Journal of Systems Architecture 90 (2018) 23–33

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

Task scheduling with fault-tolerance in real-time heterogeneous systems

Jing Liu[a,b,*], Mengxue Wei[a,b], Wei Hu[a,b], Xin Xu[a,b], Aijia Ouyang[c]

[a] *College of Computer Science and Technology, Wuhan University of Science and Technology, China*
[b] *Hubei Province Key Laboratory of Intelligent Information Processing and Real-time Industrial System, China*
[c] *School of Information Engineering, Zunyi Normal University, China*

ARTICLE INFO

Keywords:
Fault-tolerance
Permanent failures
Scheduling algorithms
Reliability
Time constraint

ABSTRACT

Nowadays, the performance of heterogeneous systems has been improved dramatically, which also increases the complexity of heterogeneous systems, leading to the growing potential of system failures. Failures can be masked through scheduling approaches. Efficient task scheduling with fault-tolerance can guarantee the execution of tasks and satisfy the real-time nature. In this paper, we address the problem of scheduling tasks on heterogeneous systems with the target to support the maximum number of permanent failures while meeting a given time constraint. The problem is NP-hard and we propose a heuristic algorithm DBSA to solve it. DBSA can dynamically calculate the number of tolerating permanent failures. Firstly, the makespan when systems tolerate a fixed number of failures is calculated. Then, DBSA gets the actual number of tolerating failures by constantly comparing the makespan with a given deadline. Finally, DBSA maps tasks to appropriate processors without violating precedence constraints. Experimental results demonstrate that DBSA can effectively tolerate failures and improves system reliability.

1. Introduction

Many heterogeneous systems, especially safety-critical systems, have real-time and fault-tolerance requirements, like automotive, aircraft and space shuttle [1]. That means such systems must guarantee their functional requirements with time constraints even in the presence of faults. Scheduling plays a substantial role in the performance of real-time systems. Effective fault-tolerant scheduling will greatly improve the reliability of systems while deadlines are satisfied. Therefore, there is an urgent need for researchers to study effective fault-tolerant scheduling algorithms to handle faults within time constraints.

Due to the effects of hardware defects, electromagnetic interferences and cosmic ray radiations, failures may occur at run-time [2]. Because the occurrence of failures is often unpredictable, fault-tolerance is considered in the design of real-time scheduling algorithms to improve the system reliability [3]. Real-time scheduling with fault-tolerant capability is implemented using redundancy either in time or space [4]. Usually, time redundancy is a viable and cost-efficient means to achieve fault-tolerance in many resource-constrained systems [1]. In this paper, we adopt time redundancy to support fault-tolerance while ensuring a time constraint. There are two main approaches to solve faults in time redundancy: the active scheme and the passive scheme. In these two schemes, each task has several copies which can be considered as a primary copy and multiple backup copies. In the active scheme, primary executes simultaneously with its backups, that is to say backups can begin to execute even though its primary does not fail [5]. If primary fails, backups will be responsible for critical functionality in the application [6]. On the contrast, the backups are allowed to execute only if the primary fails in the passive scheme [7]. This scheme assumes that real-time tasks must have enough laxity to re-execute their backups [8]. However, this assumption is unrealistic in practice, particularly when real-time systems are heavily loaded. The principal difference between these two approaches is that the active scheme entails forward error masking: all multiple copies of task are executed. In the passive scheme, a backup is executed only if the acceptance test flags a failure. The passive scheme assumes that a fault detection mechanism is employed to detect failures while the active scheme does not need this mechanism. Therefore, in this paper, we choose the active scheme to support fault-tolerance.

The two fault-tolerant approaches mentioned above are used to mask failures which can be transient or permanent [9]. When a transient failure occurs, the failed processor may soon recover and resume its duty. When a permanent failure occurs, the corresponding processor can not execute any tasks. In order to ensure the normal running of systems, tasks in the failed processor must be migrated to be executed on other available processors. It seems that comparing with transient failures, permanent failures are more intractable to deal with. In this paper, we focus on permanent failures. Using time redundancy to solve failures will inevitably increase the scheduling length of applications, which may make tasks miss deadline. Thus, how to efficiently schedule

* Corresponding author.
E-mail address: luijing_cs@wust.edu.cn (J. Liu).

https://doi.org/10.1016/j.sysarc.2018.08.007
Received 23 April 2018; Received in revised form 18 July 2018; Accepted 26 August 2018
Available online 27 August 2018

tasks with fault-tolerance while satisfying time constraint requirement has been identified as one of the most fundamental issues.

Many fault-tolerant scheduling algorithms have been developed in the last decade [10–17]. Zhao et al.[10] study energy efficient reliability scheduling targeting precedence constrained tasks, and each task has its own deadline. However, they do not consider communication time. Guo et al.[11,12] study similar problems and they focus on periodic tasks. Haque et al.[13] and Han et al.[14] study transient failures and focus on periodic tasks. Benoit et al.[15], Zhao et al.[16] and Broberg et al.[17] have made a fault-tolerance analysis on precedence tasks, and can handle multiple permanent failures. But they do not consider time constraint.

In this paper, we consider fault-tolerant scheduling for precedence constrained tasks running on heterogeneous systems. Our goal is to tolerate permanent failures to improve the system reliability as soon as possible while meeting a given time constraint. Scheduling precedence constrained tasks is NP-hard problem [18]. The problem we study falls in such problems, so it is also NP-hard. To solve our problem, a heuristic algorithm DBSA is proposed. DBSA algorithm finds out the number of faults that systems can achieved at first, then it efficiently maps tasks to proper processors with fault-tolerant capability. Besides, we propose the GetMakespan algorithm to compute the makespan when a fix number of faults are handled in an application. This algorithm can help us to find out the number of faults that can be fault-tolerant within a time constraint.

The main contributions of this paper include:

- We propose a preallocation algorithm GetMakespan to compute the makespan when it tolerates a fix number of faults.
- We propose a scheduling algorithm DBSA to find out how many faults systems can support in a given time constraint and to map tasks to the most appropriate processors.

We conduct extensive experiments on two sets of benchmarks which are synthetic benchmarks and real benchmarks and compare our algorithm with HEFT [19] and FTSA [15]. Experimental results show that the proposed algorithm DBSA can effectively tolerate failures and improve system reliability with time constraints. For instance, in the Gaussian elimination experiments, when matrix size is 17, DBSA gets shorter makespan and higher PSS than FTSA-1, FTSA-2 and FTSA-3 and gains a higher system reliability than HEFT by 13.1%.

The rest of this paper is organized as follows. Section 2 reviews the related work. In Section 3, we explain the models and define our problem. Section 4 gives a motivational example. Section 5 presents the proposed algorithm. Simulation results and the performance analyses are discussed in Section 6. And we conclude the paper in Section 7.

2. Related work

Real-time scheduling algorithms are implemented from uniprocessor to multiprocessor systems, from homogeneous to heterogeneous systems [20–25]. For instance, for a set of periodic tasks running on a uniprocessor system, the rate monotonic (RM) and earliest-deadline-first (EDF) scheduling algorithms are proven to be optimal for static and dynamic priority in preemptive scheduling algorithms, respectively [20]. The two algorithms prioritize and schedule tasks based on timing parameters. Bamakhrama and Stefanov [24] consider the problem of hard-real-time (HRT) multiprocessor scheduling of embedded streaming applications modeled as acyclic dataflow graphs. In this work, authors prove that the actors in acyclic Cyclo-Static Dataflow (CSDF) graphs can be scheduled as periodic tasks and provide a framework for computing the periodic task parameters (i.e., period and start time) of each actor, and handling sporadic input streams. Qamhieh [23] proposes two DAG scheduling approaches to solve the problem of real-time scheduling of parallel Directed Acyclic Graph (DAG) tasks on homogeneous multiprocessor systems. Xue et al.[25] propose an algorithm D_MHEFT to meet the deadlines of more high-criticality functions in heterogeneous distributed

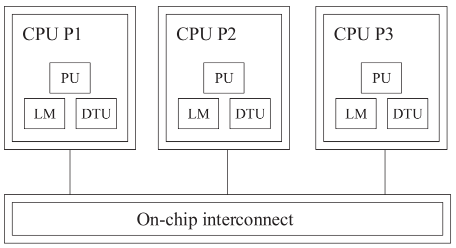


Fig. 1. An example of a system architecture model with three heterogeneous processors.

embedded systems. Liu et al.[26] propose several heuristic techniques ISGG, RLD, and RLDG to address the problem of scheduling tasks on heterogeneous multicore embedded systems with the constraints of time and resources for minimizing the total cost, while considering the communication overhead. However, these algorithms mentioned above do not provide mechanisms for fault-tolerance.

Researchers have developed many scheduling algorithms to tolerate failures to render systems highly reliable in the past decades [1,7,14,27,28]. Some focus on transient failures [13,14,29]. Haque et al.[13] propose static solutions and dynamic adaptation schemes to address the problem of achieving a given reliability target for a set of periodic real-time tasks running on a multi-core system with minimum energy consumption. Han et al.[14] develop effective scheduling methods to minimize energy consumption and, at the same time, tolerate up to K transient faults according to the EDF policy. Similarly, it also focuses on periodic tasks.

Some focus on permanent failures [5,8,15–17,30] [31]. Zhu et al.[8] present a fault-tolerant scheduling algorithm called QAFT that can tolerate one nodes permanent failures at one time instant for real-time tasks with QoS needs on heterogeneous clusters. Wang et al.[5] present a fault-tolerant mechanism which extends the primary-backup model to incorporate the features of clouds and designs a novel fault-tolerant elastic scheduling algorithm for real-time tasks in clouds named FESTAL, aiming at achieving both fault tolerance and high resource utilization in clouds. These two algorithm can only tolerate one permanent failure. Benoit et al.[15], Zhao et al.[16] and Broberg et al.[17] have made a fault-tolerance analysis on precedence tasks, and can handle multiple permanent failures. Their purpose is to reduce the schedule length as far as possible on the basis of meeting the requirement of reliability. Xue et al.[31] precisely study system reliability on the basis of Zhao et al.[16] and puts forward a specific reliability model. These studies consider similar problem to ours, but algorithms in these studies do not consider time constraints.

Different from the existing researches, the work reported in this paper integrates fault-tolerant, time-constrained and task-prioritized constraints to the scheduling problem. In addition, the proposed algorithm can dynamically decide the number of permanent failures systems can tolerate.

3. Models and problem definition

In this section, we first present the system model, task model, and fault model in detail. Then we present the problem definition.

3.1. System model

In this paper, we consider a heterogeneous system with M connected processors. Denote the set of these processors to be $P = \{p_1, p_2, p_3, \ldots, p_M\}$. Different processors have different processing speeds. Fig. 1 is an example of a system architecture model with three heterogeneous processors. Each processor includes three main elements: a processing unit (PU), a local memory (LM), and a data transfer unit

Table 1
Values of the communication time transferring a unit data between processors in Fig. 1.

P	P1	P2	P3
P1	0	2	1
P2	2	0	1
P3	1	1	0

(DTU) [32]. The PU executes a number of operations. The LM is a small-size and low-latency memory and is mainly accessed by the PU in the same processor during the PU's execution. The DUT is used to achieve data transfer operations between processors and memories. Communication between processors is through interconnect, such as bus. We assume that all interprocessor communications are performed without contention. Interconnects between processors may present different communication capacities. Communication capacity is symmetrical. That is, if denote $d(p_i, p_j)$ to be the communication time transferring a unit data from processor p_i to processor p_j, then we have $d(p_i, p_j) = d(p_j, p_i)$.

Table 1 gives an example of values of the communication time transferring a unit data between processors in Fig. 1. For example, the value 2 in the cell of row $P1$ and column $P2$ indicates that the communication time transferring a unit data from processor p_1 to p_2 is 2.

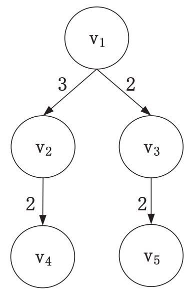
3.2. Task model

Like other studies, we model a real-time application by a weighted DAG $G =< V, E >$, where $V = \{v_1, v_2, \ldots, v_N\}$ is the set of all vertices, and each vertex that is also called node represents a real-time task. Hereafter, we use the terms "node" and "task" interchangeably in this paper. Also, $E \subseteq V \times V$ is the set of all edges between vertices, and each edge indicates a precedence relationship between tasks. If there exists an edge from node v_i to node v_j, denoted by (*vi, vj*) (namely e_{ij}), then node v_j cannot be executed until node v_i has been finished its execution. The value beside edge (*vi, vj*) represents the volume of data to be transferred from node v_i to node v_j, denoted by $data(v_i, v_j)$. Suppose that v_i is assigned to processor p_m and v_j is assigned to processor p_k, (due to different assignments between tasks executed on different processors, communication arise.) as in other studies [15,33], the communication time from task v_i to task v_j, denoted as $c(v_i, v_j)$ is calculated by the following formula: $c(v_i, v_j) = data(v_i, v_j) \times d(p_m, p_k)$. Meanwhile, if any two nodes v_i and v_j are mapped to the same processor, the communication time is assumed to be zero. Because the communication time in the same processor is negligible when it is compared with the communication time between different processors.

When a processor executes a node, it will take some time. Owing to heterogeneity existing both among processors and among tasks, different processors executing the same node will take different time. Likewise, a processor executing different tasks will also consume different time. The notation $w(v_i, p_j)$ is used to represent the taken time for processor p_j executing node v_i. Fig. 2 shows an example of values of execution time for tasks on different processors. For instance, the value 3 of cell (v_1, P1) represents that the execution time of task v_1 on p_1 is 3.

In order to support fault-tolerance in a processor, an active scheme is adopted in this paper. In the scheme, each task v_i can have multiple copies, separately labeled by $v_i{}^k (0 \le k \le ri, 0 \le ri \le M-1)$, where the copy $v_i{}^0$ represents the primary, and other copies are called backups. Copies of each task are assigned to different processors. All copies of one task have the same execution time for the same processor.

For task v_i, all its direct predecessor nodes are denoted as $pre(v_i)$ and all its direct successors are denoted as $succ(v_i)$. A task without any predecessor is called an *entry* task and a task without any successor is an *exit* task. A DAG may have multiple entry tasks and multiple exit tasks. Without loss of generality, we assume that a DAG has only one entry node and one exit node. If multiple entry tasks or multiple exit tasks exist, they may be connected with zero time weight edges to a single pseudo-entry task or a single pseudo-exit task that has zero time-weight.



(a)

Task	P1	P2	P3
v1	3	4	2
v2	2	3	3
v3	2	2	4
v4	2	1	3
v5	2	2	3

(b)

Fig. 2. An example of DAG. (a) A tree. (b) Values of execution time for tasks on different processors.

For each task v_i, $ft(v_i)$ denotes the finish time of task v_i. A DAG is completed only when its exit task is successfully executed. Therefore, the completed time (namely the makespan) of the given DAG, denoted by *makespan*, is obtained by

$$makespan = ft(v_{exit}). \tag{1}$$

3.3. Fault model

When designing a particular fault-tolerant system, the nature and frequency of faults are specified using a fault model. As in the researches [34,35], based on the common exponential distribution assumption, fault arrival rate is constant and the distribution of the fault count for any fixed time interval follows a poisson distribution, which means the fault distribution in time period t can be defined as

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

where k is the number of actual faults in time period t and λ_j denotes the fault rate of processor p_j. The reliability of task v_i, denoted by R_i, is the possibility that v_i is completed successfully. That is

$$R_i = f(0, \lambda_j) = e^{-\lambda_j w(v_i, p_j)}.$$

The system reliability is defined as the probability that none of real-time tasks fails, even in the presence of faults occur. Thus, the reliability of a system with N tasks can be computed by

$$R = \prod_{i=1}^{N} R_i. \tag{2}$$

3.4. Problem definition

We formulate our problem formally as follows: given a DAG $G =< V, E >$, a system consisting of M heterogeneous processors $P = \{p_1, p_2, \ldots, p_M\}$, and a deadline D, the goal is to find a fault-tolerant scheduling algorithm to improve system reliability as much as possible within the given deadline.

4. A motivational example

In this section, we give an example to illustrate the motivation of this research. The system model and task model that we consider are shown in Fig. 1 and Fig. 2(a). The communication time transferring a unit data between processors is shown in Table 1. The value beside each edge in Fig. 2(a) is the amount of communication data. There are five tasks in the DAG and the execution time of each task on different processors is given in Fig. 2(b). Fig. 3 gives four schedule results generated by HEFT, FTSA-1 and FTSA-2 algorithms, respectively. FTSA-1 and FTSA-2 means FTSA tolerates one and two permanent failures, respectively. In

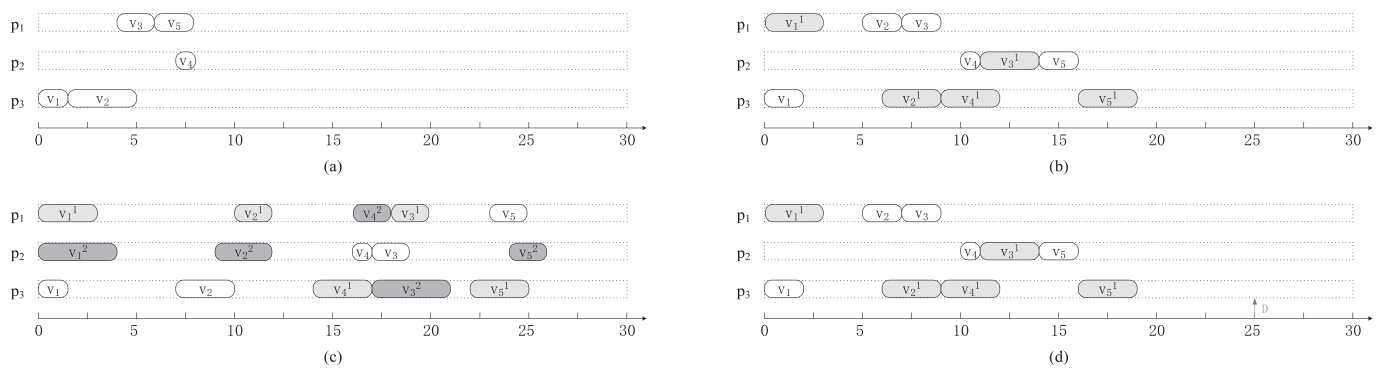


Fig. 3. A motivational example: (a) HEFT in which the makespan is 8 and the system reliability is 67.71%. (b) FTSA-1 without time constraint in which the makespan is 19 and the system reliability is 96.68%. (c) FTSA-2 without time constraint in which the makespan is 26 and the system reliability is 99.61%. (d) DBSA in which the makespan is 19 and the system reliability is 96.68%, D = 25.

Fig. 3, a rounded rectangle represents a task. The rounded rectangle with light gray color represents the first backup of a task. And the rounded rectangle with dark gray color represents the second backup of a task.

The result in Fig. 3(a) is generated by HEFT. According to the schedule, task v_3 and v_5 are assigned to processor p_1. Task v_4 is allocated to p_2. Task v_1 and v_2 are assigned to processor p_3. Then we can get the makespan is 8. The result shown in Fig. 3(b) is generated by FTSA-1. Task ${v_1}^1$, v_2 and v_3 are allocated to p_1. Task v_4, ${v_3}^1$ and v_5 are allocated to p_2. Task v_1, ${v_2}^1$, ${v_4}^1$ and ${v_5}^1$ are mapped to p_3. ${v_1}^1$ is denoted as the first backup of task v_1, and ${v_2}^1$, ${v_3}^1$, ${v_4}^1$ and ${v_5}^1$ have similar meaning. Then we get the makespan in FTSA-1 is 19. The result shown in Fig. 3(c) is generated by FTSA-2. Task ${v_1}^1$, ${v_2}^1$, ${v_4}^2$, ${v_3}^1$ and v_5 are allocated to p_1. Task ${v_1}^2$, ${v_2}^2$, v_4, v_3 and ${v_5}^2$ are allocated to p_2. Task v_1, v_2, ${v_4}^1$, ${v_3}^2$ and ${v_5}^1$ are mapped to p_3. ${v_1}^2$, ${v_2}^2$, ${v_3}^2$, ${v_4}^2$ and ${v_5}^2$ indicate the second backup of task v_1, v_2, v_3, v_4 and v_5 respectively. The makespan in FTSA-2 is 26. The results generated by HEFT, FTSA-1 and FTSA-2 are without time constraint. However, our proposed algorithm DBSA considers the time constraint. Fig. 3(d) shows the result generated by DBSA when $D = 25$. DBSA calculates that the system can only handle at most one permanent failure. So it backs up once for each task. The result is the same as the result of FTSA-1. The makespan is 19. It makes full use of time redundancy and guarantees that the makespan does not exceed deadline.

At the same time, DBSA also greatly improves the reliability of the system. Assumed $\Lambda = \{0.05, 0.04, 0.03\}$, the reliability of the system achieved by HEFT is $(0.9418 \times 0.9048 \times 0.9139 \times 0.9048 \times 0.9608) \times 100\% = 67.71\%$. FTSA-1 and FTSA-2 can respectively achieve 96.68% and 99.61% of the system reliability, and the system reliability in DBSA is 96.68%. With $D = 25$, HEFT wastes a lot of time redundancy. FTSA blindly backs up for tasks, which may lead to miss deadline, like FTSA-2. Nevertheless, DBSA can automatically adjust the number of permanent failures to meet deadline and improve the system reliability.

The notations used in this paper are summarized in Table 2.

5. The proposed algorithm

This section proposes an algorithm named Deadline-Based Scheduling Algorithm (in short, DBSA) to address the permanent fault-tolerant scheduling problem in a heterogeneous system. For a better understanding of the proposed algorithm, we first introduce some basic notations. Then we give detailed description of the algorithm.

5.1. Basic notations

Let *est*(v_i, p_k) and *eft*(v_i, p_k) denote the earliest execution start time and the earliest finish time of task v_i on processor p_k, respectively. *est*(v_i, p_k) can be computed by

$$est(v_i, p_k) = \begin{cases} 0 & \text{if } v_i = v_{entry} \\ \max\{drt(v_i, p_k), avail(p_k)\} & \text{if } v_i \neq v_{entry}, \end{cases} \tag{3}$$

where *avail*(p_k) represents the instant that processor p_k becomes idle. *drt*(v_i, p_k) represents the time when v_i has received all data from its predecessors when it is allocated to p_k. It is computed by

$$drt(v_i, p_k) = \max_{v_x \in pre(v_i)} \left\{ \max_{0 \le k \le numF} \{eft(v_x{}^k, pro(v_x{}^k)) + c(v_x, v_i)\} \right\}, \tag{4}$$

where *pro*(v_k) denotes the processor that executes task v_k. *numF* represents the number of fault-tolerant tasks, and *numF* < *M*.

eft(v_i, p_k) can be calculated by

$$eft(v_i, p_k) = est(v_i, p_k) + w(v_i, p_k). \tag{5}$$

$$rank_t(v_i) = \begin{cases} 0 \quad \text{if } v_i = v_{entry} \\ \max\limits_{v_j \in pre(v_i)} \{eft(v_j, pro(v_j) + data(v_i, v_j) \\ \quad \times \max\limits_{1 \le j \le M} \{d(pro(v_j), pro(v_i))\}\} \quad \text{if } v_i \neq v_{entry}, \end{cases} \tag{6}$$

$$rank_b(v_i) = \begin{cases} \overline{w(v_i)} & \text{if } v_i = v_{exit} \\ \overline{w(v_i)} + \max\limits_{v_j \in succ(v_i)} \{(\overline{c(v_i, v_j)} + rank_b(v_j))\} & \text{if } v_i \neq v_{exit}, \end{cases} \tag{7}$$

The executing order of tasks is determined by their priorities that are based on the *dynamic top level* and the *static bottom level*. *rank*$_t$(v_i), the *dynamic top level* of v_i is recursively defined by (6). Similarly, *rank*$_b$(v_i), the *static bottom level* of v_i is recursively computed by (7), where

$$\overline{w(v_i)} = \frac{\sum_{j=1}^{M} w(v_i, p_j)}{M}, \tag{8}$$

$$\overline{c(v_i, v_j)} = data(v_i, v_j) \times \overline{d}, \tag{9}$$

$$\overline{d} = \frac{\sum_{m=1}^{M} \sum_{k=1}^{M} d(p_m, p_k)}{M^2 - M}. \tag{10}$$

$\overline{w(v_i)}$ is the average execution time of task v_i in all processors. $\overline{c(v_i, v_j)}$ is the average communication time from task v_i to v_j. $\overline{d}$ is the average time to transfer a unit data between two processors in a system.

The task with higher $(rank_t(v_i) + rank_b(v_i))$ value has higher scheduling priority. For instance, $(rank_t(v_1) + rank_b(v_1)) > (rank_t(v_2) + rank_b(v_2))$ indicates that the priority of v_1 is higher than that of v_2.

S is the set of scheduled tasks.

U is the set of unscheduled tasks.

We define a task is free if it is unscheduled and all of its predecessors are scheduled. U_f is denoted the set of free tasks, and $U_f \subseteq U$.

Table 2
Definitions of the notations.

Notation	Definition
P	the set of processors
M	the number of processors
V	the set of nodes
E	the set of edges
N	the number of nodes in the set of V
$d(p_i, p_j)$	the communication time transferring a unit data from processor p_i to processor p_j
(v_i, v_j)	the edge from node v_i to node v_j
$data(v_i, v_j)$	the volume of data to be transferred from node v_i to node v_j
$c(v_i, v_j)$	the communication time from node v_i to node v_j
$w(v_i, p_j)$	the taken time for processor p_j executing node v_i
v_i^k	the kth copies of node v_i, and v_i^0 is the primary
$pre(v_i)$	all the direct predecessor nodes of node v_i
$succ(v_i)$	all the direct successors of node v_i
$pro(v_i)$	the executed processor of task v_i
$ft(v_i)$	the finish time of node v_i
$makespan$	the completed time of a given DAG
λ_j	the fault rate of processor p_j
$f(k, \lambda_j)$	the fault distribution in time period t when k faults occur with the fault rate λ_j of processor p_j
R_i	the possibility that node v_i is completed successfully
R	the reliability of a system
$est(v_i, p_k)$	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
$avail(p_k)$	the instant that processor p_k becomes idle
$drt(v_i, p_k)$	the time when v_i has received all data from its predecessors on processor p_k
$rank_t(v_i)$	the dynamic top level of task v_i which is used to compute the priority of v_i
$rank_b(v_i)$	the static bottom level of task v_i which is used to compute the priority of v_i
$\overline{w_(v_i)}$	the average execution time of task v_i in all processors
$\overline{c_{(v_i,v_j)}}$	the average communication time from task v_i to v_j
$\overline{d}$	the average communication time transferring a unit data between any two processors
D	the given deadline
S	the set of scheduled tasks
U	the set of unscheduled tasks
U_f	the set of free tasks

5.2. Deadline-based scheduling algorithm

In this section, we introduce the algorithm DBSA. Its basic idea is as follows. Firstly, DBSA adds all free tasks to U_f and calculates priorities of these tasks. Next, it continuously picks out the task with the highest priorities from U_f, and schedules the task to the processor which makes the task have minimal earliest finish time. When U_f is empty, a makespan is attained. Then, it compares the makespan with the given deadline to find out the maximum number of permanent failures that a system can achieve. Finally, it reschedules all the tasks including primary tasks and backup tasks to appropriate processors. The pseudo code of DBSA is shown in the Algorithm 1.

Algorithm 1 DBSA.

```
Input: A DAG $G = \{V, E\}$, a common deadline $D$, $P = \{p_1, p_2, \dots, p_M\}$;
Output: A schedule of DAG G that achieves fault-tolerance;
1: $numF \leftarrow 0$;
2: while $numF < M \&\& getMakespan(numF) \le D$ do
3:     $numF++$;
4: end while
5: $numF--$;
6: call $getMakespan(numF)$;
```

In Algorithm 1, line 1 initializes *numF* as zero. The **while** loop in lines 2–4 tries to get maximum *numF* by constantly updating the value of *numF* until *numF* is larger or equal to *M* or until the new makespan obtained by calling the function *getMakespan*(*numF*) shown in Algorithm 2 exceeds the given deadline. Line 5 obtains the value of *numF*. Line 6 reschedules all tasks to tolerate *numF* permanent failures.

Algorithm 2 computes the makespan when *numF* permanent failures occur. First, it initializes $makespan = 0, S = \emptyset, U = V, V_f = \{v_{entry}\}$. Second, it computes the $rank_b$ value of all tasks by (7) (line 2). Next, it finds out the most appropriate processors for scheduling task *t* and the backup tasks of *t* by a **while** loop(lines 3–19). The **while** loop firstly computes the $rank_u$ of all tasks in U_f and finds out the highest priority tasks *t* (lines 4–5). Then it calculates the earliest execution finish time of task *t* in all available processors (lines 6–8) and stores the finish time results in list *EFT*(*t*) with a non-decreasing order. We use the notation t^k

Algorithm 2 *getMakespan*(*numF*).

```
Input: A DAG $G = \{V, E\}$, a common deadline $D$, $P = \{p_1, p_2, \dots, p_M\}$;
Output: A new makespan and a new schedule scheme
1: $makespan \leftarrow 0, S \leftarrow \emptyset, U \leftarrow V, U_f \leftarrow \{v_{entry}\}$;
2: compute $rank_b(v_i), \forall v_i \in V$;
3: while $U \neq \emptyset$ do
4:     compute $rank_t(v_j), \forall v_j \in U_f$;
5:     $t \leftarrow$ the task with the largest $(rank_b + rank_t)$ in $U_f$
6:     for $j = 1$ to $M$ do
7:         calculate $eft(t, p_j)$ and store the results in list $EFT(t)$ with a
           non-decreasing order
8:     end for
9:     for $k = 0$ to $numF$ do
10:        $pro(t^k) \leftarrow$ the processor with kth minimum value in $EFT(t)$;
11:        scheduling $t^k$ to processor $pro(t^k)$;
12:        if $makespan < eft(t^k, pro(t^k))$ then
13:            $makespan \leftarrow eft(t^k, pro(t^k))$;
14:        end if
15:    end for
16:    $S \leftarrow S \cup \{t\}$;
17:    $U_f \leftarrow U_f \cup succ(t)$, where $\forall v \in succ(t)$ is scheduled;
18:    $U \leftarrow U \backslash \{t\}$;
19: end while
20: return $makespan$.
```

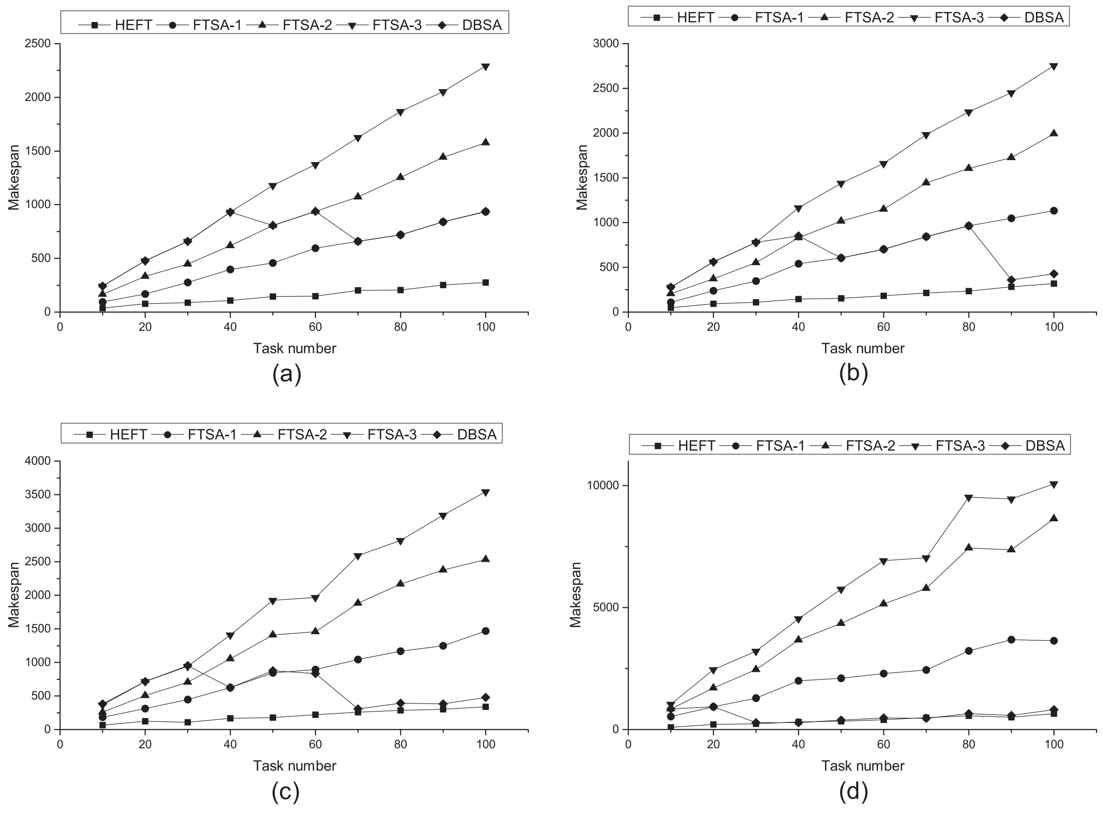


Fig. 4. Makespan of HEFT, FTSA-1, FTSA-2, FTSA-3 and DBSA for $CCR = \{0.1, 0.5, 1, 5\}, \overline{w} = 15, D = 1000$. (a) $CCR = 0.1$; (b) $CCR = 0.5$; (c) $CCR = 1$; (d) $CCR = 5$.

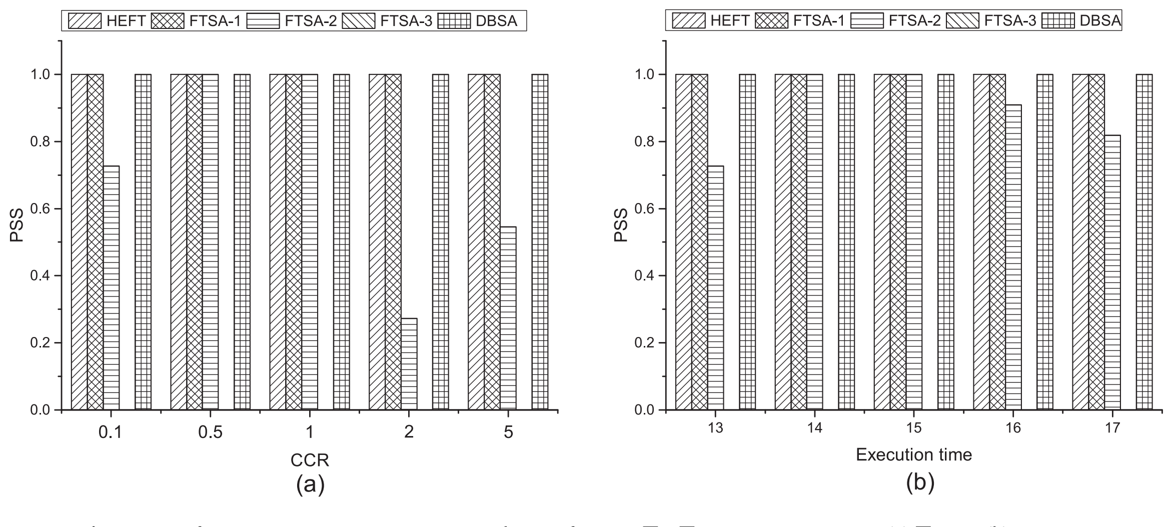


Fig. 5. PSS of HEFT, FTSA-1, FTSA-2, FTSA-3 and DBSA for $D = (\overline{w} + \overline{w} \times CCR) \times N$, $N = 40$. (a) $\overline{w} = 15$; (b) $CCR = 1$.

to denote the kth copy of task t, where $0 \leq k \leq numF$. We use the notation $pro(t^k)$ to denote the processor that t^k is assigned to. Then, we allocate t^k to the processor with kth minimum value in $EFT(t)$ (lines 9–15). After scheduling task t, it adds t into S and updates the priorities of successors of t, puts free successors of t in U_f and deletes t from S (lines 16–18). Finally, the makespan is obtained.

6. Experimental results and discussion

In this section, we evaluate the effectiveness of our proposed algorithm DBSA by comparing with two excellent scheduling algorithms HEFT [19] and FTSA [15]. HEFT is a classical static list scheduling algorithm. Although it does not take processor failures into consideration, it was proven to perform well for the task scheduling problem. FTSA is an extended version of the classic HEFT. It can tolerates arbitrary e processor failures. FTSA and DBSA both can tolerate permanent failures. The main distinction is that the purpose of FTSA is to minimize the makespan and DBSA is aiming at improving system reliability under the condition of time constraint. Two sets of simulations have been conducted to test the performance of our algorithm. They are randomly generated applications with different characteristics and real-world applications. We first present experimental parameters and per-

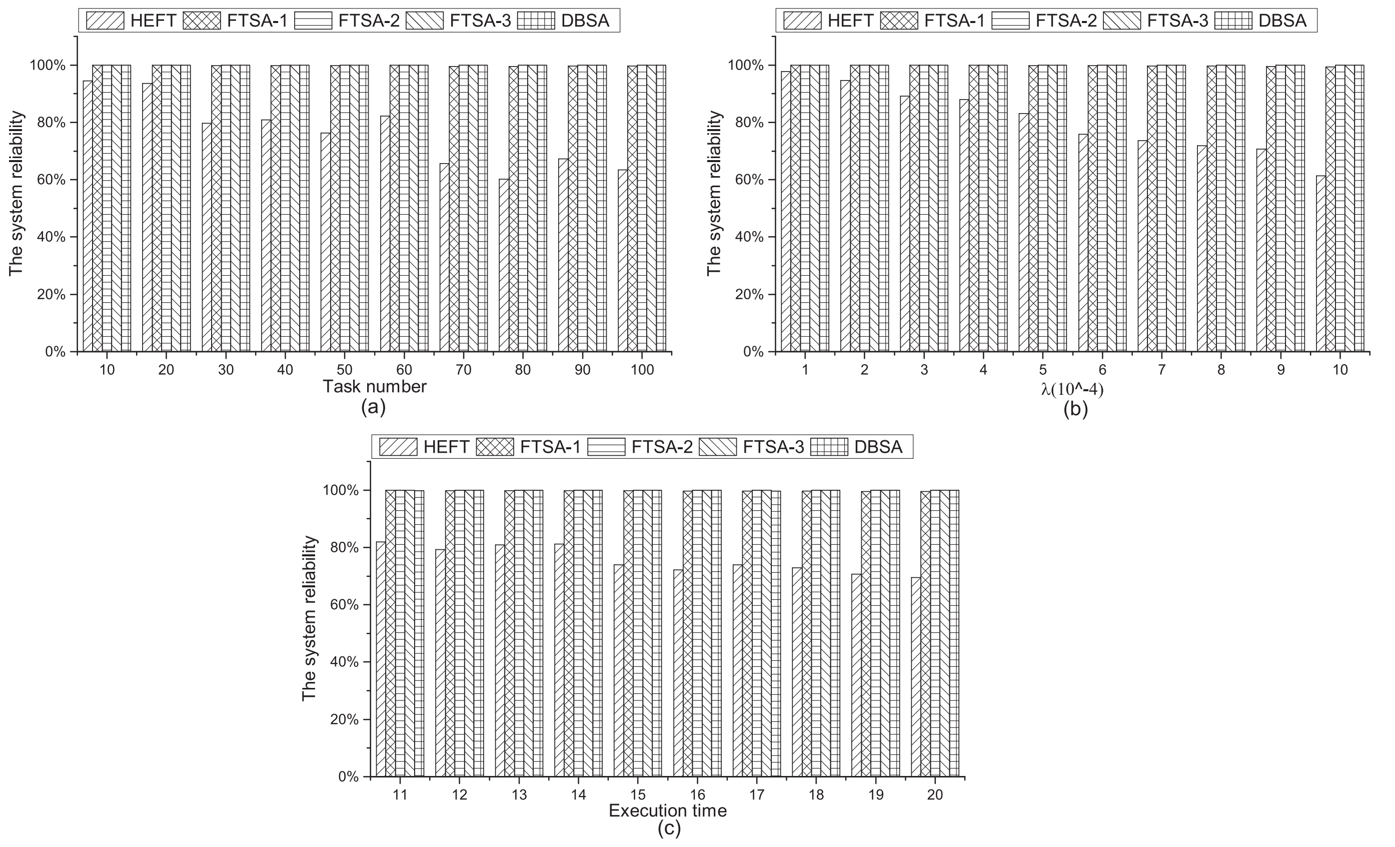


Fig. 6. The system reliability of HEFT, FTSA-1, FTSA-2, FTSA-3, DBSA algorithms. (a) $CCR = 1, \overline{w} = 15, D = (\overline{w} + \overline{w} \times CCR) \times N$; (b) $N = 40, CCR = 1, \overline{w} = 15, D = (\overline{w} + \overline{w} * CCR) * N$; (c) $CCR = 1, \overline{w} = 15, D = (\overline{w} + \overline{w} \times CCR) \times N$.

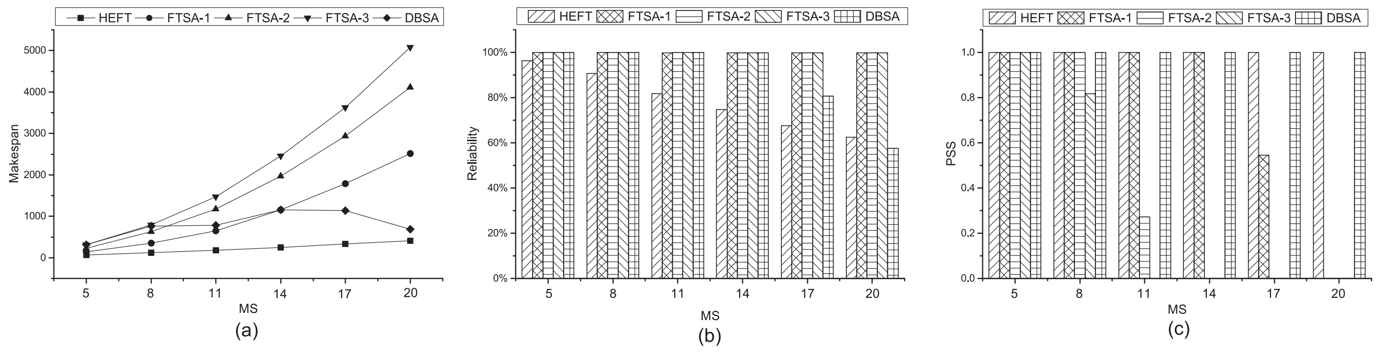


Fig. 7. Makespan, Reliability, PSS of HEFT, FTSA-1, FTSA-2, FTSA-3 and DBSA algorithms for the Gaussian elimination graph with $\overline{w} = 15, CCR = 1, D = 3.5 \times MS \times (\overline{w} + \overline{w} \times CCR)$.

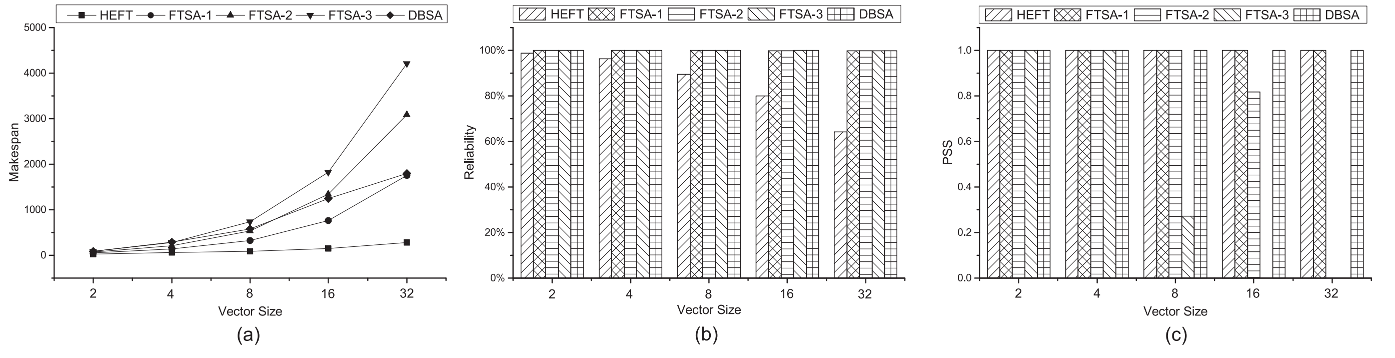


Fig. 8. Makespan, Reliability, PSS of HEFT, FTSA-1, FTSA-2, FTSA-3 and DBSA algorithms for the FFT graph with $\overline{w} = 15, CCR = 1, D = 3 \times S \times (\overline{w} + \overline{w} \times CCR)$.

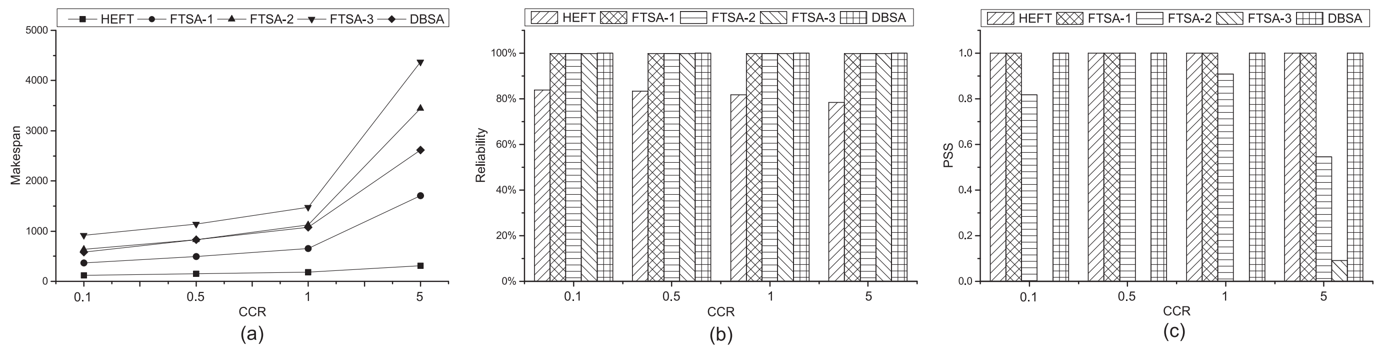


Fig. 9. Makespan, Reliability, PSS of HEFT, FTSA-1, FTSA-2, FTSA-3 and DBSA algorithms for the task graph of molecular dynamics code with $D = (\overline{w} + \overline{w} \times CCR) \times N, \overline{w} = 15, N = 40$.

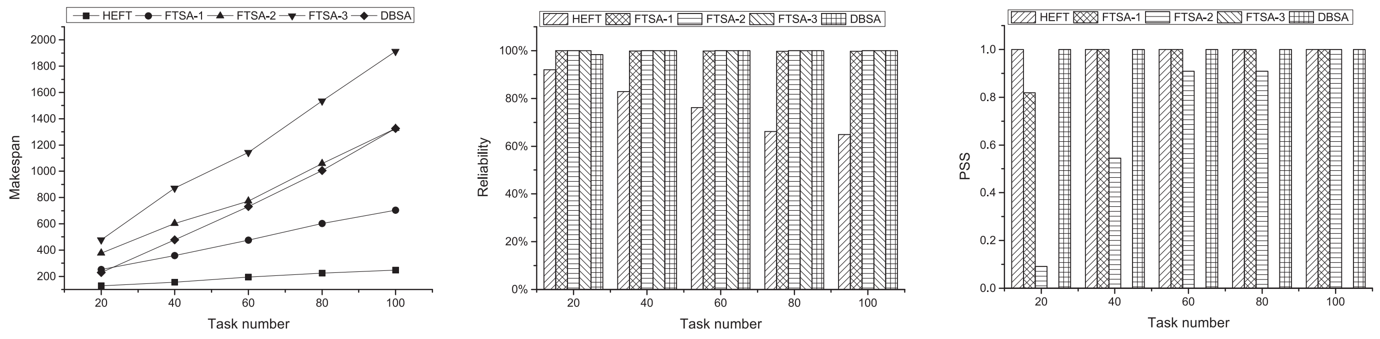


Fig. 10. Makespan, Reliability, PSS of HEFT, FTSA-1, FTSA-2, FTSA-3 and DBSA algorithms for the task graph of random with $\overline{w} = 15, CCR = 1, D = (\overline{w} + \overline{w} \times CCR) \times N$.

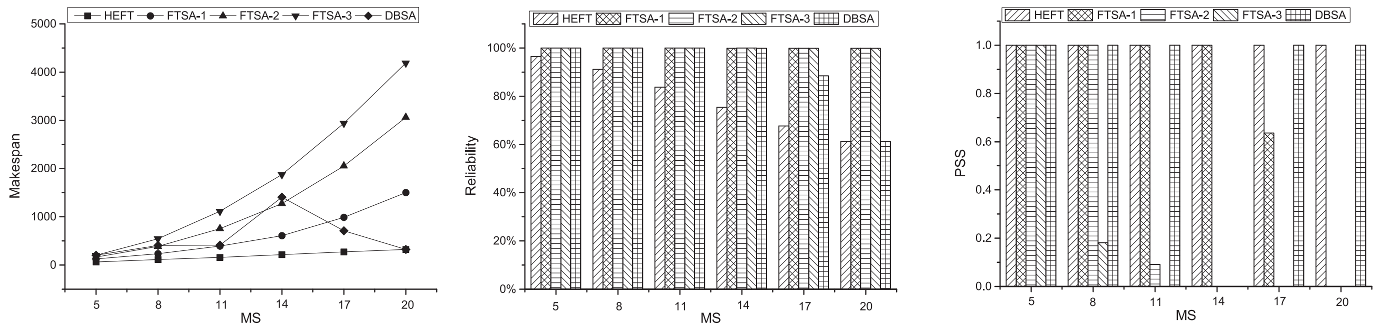


Fig. 11. Makespan, Reliability, PSS of HEFT, FTSA-1, FTSA-2, FTSA-3 and DBSA algorithms for the Gaussian elimination graph with $\overline{w} = 15, CCR = 1, D = 2 \times MS \times (\overline{w} + \overline{w} \times CCR)$.

formance metrics, and then we give detailed analysis for experimental results.

6.1. Experimental parameters

In our experiments, we evaluate our algorithms using synthetic tasks, which are generated based on a similar methodology as adopted in [15,36]. The parameters used in the task generation process are listed as following:

- *N*, the number of real-time tasks in a DAG;
- *CCR*, the communication data to computation time ratio, defined as the average communication data divided by the average computation time;
- the number of parents of a task, or the indegree of a task;
- the number of children of a task, or the outdegree of a task.

N is generated within the range of [10, 100] with the increment of 10 and *CCR* is selected from {0.1, 0.5, 1, 2, 5}. The indegree and outdegree of a task are range form [0, 4]. The computation time of each task is selected randomly from an uniform distribution with range [1, 20]. Naturally, the communication data of the task can be computed by the *CCR* value and the average computation time of the task. To account for communication heterogeneity in the system, the unit data delay of the processors is chosen uniformly from the range of [0, 2].

Besides, in order to simulate faults, we need to generate the failure rate of processors. Thus, the number of processors and the failure rate of processors are also two important parameters.

- *M*, the number of processors, set to be four and eight;
- λ, the failure rate of processors, randomly chosen from the range $[1 \times 10^{-4}, 9 \times 10^{-4}]$.

6.2. Comparison metrics

The evaluation metrics include the system reliability, makespan and the possibility of scheduling success (in short, PSS). The system reliability is an important metric to measure the performance of fault-tolerant algorithms. It has been defined in Section 3 and is computed by (2).

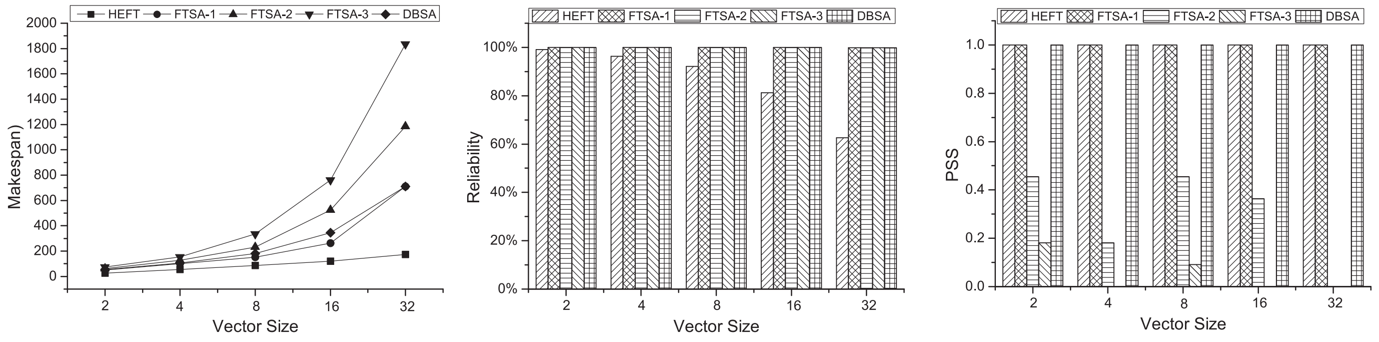


Fig. 12. Makespan, Reliability, PSS of HEFT, FTSA-1, FTSA-2, FTSA-3 and DBSA algorithms for the FFT graph with $\overline{w} = 15, CCR = 1, D = S \times (\overline{w} + \overline{w} \times CCR)$.

Makespan, which represents the instant time when the all tasks in an application are completed, is calculated by (1). PSS represents the successful possibility of applications to be completed within time constraint, and it is calculated by

$$PSS = NSC/TNS, \tag{11}$$

where NSC represents the number of scheduling successfully within deadline, and TNS represents the total number of scheduling and is set to eleven in our experiments. That the makespan exceeds the deadline will have negative effect on real-time systems. Thereby, the higher the value of PSS is, the more efficient the algorithm is.

6.3. *Performance results of randomly generated application*

In this section, we display several groups of experimental results to illustrate the performance characteristics of DBSA, FTSA and HEFT with respect to the makespan, the system reliability and PSS.

Fig. 4 shows the experimental results for the makespan obtained by HEFT, FTSA-1, FTSA-2, FTSA-3, DBSA for $CCR = \{0.1, 0.5, 1, 5\}$ and $\overline{w} = 15$ with the growing number of tasks. FTSA-1, FTSA-2, and FTSA-3 represent the FTSA algorithm tolerate one, two, and three permanent failures respectively. Fig. 5 shows the results for the PSS obtained by the above five algorithms when the *CCR* and execution time are changed. Fig. 6 shows experimental results for the systems reliability when task number, fault rate and execution time are changed.

Fig. 4 illustrates the impact of task number and *CCR* on the makespan. It is found that the makespan is higher when task number is larger or *CCR* is larger for HEFT, FTSA-1, FTSA-2, FTSA-3 algorithms. However, when *CCR* is fixed, the makespan obtained by DBSA sometimes decreases. When the makespan approaches the deadline, the DBSA algorithm dynamically calculates the number of tolerance permanent failures in order to satisfy the deadline. As the number of tolerance permanent failures decreases, the makespan decreases. From Fig. 4(a), the makespan of FTSA-3, FTSA-2, FTSA-1 turns out to be higher than the makespan of HEFT. This result can be attributed to the fact that the backup of tasks increases the total execution time. In addition, the makespan of DBSA is between the makesapn of FTSA-3 and HEFT because DBSA can adaptively adjust the number of tolerance permanent failures. Also, Fig. 4(b)–Fig. 4(d) show similar information to that Fig. 4(a) does when *CCR* increases.

Fig. 5 shows the impact of *CCR* and execution time on PSS. Fig. 5(a) demonstrates that when *CCR* varies from 0.1 to 5, HEFT, FTSA-1 and DBSA always have a higher PSS than FTSA-2 and FTSA-3. The reason for this is that the communication cost is increased with the value of *CCR* increasing, which leads to less time to tolerate failures. Thus, FTSA-2 and FTSA-3 often gets small PSS. The same trend can be seen in Fig. 5(b) as in Fig. 5(a). In short, from Fig. 5, PSS obtained by DBSA, HEFT and FTSA-1 turns out to be higher than that of FTSA-2 and FTSA-3. And PSS of DBSA always remains higher than that of FTSA-2 and FTSA-3. The reason is that, DBSA employs dynamically adjustment technology to guarantee deadline.

Fig. 6 gives the impact of task number, λ and execution time on the system reliability. Fig. 6(a) shows FTSA-2, FTSA-3 and DBSA have a higher system reliability than HEFT and FTSA-1 when the task number ranges from 10 to 100. This means that whether the task number is changed or not, FTSA-2, FTSA-3 and DBSA displays better performance. The reason is that HEFT cannot tolerate failures. Generally, with the increase of task number, the system reliability of HEFT gets decreased. On the other hand, FTSA-1 can tolerate one permanent failure. So the reliability of FTSA-1 is improved a lot. But, when more failures occur, FTSA-3 and DBSA show better performance. The Fig. 6(b) and Fig. 6(c) get the similar results due to the same reason just mentioned.

From above three figures, we observe that HEFT and DBSA outperform FTSA-1, FTSA-2 and FTSA-3 in terms of makespan. HEFT, FTSA-1 and DBSA outperform FTSA-2 and FTSA-3 in terms of PSS. FTSA-2, FTSA-3 and DBSA outperform HEFT and FTSA-1 in terms of the system reliability. In addition, we note that DBSA can dynamically adjust the number of tolerance permanent failures based on the given deadline. However, HEFT does not tolerate failures, and FTSA just blindly back up tasks to tolerate permanent failures without considering whether the makespan obtained by the algorithm has exceeded the deadline.

6.4. *Performance results of real-world application*

In this section, we consider three types of application graphs for real-world problems to test the performance of the proposed algorithm. They are Gaussian Elimination [37], Fast Fourier Transform [38] and Molecular Dynamics Code graphs [39]. Since structures of these graphs are known, only the mean execution time and CCR are needed.

Gaussian elimination: In the Gaussian elimination experiments, the matrix size of the coefficient matrix for the Gaussian elimination problem is denoted by *MS* which is used as variable to compute the number of tasks in the task graph *N*. That is, $N = \frac{MS^2+MS-2}{2}$.

Fig. 7 shows the makespan, reliability and PSS of HEFT, FTSA-1, FTSA-2, FTSA-3 and DBSA algorithms at different matrix sizes from 5 to 20 with the increment of three when $\overline{w} = 15, CCR = 1$ and $D = 3.5 \times MS \times (\overline{w} + \overline{w} \times CCR)$. From Fig. 7(a), we can see that the makespan of FTSA-1, FTSA-2, FTSA-3 and HEFT gets larger as MS gets larger. However, the makespan of DBSA presents a different trend. For example, when $MS > 17$, the makespan decreases. The reason is that if DBSA provides fault-tolerance, it will miss deadline as can be seen in Fig. 7(c). Therefore, DBSA choose not to support fault-tolerance which leads to a lower makespan. Fig. 7(b) demonstrates that FTSA-1, FTSA-2, FTSA-3 and DBSA can get high system reliability when $MS < 17$. The reason why the reliability of DBSA gets lower when $MS > 17$ has been just mentioned. The results in Fig. 7(c) also confirm this point. Fig. 7(c) shows that DBSA can always guarantee high PSS and the PSS of FTSA-1, FTSA-2 and FTSA-3 drops gradually as the *MS* increases.

Fast Fourier transform: In the FFT-related experiments, the size of the input vector which represents the coefficients of the polynomial, denoted by *S*, can be used to compute the DAG size. There are $2S - 1$

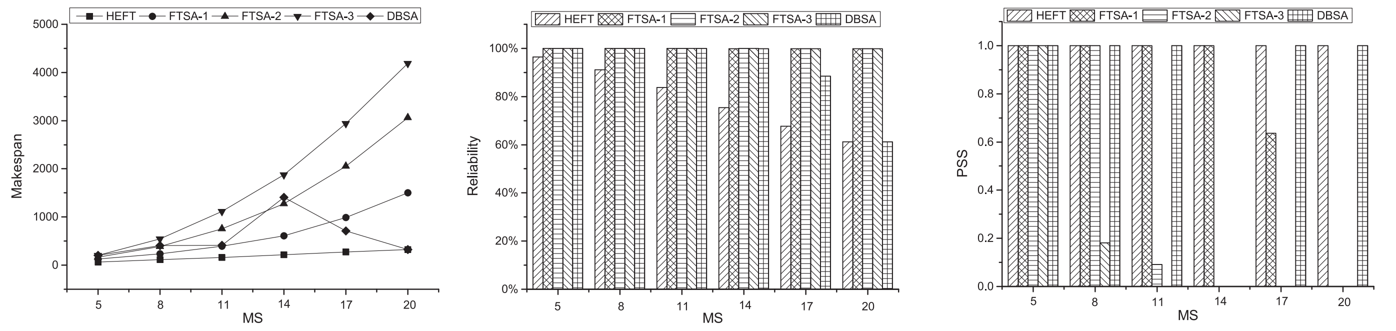


Fig. 13. Makespan, Reliability, PSS of HEFT, FTSA-1, FTSA-2, FTSA-3 and DBSA algorithms for the task graph of molecular dynamics code with $D = (\overline{w} + \overline{w} \times CCR) \times N/2, \overline{w} = 15, N = 40$.

recursive call tasks and $S \times \log_2 S$ butterfly operation tasks. S varies from 2 to 32 with a multiplier of 2.

Fig. 8 shows the makespan, reliability and PSS of HEFT, DB-FTSA and FTSA algorithms at different vector sizes when $\overline{w} = 15, CCR = 1$ and $D = 3 \times S \times (\overline{w} + \overline{w} \times CCR)$. Fig. 8(a) shows that the makespan of DBSA is shorter than that of FTSA-2 and FTSA-3 by (20, 0), (79, 0), (70, 204), (101, 579) and (1295, 2416). The parenthesized pairs are arranged to present the results achieved with S of 2, 4, 8, 16 and 32, respectively. In this paragraph, all the results are presented with respect to the size of input vector. From Fig. 8(b), DBSA gains a higher system reliability than HEFT by (1.14%, 3.62%, 10.6%, 20.5%, 35.54%). From Fig. 8(c), DBSA outperforms FTSA-2 and FTSA-3 in terms of PSS by (0, 0), (0, 0), (0, 0.73), (0.18, 1) and (1, 1).

Molecular dynamics code: This application is part of performance evaluation experiments. It has an irregular task graph. In the experiments, the number of tasks is set to be 40 and CCR is selected from {0.1, 0.5, 1, 5}.

Fig. 9 shows the makespan, reliability, PSS of HEFT, DB-FTSA and FTSA algorithms at different value of CCR. As it can be seen from Fig. 9(a), the makespan obtained by DBSA is shorter than those of FTSA-2 and FTSA-3 by (52, 331), (0, 312), (49, 404) and (824, 1751). The parenthesized pairs are arranged to present the results achieved with CCR values of 0.1, 0.5, 1.0, and 5.0, respectively. In this paragraph, all the results are presented with respect to the value of CCR. From Fig. 9(b), DBSA gains a higher system reliability than HEFT by (16.13%, 16.73%, 18.24%, 11.55%). From Fig. 8(c), DBSA outperforms FTSA-2 and FTSA-3 in terms of PSS by (0.18, 1), (0, 1), (0.09, 1) and (0.45, 0.91).

For real-world applications, DBSA outperforms FTSA-2 and FTSA-3 in terms of schedule length and PSS. In addition, DBSA outperforms HEFT in terms of the system reliability. When the deadline is relatively small, DBSA outperforms FTSA-1, FTSA-2, FTSA-3 algorithms in terms of schedule length and PSS. Although the system reliability of DBSA varies, the general trend of reliability improves. This provides clear indication that when the time constrain is relatively strict, the superiority of the DBSA algorithm can be reflected.

Fig. 10,11,12,13 show the makespan, reliability and PSS of HEFT, DB-FTSA and FTSA algorithms for randomly generated graphs, the Gaussian Elimination application, the FFT application, and the Molecular Dynamics Code graph when the given system has eight heterogeneous processors. Obviously, these four figures reflect similar information as that the preceding experiments do.

7. Conclusion

We present a fault-tolerance scheduling algorithm, called DBSA algorithm, for solving the problem of scheduling in heterogeneous systems. The DBSA algorithm uses an active replication scheme to support permanent fault-tolerance. In the scheme, tasks can have many copies. Too many copies may exceed the given time constraint. To avoid this case, we should find out the reasonable number of copies for each task. To get the number (denoted by e), the algorithm *getMakespan*(*NumF*) is proposed. First, it initializes $NumF = 0$ and computes the makespan when *getMakespan*(*NumF*) provides *NumF* permanent failures using the pre-allocation method. Then, we compare the makespan with the given deadline. If makespan is smaller than deadline, *NumF* is increased by one and the makespan is recalculated. Repeat the above step until the obtained makespan is larger than the given deadline. The value of the last *NumF* is we need. That is, the number e is equal to $NumF - 1$. After getting e, DBSA maps all tasks as well as their backups to appropriate processors. DBSA algorithm is mainly for handling permanent failures, so different copies of a task cannot be assigned to the same processor.

We conduct extensive experiments to test the efficiency of our proposed algorithm and compare DBSA algorithm with two of existing scheduling algorithms, HEFT and FTSA. Experimental results demonstrate that on the whole, the DBSA algorithm is a practical solution for task scheduling with fault-tolerance in heterogeneous systems.

In the future, we will study the problem of scheduling algorithms that consider how to tolerate failures while ensuring the given reliability and time constrains. By doing so, we can save the system resources while meeting the requirements of users.

Acknowledgment

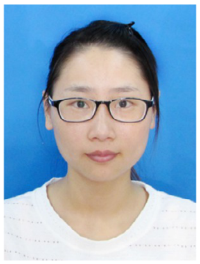
The authors would like to express their sincere gratitude to the editors and the reviewers. This work was supported by the National Natural Science Foundation of China (Grant Nos.61602350, 61602349, 61702382), the Open Foundation of Hubei Province Key Laboratory of Intelligent Information Processing and Real-time Industrial System (2016znss26C).

References

[1] R.M. Pathan, Fault-tolerant and real-time scheduling for mixed-criticality systems, Real-Time Systems 50 (4) (2014) 509–547.
[2] D. Zhu, H. Aydin, Reliability-aware energy management for periodic real-time tasks, in: Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07. IEEE, 2007, pp. 225–235.
[3] F. Liberato, S. Lauzac, R. Melhem, D. Mosse, Fault tolerant real-time global scheduling on multiprocessors, in: Real-Time Systems, 1999. Proceedings of the Euromicro Conference on, 1999, pp. 252–259.
[4] I. Koren, C.M. Krishna, Fault-Tolerant systems, Morgan Kaufmann Publishers Inc., 2007.
[5] J. Wang, W. Bao, X. Zhu, L.T. Yang, Y. Xiang, Festal: fault-tolerant elastic scheduling algorithm for real-time tasks in virtualized clouds, IEEE Trans. Comput. 64 (9) (2015) 2545–2558.
[6] M. Treaster, A survey of fault-tolerance and fault-recovery techniques in parallel systems, ACM Computing Research Repository (CoRR 501002 (2005) 1–11.
[7] C.M. Krishna, Fault-tolerant scheduling in homogeneous real-time systems, ACM Comput. Surv. 46 (4) (2014) 1–34.
[8] X. Zhu, X. Qin, M. Qiu, Qos-aware fault-tolerant scheduling for real-time tasks on heterogeneous clusters, IEEE Trans. Comput. 60 (6) (2011) 800–812.
[9] D.K. Pradhan, Fault-tolerant computer system design, Prentice-Hall, Inc., 1996.
[10] B. Zhao, H. Aydin, D. Zhu, Shared recovery for energy efficiency and reliability enhancements in real-time applications with precedence constraints, ACM Trans. Des. Autom. Electron. Syst. 18 (2) (2013) 1–21.

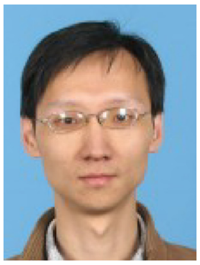
[11] Y. Guo, D. Zhu, H. Aydin, L.T. Yang, Energy-efficient scheduling of primary/backup tasks in multiprocessor real-time systems (extended version), in: Proc. of IEEE International Symposium on Parallel and Distributed Processing, 2013, pp. 896–902.
[12] Y. Guo, D. Zhu, H. Aydin, Generalized standby-sparing techniques for energy-efficient fault tolerance in multiprocessor real-time systems, in: IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2013, pp. 62–71.
[13] M.A. Haque, H. Aydin, D. Zhu, On reliability management of energy-aware real-time systems through task replication, IEEE Trans. Parallel Distrib. Syst. 28 (3) (2017) 813–825.
[14] Q. Han, L. Niu, G. Quan, S. Ren, S. Ren, Energy efficient fault-tolerant earliest deadline first scheduling for hard real-time systems, Real-Time Systems 50 (5) (2014) 592–619.
[15] A. Benoit, M. Hakem, Y. Robert, Fault tolerant scheduling of precedence task graphs on heterogeneous platforms, in: Parallel and Distributed Processing, Ipdps, IEEE International Symposium on, 2008, pp. 1–8.
[16] L. Zhao, Y. Ren, X. Yang, K. Sakurai, Fault-tolerant scheduling with dynamic number of replicas in heterogeneous systems, in: IEEE International Conference on High PERFORMANCE Computing and Communications, 2010, pp. 434–441.
[17] J. Broberg, P. Ståhl, Dynamic fault tolerance and task scheduling in distributed systems (2016).
[18] M.R. Garey, D.S. Johnson, Computers and intractability: A Guide to the theory of NP-Completeness, W. H. Freeman, 1979.
[19] H. Topcuoglu, S. Hariri, M.-Y. Wu, Performance-effective and low-complexity task scheduling for heterogeneous computing, IEEE Trans. Parallel Distrib. Syst. 13 (3) (2002) 260–274.
[20] C.L. Liu, J.W. Layland, Scheduling algorithms for multiprogramming in a hard-real-time environment, Readings in Hardware/Software Co-Design 20 (1) (2002) 179–194.
[21] B. Andersson, G. Raravi, Real-time scheduling with resource sharing on heterogeneous multiprocessors, Real-Time Systems 50 (2) (2014) 270–314.
[22] V. Legout, M. Jan, L. Pautet, Scheduling algorithms to reduce the static energy consumption of real-time systems, Real-Time Systems 51 (2) (2015) 153–191.
[23] M. Qamhieh, Scheduling of parallel real-time dag tasks on multiprocessor systems (2015).
[24] M.A. Bamakhrama, T.P. Stefanov, On the hard-real-time scheduling of embedded streaming applications, Design Automation for Embedded Systems 17 (2) (2013) 221–249.
[25] G. Xie, G. Zeng, L. Liu, R. Li, K. Li, High performance real-time scheduling of multiple mixed-criticality functions in heterogeneous distributed embedded systems, J. Syst. Archit. 70 (2016) 3–14.
[26] J. Liu, K. Li, D. Zhu, J. Han, K. Li, Minimizing cost of scheduling tasks on heterogeneous multicore embedded systems 16 (2) (2016) 1–25.
[27] P. Keerthika, N. Kasthuri, An efficient grid scheduling algorithm with fault tolerance and user satisfaction, Mathematical Problems in Engineering 2013 (4) (2013) 321–341.
[28] S. Gui, L. Luo, Reliability analysis of real-time fault-tolerant task models, Design Automation for Embedded Systems 17 (1) (2013) 87–107.
[29] A. Kumar, B. Alam, Real-time fault tolerance task scheduling algorithm with minimum energy consumption, Springer India, 2016.
[30] Z. Albayati, B.H. Meyer, H. Zeng, Fault-tolerant scheduling of multicore mixed-criticality systems under permanent failures, in: IEEE International Symposium on Defect and Fault Tolerance in Vlsi and Nanotechnology Systems, 2016, pp. 57–62.
[31] G.Q. Xie, R.F. Li, L. Liu, F. Yang, Dag reliability model and fault-tolerant algorithm for heterogeneous distributed systems, Chin. J. Comp. 36 (10) (2013) 2019–2032.
[32] K. Uchiyama, F. Arakawa, H. Kasahara, T. Nojiri, H. Noda, Y. Tawara, A. Idehara, K. Iwata, H. Shikano, Heterogeneous multicore architecture, Springer New York, 2012.
[33] X. Qin, H. Jiang, A novel fault-tolerant scheduling algorithm for precedence constrained tasks in real-time heterogeneous systems, Elsevier Science Publishers B. V., 2006.
[34] H. Jin, X.H. Sun, Z. Zheng, Z. Lan, B. Xie, Performance under failures of dag-based parallel computing, in: Ieee/acm International Symposium on CLUSTER Computing and the Grid, 2009, pp. 236–243.
[35] J.W. Young, A first order approximation to the optimum checkpoint interval, Commun. ACM 17 (9) (1974) 530–531.
[36] C.Y. Chen, Task scheduling for maximizing performance and reliability considering fault recovery in heterogeneous distributed systems, IEEE Press, 2016.
[37] M.Y. Wu, D.D. Gajski, Hypertool: a programming aid for message-passing systems. ieee trans parallel distrib syst, Parallel and Distributed Systems IEEE Transactions on 1 (3) (1990) 330–343.
[38] Y.C. Chung, S. Ranka, Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors, IEEE, 1995.
[39] M.I. Daoud, N. Kharma, A high performance algorithm for static task scheduling in heterogeneous distributed computing systems, J. Parallel Distrib. Comput. 68 (4) (2008) 399–409.

Jing Liu received her BS degree from the college of mathematics and econometrics and PhD degree from the college of information science and engineering, Hunan University, Changsha, China, in 2009 and 2015, respectively. She is a lecturer in the School of Computer Science and Technology, Wuhan University of Science and Technology, Wuhan, China. Her current research interests include scheduling, fault tolerance, embedded systems, real-time systems.



Mengxue Wei is a postgraduate student in the school of computer science and technology, Wuhan University of Science and Technology. Her research interest include real-time systems and fault-tolerant scheduling.

Wei Hu received his Master degree and Ph.D degree in Computer Science from Zhejiang University in 2005 and 2008 respectively. He was an Assistant Professor in Zhejiang University in 2008–2010. He is currently Associate Professor in Wuhan University of Science and Technology. His current research interests include computer architecture and embedded systems.



Xin Xu received the B.Sc. and Ph.D. degree in computer science and engineering from Shanghai Jiao Tong University, China, in 2004 and 2012 respectively.He is an associate professor in the School of Computer Science and Technology, Wuhan University of Science and Technology, Wuhan, China. His current research interests include computer vision, pattern recognition, and visual surveillance.

Aijia Ouyang received his Ph.D. degree from the college of information science and engineering, Hunan University, Changsha, China, 2015. His research interests include intelligence computing, parallel computing, cloud computing, and machine learning. He has published more than 50 research papers in international conferences and journals on intelligence optimization algorithm, data mining and parallel algorithm.