A Hybrid Real-time Fault-tolerant Scheduling Algorithm for Partial Reconfigurable System

Jinyong Yin  
Jiangsu Automation Research Institute, Lianyungang, China  
Email: yinjinyong@yahoo.com

Boxiang Zheng  
Jiangsu Automation Research Institute, Lianyungang, China  
Email: zboxiang@sohu.com

Zhongyi Sun  
Jiangsu Automation Research Institute, Lianyungang, China  
Email: sunbigman@126.com

Abstract—Partial reconfigurable system is an architecture consisting general purpose processors and FPGAs, in which FPGA can be reconfigured in run-time. Based on the architecture, software tasks and hardware tasks that are executed on processor and FPGA respectively co-exist. In this paper, a real-time fault-tolerant scheduling algorithm is proposed to schedule software/hardware hybrid tasks. In the algorithm, the sufficient condition for schedulable hybrid tasks is derived from analyzing system operation conditions when the first deadline is missed, and rollback/recovery and TMR approaches are used respectively to schedule software subtasks and hardware subtasks for fault tolerance. The experimental results demonstrate that all deadlines of accepted hybrid tasks are met and processor’s utilization ratio is increased greatly compared with that of the exiting approaches when multiple faults occur.

Index Terms—Partial reconfigurable system, Real-time scheduling algorithm, Fault-tolerant scheduling algorithm, Software/Hardware hybrid tasks

I. INTRODUCTION

As we all know, the FPGA’s configuration information is stored in static RAM, which is easily affected by space particles and electromagnetic wave. When the static RAM is affected, the SEU (Single Event Upset) occurs and internal circuit fails. In order to avoid the serious consequences caused by system failure, we need to provide the fault-tolerant ability for FPGA to ensure that all real-time tasks’ deadlines can be met even internal partial circuit failed.

The fault-tolerant solutions for FPGA are usually divided into two categories which are based on hardware and software respectively. The first one is still thinking along the hardware redundancy which set the spare resources in the FPGA chip to achieve fault tolerance. If the resources are damaged somewhere, it will be replaced by the spare resources [1]. Doumar et al [2] proposed a solution which can move the configuration data between the row, the column and the modules of I/O by the special designing of the SRAM. When a failure occurs, the configuration data of failure resource will be transferred to the adjacent free resources according to specific rules to make the system back to normal. The second approach is based on hardware/software coordination which firstly tests the FPGA chips and stores the damaged data in the database by software, and then the test results are read from the database to ensure that the damaged parts are not used and finally re-layout to resume normal operation of FPGA [3]. Above solutions require the system to stop working when detecting the FPGA chip which will reduce performance and flexibility of the system. So some researchers proposed schemes of online detection and dynamic reconfiguration while the system is still working [4, 5]. These solutions are mainly made for non-real-time systems offline or online fault detection while real-time tasks can not be guaranteed.

Real-time fault-tolerant scheduling is a technology which can achieve the ability of system fault tolerance through software and improve the reliability of the system with limited hardware spending. The proposed fault-tolerant scheduling algorithms [6-15] were mainly developed based on the technology of primary/backup version, and when multiple processors fail at the same time in the system, the multiple backups version is needed for each real-time task which will result in the processor utilization decrease rapidly.

To resolve the rapid decrease of processor utilization and the problem of the real-time hardware task’s fault-tolerance, this paper proposes a fault-tolerant scheduling algorithm FT-SSHTNB (Fault Tolerant SSHTNB) for real-time hardware/software hybrid tasks, the FT-SSHTNB algorithm is based on SSHTNB algorithm [17].

II. TASK AND SYSTEM MODELS
In the system, hybrid tasks are presented by a set $T = \{T_1, T_2, \ldots, T_n\}$ and each task $T_i \in T$ is represented by a directed acyclic graph (DAG): $T_i = (ST_i, E_i, C_i, D_i, P_i)$, where $ST_i = \{s_{t_1}, s_{t_2}, \ldots, s_{t_k}\}$ represents the hardware and software subtasks of $T_i$, $E_i \subseteq ST_i \times ST_i$ represents the relationship of constraints between subtasks, $C_i$, $D_i$ and $P_i$ represent the execution time, relative deadline and period of the task $T_i$, respectively, where $D_i = P_i$.

$\text{IPred}(s_{t_k}) = \{s_{t_1} | (s_{t_1}, s_{t_k}) \in E_i\}$ represents the immediate predecessor subtasks of the subtask $s_{t_k}$, and if $|\text{IPred}(s_{t_k})| = 0$, then the $s_{t_k}$ is defined as entry subtask.

$\text{ISucc}(s_{t_k}) = \{s_{t_1} | (s_{t_k}, s_{t_1}) \in E_i\}$ represents the immediate successor subtasks of the subtask $s_{t_k}$, and if $|\text{ISucc}(s_{t_k})| = 0$, then the $s_{t_k}$ is defined as exit subtask.

Figure 1 shows a task's instance which contains 8 subtasks, where rectangle presents hardware subtask, circle presents software subtask, and line presents the longest path from the entry subtask to the exit subtask.

Not lose the general case, assume that each task $T_i$ only has an entry subtask $Entry_i$ and an exit subtask $Exit_i$. Figure 1 shows a task’s instance which contains 8 subtasks, where rectangle presents hardware subtask and circle presents software subtask.

Let $C_{ij}^{cp}$ presents the sum execution time of subtasks in task $T_i$’s critical path; Let $C_{ij}^{det}$ presents all software subtasks’ execution time of task $T_i$; Let $C_{ij}^{top}$ presents the longest path from the entry subtask to the subtask $s_{t_j}$; Let $C_{ij}^{bot}$ presents the longest path from subtask $s_{t_j}$ to the exit subtask.

Subtask $s_{t_j}$ is presented by a set $s_{t_j} = \{C_{ij}, w_{ij}, h_{ij}, type_{ij}\}$, where $C_{ij}$ presents execution time of subtask $s_{t_j}$, $w_{ij}$ presents the width of subtasks, $h_{ij}$ presents the height of subtasks, and $type_{ij} = \{H, S\}$ presents subtask’s type where $H$ presents that $s_{t_j}$ is a hardware subtask and $S$ presents that $s_{t_j}$ is a software subtask and if $s_{t_j}$ is a software subtask, then $w_{ij} = h_{ij} = 0$.

Figure 2. Reconfigurable system architecture. (a) System architecture (b) The architecture in FPGA

Shown in Figure 2, reconfigurable system consists of multiple processors and multiple FPGAs. Processors and FPGAs are linked together through high speed serial bus. Some free resources (Slots) are reserved for hardware tasks in FPGA, where hardware subtasks can be configured to the slots dynamically when the system is still running. The hardware subtasks can communicate with the software subtasks through the serial bus interface and hardware task interfaces (HTI).

III. FT-SSHTNB SCHEDULING ALGORITHM

FT-SSHTNB algorithm has some restrictions on the fault model and fault number. We make the following assumptions for failure model:

1) At most $f$ computing units fail at a time, when the $f+1$th failure occurs, there is at least one of the former $f$ faults has been repaired and put into operation.
2) Suppose that the interval between two failures is longer than the period of real-time tasks, namely, there is at most one failure occurs when a real-time task executes.

Definition 1: For a task $T_i$, if it can tolerate $k$ failures, that is, when the failures are not more than $k$, task $T_i$ can be fault-tolerant scheduled by a algorithm; when the $k+1$ failure occurs, the algorithm will go into the exception process, then the algorithm is called $k$-fault-tolerant for the task $T_i$.

Definition 2: For a task set $T = \{T_1, T_2, \ldots, T_n\}$, if a scheduling algorithm is $k$-fault-tolerant for task $T_i$, then the algorithm is $(k_1, k_2, \ldots, k_n)$ - fault-tolerant for the task set $T$.

We can see from the fault model that FT-SSHTNB algorithm is $f$-fault-tolerant for each task $T_i$.

According to the place where SEU occurs, the system failure can be divided into two categories: software failure and hardware failure. If the SEU occurs in the main memory, the program’s data flow and (or) instruction flow are prone to error and if the SEU occurs in the configuration RAM, the hardware error will occurs. FT-SSHTNB algorithm schedules the software subtask on two processors at one time and compares subtask execution state in check point. By this way, processor error and software task error can be found. And hardware subtask error is detected and tolerated through the TMR structure.
A. Fault-tolerance for Software Subtasks

![Diagram](image)

Figure 3. Fault checking principle (a) Software subtask executing model (b) Check point comparison and threads synchronization.

As figure 3(a) shows, each software subtask $st_i$ is correspond to two threads $thread_j^i$ and $thread_k^i$ which were scheduled to two processors and executes simultaneously. By checking the consistency of checkpoint and the synchronization of the two threads, processor error and software task error can be detected. If a failure is detected, the threads are rolled back to the appropriate checkpoint.

From the point view of $thread_j^i$, fault detection process is shown in Figure 3 (b), where $CP_i$ presents the $ith$ checkpoint, up and down arrows indicate the current position of the thread. When $thread_j^i$ is at the checkpoint $CP_i$, if the thread $thread_k^i$ is at position $1$, then thread $thread_j^i$ and $thread_k^i$ roll back to the checkpoint $CP_{i-1}$; if the $thread_k^i$ is at position $2$, then the thread $thread_j^i$ compares the states of checkpoint $CP_{i-2}$, and if the states are not identical, then the thread $thread_j^i$ and $thread_k^i$ roll back to the checkpoint $CP_{i-2}$ at the same time, otherwise the thread $thread_j^i$ saves the state of checkpoint $CP_i$ and continues executing; if the thread $thread_j^i$ is at position $3$, then the thread $thread_j^i$ saves the state of checkpoint $CP_i$, and continues executing.

**Definition 3:** When $thread_j^i$ reaches the checkpoint $CP_i$, thread $thread_j^i$ will take at least $sd_i$ time to reach checkpoint $CP_i$, we call $sd_i$ as synchronization distance at checkpoint $CP_i$.

**Definition 4:** Given a constant $\zeta$, if the sdk \(<\zeta$, thread$1$ and thread$2$ synchronize at checkpoint $CP_i$, otherwise $thread_j^i$ and $thread_k^i$ are not synchronized.

In Figure 3, the checkpoint number is generated dynamically during the execution of thread, each checkpoint number increases 1 when the thread encounters a checkpoint. The checkpoint’s state is stored as an information block and expressed as a 4-tuple $ckp = \{id_s, data_1, id_s, data_2\}$, where $id_s$ and $id_s$ are the current checkpoint number of thread $thread_j^i$ and $thread_k^i$, $data_1$ and $data_2$ are data saved in the checkpoint. Comparison and synchronization algorithm $CmpSync$ is described as follows:

**Step 1:** Determining which thread will execute the algorithm

- If the thread is $thread_j^i$, then $thisid = id_s$, otherid = id_s;
- If the thread is $thread_k^i$, then $thisid = id_s$, otherid = id_s;

**Step 2:** Comparing checkpoint number

1. If $thisid > otherid$, then the restore task $T_{ru}$ is started to recover the status of checkpoint $CP_{i-2}$.

2. If $thisid = otherid$, the status of checkpoint $CP_{i-1}$ are compared. If the statuses are identical, then the save task $T_{sa}$ is started to save the checkpoint $CP_i$; and $thisid$ increases 1, otherwise the task $T_{re}$ is started to recover the state of checkpoint $CP_{i-2}$.

3. If $thisid < otherid$, the task $T_{sa}$ is started to save the state of checkpoint $CP_i$ and $thisid$ increases 1.

In algorithm $CmpSync$, $thisid$ and $otherid$ are the checkpoint number of current thread and another thread respectively, If $thisid > otherid$, then the interval of the two threads is greater than a detection length and we can determine that the other processor fails and recovers checkpoint’s state. If $thisid = otherid$, then the thread is ahead of the thread on the other processor and the checkpoint is not compared, so the checkpoint will be compared and it is determined to save the new checkpoint or roll back to the previous checkpoint according to the consistency of checkpoint. If $thisid < otherid$, the thread is behind the thread on the other processor and the caparison of checkpoint has complete, so it is only needed to save the checkpoint’s state.

Both $thread_j^i$ and $thread_k^i$ should execute algorithm $CmpSync$ when they reach checkpoint, so the number of execution is same, but the execution load of the algorithm $CmpSync$ is not same, because sometimes checkpoint’s state need to be compared and sometimes not. The following theorem will prove that the load asymmetry does not lead threads $thread_j^i$ and $thread_k^i$ to lose synchronization.

**Theorem 1:** Threads $thread_j^i$ and $thread_k^i$ will not lose synchronization as executing $CmpSync$ algorithm.

**Proof:** According to Figure 3, when the thread $thread_j^i$ encounters a checkpoint $CP_i$, under normal circumstances, the thread $thread_j^i$ may be at the position $2$ or $3$. When the thread $thread_j^i$ is at position $2$, the thread $thread_j^i$ compares state of checkpoint $CP_{i-1}$; When the thread $thread_j^i$ is at position $3$, the thread $thread_j^i$ just saves the state of checkpoint $CP_i$.

From the above analysis, we can see that the comparison operation of the checkpoint is always implemented by thread at ahead, so the threads $thread_j^i$ and $thread_k^i$ will not lose synchronization due to the implementation of algorithm $CmpSync$. $\square$
As shown in Figure 4, when the thread \( \text{thread}_{ij} \) encounters a checkpoint, the maximum rollback length of task is different according to the location of the thread \( \text{thread}_{ij} \).

Suppose that the detection interval length of subtasks \( T_{ij} \) is \( C_{ij} \), when the thread \( \text{thread}_{ij} \) is at position \( \circ \), the task’s rollback length is \( 2C_{ij} + C_{sa} + C_{re} \); when the thread \( \text{thread}_{ij} \) is at position \( \circ \), the task’s rollback length is \( C_{ij} + C_{sa} + C_{re} \); when the thread \( \text{thread}_{ij} \) is at position \( \circ \), the task’s rollback length is the longest and denoted as:

\[
C_{ij} = 2C_{ij} + C_{sa} + C_{re}
\]  

(1)

Assuming that the subtask \( st_{ij} \) sets \( m_{ij} \) checkpoints and the detection intervals are equal, that is \( C_{in} = \frac{C_{ij}}{m_{ij} - 1} \), when no fault occurs during the execution of the subtask \( st_{ij} \), the execution time of \( st_{ij} \) is:

\[
C_{in} = C_{ij} + m_{ij}C_{sa}
\]  

(2)

When one fault occurs during the execution of the subtask \( st_{ij} \), the maximum execution time of \( st_{ij} \) is:

\[
C_{iT} = C_{in} + C_{ir}
\]  

(3)

When \( r \) faults occur, the maximum execution time of task \( T_i \)’s software subtask is:

\[
C_{soft} = \sum C_{ij} + \max \left\{ \sum C_{ij} \right\},
\]  

\( \forall st_{ij} \in ST_i \land st_{ij}.type = S \)  

(4)

Checkpoints can effectively reduce the task execution time, but if the cost of checkpoint is large or too many checkpoints are set, the total execution time may be longer than the execution time with no checkpoint \([16]\). According to equation (3), when the task’s execution time \( C_{in} \), checkpoint recovery time \( C_{sa} \) and saving time \( C_{sa} \) are certain, the task’s maximum execution time depends on checkpoint’s number \( m_{ij} \).

**Theorem 2:** Suppose \( C_{sa} < C_{ij} \), when \( m_{ij} = \left[ \frac{2C_{ij}}{C_{sa}} + 1 \right] \) or \( \left[ \frac{2C_{ij}}{C_{sa}} + 1 \right] \), the execution time of subtask \( st_{ij} \) reaches the minimum.

**Proof:** Take the equation (1) and (2) into equation (3), we can get:

\[
C_{iT} = C_{ij} + C_{ir} = m_{ij}C_{sa} + 2\frac{C_{ij}}{m_{ij} - 1} + C_{sa} + C_{re}.
\]

The first order differential equation and second order differential equation of \( C_{iT} \) for \( m_{ij} \) are:

\[
\frac{dC_{ij}}{dm_{ij}} = C_{sa} - 2\frac{C_{ij}}{(m_{ij} - 1)^2}, \quad \frac{d^2C_{ij}}{dm_{ij}^2} = 4\frac{C_{ij}}{(m_{ij} - 1)^3}.
\]

Because \( m_{ij} \geq 2 \) (we set a checkpoint at the entry and the exit of subtask \( st_{ij} \) respectively), so the second order differential equation \( \frac{d^2C_{ij}}{dm_{ij}^2} > 0 \). When the first order differential equation \( \frac{dC_{ij}}{dm_{ij}} = 0 \), that is \( m_{ij} = \sqrt{\frac{2C_{ij}}{C_{sa}} + 1} \), \( C_{ij} \) will obtain the minimum value. As \( m_{ij} \) is a positive integer, so when \( m_{ij} = \left[ \frac{2C_{ij}}{C_{sa}} + 1 \right] \) or \( \left[ \frac{2C_{ij}}{C_{sa}} + 1 \right] \), \( C_{ij} \) will obtain the minimum value. □

**B. Fault-tolerance for Hardware Subtasks**

Based on the existing technology, saving and recovering circuit’s state still can not be implemented or the cost is too large, so when the hardware subtask fails, strategy of rollback/recovery should not be taken. In this paper, we use TMR technology to detect and tolerate hardware subtask’s failure.

In order to increase the utilization of reconfigurable resource, hardware subtasks with precedence constrains are partitioned into the same group and all subtasks in a group are configured into one slot. The partition algorithm is described as fellows:

**Partition (\( T_i \))**

\[
\text{// each while circle partition a group of subtasks}
\]

\[
\text{while}(T_i \text{ has unmarked hardware subtask})
\]

\[
\text{Group = } \Phi;
\]

\[
\text{MaxTask = the unmarked widest subtask of task } T_i; \text{ Mark the MaxTask};
\]

\[
\text{Group} = \text{Group } \cup \{ \text{MaxTask} \};
\]

\[
\text{// mark MaxTask’s predecessor subtasks}
\]

\[
\text{CurTask} = \text{MaxTask};
\]

\[
\text{while}(\text{CurTask has predecessor subtask})
\]

\[
\text{if} \text{CurTask has unmarked predecessor subtask, then } \text{Pred} = \text{CurTask's unmarked widest predecessor subtask}, \text{ otherwise Pred = CurTask's first predecessor subtask;}
\]

\[
\text{if} \text{Pred is unmarked, then Group} = \text{Group} \cup \{ \text{Pred} \} \text{ and mark Pred;}
\]

\[
\text{CurTask} = \text{Pred};
\]

\[
\text{// mark MaxTask’s successor subtasks}
\]

\[
\text{CurTask} = \text{MaxTask};
\]

\[
\text{while}(\text{CurTask has successor subtask})
\]

\[
\text{if} \text{CurTask has unmarked successor subtask, then } \text{Succ} = \text{CurTask's unmarked widest successor subtask, otherwise Succ = CurTask's first successor subtask;}
\]

\[
\text{if} \text{Succ is unmarked, then Group} = \text{Group} \cup \{ \text{Succ} \} \text{ and mark Succ;}
\]
After the subtasks are grouped by Partition algorithm, subtasks belonging to the same group are configured to three slots at the same time. The hardware subtask \( st_{ij} \) is presented as \( st_{ij}^1, st_{ij}^2 \) and \( st_{ij}^3 \) in three slots. By comparing the execution results of \( st_{ij}^1 \), \( st_{ij}^2 \) and \( st_{ij}^3 \), hardware subtask \( st_{ij}^k \)'s error can be masked off.

### C. Tasks’ Schedulability Test

According to the theorem 4-1 in paper [17], if a set of periodic tasks \( T = \{T_1, T_2, \ldots, T_n\} \) satisfies the inequality \( \sum_{i=1}^{k} \left[ D_k / P_i \right] C_{i}^{soft} \leq mD_k - (m - 1)C_{k}^{CP} \), then \( T \) can be scheduled by SSHTNB algorithm on \( m \) processors without fault-tolerant requirements. During the execution of task \( T_k \), the critical path will be changed if some processors fail. So the critical path must be found when different processors fail.

**Theorem 3:** For a set of periodic tasks \( T = \{T_1, T_2, \ldots, T_n\} \) satisfies \( T \) and the critical path is at most \( 3 \) slots at the same time. The hardware subtask \( st_{ij} \) is presented as \( st_{ij}^1, st_{ij}^2, st_{ij}^3. \) By comparing the execution results of \( st_{ij}^1, st_{ij}^2 \) and \( st_{ij}^3 \), hardware subtask \( st_{ij}^k \)'s error can be masked off.

**Proof:** Just need to prove that when \( f \) processors fail, the task set \( T \) can be scheduled by FT-SSHTNB algorithm on \( m-f \) processors.

The second term of left side in inequation 5 represents the sum of \( k-1 \) high-priority tasks’ \( f \)-longest rollback length. During the execution of task \( T_k \), if the \( T_k \)'s processor fails \( r \) times, then the other processors in the system at most fail \( f-r \) times. So the demands of \( k-1 \) high-priority tasks at most increase \( \max \{ \sum_{r} C_{ij}^{R} \} \) and the demand of task \( T_k \) is \( C_{ij}^{soft-r} \) in one period of task \( T_k \). Because the increase of each path of task \( T_k \) is less than or equal to \( \max \{ \sum_{r} C_{ij}^{R} \} \), the length of task \( T_k \)'s critical path is at most \( C_{ij}^{CP} + \max \{ \sum_{r} C_{ij}^{R} \} \) when processors fail \( r \) times during the execution of task \( T_k \).

**E. The Analysis of Real-time Capability**

Because of the reliability of computing units, all real-time systems can not guarantee each real-time task's deadline when some computing units fail. So we give the definition of real-time capability for a task as follows:

**Definition 5:** the real-time capability of a task \( T_k \) is the probability \( P(T) \) that task \( T_k \) can be finished within its deadline.

SEU is a random event with the characters: (1) in time interval \( [t, t+\Delta t] \), the probability of SEU happening \( k \) \((k<0) \) times only depends on interval length \( \Delta t \) and has no relation with interval endpoints \( t, t+\Delta t \); (2) One SEU happens independently to the others in time intervals without overlaps; (3) the probability of SEU happening two or more times can be thought as zero when the time interval is small enough. So SEU flow can be regarded as Poisson flow. Let \( X_t \) represents the times of SEU.
occurring to a computing unit within task $T_k$’s period $P_k$, and suppose the intensity of SEU flow as $\lambda$, then the probability of the computing unit failing during task $T_i$ executing is:

$$P(X_i \geq 1) = 1 - P(X_i = 0) = 1 - e^{-\lambda t_i} \quad (6)$$

Let $n_f$ represents the number of computing units that fail. Because the computing unit fails independently, the probability of $q$ computing units failing in the same time within a system containing $M$ computing units during task $T_k$ executing is:

$$P(n_f = q) = C_M^q \left(1 - e^{-\lambda P_k}\right)^q \left(e^{-\lambda P_k}\right)^{M-q} \quad (7)$$

When $q \leq f$, the task $T_k$ can be finished within its deadline according to FT-SSHTNB algorithm, so the real-time capability of task $T_k$ is:

$$P(T_k) = P(n_f = 0) + P(n_f = 1) + \cdots + P(n_f = f) \quad (8)$$

### TABLE I.

**REAL-TIME CAPABILITY OF TASK $T_k$ UNDER DIFFERENT CONDITIONS**

<table>
<thead>
<tr>
<th>Faults number $f$</th>
<th>$P_k = 100$</th>
<th>$P_k = 200$</th>
<th>$P_k = 300$</th>
<th>$P_k = 400$</th>
<th>$P_k = 500$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.951229</td>
<td>0.964837</td>
<td>0.860709</td>
<td>0.818730</td>
<td>0.778800</td>
</tr>
<tr>
<td>1</td>
<td>0.998814</td>
<td>0.995411</td>
<td>0.990009</td>
<td>0.982804</td>
<td>0.973987</td>
</tr>
<tr>
<td>2</td>
<td>0.999981</td>
<td>0.999854</td>
<td>0.999527</td>
<td>0.998915</td>
<td>0.997957</td>
</tr>
<tr>
<td>3</td>
<td>0.999999</td>
<td>0.999996</td>
<td>0.999985</td>
<td>0.999948</td>
<td>0.999880</td>
</tr>
</tbody>
</table>

Suppose the real-time system contain 50 computing units and the SEU flow intensity $\lambda = 10^{-5}$, the real-time capability of task $T_k$ is shown in table I according to formula 8.

### IV. EXPERIMENTAL RESULTS AND ANALYSIS

In these experiments, we have simulated processor utilization under different conditions and compared with the related algorithms, and the results are shown in table II, table III and table IV as below.

### TABLE II.

**CAPABILITY COMPARISON OF FT-SSHTNB WITH RELATED ALGORITHMS**

<table>
<thead>
<tr>
<th>Items</th>
<th>FT-RMFF</th>
<th>HTFS</th>
<th>Liu</th>
<th>DABCBF</th>
<th>FT-SSHTNB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent tasks</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Precedence Constraint tasks</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>S/H hybrid tasks</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>√</td>
</tr>
<tr>
<td>One processor fault</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>Multi processor faults</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>√</td>
</tr>
</tbody>
</table>

FT-RMFF [6] algorithm is a classic fault-tolerant scheduling algorithm for periodic tasks base on RM scheduling algorithm; HTFS algorithm [12] uses FT-RMFF algorithm to test the schedulability of periodic tasks and aperiodic tasks also can be fault-tolerant scheduled; DABCBF algorithm [13] improved the FT-RMFF algorithm by deferring the execution of task’s active slave copy to increase the processor utilization; Liu [8] presents a new algorithm to test the schedulability of periodic tasks with fault-tolerant requirement. As can be seen from Table II, the above mentioned algorithms can only schedule independent software tasks and tolerate one processor failure. Compared with these algorithms, FT-SSHTNB algorithm’s capability is much stronger.
TABLE III.
PERFORMANCE COMPARISON OF FT-SSHTNB WITH RELATED ALGORITHMS

<table>
<thead>
<tr>
<th>Fault numbers</th>
<th>Algorithms</th>
<th>FT-RMFF</th>
<th>HTFS</th>
<th>Liu</th>
<th>DABCBF</th>
<th>FT-SSHTNB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>61.582990</td>
<td>59.347181</td>
<td>48.993356</td>
<td>71.990353</td>
<td>32.932836</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>41.051221</td>
<td>39.564787</td>
<td>32.662204</td>
<td>47.993568</td>
<td>31.870486</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>30.788416</td>
<td>29.673590</td>
<td>24.496653</td>
<td>35.995176</td>
<td>30.808136</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>24.630732</td>
<td>23.738872</td>
<td>19.597322</td>
<td>28.796141</td>
<td>29.745787</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>20.525610</td>
<td>19.782393</td>
<td>16.331102</td>
<td>23.996784</td>
<td>28.683437</td>
</tr>
</tbody>
</table>

Because FT-RMFF, HTFS, Liu and DABCBF algorithms adopted primary/slave copy technology, if each real-time task has multiple slave copies, these algorithms also can tolerate multiple processor failures. In this experiment, task’s execution time accords to the uniform distribution in \((0, 0.5\pi]\) and independent software periodic tasks are fault-tolerant scheduled on 32 processors, and the scheduling results are shown in Table III. From Table III we can see that FT-SSHTNB algorithm has low processor utilization when fault-tolerant number is few, however the processor utilization of FT-SSHTNB algorithm is higher than that of other algorithms with the fault-tolerant number increases.

TABLE IV.
AVERAGE PROCESSOR UTILIZATION UNDER DIFFERENT CONDITIONS (%)

<table>
<thead>
<tr>
<th>Fault numbers</th>
<th>Processor numbers (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>39.820220</td>
</tr>
<tr>
<td>1</td>
<td>36.184924</td>
</tr>
<tr>
<td>2</td>
<td>30.142350</td>
</tr>
<tr>
<td>3</td>
<td>26.073210</td>
</tr>
<tr>
<td>4</td>
<td>20.432851</td>
</tr>
<tr>
<td>5</td>
<td>15.167190</td>
</tr>
</tbody>
</table>

Table IV shows the scheduling results of software/hardware hybrid tasks with parameters same to SSHTNB algorithm. As can be seen from Table IV, the average utilization ratio of processor decreases with the fault-tolerant number increase, however the decrease becomes unobvious with the processor number increase. For example, when there are 8 processors in the system, the processor’s utilization ratio decreases by 24.7%, however when there are 64 processors, the ratio only decreases 2.8%. The reason is that if the fault-tolerant number is \(f\), the task set \(T\) must be scheduled on \(m-f\) processors according theorem 3. When \(m\) is few, the task’s load will too large to be scheduled on \(m-f\) processors, so the processor utilization ratio is lower.

V. CONCLUSIONS

In this paper, a method of processor and software task fault detection and tolerance is given firstly. When there are multiple processor failures, this method can effectively improve the processor utilization. Secondly, the hardware subtask fault detection and tolerance issues are researched, and each hardware subtask is configured to 3 slots in FPGA and fault tolerance is realized by TMR technology. Finally, a real-time fault-tolerant algorithm (FT-SSHTNB) is proposed to schedule software/hardware hybrid tasks. The experimental results show that FT-SSHTNB algorithm can tolerate multiple hardware failures and guarantee all real-time task deadlines to be met with low hardware cost.

REFERENCES


Jinyong Yin was born in Shandong, China in September 1978. He received the bachelor’s degree from Department of mechanical engineering, Wuhan University of Science and Technology in 2001. And he received the MS and Ph.D degrees from College of Computer Science and Technology, Harbin Engineering University, China in 2007 and 2010 respectively. Now he is a senior engineer in Jiangsu Automation Research Institute, Lianyungang, China. His current research interests include high performance computing, reconfigurable computing and real-time embedded system.