT(v_i, pro(v_i))$;
10: **end if**
11: **for** $j = 1$ to CN **do**
12: **for** $k = 1$ to M **do**
13: calculate $EFT(v_i^{b_j}, p_k)$ by (9);
14: **end for**
15: **if** $clist != \emptyset \&\& v_i \in clist$ **then**
16: schedule $v_i^{b_j}$ on processor $pro(v_i^{b_j})$ which makes $v_i^{b_j}$ have the minimum *EFT*;
17: $BAFT(v_i^{b_j}, pro(v_i^{b_j})) \leftarrow EFT(v_i, pro(v_i^{b_j}))$ by (11);
18: **if** $makespan < BAFT(v_i^{b_j}, pro(v_i^{b_j}))$ **then**
19: $makespan \leftarrow BAFT(v_i^{b_j}, pro(v_i^{b_j}))$;
20: **end if**
21: **end if**
22: **end for**
23: **end while**
24: **return** makespan.

value of *CN* is set to be two. That is, the number of replicas for each primary task shall be no larger than two. Fifthly, it checks if the makespan exceeds the given deadline. If the makespan is larger than D, then the last task added to the list *clist* will be taken off (line 17). Finally, it updates the makespan and calculates the reliability that the system can achieve under the final scheduling scheme by Equation (5) (lines 19-20). The time complexity of Algorithm 1 is $O(N + |E|) + O(NlogN) + O(N^2M|E|) = O(N^2M|E|)$, where $|E|$ is the number of edges.

Algorithm 2 shows how to map the primary tasks and their replicas to proper processors. The Input is three arrays *list*, *clist*, *P* and a parameter *CN*. The Output is the *makespan*. While *list* is not empty, the algorithm first select the task with the maximum uprank value, calculates its earliest execution finish time on all processors, assigns it to the processor with the minimum earliest execution finish time, and updates the makespan. Then, the algorithm calculates the assignment for the replicas of the selected task and updates the makesapn. The time complexity of Algorithm 2 is $O(NM|E|)$.

V. EXPERIMENT

To evaluate the effectiveness of the proposed algorithm, several series of experiments have been conducted on a computer with the 64-bit Windows 10 operating system, a dual processor Intel(R) Core (TM) CPU @ 2.2GHz and an 8GB RAM. Two algorithms DB-FTSA [20] and FTSA [32] are selected as comparison baselines. FTSA is an excellent fault tolerance algorithm for solving reliability problems with no timing constraint. It requires that the primary task and its replicas must be assigned to different processors. DB-FTSA is the latest and efficient fault tolerance algorithm to solve the problem similar to ours. It only supports up to one replicas and allows the primary task and its replicas to be assigned to the same processor. Randomly generated application graphs with various characteristics and real-world applications are adopted as test instances. In the following, we present the experimental parameters and metrics.

A. EXPERIMENTAL SETTING

In the experiments, we use randomly generated directed acyclic graphs which have been widely used in many studies, such as [38], [39]. To generate a directed acyclic graph, several parameters are needed:

- N, the number of tasks in a directed acyclic graph, its value is ranged from 20 to 100 with the increment of 20.
- the indegree of a task, its value is randomly generated from the interval [0, 4].
- the outdegree of a task, its value is randomly generated from the interval [0, 4].

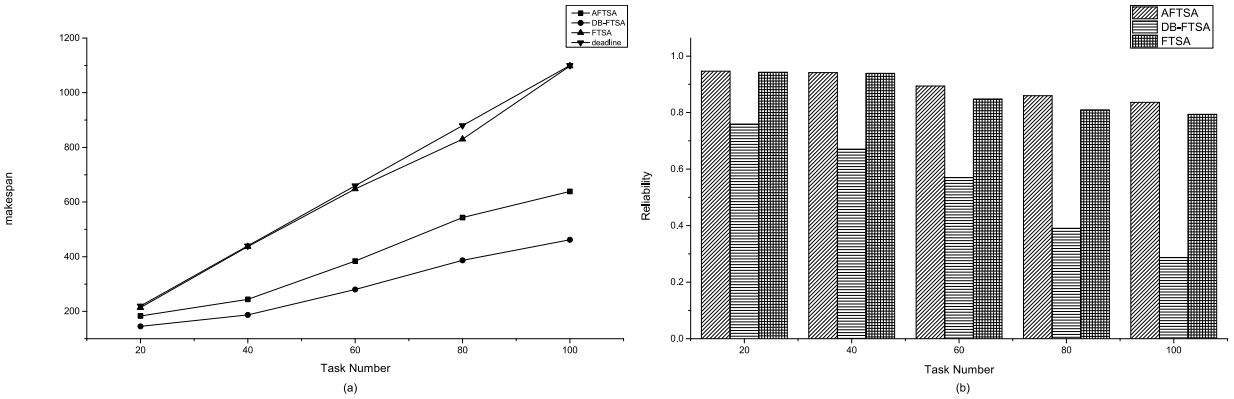


FIGURE 4. Comparisons of the makespan (a) and reliability (b) generated by AFTSA, DB-FTSA, and FTSA for $CCR = 1$, $\overline{ET} = 11$, $M = 4$.

- *CCR*, the communication to computation ratio which is equal to the value of the average communication time divided by the average computation time in a system. Like many other studies, the value of CCR is selected from {0.1, 0.5, 1, 2, 5}.

To account for communication heterogeneity in the system, the unit data delay of the processors is chosen uniformly from the range of [0, 2]. In addition, we consider two system platforms. One is composed of four connected processors and the other consists of eight connected processors. The failure rate λ of processors is randomly generated from the range of $[1 \times 10^{-2}, 2 \times 10^{-2}]$.

We choose two metrics to assess the performance of the proposed algorithm as follows:

- *Reliability*: it is an important metric to measure whether a fault-tolerant algorithm is effective. A higher reliability means a better performance in system reliability.
- *Makespan*: it is the time to complete all tasks of a given application. A shorter makespan means a lower system delay.

In what follows, we will present and discuss the experimental results in detail.

B. EXPERIMENTAL RESULTS AND DISCUSSIONS FOR RANDOMLY GENERATED APPLICATIONS

Fig.4 shows the makespan and reliability produced by algorithms AFTSA, DB-FTSA, and FTSA for different benchmarks with different numbers of tasks when $CCR = 1$, the average execution time $\overline{ET} = 15$, and the number of processor $M = 4$ under the given deadline. It is observed that from Fig.4(a), DB-FTSA performs best in terms of *makespan*, while AFTSA performs slightly worse than DB-FTSA but better than FTSA. This is probably because DB-FTSA takes up to one replica into consideration while AFTSA and FTSA allow multiple replicas. More replicas will take more execution time. Also, AFTSA allows the primary task and its replicas to be assigned to the same processor while FTSA requires the primary task and its replica to be assigned to different processors, which incur additional communication time and a longer makespan. With the number of tasks increasing, the makespan produced by all above three algorithms increases. Fig.4(b) shows that in terms of *reliability*, AFTSA always maintains higher reliability compared with the other algorithms. Reasons are multifold: AFTSA can generate multiple replicas for each primary task which has a good effect on improving reliability; FTSA needs much more time to generate more replicas to provide higher reliability, and DB-FTSA cannot make full use of the given time which leads to a lower reliability. With the number of tasks increasing, the reliability generated by all above three algorithms decreases.

Fig.5 shows the makespan and reliability produced by algorithms AFTSA, DB-FTSA, and FTSA when $CCR = 1$, the average execution time $\overline{ET} = 15$, and the number of processor $M = 8$ under given deadline. On the whole, Fig.4 and Fig.5 convey to us similar information: For all three algorithms, with the number of tasks increasing, the makespan increases and the reliability decreases. It is worth pointing out that AFTSA can obtain higher reliability. The difference lies in that for the same benchmark, the platform with eight processors leads to lower makespan and higher reliability in comparison with the platform with four processors. This is because more resources is helpful to reduce the burden on the system and improve its performance. This shapes the very advantage of the parallel computing with multiple processors.

Table 2 shows the makespan and reliability of benchmarks in Fig. 4 produced by algorithms AFTSA, DB-FTSA, and FTSA when the given deadline increases. It is observed that when the deadline increases, the makespan and reliability generated by all three algorithms also increase. When the deadline increases to a certain value, if it continues to increase, the makespan and reliability generated by AFTSA and DB-FTSA keep unchanged. Because AFTSA can support one replica at most and DB-FTSA can tolerate up to one replica, when the deadline is small, the two algorithms cannot guarantee the maximum number of replicas for each task and get small makespan and reliability; when the deadline is large enough, the two algorithms can achieve the maximum

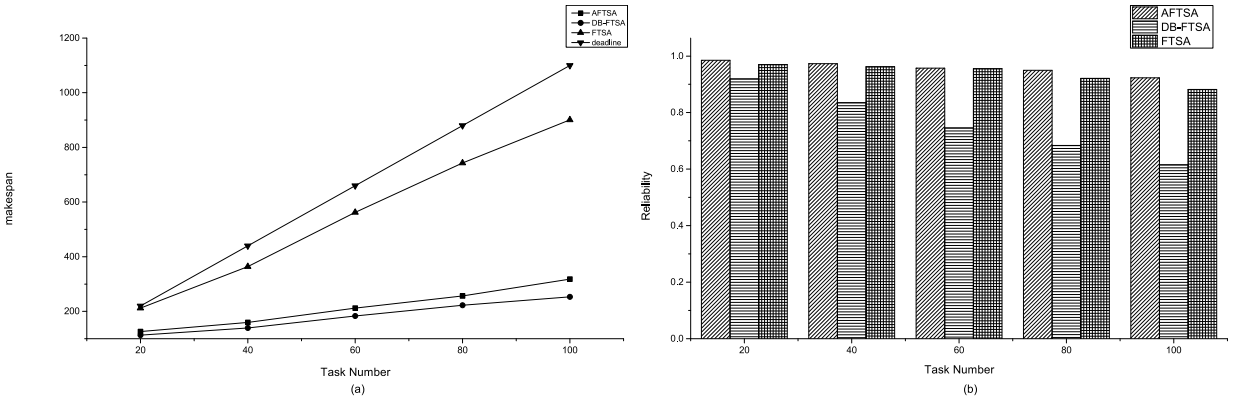


FIGURE 5. Comparisons of the makespan (a) and reliability (b) generated by AFTSA, DB-FTSA, and FTSA for $CCR = 1$, $\overline{ET} = 11$, $M = 8$.

TABLE 2. Changes of makespan and reliability of benchmarks in Fig. 4 when deadline increases.

Number of tasks	Deadline	AFTSA		DB-FTSA		FTSA	
		Makespan	Reliability	Makespan	Reliability	Makespan	Reliability
20	120	118	23.96%	118	23.96%	60	16.23%
	170	149	75.72%	149	75.72%	162	71.65%
	220	178	96.30%	149	75.72%	214	94.24%
	620	178	96.30%	149	75.72%	409	99.38%
	1020	178	96.30%	149	75.72%	887	100.00%
40	340	244	94.16%	187	67.05%	303	50.93%
	390	244	94.16%	187	67.05%	344	58.33%
	440	244	94.16%	187	67.05%	437	93.91%
	840	244	94.16%	187	67.05%	799	98.53%
	1240	244	94.16%	187	67.05%	799	98.53%
60	560	384	89.37%	280	47.05%	540	41.35%
	660	384	89.37%	280	47.05%	648	84.75%
	1060	384	89.37%	280	47.05%	840	86.13%
	1460	384	89.37%	280	47.05%	1275	96.46%
80	780	543	85.95%	387	39.05%	738	33.62%
	880	543	85.95%	387	39.05%	831	80.92%
	1280	543	85.95%	387	39.05%	1061	82.57%
	1680	543	85.95%	387	39.05%	1536	95.68%
100	1000	639	83.66%	462	28.75%	867	21.46%
	1100	639	83.66%	462	28.75%	1098	79.42%
	1500	639	83.66%	462	28.75%	1374	81.28%
	1900	639	83.66%	462	28.75%	1896	95.34%

number of replicas for each task and get large makespan reliability. If the deadline further becomes larger, the makespan and reliability will keep invariant. However, FTSA can get increasing makespan and reliability because it can support more and more replicas with deadline increasing. What is more, AFTSA can obtain higher reliability and substantially less makespan compared with FTSA, and this helps to save time and resources.

Fig.6 shows the reliability produced by algorithms AFTSA, DB-FTSA, and FTSA for different benchmarks when the average execution time and *CCR* change their values with the platform of $M = 4$ processors under given deadline. Fig.6(a) shows the reliability gained for benchmarks with $N = 40$, $CCR = 1$, and different average execution time. Table 3 shows the deadline and the makespan generated by algorithms AFTSA, DB-FTSA, and FTSA for each benchmark. It can be seen that AFTSA can obtain the highest reliability, followed by FTSA, and followed by DB-FTSA.

TABLE 3. Deadline and makespan for benchmarks in Fig. 6.

$\overline{ET}$	Deadline	AFTSA	DB-FTSA	FTSA
12	480	159	160	375
14	560	220	199	501
16	640	257	222	590
18	720	282	264	641
20	800	283	260	712
CCR	Deadline	AFTSA	DB-FTSA	FTSA
0.1	600	123	89	387
0.5	600	159	137	437
1	600	211	185	590
5	1800	235	208	1750

This is because AFTSA can produce more replicas for each primary task and get higher reliability than FTSA, and DB-FTSA supports no more than two replicas for each primary task. More replicas usually denote higher reliability. Fig.6(b) shows the reliability gained for benchmarks

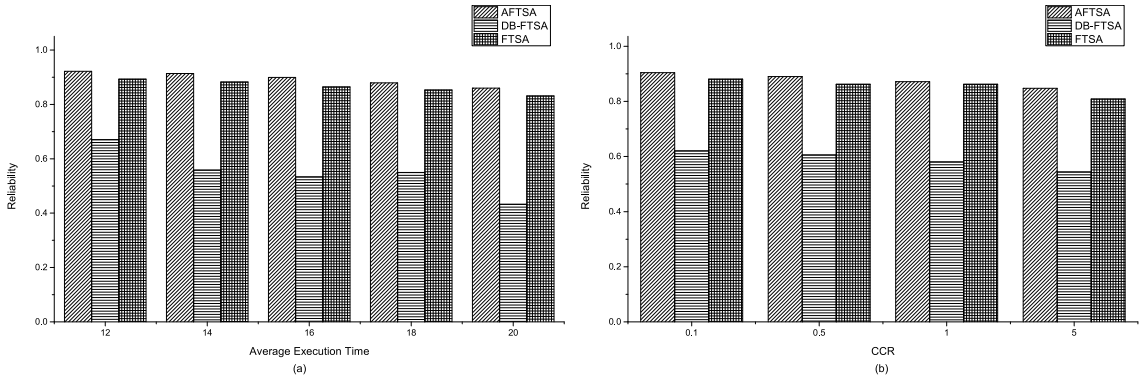


FIGURE 6. Comparison of the reliability generated by AFTSA, DB-FTSA, FTSA with $M = 4$. (a)$N = 40$, $CCR = 1$; (b)$N = 50$, $\overline{ET} = 15$.

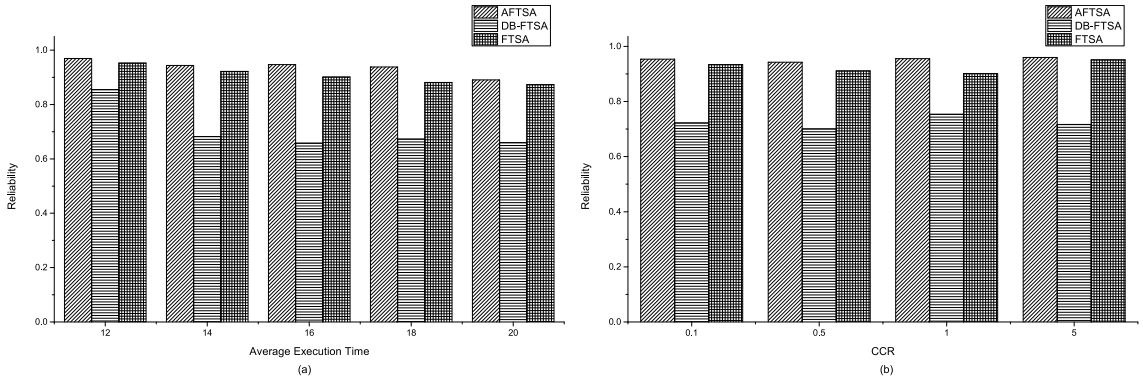


FIGURE 7. Comparison of the reliability generated by AFTSA, DB-FTSA, FTSA with $M = 8$. (a)$N = 40$, $CCR = 1$; (b)$N = 50$, $\overline{ET} = 15$.

TABLE 4. Deadline and makespan for benchmarks in Fig. 7.

$\overline{ET}$	Deadline	AFTSA	DB-FTSA	FTSA
12	480	323	249	433
14	560	347	297	511
16	640	429	327	577
18	720	440	370	678
20	800	494	360	765
CCR	Deadline	AFTSA	DB-FTSA	FTSA
0.1	600	334	237	453
0.5	600	369	263	506
1	600	403	306	593
5	1800	693	603	1511

with $N = 50$, $\overline{ET} = 15$, and different CCR value. It can be seen that AFTSA can still achieve better reliability than the other two algorithms.

Fig.7 shows the reliability produced by algorithms AFTSA, DB-FTSA, and FTSA for different benchmarks when the average execution time and CCR change their values with the platform of $M = 8$ processor under the deadline. Table 4 shows the deadline and the makespan generated by algorithms AFTSA, DB-FTSA, and FTSA for each benchmark in Fig. 7. Similar to Fig. 6, it reveals to us that AFTSA is more reliable than the other two algorithms within deadline. The only difference lies in that for the same benchmark, the platform of eight processors achieves higher reliability than the platform of four processors since more resources are beneficial to reducing the burden of systems and improve efficiency.

C. EXPERIMENTAL RESULTS AND DISCUSSIONS FOR REAL-WORLD APPLICATIONS

This subsection considers three types of real-word applications: Gaussian elimination [40], Fast fourier transform (FFT) [39] and a molecular dynamics code [38] which are adopted to test the effectiveness of the proposed algorithm. Task graph instances for each of these three applications from [40] are shown in Fig. 8, Fig. 9 and Fig. 10.

1) GAUSSIAN ELIMINATION

The structure of data-flow graph for gaussian elimination applications is already known, and we only need to know the number of tasks N and the execution time of tasks for these applications. N is computed by a formula $N = \frac{MS^2+MS-2}{2}$ [40], where MS is the matrix size of the coefficient matrix for gaussian elimination applications.

Fig.11 shows the makespan and reliability produced by algorithms AFTSA, DB-FTSA, and FTSA for Gaussian elimination applications with different matrix size when $CCR = 1$, $\overline{ET} = 15$, and the deadline $D = MS \times (\overline{ET} + \overline{ET} \times CCR)$ on the platform of four processors. The matrix

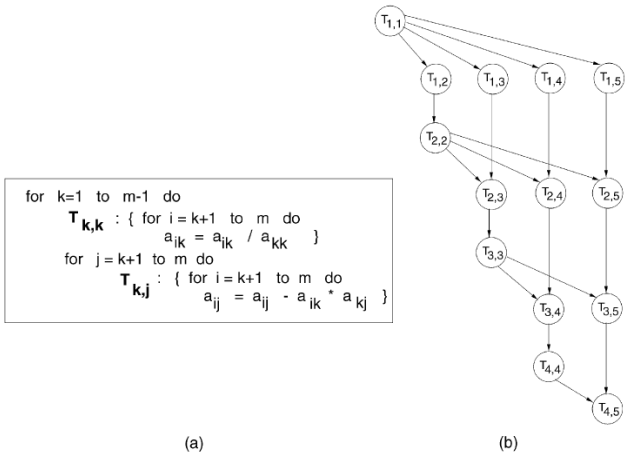


FIGURE 8. (a) Gaussian elimination, (b) task graph for matrix of size 5 [40].

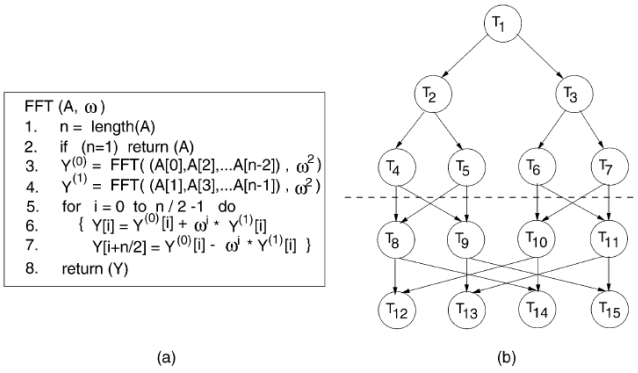


FIGURE 9. (a) FFT algorithm, (b) the generated DAG of FFT with four points [40].

size of gaussian elimination applications increases from 5 (corresponding to 14 tasks) to 20 (corresponding to 209 tasks) with an increment of 3. In terms of *makespan*, DB-FTSA performs best, AFTSA performs slightly worse than DB-FTSA, and FTSA obtains a much larger makespan than the former two algorithms. In terms of *Reliability*, AFTSA always performs best, and DB-FTSA does better than FTSA in most cases. Fig.12 shows the makespan and reliability produced by algorithms AFTSA, DB-FTSA, and FTSA for Gaussian elimination applications with different matrix size when $CCR = 1$, $\overline{ET} = 15$, and the deadline $D = MS \times (\overline{ET} + \overline{ET} \times CCR)$ on the platform of eight processors. It tells a similar story as Fig.11. One difference is that in reliability, AFTSA performs best, followed by FTSA, and further by DB-FTSA. The other difference is that on the platform with eight processors, all three algorithms can obtain a smaller *makespan* and a higher reliability, and the makespan generated by AFTSA is very close to that generated by DB-FTSA. When the matrix size is less than or equal to 14, AFTSA and DB-FTSA have the same makespan. The reason is that more resources generally provide more opportunities for reducing makespan and improving reliability.

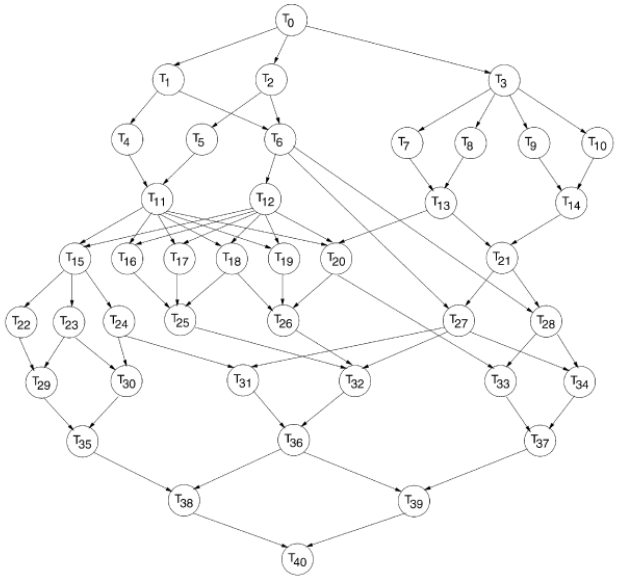


FIGURE 10. The task graph of the molecular dynamics code [40].

2) FAST FOURIER TRANSFORM

The structure of data-flow graph for one-dimensional and recursive FFT applications is known [39], and we only need to know the number of tasks and the execution time of tasks for these applications. The FFT algorithm with an input vector size of S has $2S - 1$ recursive call tasks and $S \log S$ butterfly operation tasks.

Fig.13 shows the makespan and reliability produced by algorithms AFTSA, DB-FTSA, and FTSA for FFT applications at different vector size when $\overline{ET} = 15$, $CCR = 1$, and the deadline $D = (2 \times \log_2 S + 1) \times (\overline{ET} + \overline{ET} \times CCR)$ on the platform of four processors. S varies from 2 to 32 with a multiplier of 2. In view of *makespan*, DB-FTSA performs best, followed by AFTSA, and further by FTSA, and the gap between them becomes wider and wider with the increase of the vector size. In view of *Reliability*, AFTSA always keeps higher reliability among three algorithms, especially when the vector size is large. After all, more replicas generally mean longer makespan and higher reliability with limited resource, and that the primary task and their replicas are assigned to different processors will increase communication time and cause a longer makespan. Fig.14 shows the makespan and reliability produced by algorithms AFTSA, DB-FTSA, and FTSA for FFT applications at different vector size when $\overline{ET} = 15$, $CCR = 1$, and the deadline $D = (2 \times \log_2 S + 1) \times (\overline{ET} + \overline{ET} \times CCR)$ on the platform of eight processors. Fig.13 and Fig.14 reflect similar insight. All three algorithms can obtain a smaller *makespan* and a higher reliability for each benchmark on the platform of eight processors, and the makespan obtained by AFTSA and DB-FTSA are very close but much smaller than that by FTSA, especially when the vector size is large. The reason is that more processors are more likely to reduce makespan and improve reliability.

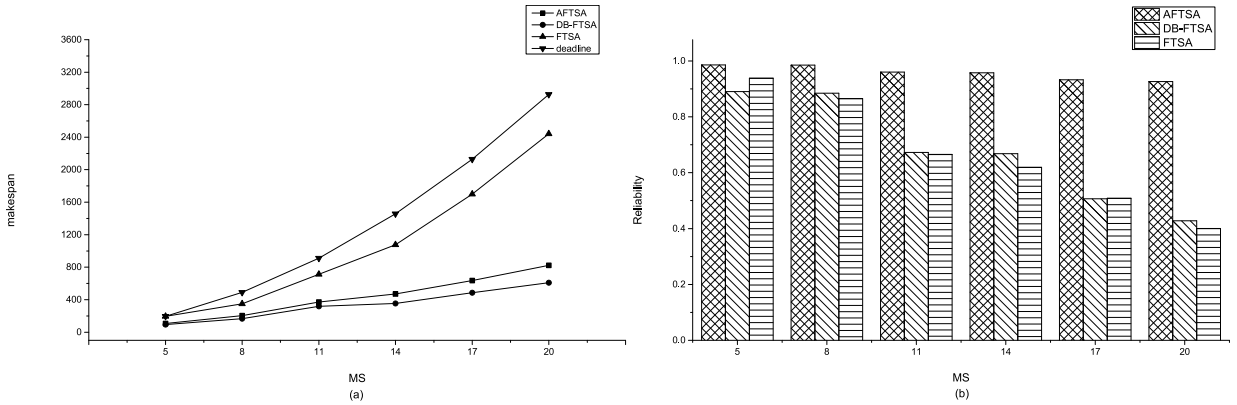


FIGURE 11. Makespan, Reliability generated by AFTSA, DB-FTSA and FTSA for Gaussian elimination applications with $\overline{ET} = 15$, $CCR = 1$, $D = MS \times (\overline{ET} + \overline{ET} \times CCR)$, $M = 4$.

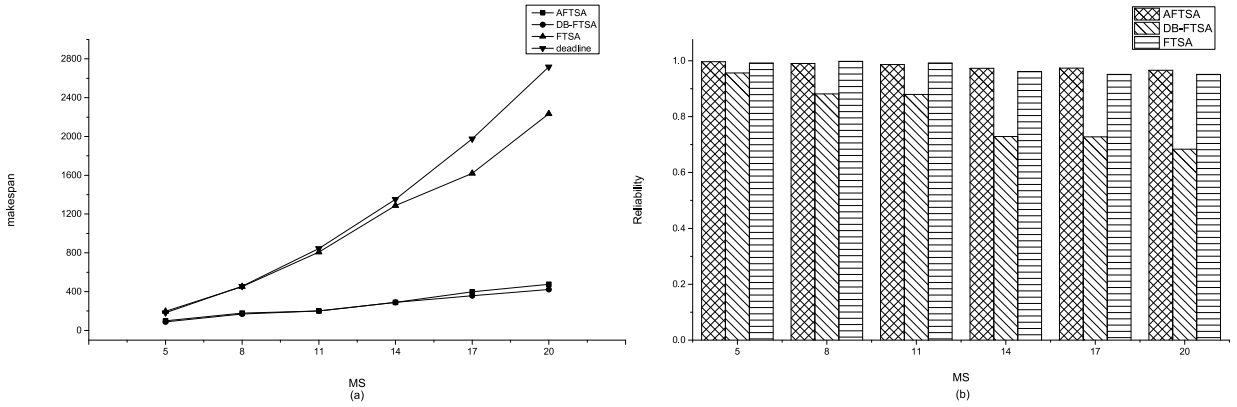


FIGURE 12. Makespan, Reliability generated by AFTSA, DB-FTSA and FTSA for the Gaussian elimination applications with $\overline{ET} = 15$, $CCR = 1$, $D = MS \times (\overline{ET} + \overline{ET} \times CCR)$, $M = 8$.

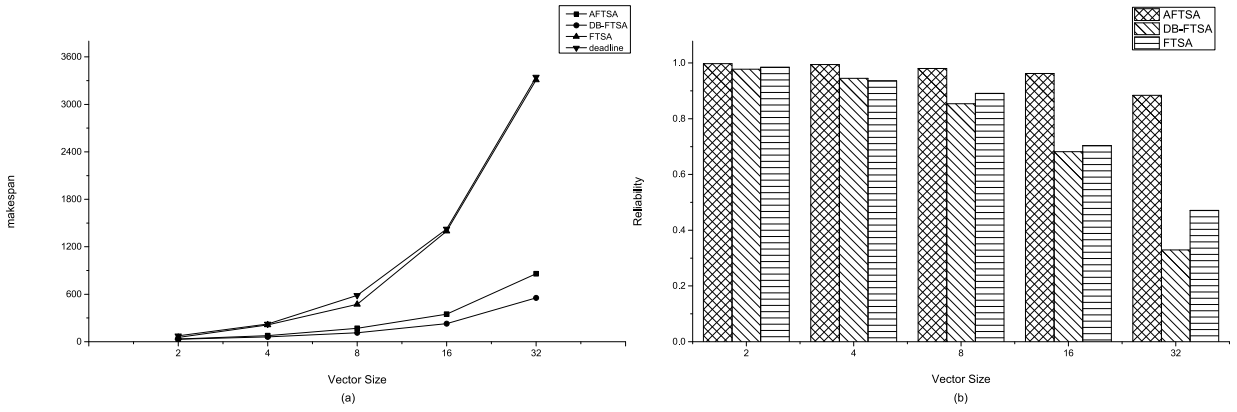


FIGURE 13. Makespan, Reliability generated by AFTSA, DB-FTSA and FTSA for FFT applications with $\overline{ET} = 15$, $CCR = 1$, $D = (2 \times \log_2 S + 1) \times (\overline{ET} + \overline{ET} \times CCR)$, $M = 4$.

3) MOLECULAR DYNAMICS CODE

The structure of data-flow graph for molecular dynamics code applications and the number of tasks are known [40], and we only consider two factors: *CCR* and the average execution time of tasks in the experiments.

Fig.15 shows the makespan and reliability generated by algorithms AFTSA, DB-FTSA, and FTSA for molecular dynamics code applications at different values of *CCR* when $\overline{ET} = 15$ and $M = 4$ within the given deadline. It can be seen that AFTSA can obtain the highest reliability, followed by FTSA, and followed by DB-FTSA. FTSA can obtain the largest makespan, followed by AFTSA, and followed by DB-FTSA. Because AFTSA and FTSA can support more replicas for each task than DB-FTSA, and more replicas usually mean higher reliability and longer makespan. In addition, the primary task and its replicas can be allowed to be assigned

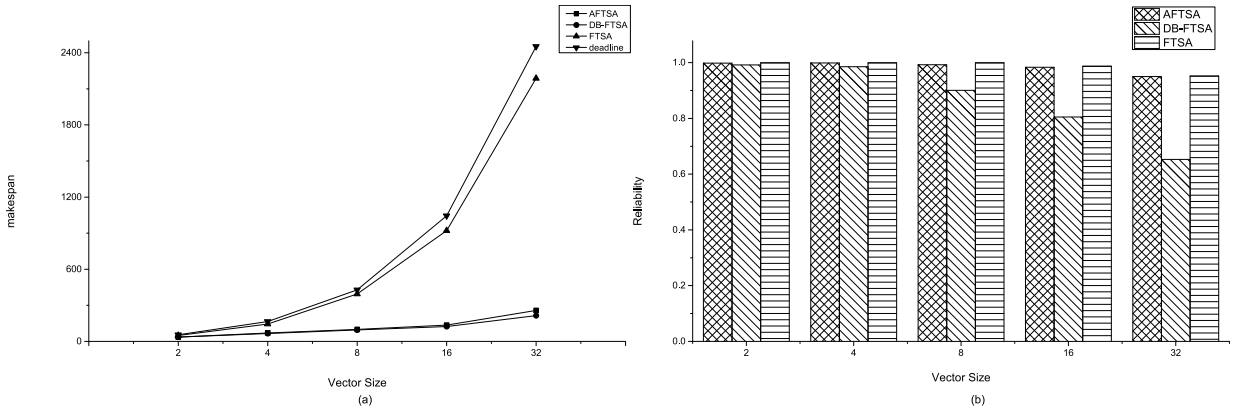


FIGURE 14. Makespan, Reliability generated by AFTSA, DB-FTSA and FTSA for FFT applications with $\overline{ET} = 15$, $CCR = 1$, $D = (2 \times \log_2 S + 1) \times (\overline{ET} + \overline{ET} \times CCR)$, $M = 8$.

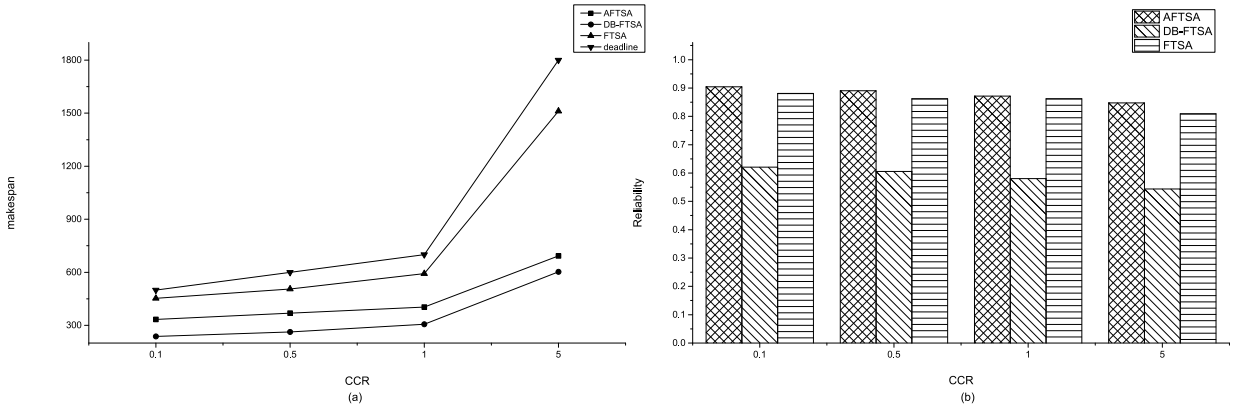


FIGURE 15. Makespan, Reliability generated by AFTSA, DB-FTSA and FTSA for the task graph of a molecular dynamics code with $\overline{ET} = 15$, $CCR = 1$, $D = 500$, $M = 4$.

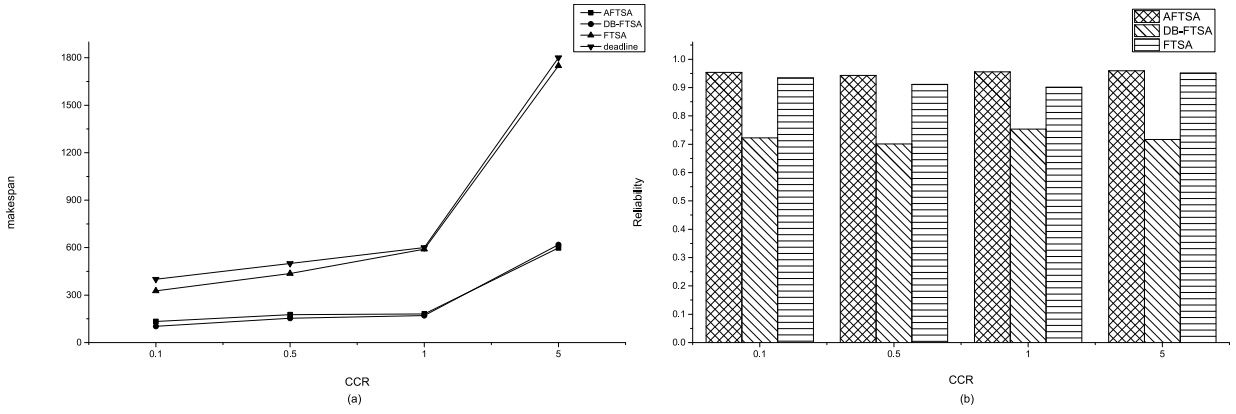


FIGURE 16. Makespan, Reliability generated by AFTSA, DB-FTSA and FTSA for the task graph of a molecular dynamics code with $\overline{ET} = 15$, $CCR = 1$, $D = 500$, $M = 8$.

to the same processor to reduce communication time and makespan for AFTSA. When the value of *CCR* is smaller than or equal to 1, the gap of the makespan obtained from any two algorithm is not very large; when the value of *CCR* is larger than 1, the makespan obtained by three algorithms increase dramatically, and the makespan obtained by AFTSA and DB-FTSA are quite close but much smaller than that by FTSA. Fig.16 shows the makespan and reliability generated by algorithms AFTSA, DB-FTSA, and FTSA for molecular dynamics code applications at different values of *CCR* when $\overline{ET} = 15$ and $M = 8$ within the given deadline. It takes on similar information as Fig.15. The difference is that on the platform of eight processors, all three algorithms can obtain a smaller makespan and a higher reliability for the same benchmark. This is because more processors embody more likelihood to reduce makespan and improve reliability.

According to all above experimental results and analysis, we conclude that AFTSA can always get higher reliability within limited time compared with DB-FTSA and FTSA, and more resources provide more opportunities to reduce makespan and improve reliability.

VI. CONCLUSION

In this paper, we propose a novel, adaptive and transient fault-tolerant scheduling algorithm to solve the fault-tolerant problem on heterogeneous systems, aiming to optimize the reliability in a shared deadline. The proposed algorithm allows multiple replicas, which has been proved in its capability to improve reliability. The primary task and its replicas can be assigned to the same processor which helps to decrease makespan. The proposed algorithm first works out the maximum number of replicas for each primary task that the system can tolerate within the given deadline. Then it assigns all primary tasks and their replicas to appropriate processors and computes the maximum reliability that the system can achieve. Simulated results show that the proposed algorithm can keep a higher reliability in comparison with the existing and related fault-tolerant algorithms. When the number of tasks for an application is large and the given deadline is large enough, the reliability obtained by our proposed algorithm will be smaller than that obtained by FTSA. In the future, we will continue to find a better way to solve this problem. Also, we will explore how to reduce energy consumption on the basis of improving reliability, and use more complicated fault model to solve this type of problems.

ACKNOWLEDGMENT

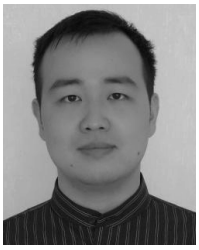
The authors would like to thank the editors and the referees. This article was presented in part at the 14th IEEE Conference on Industrial Electronics and Applications (ICIEA 2019).

REFERENCES

[1] K. Li, "Scheduling precedence constrained tasks with reduced processor energy on multiprocessor computers," *IEEE Trans. Comput.*, vol. 61, no. 12, pp. 1668–1681, Dec. 2012.
[2] J. Villamayor, D. Rexachs, E. Luque, and D. Lugones, "Raas: Resilience as a service," in *Proc. 18th IEEE/ACM Int. Symp. Cluster, Cloud Grid Comput. (CCGRID)*, May 2018, pp. 356–359.
[3] C. George and S. Vadhiyar, "Fault tolerance on large scale systems using adaptive process replication," *IEEE Trans. Comput.*, vol. 64, no. 8, pp. 2213–2225, Aug. 2015.
[4] F. Abdi, R. Tabish, M. Rungger, M. Zamani, and M. Caccamo, "Application and system-level software fault tolerance through full system restarts," in *Proc. 8th Int. Conf. Cyber-Phys. Syst.*, Apr. 2017, pp. 197–206.
[5] B. Zhao, H. Aydin, and D. Zhu, "Energy management under general task-level reliability constraints," in *Proc. IEEE 18th Real Time Embedded Technol. Appl. Symp.*, Apr. 2012, pp. 285–294.
[6] C. M. Krishna, "Fault-tolerant scheduling in homogeneous real-time systems," *ACM Comput. Surv.*, vol. 46, no. 4, p. 48, 2014.
[7] A. Das, A. Kumar, and B. Veeravalli, "Energy-aware task mapping and scheduling for reliable embedded computing systems," *Acm Trans. Embedded Comput. Syst.*, vol. 13, no. 2s, pp. 479–496, 2014.
[8] A. Aviziens, "Fault-tolerant systems," *IEEE Trans. Comput.*, vol. COM-100, no. 12, pp. 1304–1312, Dec. 1976.
[9] E. Dubrova, *Fault-Tolerant Design*. New York, NY, USA: Springer, 2013.
[10] R. Devaraj, A. Sarkar, and S. Biswas, "Fault-tolerant scheduling of non-preemptive periodic tasks using sct of timed des on uniprocessor systems," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 9315–9320, Jul. 2017.
[11] Y. Guo, D. Zhu, H. Aydin, and L. T. Yang, "Energy-efficient scheduling of primary/backup tasks in multiprocessor real-time systems (extended version)," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, Oct. 2013, pp. 896–902.
[12] H. Agarwal and A. Sharma, "A comprehensive survey of fault tolerance techniques in cloud computing," in *Proc. Int. Conf. Comput. Netw. Commun. (CoCoNet)*, Dec. 2015, pp. 408–413.
[13] X.-T. Cui, K.-J. Wu, T.-Q. Wei, and E. H.-M. Sha, "Worst-case finish time analysis for DAG-based applications in the presence of transient faults," *J. Comput. Sci. Technol.*, vol. 31, no. 2, pp. 267–283, Mar. 2016.
[14] J. J. Dongarra, E. Jeannot, E. Saule, and Z. Shi, "Bi-objective scheduling algorithms for optimizing makespan and reliability on heterogeneous systems," in *Proc. 19th Annu. ACM Symp. Parallel Algorithms Archit. SPAA*, 2007, pp. 280–288.
[15] W. Luo, F. Yang, L. Pang, and X. Qin, "Fault-tolerant scheduling based on periodic tasks for heterogeneous systems," in *Autonomic and Trusted Computing*, vol. 4158. Berlin, Germany: Springer, 2006, pp. 571–580.
[16] H. Yan, X. Zhu, H. Chen, H. Guo, W. Zhou, and W. Bao, "DEFT: Dynamic fault-tolerant elastic scheduling for tasks with uncertain runtime in cloud," *Inf. Sci.*, vol. 477, pp. 30–46, Mar. 2019.
[17] S. M. Abdulhamid, M. S. Abd Latiff, S. H. H. Madni, and M. Abdullahi, "Fault tolerance aware scheduling technique for cloud computing environment using dynamic clustering algorithm," *Neural Comput. Appl.*, vol. 29, no. 1, pp. 279–293, Jan. 2018.
[18] L. Zhang, K. Li, Y. Xu, J. Mei, F. Zhang, and K. Li, "Maximizing reliability with energy conservation for parallel task scheduling in a heterogeneous cluster," *Inf. Sci.*, vol. 319, pp. 113–131, Oct. 2015.
[19] M. H. Mottaghi and H. R. Zarandi, "DFTS: A dynamic fault-tolerant scheduling for real-time tasks in multicore processors," *Microprocessors Microsyst.*, vol. 38, no. 1, pp. 88–97, Feb. 2014.
[20] M. Wei, J. Liu, T. Li, X. Xu, W. Hu, and D. Zhao, "Fault-tolerant scheduling of real-time tasks on heterogeneous systems," in *Proc. 12th IEEE Conf. Ind. Electron. Appl. (ICIEA)*, Jun. 2017, pp. 1006–1011.
[21] Y. Guo, D. Zhu, and H. Aydin, "Generalized standby-sparing techniques for energy-efficient fault tolerance in multiprocessor real-time systems," in *Proc. IEEE 19th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, Aug. 2013, pp. 62–71.
[22] B. Zhao, H. Aydin, and D. Zhu, "Shared recovery for energy efficiency and reliability enhancements in real-time applications with precedence constraints," *Acm Trans. Design Autom. Electron. Syst.*, vol. 18, no. 2, pp. 99–109, 2013.
[23] G. Xie, Y. Chen, X. Xiao, C. Xu, R. Li, and K. Li, "Energy-efficient fault-tolerant scheduling of reliable parallel applications on heterogeneous distributed embedded systems," *IEEE Trans. Sustain. Comput.*, vol. 3, no. 3, pp. 167–181, Jul. 2018.
[24] N. Chatterjee, S. Paul, and S. Chattopadhyay, "Fault-tolerant dynamic task mapping and scheduling for network-on-chip-based multicore platform," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 16, no. 4, p. 108, 2017.
[25] P. P. Nair, R. Devaraj, and A. Sarkar, "FEST: Fault-tolerant energy-aware scheduling on two-core heterogeneous platform," in *Proc. 8th Int. Symp. Embedded Comput. Syst. Design (ISED)*, Dec. 2018, pp. 63–68.
[26] R. Devaraj, A. Sarkar, and S. Biswas, "Fault-tolerant preemptive aperiodic rt scheduling by supervisory control of tdes on multiprocessors," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 3, pp. 1–25, 2017.
[27] L. A. R. Duque, J. M. M. Diaz, and C. Yang, "Improving MPSoC reliability through adapting runtime task schedule based on time-correlated fault behavior," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2015, pp. 818–823.
[28] M. A. Haque, H. Aydin, and D. Zhu, "On reliability management of energy-aware real-time systems through task replication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 813–825, Mar. 2017.
[29] J. Zhou, X. S. Hu, Y. Ma, J. Sun, T. Wei, and S. Hu, "Improving availability of multicore real-time systems suffering both permanent and transient faults," *IEEE Trans. Comput.*, vol. 68, no. 12, pp. 1785–1801, Dec. 2019.
[30] J. Zhou, J. Sun, X. Zhou, T. Wei, M. Chen, S. Hu, and X. S. Hu, "Resource management for improving soft-error and lifetime reliability of real-time MPSoCs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 12, pp. 2215–2228, Dec. 2019.
[31] A. Benoit, M. Hakem, and Y. Robert, "Fault tolerant scheduling of precedence task graphs on heterogeneous platforms," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, Apr. 2008, pp. 1–8.

[32] L. Zhao, Y. Ren, Y. Xiang, and K. Sakurai, "Fault-tolerant scheduling with dynamic number of replicas in heterogeneous systems," in *Proc. IEEE 12th Int. Conf. High Perform. Comput. Commun. (HPCC)*, Sep. 2010, pp. 434–441.
[33] A. K. Samal, R. Mall, and C. Tripathy, "Fault tolerant scheduling of hard real-time tasks on multiprocessor system using a hybrid genetic algorithm," *Swarm Evol. Comput.*, vol. 14, pp. 92–105, Feb. 2014.
[34] M. C. Kurt, S. Krishnamoorthy, K. Agrawal, and G. Agrawal, "Fault-tolerant dynamic task graph scheduling," in *Proc. SC14: Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2014, pp. 719–730.
[35] M. Lauer, M. Amy, J.-C. Fabre, M. Roy, W. Excoffon, and M. Stoicescu, "Engineering adaptive fault-tolerance mechanisms for resilient computing on ROS," in *Proc. IEEE 17th Int. Symp. High Assurance Syst. Eng. (HASE)*, Jan. 2016, pp. 94–101.
[36] C. Wongyai, "Improve fault tolerance in cell-based evolve hardware architecture," in *Proc. Int. Conf. Adv. Comput. Sci. Inf. Syst.*, Oct. 2014, pp. 13–18.
[37] H. Jin, X.-H. Sun, Z. Zheng, Z. Lan, and B. Xie, "Performance under failures of DAG-based parallel computing," in *Proc. 9th IEEE/ACM Int. Symp. Cluster Comput. Grid*, May 2009, pp. 236–243.
[38] M. I. Daoud and N. Kharma, "A high performance algorithm for static task scheduling in heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 68, no. 4, pp. 399–409, Apr. 2008.
[39] C.-Y. Chen, "Task scheduling for maximizing performance and reliability considering fault recovery in heterogeneous distributed systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 2, pp. 521–532, Feb. 2016.
[40] H. Topcuoglu, S. Hariri, and M. Y. Wu, "Performance-effective and low-complexity task scheduling forheterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, 2002.

JING LIU received the B.S. degree from the College of Mathematics and Econometrics, in 2009, and the Ph.D. degree from the College of Information Science and Engineering, Hunan University, Changsha, China, in 2015. She is currently a Lecturer with the School of Computer Science and Technology, Wuhan University of Science and Technology, Wuhan, China. Her current research interests include scheduling, fault tolerance, real-time systems, and edge computing.



ZIQI ZHU received the B.S. degree in computer science from Wuhan University, Wuhan, China, in 2005, and the Ph.D. degree from the Huazhong University of Science and Technology, Wuhan, in 2011. He is currently an Associate Professor with the School of Computer Science and Technology, Wuhan University of Science and Technology, Wuhan. His current research interests include scheduling, machine learning, pattern recognition, and computer vision.

CHUNHUA DENG received the Ph.D. degree in pattern recognition and intelligent systems from the School of Automation, Huazhong University of Science and Technology, Wuhan, China, in 2016. He is currently a Lecturer with the School of Computer Science and Technology, Wuhan University of Science and Technology, Wuhan. His current research interests include computer vision, pattern recognition, and machine learning.