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

doi:10.4304/jcp.7.11.2773-2780

In the system, hybrid tasks are presented by a set $T=\{T_1,T_2, \dots ,T_n\}$ 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 $ST_i=\{st_{i1}, st_{i2}, \dots, st_{iq}\}$ represents the hardware and software subtasks of T_i, $E_i \in ST_i \times ST_i$ 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 $D_i=P_i$.

$\mathrm{IPred}(st_{ik})=\{st_{ij}|(st_{ij},st_{ik}) \in E_i\}$ represents the immediate predecessor subtasks of the subtask st_{ik}, and if $|\mathrm{IPred}(st_{ik})|=0$, then the st_{ik} is defined as entry subtask .

$\mathrm{ISucc}(st_{ij})=\{st_{ik}|(st_{ij},st_{ik}) \in E_i\}$ represents the immediate successor subtasks of the subtask st_{ij}, and if $|\mathrm{ISucc}(st_{ij})|=0$, then the st_{ij} is defined as exit subtask .

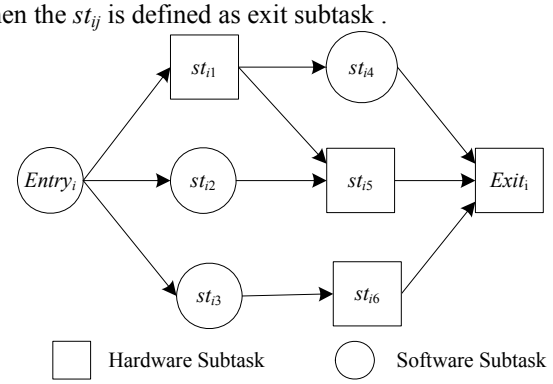
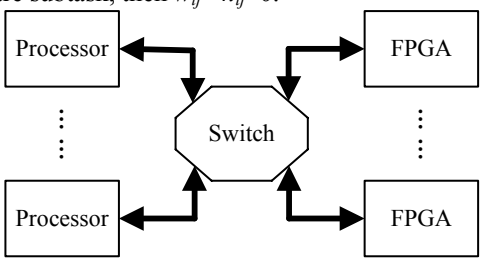


Figure 1. Real-time task architecture.

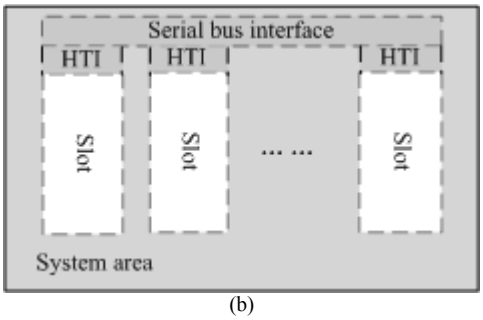
Not lose the general case, assume that each task T_i only has an entry subtask $Entry_i$ and an exit subtask $Exit_i$. Figure 1 shows a task's instance which contains 8 subtasks, where rectangle presents hardware subtask and circle presents software subtask.

Let C_i^{CP} presents the sum execution time of subtasks in task T_i's critical path; Let C_i^{soft} 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 st_{ij}; Let C_{ij}^{bot} presents the longest path from subtask st_{ij} to the exit subtask.

Subtask st_{ij} is presented by a set $st_{ij}=\{C_{ij}, w_{ij}, h_{ij}, type_{ij}\}$, where C_{ij} presents execution time of subtask st_{ij}, 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 st_{ij} is a hardware subtask and S presents that st_{ij} is a software subtask and if st_{ij} is a software subtask, then $w_{ij}= h_{ij}=0$.



(a)



(b)

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+1th 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[17].

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

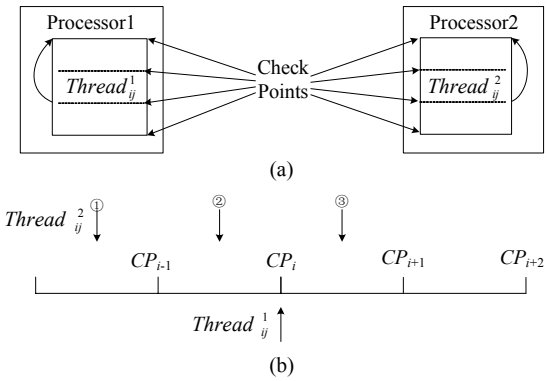


Figure 3. Fault checking principle (a) Software subtask executing model (b) Check point comparison and threads synchronization.

As figure 3(a) shows, each software subtask st_{ij} is correspond to two threads $thread_{ij}^1$and $thread_{ij}^2$ 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 $thread_{ij}^1$, 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 $thread_{ij}^1$ is at the checkpoint CP_i, if the thread $thread_{ij}^2$ is at position ①, then thread $thread_{ij}^1$ and $thread_{ij}^2$ roll back to the checkpoint CP_{i-2}; if the $thread_{ij}^2$ is at position ②, then the thread $thread_{ij}^1$ compares the states of checkpoint CP_{i-1}, and if the states are not identical, then the thread $thread_{ij}^1$ and $thread_{ij}^2$ roll back to the checkpoint CP_{i-2} at the same time, otherwise the thread $thread_{ij}^1$ saves the state of checkpoint CP_i and continues executing; if the thread $thread_{ij}^2$ is at position ③, then the thread $thread_{ij}^1$ saves the state of checkpoint CP_i and continues executing.

Definition 3: When $thread_{ij}^1$ reaches the checkpoint CP_k, $thread_{ij}^2$ will take at least sd_k time to reach checkpoint CP_k, we call sd_k as synchronization distance at checkpoint CP_k .

Definition 4: Given a constant ζ, if the sdk $<\zeta$, threadij1 and threadij2 are synchronized at checkpoint CP_k, otherwise $thread_{ij}^1$ and $thread_{ij}^2$ 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 $ckp = \{id_1, data_1, id_2, data_2\}$, where id_1 and id_2 are the current checkpoint number of thread $thread_{ij}^1$ and $thread_{ij}^2$, $data_1$ and $data_2$ are data saved in the checkpoint. Comparison and synchronization algorithm *CmpSync* is described as follows:

Step1: Determining which thread will execute the algorithm

If the thread is $thread_{ij}^1$, then $thisid = id_1$, $otherid = id_2$;

If the thread is $thread_{ij}^2$, then $thisid = id_2$, $otherid = id_1$;

Step2: Comparing checkpoint number

(1) If $thisid > otherid$, then the restore task T_{re} 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 $thread_{ij}^1$ and $thread_{ij}^2$ 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 $thread_{ij}^1$ and $thread_{ij}^2$ to lose synchronization.

Theorem 1: Threads $thread_{ij}^1$ and $thread_{ij}^2$ will not lose synchronization as executing *CmpSync* algorithm.

Proof: According to Figure 3, when the thread $thread_{ij}^1$ encounters a checkpoint CP_i under normal circumstances, the thread $thread_{ij}^2$ may be at the position ② or ③.When the thread $thread_{ij}^2$ is at position ②, the thread $thread_{ij}^1$ compares state of checkpoint CP_{i-1}; When the thread $thread_{ij}^2$ is at position ③, the thread $thread_{ij}^2$ compares state of checkpoint CP_{i-1}, the thread $thread_{ij}^1$ 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 $thread_{ij}^1$ and $thread_{ij}^2$ will not lose synchronization due to the implementation of algorithm *CmpSync*. □

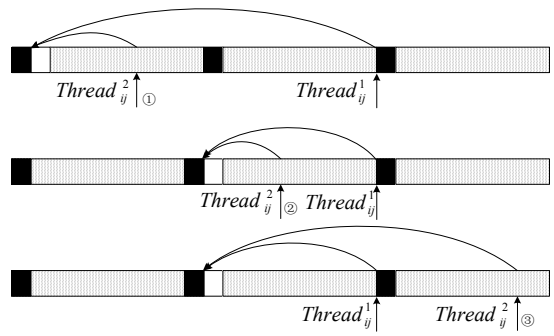


Figure 4. Rollback length when fault occurs in different parts.

As shown in Figure 4, when the thread $thread_{ij}^1$ encounters a checkpoint, the maximum rollback length of task is different according to the location of the thread $thread_{ij}^2$.

Suppose that the detection interval length of subtasks st_{ij} is C_{in}^{ij}, when the thread $thread_{ij}^2$ is at position ①, the task's rollback length is $2C_{in}^{ij} + C_{sa} + C_{re}$; when the thread $thread_{ij}^2$ is at position ②, the task's rollback length is $C_{in}^{ij} + C_{sa} + C_{re}$; when the thread $thread_{ij}^2$ is at position③, the task's rollback length is longer than $C_{in}^{ij} + C_{sa} + C_{re}$ but less than $2C_{in}^{ij} + C_{sa} + C_{re}$. Therefore, when the thread $thread_{ij}^2$ is in position ①, the rollback length is the longest, and denoted as:

$$C_{ij}^R = 2C_{in}^{ij} + C_{sa} + C_{re} \tag{1}$$

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

$$C_{ij}^N = C_{ij} + m_{ij}C_{sa} \tag{2}$$

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

$$C_{ij}^T = C_{ij}^N + C_{ij}^R \tag{3}$$

When r faults occur, the maximum execution time of task T_i's software subtask is:

$$C_i^{soft-r} = \sum C_{ij}^N + \max\left\{\sum_r C_{ij}^R\right\}, \quad \forall st_{ij} \in ST_i \wedge st_{ij}.type = S \tag{4}$$

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_{ij}, checkpoint recovery time C_{re} 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 $m_{ij} = \left\lceil \sqrt{\dfrac{2C_{ij}}{C_{sa}}} + 1 \right\rceil$ or $\left\lfloor \sqrt{\dfrac{2C_{ij}}{C_{sa}}} + 1 \right\rfloor$, the execution time of subtask st_{ij} reaches the minimum.

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

$$C_{ij}^T = C_{ij}^N + C_{ij}^R = m_{ij}C_{sa} + 2\frac{C_{ij}}{m_{ij}-1} + C_{ij} + C_{sa} + C_{re}.$$

The first order differential equation and second order differential equation of C_{ij}^T for m_{ij} are:

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

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 $\dfrac{d^2C_{ij}^T}{dm_{ij}^2} > 0$. When the first order differential equation $\dfrac{dC_{ij}^T}{dm_{ij}} = 0$, that is $m_{ij} = \sqrt{\dfrac{2C_{ij}}{C_{sa}}} + 1$, C_{ij}^T will obtain the minimum value. As m_{ij} is a positive integer, so when $m_{ij} = \left\lceil \sqrt{\dfrac{2C_{ij}}{C_{sa}}} + 1 \right\rceil$ or $\left\lfloor \sqrt{\dfrac{2C_{ij}}{C_{sa}}} + 1 \right\rfloor$, C_{ij}^T 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 (Ti)
{
  // each while circle partition a group of subtasks
  while(Ti has unmarked hardware subtask)
  {
    Group = Φ;
    MaxTask =the unmarked widest subtask of task Ti;
    Mark the MaxTask;
    Group= Group∪ { MaxTask };
    // mark MaxTask's predecessor subtasks
    CurTask= MaxTask;
    while(CurTask has predecessor subtask)
    {
      if CurTask has unmarked predecessor subtask,
      then Pred = CurTask's unmarked widest
      predecessor subtask, otherwise Pred = CurTask's
      first predecessor subtask;
      if Pred is unmarked, then Group=Group∪ {Pred}
      and mark Pred;
       CurTask=Pred;
    }
    // mark MaxTask's successor subtasks
    CurTask= MaxTask;
    while(CurTask has successor subtask)
    {
      if CurTask has unmarked successor subtask, then
      Succ = CurTask's unmarked widest successor
      subtask, otherwise Succ = CurTask's first
      successor subtask;
      if Succ is unmarked, then Group=Group∪ {Succ}
      and mark Succ;
```

```
        CurTask=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} are 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}'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 $\sum_{i=1}^{k} \lceil D_k / P_i \rceil C_i^{soft} \le mD_k - (m-1)C_k^{CP}$, 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 $T = \{T_1, T_2,\ldots,T_n\}$, if each task $T_i \in T$ satisfies the inequation(5), then tasks set T can be *f-fault-tolerant* scheduled by FT-SSHTNB algorithm on m processors.

$$\sum_{i=1}^{k-1} \left\lceil \frac{D_k}{P_i} \right\rceil C_i^{soft} + \max\left\{ \sum_{f-r} C_{ij}^R \right\} + C_k^{soft-r}$$

$$\le gD_k - (g-1)\left(C_k^{CP} + \max\left\{ \sum_{r} C_{kj}^R \right\} \right), \qquad (5)$$

$$g = m - f, r = 0, \cdots, f$$

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-r 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 $\max\left\{ \sum_{f-r} C_{ij}^R \right\}$ and the demand of task T_k is C_k^{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 $\max\left\{ \sum_{r} C_{kj}^R \right\}$, the length of task T_k's critical path is at most $C_k^{CP} + \max\left\{ \sum_{r} C_{kj}^R \right\}$ when processors fail r times during the execution of task T_k.

According to theorem 4-1 [17], if the inequation (5) holds, the tasks set T can be scheduled by FT-SSHTNB algorithm on m-f processors. □

D. FT-SSHTNB Algorithm Description

The SSHTNB algorithm [17] can schedule software/hardware hybrid real-time tasks without fault-tolerant requirement. Based on this, rollback/recovery mechanism is introduced to FT-SSHTNB algorithm to schedule tasks with multiple software and hardware faults. The following part only describes the content related to fault tolerance in FT-SSHTNB and the other content is similar to SSHTNB in paper [17].

FT-SSHTNB algorithm includes two parts: static algorithm and dynamic algorithm. In static algorithm, each real-time task T_i's fault-tolerant number k_i is determined and schedulability of tasks is tested according theorem 3. Dynamic algorithm includes following parts:

Scheduling hardware subtasks:

(1) A thread $thread_{ij}^k$ is created for each subtask st_{ij}^k(k = 1,2,3) , and the thread is responsible for passing parameters from processor to the hardware subtask st_{ij}^k and reading the execution results from subtask st_{ij}^k to processor.

(2) If two subtasks in st_{ij}^k (k = 1, 2, 3) are completed, then the execution results should be compared. If the execution results are consistent, the results as the final results are accepted and the 3rd subtask is canceled; otherwise three execution results are compared.

Scheduling software subtasks:

(1) If a processor start executing the $thread_{ij}^1$, then the other processor is notified to start executing $thread_{ij}^2$.

(2) Suppose that $thread_{ij}^1$ and $thread_{ij}^2$ are executing on processor Pro_1 and Pro_2 respectively and $thread_{im}^1$ and $thread_{im}^2$ are executing on processor Pro_3 and Pro_4 respectively. If the states of $thread_{ij}^1$ and $thread_{ij}^2$ are not consistent or $thread_{ij}^1$ and $thread_{ij}^2$ lose synchronization at checkpoint, $thread_{ij}^1$ and $thread_{ij}^2$ will be rolled back to the pervious checkpoint and $thread_{ij}^1$ and $thread_{im}^1$ are scheduled to Pro_3 and Pro_1. The states will be compared again at checkpoint, if the states of $thread_{ij}^1$ and $thread_{ij}^2$ are not consistent or $thread_{ij}^1$ and $thread_{ij}^2$ lose synchronization, then the processor Pro_2 fails, if the states of $thread_{im}^1$ and $thread_{im}^2$ are not consistent or $thread_{im}^1$ and $thread_{im}^2$ lose synchronization, then the processor Pro_1 fails, otherwise the subtask st_{ij} fails.

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_i is the probability $P(T_i)$ that task T_i 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 Δt 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_i 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 λ, and then the probability of the computing unit failing during task T_i executing is:

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

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

Faults number *f*	**$P(T_k)$**				
	P_k=100	**P_k =200**	**P_k =300**	**P_k =400**	**P_k =500**
0	0.951229	0.904837	0.860709	0.818730	0.778800
1	0.998814	0.995411	0.990009	0.982804	0.973987
2	0.999981	0.999854	0.999527	0.998915	0.997957
3	0.999999	0.999996	0.999985	0.999948	0.999880

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.

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.

IV. EXPERIMENTAL RESULTS AND ANALYSIS

TABLE II.
CAPABILITY COMPARISON OF FT-SSHTNB WITH RELATED ALGORITHMS

Items \ **Algorithms**	**FT-RMFF**	**HTFS**	**Liu**	**DABCBF**	**FT-SSHTNB**
Independent tasks	√	√	√	√	√
Precedence Constraint tasks	×	×	×	×	√
S/H hybrid tasks	×	×	×	×	√
One processor fault	√	√	√	√	√
Multi processor faults	×	×	×	×	√

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

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

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, 0.5Pi] 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 (%)

Fault numbers	Processor numbers *m*			
	8	16	32	64
0	39.820220	38.293705	36.966333	35.613902
1	36.184924	35.493393	36.533091	35.420144
2	30.142350	33.961544	34.924395	34.499440
3	26.073210	30.281818	33.183477	34.133673
4	20.432851	28.085570	32.150187	33.606462
5	15.167190	26.550720	30.587747	32.861652

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

[1] CH. Chapman, B. Dufoet, "Using laser defect avoidance to build large-area FPGAs," IEEE Design & Test of Computers, 1998, 15(4): 75-81.

[2] Doumar A, Kaneko S, Ito H, "Defect and fault tolerance FPGAs by shifting the configuration data," Proceedings of the 14th International Symposium on Defect and Fault-Tolerance in VLSI Systems, Albuquerque, NM, 1999: 377-385.

[3] Hanchek F, Dutt S, "Methodologies for tolerating cell and interconnect faults in FPGAs," IEEE Transactions on Computers, 1998, 47(9):15-33

[4] Abramovici M, Stroud C, Wijesuriya S, et al., "Using Roving STARs for on-line testing and diagnosis of FPGAs in fault-tolerant applications," IEEE International Test Conference, Atlantic City, NJ, 1999: 973-982.

[5] Emmert J, Stround C, Skaggs B, et al., "Dynamic fault tolerance in FPGAs via partial reconfiguration," IEEE Symposium on Field-Programmable Custom Computing Machines, Napa Valley, CA, 2000: 165-174.

[6] Bertossi A, Mancini LV, Rossini F, "Fault-tolerant rate-monotonic first-fit scheduling in hard real-time systems," IEEE Transactions on Parallel and Distributed Systems. l999, 10(9) 934- 945.

[7] QIN Xiao, HAN Zong-Fen, PANG Li-Ping, "Real-Time Scheduling with Fault-Tolerance in Heterogeneous Distributed Systems," Chinese Journal of Computers, 2002,25(1): 49-56

[8] LIU Huai, FEI Shu-Min, "A Fault-Tolerant Scheduling Algorithm Based on EDF for Distributed Control Systems. Journal of Software," 2003,14(8): 1371-1378

[9] Al-Omari R, Somani AK, Manimaran G, "A new fault-tolerant technique for improving schedulability in multiprocessor real-time systems," Proceedings of the 15th IEEE Parallel and Distributed Processing Symposium, 2001: 32-33

[10] ZHANG Yong-Jun,ZHANGYi, PENGYu-Xing, et al., "A Multiprocessor-Based Fault-Tolerant Real-Time Task Scheduling Algorithm," Computer Research and Development, 2000,37(4): 425-429

[11] Yang CH, Deconinck G, "A fault-tolerant reservation-based strategy for scheduling aperiodic tasks in multiprocessor systems," Proceedings of the l0th IEEE Euromicro Workshop on Parallel, Distributed and Network-based Processing, 2002: 3l9-326.

[12] YANG Chun-Hua, GUI Wei-Hua, JI Li, "A Fault-Tolerant Scheduling Algorithm of Hybrid Real-Time Tasks Based on Multiprocessors," Chinese Journal of Computers, 2003, 26(11): 1480-1486

[13] Luo Wei, Yang Fumin, Pang Liping, et al., "A Real-Time Fault-Tolerant Scheduling Algorithm for Distributed Systems Based Deferred Active Backup-Copy," Journal of Computer Research and Development, 2007, 44(3):521-528

[14] Luo Wei, Yang Fumin, Pang Liping, et al., "A Real-Time Fault-Tolerant Scheduling Algorithm of Periodic Tasks in Heterogeneous Distributed Systems," Chinese Journal of Computers, 2007, 30(10): 1740-1749

[15] WU Jun, "Fault-tolerant scheduling algorithm for heterogeneous distributed control systems based on dual priorities queues," Journal of Southeast University (Natural Science Edition), 2008, 38(3):407-412

[16] Punnekkat S, Burns B, Davis R, "Analysis of checkpointing for real-time systems," Real-Time Systems, 2001, 20(1):83-102

[17] Yin Jinyong, "Research on Real-time Task Scheduling Algorithm of Reconfigurable System," PhD dissertation, College of Computer Science & Technology，Harbin Engineering University，June 2010

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.