Dynamic Scheduling Real-Time Task Using Primary-Backup Overloading Strategy for Multiprocessor Systems*

Wei SUN†, Chen YU‡, Nonmembers, Xavier DÉFAGO†, and Yasushi INOGUCHI†, Members

SUMMARY The scheduling of real-time tasks with fault-tolerant requirements has been an important problem in multiprocessor systems. The primary-backup (PB) approach is often used as a fault-tolerant technique to guarantee the deadlines of tasks despite the presence of faults. In this paper we propose a dynamic PB-based task scheduling approach, wherein an allocation parameter is used to search the available time slots for a newly arriving task, and the previously scheduled tasks can be re-scheduled when there is no available time slot for the newly arriving task. In order to improve the schedulability we also propose an overloading strategy for PB-overloading and Backup-backup (BB) overloading. Our proposed task scheduling algorithm is compared with some existing scheduling algorithms in the literature through simulation studies. The results have shown that the task rejection ratio of our real-time task scheduling algorithm is almost 50% lower than the compared algorithms.

key words: real-time, multiprocessor, fault tolerance, dynamic task scheduling

1. Introduction

Multiprocessor and multicomputer systems have been a powerful means for real-time applications due to their high performance. For most real-time applications, the correctness of tasks depends not only on the logical correctness but also on the finish time of tasks [1]. Thus, it is essential that tasks complete before their deadlines even in the presence of processor failures. This makes fault-tolerance an inherent requirement of real-time systems.

In a multiprocessor system, fault-tolerance can be provided by scheduling multiple copies of tasks on different processors [1]–[8]. Primary-backup approach is one of fault tolerant scheduling techniques. In the PB-based task scheduling, two versions of a task (primary version and backup version) are scheduled on two different processors and the acceptance test is used to check the correctness of the execution result [4]–[8].

In order to improve the schedulability, overloading techniques are often used. PB-overloading schedules the primary of a task onto the same or overlapping time slot with the backup of another task on a processor [8]. BB-overloading schedules the backups of multiple tasks onto the same or overlapping time slot on a processor [4], [7], [8]. In [8], R. Al-Omari et al. drew a conclusion that the PB-overloading is able to achieve better performance than BB-overloading, and BB-overloading algorithm is better than non-overloading algorithm. They also left the unsolved problem of whether both PB-overloading and BB-overloading co-exist in a single scheduling algorithm.

In most scheduling algorithms, the newly arriving tasks do not affect the previously scheduled tasks [2], [3], [6]–[8]. Usually a new task will be rejected if the scheduler cannot find a feasible time slot for it. In [4], the previously scheduled primaries of tasks can be moved if scheduler cannot find an empty time slot for a new task, which is known as re-scheduling.

A k-timely-fault-tolerant (k-TFT) schedule is defined as the schedule in which no task deadlines are missed, despite k arbitrary processor failures [10]. k-TFT can be achieved by grouping techniques [8], [9], which divide processors into groups and allow overloading to take place only within a group.

In this paper, we address the dynamic PB-based scheduling of non-preemptive aperiodic real-time tasks with fault-tolerant requirements. In this PB-based scheduling, both PB-overloading and BB-overloading exist, and an overloading strategy is used to make the overloading more flexible and efficient. Our scheduling algorithm can re-schedule the previously scheduled tasks on one processor. We can achieve k-TFT by grouping techniques, but this will greatly increase the complexity of algorithm. Therefore, we only consider 1-TFT in this paper. The objective of the paper is to decrease task rejection ratio. A short version of this paper has appeared in [13].

The rest of the paper is organized as follows. In Sect. 2, related work and motivation for our work are presented. Section 3 discusses the task, scheduler, and fault models. The proposed scheduling algorithm and overloading strategy are introduced in Sect. 4. We analyze the time complexity of the proposed scheduling algorithm in Sect. 5. In Sect. 6, the performance of our scheduler is illustrated through simulation studies. Finally, we conclude our work in Sect. 7.
2. Related Work and Motivation

In this section, we first introduce the previous work related to this paper and then show the existing problems which lead to our motivation for our work.

2.1 Related Work

In PB-based task scheduling a backup is deallocated when its primary is finished successfully [4], [6], [8]. Resource reclaiming, which refers to the problem of utilizing resources left unused by a task version [11], is used to improve the processor utilization [7]. The resource reclaiming includes not only the deallocation of task but also the left time slot when the actual execution time is less than the worst case execution time. Thus there might be some empty time slots in a previous schedule due to the resource reclaiming. The empty time slots will be reused for new tasks.

Backups are scheduled as late as possible or overloaded on other backups as much as possible, and a function is used to control the length of the overlapping parts of the overloaded backups [4]. In order to tolerate more faults, backup overloading can take place only among the processor in a group [7], known as static grouping. In [8], the author extends static grouping to dynamic grouping and presents a PB-overloading technique in order to improve schedulability.

Dertouzos et al. showed that an optimal algorithm does not exist for dynamically scheduling tasks on a multiprocessor system [12]. Aperiodic tasks, whose arrival times and deadlines are not known in advance, require dynamic scheduling algorithm. In [4], when scheduler cannot find a proper time slot for a new task, a primary will be rescheduled by moving it forward while any backup cannot be re-scheduled. In [6], [8] the scheduling algorithms are based on the Spring scheduling approach [9], which is a heuristic algorithm and dynamically schedules tasks with resource requirements. The algorithms in [2], [3], [6]-[8] cannot re-schedule tasks.

2.2 Motivation for Our Work

For those algorithms without re-scheduling, the empty time slots could not be re-used well. For example, in Fig. 1, because of resource reclaiming, there are some empty time slots in the current schedule. The new task cannot be allocated properly between its ready time and deadline (this will be explained later). If we move pr3 and pr5, it is possible to allocate the primary of the new task. Therefore, to re-schedule the previously scheduled tasks is useful for real-time PB-based task scheduling.

The rescheduling in [4] can move primaries forward when a new task requires the occupied time slots while the backups cannot be moved. However, sometimes it is necessary to move tasks backward. For example, in Fig. 1, pr3 can be moved backward within the reasonable scope. The backups can also be moved even if the backups are overloaded with each other.

In [8], PB-overloading chain will not contain more than two tasks at the same time, for example the chain A in Fig. 2. But, in theory, as long as the time between the first task and the last task in the PB-overloading chain is less than the double of the minimum time interval of faults, a PB overloading chain can contain more than two tasks, for example the chain B in Fig. 2, for a PB-overloading chain can only tolerate one failure [8]. Moreover, PB-overloading chain should be opened but looped, and the looped PB-chain will fail eventually, for example the looped chain C, in Fig. 2, will fail, if Processor 4 fails. The chain A and B are the opened chains.

PB-overloading and BB-overloading are used to improve schedulability, but the overloading will increase the coupling of processors. So in [7], [8] grouping techniques are used as a kind of overloading strategy to limit overloading to take place only on a subset of processors.

Considering these existing problems, our task scheduling algorithm can re-schedule tasks by moving them forward or backward within the reasonable scope. The backups can also be moved even if the backups are overloaded with each other.

In [8], PB-overloading chain will not contain more than two tasks at the same time, for example the chain A in Fig. 2. But, in theory, as long as the time between the first task and the last task in the PB-overloading chain is less than the double of the minimum time interval of faults, a PB overloading chain can contain more than two tasks, for example the chain B in Fig. 2, for a PB-overloading chain can only tolerate one failure [8]. Moreover, PB-overloading chain should be opened but looped, and the looped PB-chain will fail eventually, for example the looped chain C, in Fig. 2, will fail, if Processor 4 fails. The chain A and B are the opened chains.

PB-overloading and BB-overloading are used to improve schedulability, but the overloading will increase the coupling of processors. So in [7], [8] grouping techniques are used as a kind of overloading strategy to limit overloading to take place only on a subset of processors.

Considering these existing problems, our task scheduling algorithm can re-schedule tasks by moving them forward or backward within the reasonable scope. The backups can also be moved. Because of the large time cost to re-schedule tasks on all processors, the re-scheduling only takes place on one processor and the relationship of overloaded tasks cannot be changed. The PB- and BB-overloading co-exist in our algorithm and are managed by an overloading strategy. In fact most PB-based fault tolerant task scheduling algorithms mainly deal with the tradeoff between schedulability and reliability. Our motivation is to
reduce the task rejection ratio as much as possible and guarantee reliability.

3. Models

3.1 Scheduler Model

The scheduler model used in this paper is similar with those in [2], [6]–[8]. All tasks arrive at the scheduler, a central processor, and then are dispatched to other processors in terms of the scheduling result. All processors have identical computing capability and are connected through a shared medium. The scheduler is running in parallel with the processors. Each processor has its own task queue. A tuner is in front of local processor task queue and in charge of inserting a new task into this task queue or changing the previous schedules. The structure of scheduler is shown in Fig. 3. It is assumed that the scheduler has been made fault tolerant by other fault tolerant technique, for example, modular redundancy technique [8].

3.2 Task Model

Tasks have the following attributes:

1. Tasks are aperiodic, i.e., task arrivals are not known in advance. Each task $T_i$ has the numeric characteristics, where $i$ is the sequence number of task: arrival time ($a_i$), ready time ($r_i$), worst case computation time ($c_i$), actual computation time ($a_{c_i}$) and deadline ($d_i$). The actual computation time is the true time that a processor takes to finish a task. The worst case computation time is assumed always larger than the actual computation time.

2. Each task has two identical versions. The version to be scheduled earlier in a schedule is marked as primary ($p_{ri}$) and the other one is marked as backup ($b_{ki}$). When a primary is finished successfully, its backup will be deallocated at once.

3. Tasks are not parallelizable. A task can be executed only on one processor.

4. Tasks are independent. For tasks with precedence constraints, ready times and deadlines of tasks can be modified to make tasks comply with their precedence constraints [7].

5. Tasks are non-preemptable.

3.3 Fault Model

Each processor, except the scheduler, may fail due to hardware or software faults which result in task failures. The faults can be transient or permanent. Each fault is independent to the others and exists in one processor.

$MTBF$ is defined to be the expected time between two failures of a system. $TTSF$ is defined to be the time to the second failure. The maximum number of processors that are expected to fail at any point of time is assumed to be one, because only $1-TFT$ is considered in this paper. We also assume that $\forall T_i(d_i - r_i)$ is much less than $MTBF$. If any overloading does not happen, in the worst case, $TTSF$ will be equal to $\max(d_i - r_i)$. Because of the overloading, $TTSF$ will be discussed again in Sect. 3.4.

A fault-detection is assumed to announce failures on time. The scheduler will not schedule tasks to a known failed processor.

3.4 Definitions

1. $st(\cdot)$ is the start time of $p_{ri}$ or $b_{ki}$. $ft(\cdot)$ is the finish time of $p_{ri}$ or $b_{ki}$.

Constraint 1: $r_i \leq st(p_{ri}) < ft(p_{ri}) < st(b_{ki}) < ft(b_{ki}) \leq d_i$.

2. proc(\cdot) is the processor on which the primary or backup is scheduled.

Constraint 2: proc($p_{ri}$) $\neq$ proc($b_{ki}$).

3. $ti(\cdot)$ is the time interval from $st(\cdot)$ to $ft(\cdot)$ on which the primary or backup is scheduled.

Constraint 3: $ti(p_{ri}) + ti(b_{ki}) = 0$.

4. $n_{\text{cascade}}$ is the cascade number of overloaded tasks within a time slot. $m$ is the number of processors. when $n_{\text{cascade}} = 1$, it means the task is scheduled without overloading; when $n_{\text{cascade}} = m$, it means no task can be overloaded on this time slot again.

Constraint 4: $1 \leq n_{\text{cascade}} \leq m$.

5. $t_{\text{overload}}$ is the part time of a task overloaded on other tasks.

Constraint 5: $0 \leq t_{\text{overload}} \leq c_i$.

6. A set of tasks which are overloaded with each other within a time slot is named an overloading task set. This task set is denoted as $\tau$. $st(\tau)$ is the start time of the first task to be executed, and $ft(\tau)$ is the finish time of the last task to be finished in $\tau$. A single task is also a task set with only one task.

7. The shift window $Win_s(\cdot) < back, for >$ is the time interval on which a previously scheduled task can move. $for$ is the time of a task being moved forward. $back$ is the time of a task being moved backward. All tasks in an overloading task set ($\tau$) have the same $Win_s$, which is the $Win_s(\tau)$.

8. A single PB-chain is defined to be that any primary in this chain exists in only one overloading task set. If a primary in a PB chain and a primary in another PB chain are in the same overloading task set, the two chains are coupled.

9. The maximum space length $L_s$ of a single PB-overloading chain is defined to be the maximum number of primaries.
The maximum time length $L_t$ of a single PB-overloading chain is defined to be the time interval between the earliest start time of tasks and the latest finish time of tasks in this chain.

Some definitions are shown in Fig. 4. The detailed example of the overloading task set and the shift window will be shown in Sect. 4 with the scheduling algorithm.

PB-overloading, compared with BB-overloading, is quite complex. An example, in Fig. 5, illustrates a single PB-overloading chain and two coupled PB-overloading chains. Chain A is a single PB chain. Chain B and Chain C are coupled on Processor 2. $L_t$ of each chain is shown in the figure. B and C have the same $L_t$. When $pr_3$ is finished, B and C will be decoupled.

BB-overloading will not increase $TTSF$ dramatically. Thus, $L_t$ will decide $TTSF$ at most time. Usually $TTSF$ is expected to be as small as possible. The smaller the $TTSF$, the better the fault-tolerant technique is. However, as long as the overloading exists, $TTSF$ will increase. It is a tradeoff between reliability and schedulability. The scheduler can know the minimum time interval of faults from the history. In this paper, $L_t$ is set simply to be a half of the minimum time interval between faults in the system history.

### 4. Proposed Task Scheduling Algorithm

#### 4.1 Overview

The proposed task scheduling approach in this paper consists of the elementary scheduling principles, the scheduling algorithms (the chief scheduling algorithm and the rescheduling algorithm), and the overloading strategy. The elementary scheduling principles are the outline of our task scheduling approach. The chief scheduling algorithm is used when it is easy to find available time slot for new tasks, for example, in the initiation of scheduling and the low task arrival rate. If the chief scheduling algorithm cannot find available time slots for new tasks, then the previously scheduled tasks will be re-scheduled. This is the re-scheduling algorithm, which is realized by moving previously scheduled tasks on one processor. The overloading strategy is used to achieve good schedulability and reliability when overloading takes place.

What time to start the new task and on which processor to allocate the task is a common problem for real-time task scheduling algorithms. In this paper, an allocation parameter (AP) is used to evaluate every possible task allocation.

#### 4.2 Allocation Parameter (AP)

For a new primary $pr_i$ to be allocated on processor $j$, $AP[pr_i, p_j, ft(pr_i)]$ of each possible allocation (possible $ft(pr_i)$) is defined as follows:

$$AP[pr_i, p_j, ft(pr_i)] =$$

$$\begin{cases} 
\frac{d - ft(pr_i)}{d - r_i} + \frac{1}{m}, & \text{for } n_{cascade} = 1, \text{non-overloading}, \\
\frac{d - ft(pr_i) - r_i}{d - r_i} + \frac{\text{cascade} \cdot \text{latency}}{m}, & \text{for } 1 < n_{cascade} \leq m, \text{overloading},
\end{cases}$$

and $AP[pr_i, p_j]$ is defined as:

$$AP[pr_i, p_j] = \max\{AP[pr_i, p_j, ft(pr_i)]\}. \quad (2)$$

For a new backup $bk_i$ to be allocated on processor $j$, $AP[bk_i, p_j, st(bk_i)]$ of each possible allocation (possible $st(bk_i)$) is defined as follows:

$$AP[bk_i, p_j, st(bk_i)] =$$

$$\begin{cases} 
\frac{st(bk_i) - r_i}{d - r_i} + \frac{1}{m}, & \text{for } n_{cascade} = 1, \text{non-overloading}, \\
\frac{st(bk_i) - r_i - \text{cascade} \cdot \text{latency}}{d - r_i} + \frac{\text{cascade} \cdot \text{latency}}{m}, & \text{for } 1 < n_{cascade} \leq m, \text{overloading},
\end{cases}$$

and $AP[bk_i, p_j]$ is defined as:

$$AP[bk_i, p_j] = \max\{AP[bk_i, p_j, st(bk_i)]\}. \quad (4)$$

Thus, we define the allocation parameter for a new task:
\[ AP = \begin{cases} \max\{AP(p_i, p_j)\} & \text{for } 1 \leq j \leq m, \\
& \text{when a primary is to be scheduled,} \\
\max\{AP(b_k, p_j)\} & \text{for } 1 \leq j \leq m, \\
& \text{when a backup is to be scheduled.} \end{cases} \quad (5) \]

Since we have
\[ 0 < \frac{\text{st}(bk_j) - r_i}{d_i - r_i} < 1, \quad (6) \]
\[ 0 < \frac{d_i - \text{ft}(pr_i)}{d_i - r_i} < 1, \quad (7) \]
\[ 0 < \frac{n_{\text{cascade}}}{m} < 1 \quad (8) \]

and
\[ 0 < \frac{t_{\text{overload}}}{c_i} < 1, \quad (9) \]

the value of AP is between 0 and 1.

PB-based fault tolerant real-time task scheduling is affected by many factors, for example location, cascade number of overloading, and depth of overloading (overlapping length). AP is the result of the trade off among these factors.

4.3 Elementary Scheduling Principles

1. A new task \( T_i \) will be scheduled on its arrival, i.e., FCFS (First Come First Serve).
2. The APs of both the primary and the backup of the new task are calculated, and the primary and the backup of this task are scheduled to the corresponding allocations.
3. It is possible that a processor has the same AP value as the other one. If two same AP values exist, the processor on which the task can achieve larger \( n_{\text{cascade}} \) is selected. If the two \( n_{\text{cascade}} \) are still identical, the processor on which the task can achieve larger \( t_{\text{overload}} \) is selected. If they are also identical, then a processor will be selected randomly.
4. The primary is scheduled as early as possible that is \( \text{st}(pr_i) \) tries to be closer to \( r_i \), the backup is scheduled as late as possible that is \( \text{ft}(bk_i) \) tries to be closer to \( d_i \). Equations (4) and (5) have effect on this principle.
5. If it is necessary to overload the task on the others, the larger \( n_{\text{cascade}} \) and \( t_{\text{overload}} \), the better schedulability is. Equations (8) and (9) have effect on this principle.
6. For the task, which cannot be allocated by the chief scheduling algorithm, the re-scheduling algorithm will try to move the previously scheduled tasks and find available time slots.
7. If a task still cannot find its available allocation, it will be rejected.
8. The constraints, the principles and the overloading strategy must be guaranteed in any operation.

4.4 Overloading Strategy

1. Backups can be overloaded on any task. Primaries can be overloaded only on backups.
2. If a primary \( pr_i \) is overloaded on a backup \( bk_j \), \( \text{st}(pr_i) \) must be later than \( \text{ft}(bk_j) \).
3. An overloading task set is scheduled to contain as many tasks as possible, and the time slot occupied by an overloading task set is scheduled as short as possible. This means the same as the principle 3.
4. A new task can only be overloaded on one task set.
5. If the number of processors is \( m \), the maximum \( L_\pi \) is \( m - 1 \).
6. The maximum \( L_\tau \) is equal to or larger than \( \max(d_i - r_i) \) and much less than \( MTBF \).
7. In the re-scheduling algorithm, an overloading task set is moved as a whole. After task re-scheduling, the relationship of the overloaded tasks cannot be changed.
8. A single PB-overloading chain should be opened but looped, that is the task sets of a chain can not exist on the same processor (a PB chain can also be looked as a chain of overloading task sets, see Definition 6). A looped chain has been shown in Fig. 2.

4.5 Scheduling Algorithms

In terms of the elementary scheduling principles and the overloading strategy, the scheduling algorithms are introduced simply as follows. The forward moving time of task set \( \tau \) is marked as \( \text{Win}_f(\tau) \). For the backward moving time of task set \( \tau \) is marked as \( \text{Win}_b(\tau) \). The validity checking consists of all constraints mentioned in Sect. 3.4, the elementary scheduling principles and the overloading strategy.

4.5.1 Chief Scheduling Algorithm

1. On a new task arrival
   a. Within \([r_i, d_i]\), on each processor, if the AP of the primary \( pr_i \) and the AP of the backup \( bk_i \) for the new task \( T_i \) exist and pass the validity checking,
      i. schedule \( pr_i \) and \( bk_i \) to the corresponding locations.
      ii. set \( \text{Win}_f(pr_i) \) and \( \text{Win}_b(bk_i) \), and if the new task is overloaded on a previously scheduled task set \( \tau \), update \( \text{Win}_b(\tau) \).
   b. If any one AP does not exist,
      i. call rescheduling algorithm,
      ii. if both the AP of \( pr_i \) and the AP of \( bk_i \) exist after the rescheduling and pass the validity checking,
         A. \( pr_i \) and \( bk_i \) are scheduled,
         B. set \( \text{Win}_f(pr_i) \) and \( \text{Win}_b(bk_i) \), and if the new task is overloaded on a previously scheduled task set \( \tau \), update \( \text{Win}_b(\tau) \).
SUN et al.: DYNAMIC SCHEDULING REAL-TIME TASK USING PRIMARY-BACKUP OVERLOADING STRATEGY

801

1. Re-scheduling

a. $T_i$ is the task to be scheduled; $\tau_j$ is a previously scheduled overloading task set between $r_i$ and $d_i$. $\tau_j$ can include one or more tasks. The $\text{Win}(\tau_i)$ is $\langle \text{back}, \text{for} \rangle$. The number of all $\tau_j$ is $s$, that is $1 \leq j \leq s$.

b. If $\text{back}_{j+1} + \text{for}_{j+1} < c_i$, then enlarge properly the interval between two neighboring task sets $\tau_j$ and $\tau_{j+1}$ by moving them backward and forward respectively to find a time slot for $T_i$. (If $\text{st}(\tau_1)$ is less than $r_i$, or $\text{ft}(\tau_s)$ is larger than $d_i$, then move $\tau_1$, $\tau_s$ forward or backward to find an available time slot.)

c. If a time slot is available, calculate $\text{AP}(pr_i, p_j)$ of this processor.

d. repeat (a) to (c) on each processor for all $\tau_j$ and $\tau_{j+1}$.

e. if $\text{AP}$ exists, then return $\text{AP}$. Otherwise, reject $T_i$.

2. Moving

a. If try to move the overloading task set $\tau_j$ forward and the forward time is $t$.

i. if $\text{Win}(\tau_j).\text{for} + t \geq \text{st}(\tau_{j+1})$, iteratively move $\tau_{j+1}$ forward with the forward time $t = \text{Win}(\tau_j).\text{for} + t - \text{st}(\tau_{j+1})$.

ii. if $\text{Win}(\tau_j).\text{for} + t < \text{st}(\tau_{j+1})$, the moving is successful. Otherwise the moving is failed.

b. If try to move the overloading task set $\tau_j$ backward and the backward time is $t$.

i. if $\text{Win}(\tau_j).\text{back} - t \leq \text{ft}(\tau_{j-1})$, iteratively move $\tau_{j-1}$ backward with the backward time $t = \text{ft}(\tau_{j-1}) - \text{Win}(\tau_j).\text{back} + t$.

ii. if $\text{Win}(\tau_j).\text{back} - t > \text{ft}(\tau_{j-1})$, the moving is successful. Otherwise, the moving is failed.

c. If the moving is successful eventually, change the previous schedule. Otherwise, keep the previous schedule and return.

The relationship among all components is shown in Fig. 6.

4.6 Examples

At the end of this section, some examples, which are helpful in understanding the content of this section, are shown as follows.

Example 1. $T_1, T_2, T_3,$ and $T_4,$ are previously scheduled tasks. Now, $T_5$ is going to be scheduled. After $T_5$ is scheduled, the schedule is shown in Fig. 7. $c_5$ is 1, $r_5$ is 1.5, $d_5$ is 4. All the other values follow the scale in the figure. The primary $pr_5$ can be scheduled on Processor 1 and Processor 2 and has three candidate locations, on Processor 1 within the time 1.5 to 2.5 ($A: pr_5$), on Processor 2 within the time 1.75 to 2.75 ($B: pr_5$), and on Processor 2 within the time 1.5 to 2.5 ($pr_5$). $pr_5$ cannot be scheduled on Processor 2 within the time 2 to 3 because of the overloading strategy 4. The three results of Eq. (1) are

$$\text{AP}(pr_5, p_1, 2.5) = \frac{4 - 2.5}{4 - 1.5} \times \frac{1}{3} = 0.2$$

$$\text{AP}(pr_5, p_2, 2.75) = \frac{4 - 2.75}{4 - 1.5} \times \frac{2}{3} \times \frac{0.75}{1} = 0.25$$

$$\text{AP}(pr_5, p_3, 2.5) = \frac{4 - 2.5}{4 - 1.5} \times \frac{2}{3} \times \frac{1}{1} = 0.4$$

Thus $\text{AP}$ of $pr_5$ is 0.4, and $pr_5$ is finally scheduled to Processor 2 with its finish time 2.5. After $pr_5$ is scheduled, $bk_5$ has four candidate locations, on Processor 1 within the time 3 to 4, on Processor 1 within the time 2.75 to 3.75, on Processor 2 within the time 3 to 4, and on Processor 3 within the time 3 to 4. The four results of Eq. (3) are
Thus AP of bk5 is 0.2, but there are two processors which have the same AP. In terms of the principle 3, bk5 is scheduled to Processor 1.

**Example 2.** The task set and the change of shift window when a new task T2 is allocated are shown in Fig. 8. In the case of A, before T2 is scheduled, the shift window of pr1 is (0, 0.75) and the shift window of bk1 is (0.75, 0). After T2 is scheduled, bk1 and bk2 form an overloading task set and have the same shift window. The shift window of bk1 and bk2 is also the same as that of the task set, which is (0.5, 0). In the case of B, pr2 and bk1 form the task set. Before overloading, the shift window of bk1 is (0.5, 0). After overloading, the shift window of bk1 is (0, 0). The more overloading, the smaller the shift window is.

**Example 3.** The re-scheduling usually happens when the load of each processor is heavy and the task arrival rate is high. An example of the re-scheduling is shown in Fig. 9. Because of the task deallocation, there exists an empty time...
slot before \( pr_2 \). This is reasonable because the deallocated task in front of \( pr_2 \) might be a backup which is the tail of a PB chain with the maximum length or a task whose actual computation time is much less than its worst case computation time, so in the previous schedule \( pr_2 \) cannot be overloaded on this empty time slot. The newly arriving task is \( T_7 \). The \( pr_7 \) cannot be scheduled without the re-scheduling. The ready time, deadline, and the shift window are displayed in the figure. The re-scheduling is the only choice for \( pr_7 \). Therefore, \( pr_7 \) is moved to the time slot from 0 to 0.5 and \( pr_7 \) is scheduled to the time slot from 0.5 to 1. \( bk_7 \) has many choices for its allocation.

5. Analysis of Algorithm

PB overloading has been proved to be better than BB overloading in schedulability and worse than BB overloading in reliability [8]. In this paper BB overloading and PB overloading can co-exist in a single overloading chain, and overloading chains can be coupled, for example in Fig. 5. Thus, our overloading strategy is more flexible than pure BB overloading and pure PB overloading. Because this paper focuses on the dynamic scheduling algorithm and the similar performance analysis has been in [8], we analyze the proposed scheduling algorithm in this section rather than to analyze the performance of overloading strategy. The analysis of the performance of overloading strategy has been shown in [14].

In the proposed scheduling algorithm, the main work is to search the best valid AP. Hence, the time complexity of our scheduling algorithm is the complexity of searching AP for a new \( pr_i \) and its \( bk_i \). The worst case of searching AP for a new task is

1. to search on all processors and fail to find a valid AP in the chief scheduling algorithm, then to move the previously scheduled tasks on each processor and find a valid AP for \( pr_i \),
2. to search on all processors and fail to find a valid AP in the chief scheduling algorithm, then to move the previously scheduled tasks on each processor and find no valid AP or only one AP for \( bk_i \).

The search operation in the chief scheduling algorithm always takes less time than that in re-scheduling algorithm, since in the worst case the search operation in the chief scheduling algorithm only need to check if there are previously scheduled task sets which can be overloaded and the intervals between task sets are large enough to contain the new task. Thus, the time complexity of rescheduling algorithm should be considered.

Let \( m \) denote the number of processors. Let \( N \) denote the average number of previously scheduled task sets on a processor. The worst case computation time follows uniform distribution within \([\text{Min}_c, \text{Max}_c]\). Let \( l \) denote task laxity. For task \( i \), we have \( d_i - r_i = l_i \cdot c_i \). In this paper, \( l \) follows uniform distribution within \([\text{Min}_l, \text{Max}_l]\). Because tasks are always scheduled to overload with each other tightly, we assume the time slot occupied by a task set is approximately the same as that occupied by a task. Thus, the maximum \( N \) can be represented as

\[
N_{\text{max}} = \frac{\text{Min}_c \cdot l + \text{Max}_c \cdot \text{Max}_c}{\text{Min}_c + \text{Max}_c}.
\]

We assume in the worst case the \( N \) previously scheduled task sets distribute within \([r_i, d_i]\) of the new task \( T_i \) as in Fig. 10.

When the rescheduling algorithm is invoked, it means the search operation in the chief scheduling algorithm is failed for the new primary or the new backup. In the worst case, the primary will search all \( m \) processors and the backup will search \( m - 1 \) processors because of the constraint 2. A version of a new task will be inserted into the intervals between \( r_i \) in Fig. 10. For a new primary the number of intervals is \( N \), since the last interval after \( r_n \) cannot be used by a primary (if a primary is inserted into this interval its backup cannot be scheduled for constraint 3). It is similar for a new backup that only the last \( N \) intervals can be used except the first one. Only if the rescheduling algorithm finds a valid AP for the new primary and a valid AP for its backup successfully, the previous schedules, only on one processor, will be changed. For an available interval, the moving operation only need to check if this interval can be enlarged to contain the new task according to the shift windows of those previous task sets. Since the shift windows are known in advance, we assume this operation only take one unite time.

Finally, we represent the time cost for re-scheduling a new primary on \( m \) processor as \( N \cdot m \), and the time cost for rescheduling a new backup on \( m - 1 \) processor as \( N \cdot (m - 1) \). The time complexity of proposed task scheduling algorithm is \( O(N^2 \cdot m \cdot (m - 1)) \).

6. Simulation Studies

To evaluate our task scheduling presented above, we have performed a series of simulations. We use the performance metric, Task Rejection Ratio [4], [12], to evaluate the experimental results. Rejection Ratio (RR) is defined to be the ratio of the number of tasks rejected to the total number of tasks that arrive at the system.

\[
RR = \frac{\text{the number of rejected tasks}}{\text{the number of arrived tasks}}
\]

6.1 Simulation Setup

The parameters used in our simulations are summarized in Table 1.

Some parameters are generated as follows:
Table 1 Parameter setting.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>number of tasks</td>
<td>10000</td>
</tr>
<tr>
<td>m</td>
<td>number of processors</td>
<td>2, 3, ..., 10</td>
</tr>
<tr>
<td>l</td>
<td>task laxity</td>
<td>3, 4, ..., 10</td>
</tr>
<tr>
<td>Max_c</td>
<td>maximum computation</td>
<td>100 s</td>
</tr>
<tr>
<td>Min_c</td>
<td>minimum computation</td>
<td>20 s</td>
</tr>
<tr>
<td>ac_ratio</td>
<td>ratio of actual to worst case computation time</td>
<td>0.5 ~ 1.0</td>
</tr>
<tr>
<td>load</td>
<td>task load</td>
<td>0.5, 0.6, ..., 1.0</td>
</tr>
</tbody>
</table>

- \( c_i \) is a random number following uniform distribution between \( \text{Max}_c \) and \( \text{Min}_c \).  
- \( ac_i \) is the multiple of \( c_i \) and \( ac\_ratio \) which follows uniform distribution.  
- The deadline of a task \( T_i \) is uniformly chosen between \( r_i + 2 \cdot c_i \) and \( r_i + l \cdot c_i \). \( r_i = a_i + \delta \). \( \delta \) is a random number between 1 s and 10 s.  
- The time interval of faults follows exponential distribution with the mean \( (\text{MTBF}) \). The minimum value is \( 2l \cdot \text{Max}_c \).  
- The interval between task arrivals follows exponential distribution.  
- The task load is defined as the expected number of task arrivals per mean service time and its value is approximately equal to the ratio of the mean computation time of tasks to the mean time interval of task arrivals.

According to the introduction of Sect. 2, we compare our task scheduling algorithm with the algorithms in [4] and [8], which are named SG and RA. Because PB-overloading and BB-overloading exist in two algorithms in [8], RA includes RAPB and RABB, where the dynamic grouping is omitted considering the 1-TFT. In order to make a fair comparison, all algorithms will share the parameter setting with our algorithm and the parameter weight in [4] is set to be 0. Our task scheduling approach with the re-scheduling algorithm is denoted as OR. The re-scheduling algorithm can be reduced in our task scheduling approach, and this reduced algorithm is denoted as ONR.

6.2 Results

The effect of varying the number of processors on \( RR \) is shown in Fig. 11. As the increase of the number of processors \( RR \) decreases and tends to be close to 0. This is a commonness for most fault tolerant task scheduling algorithms, since the large number of processors means the large computation capacity. The larger task laxity leads to the more flexible allocation of tasks. Hence, the larger task laxity, the smaller \( RR \) is. This effect is shown in Fig. 12. In Fig. 13, \( RR \) increases as task load does. The larger task load means more tasks arrive in the unit time.

OR achieves the lower \( RR \) than the other algorithms in the above three experiments. The principle 4 tries to enlarge the distance between a primary and its backup. The time and space exclusion of primary and backup tasks can save the used resources because of the task deallocation. The overloading strategy is flexible, since to schedule a task with PB overloading or BB overloading is decided only by the task attributes and the system condition. Therefore, OR and ONR have the lower \( RR \) due to the higher processor utilization. The rescheduling can help to squeeze more empty time slots from the previous schedules, so OR is better than ONR.

The maximum time length of PB chain is affected by \( \text{MTBF} \). Moreover, when a fault happens, the scheduler will not schedule tasks to the failed processor. Thus, \( \text{MTBF} \) has the effect on \( RR \). Under different task load, the effect is
6.3 Time Complexities

To evaluate the efficiency of the task scheduling algorithms, we should consider their time costs of scheduling a task. A simple method is to compare the time complexities of these scheduling algorithms. The time complexity of our algorithm has been analyzed in Sect. 5. In [4] and [8], the authors did not give the time complexities of their algorithms explicitly. Hence, in this section, we first list the steps of SG and RA, and then analyze their time complexities.

SG will perform the following steps to schedule a new primary and a new backup.

1. Search all processors to find if the new task can be scheduled between $r_i$ and $d_i$. The primary is scheduled as early as possible.
2. If Step 1 is failed to schedule the primary, all previous primaries will be checked to move forward. If a primary can be successfully moved forward, the new primary will be allocated in the time slot for the moved primary.
3. Based on the schedule produced by Step 2, the backup will be scheduled to the processors not including the processor the primary has been allocated to. The backup will be scheduled using a heuristic (as late as possible or to maximize overloading in [4]).
4. Commit the task including its primary and backup.

SG is similar to our algorithm for rescheduling. From the steps of SG, it is obvious that Step 2 is the worst case of the scheduling. We also assume that there are $N$ task sets on a processor. Therefore, the time cost of scheduling a primary is $N \cdot m$, the same as our algorithm. SG will not reschedule the previous tasks for a backup. Thus, we assume that the time cost of scheduling a backup is $K$, which depends on the heuristic. For $m - 1$ processors, the time cost is $K \cdot (m - 1)$. Obviously, the worst case of $K$ is equal to $N$. Finally, we can represent the time complexity of SG as $O(N \cdot K \cdot m \cdot (m - 1))$. Thus, if the heuristic is to schedule backup as late as possible, the time complexity of SG is a little less than that of our algorithm. But in the worst case, SG has the same time complexity as ours.

RA including RABB and RAPB is not to schedule task in the way of FCFS like SG and ours. RA is based on the Spring kernel [9], where the most feasible task is chosen out of all the waiting tasks, and then if this task can be scheduled successfully, the task will be accepted, or this task will be rejected. The Spring scheduling algorithm does not include the task allocation. In [8], there is no clear statement for the task allocation after the task is chosen. Hence, in this paper, we allocate the primary as early as possible and the backup as late as possible. We also assume that there are $N$ previous tasks between $r_i$ and $d_i$ of the new task. The allocation process, in fact, is the same as our algorithm. In the worst case, all the time intervals among the previous tasks must be checked to accept the new task. Referring to the time complexity analysis of our algorithm, the time complexity of the task allocation for RA is the same as our algorithm. But if we assume that the time cost of Spring kernel is $K'$, the time complexity of RA can be represented as $O(K' \cdot N \cdot K \cdot m \cdot (m - 1))$, where $K$ is the same as SG because the backup is schedule in the same way.

Based on the above analysis, SG and our algorithm have the same time complexity and RA has the higher time complexity. SG can reschedule the primary. Because SG does not use task sets to control the relation of overloaded tasks and the rescheduling, to reschedule backups, i.e., to move backups, is difficult. Therefore, backups cannot be rescheduled in SG. Although SG has a lower time complexity, its performance is worse than RA because RA uses the Spring scheduling algorithm to choose the most feasible task before scheduling a new task. Our scheduling algorithm does not choose the most feasible task, but the rescheduling with forward and backward moving can efficiently use the empty time slots or overload more tasks together. Thus, our algorithm can achieve the better performance with the relatively lower time complexity than RA.

7. Conclusion

In this paper, we have proposed a fault tolerant dynamic
real-time task scheduling approach, which is based on the primary-backup fault tolerant technique. The scheduler uses the allocation parameter to search the proper time slots for a new task, and the scheduler uses the re-scheduling algorithm to change the previous task schedule for possible empty time slots by moving previously scheduled tasks on one processor. The simulations have shown that our approach is better than the others. The results have shown that the task rejection ratio of our real-time task scheduling algorithm is almost 50% lower than the compared algorithms.

References


