Efficient Scheduling Algorithm with Fault-tolerance for Real-time Tasks in Distributed Systems

Qin Xiao, Pang Liping, Li Shengli and Han Zongfen
Department of Computer Science, Huazhong University of Science and Technology
Wuhan, China, Post Code 430074

ABSTRACT

As real-time fault-tolerant scheduling is one of the main research areas in real-time fault-tolerant techniques, this paper proposes an efficient scheduling algorithm for BKCL(EBKCL). EBKCL can schedule the tasks with the fault-tolerant requirements(FTR) together with tasks without FTR. It is assumed in BKCL that there are no overlaps between the backup copies, however, the backup copies are allowed to be overlapped in EBKCL. The simulation experiment shows that the performance of EBKCL is better than that of BKCL. In order to enhance the performance of the BKCL, a dynamic scheduling algorithm is proposed, the performance simulation and analysis of DSA are also given in this paper.

KEY WORDS

Fault-tolerant, real-time task, efficient scheduling, performance analysis, heuristics, distributed systems

1 INTRODUCTION

Many hard real-time systems must finish before deadlines despite the presence of hardware and software failures[1][2][3]. These hard real-time systems are mission-critical systems, which have two characters, one is guaranteeing real-time applications to meet timing constraints and another is continuing the specified operations even hardware or software has errors.

Many fault-tolerant mechanisms have been invested to make computer system highly reliable. Three error detection mechanisms, using physical fault-injection, are presented in [4]. Voting in distributed systems and optimal rollback recovery, two efficient techniques of fault-tolerant, are studied in [5] and [6] respectively. However, these works do not take real-time into consideration. Also, there are many papers deal with real-time scheduling[7][8][9]. Some algorithms are preemptive and others are non-preemptive, some are static while some are dynamic. Many of these real-time scheduling algorithms assume that no error occurs in real-time systems.

Fault-tolerant and real-time techniques must be studied simultaneously[10][11]. Paper [12] presents extended recovery blocks technique for real-time applications, and scheduling algorithms for imprecise computations are studied in paper [13]. Real-time scheduling with fault-tolerance is one of the important techniques used to achieve fault-tolerance for mission-critical real-time systems. Paper [14] proposes a dynamic algorithm to ensure that backup schedules can be efficiently embedded within the original primary schedule, the shortcoming of this algorithm is that it is based on the solution to two NP-Complete problems. Paper [15] studies a fault-tolerant algorithm to dynamically distribute the workload of a failed processor to other operable processors. Several special cases of the real-time fault-tolerant scheduling problem have been explored in [16][17]. Paper [18] presents a real-time scheduling algorithm that can schedule real-time tasks with fault-tolerant requirements together with those without the requirements of fault-tolerance. In [18], it is assumed that all backup copies are not allowed to be overlapped. In this paper, such assumption is relaxed and an efficient fault-tolerant scheduling algorithm is explored. The paper is organized as follows. In Section 2, the model of scheduling for real-time tasks and two lemmas are presented. An efficient algorithm for fault-tolerant scheduling is proposed in Section 3. The performance analysis is presented in Section 4. In Section 5, we summarize the contributions and the open problems of this work.

2 SYSTEM MODEL

In many real-time distributed applications, there are hard real-time tasks and soft real-time tasks. Most of the hard real-time tasks have fault-tolerant requirements, and soft real-time tasks, that is not mission critical, usually have no fault-tolerant requirement. Assume that all the tasks are independent to each other and non-preemptive. The system model is given as follows:

Definition 1: A set of periodic real-time tasks is defined as a pair: $\Psi_{pd} = (\Psi_{ft}, \Psi_{nf})$, where Ψ_{ft} contains all the real-time tasks with fault-tolerant requirements, and Ψ_{nf} real-time tasks in need no requirement of fault-tolerant.

Definition 2: $\Psi_{ft} = \{t_1, t_2, ... ,t_n\}$, where $t \in \Psi_{ft}$ is a tuple, t = (p, d, tp, tb), where p and d are the period and the deadline respectively, tp and tb are the primary copy and the backup copy of the task; tp and tb are characterized by a tuple, tp = tb = (c, s, pcs), where c and s are computation time and start time of task tp and tb respectively, pcs indicates to which processor task tp, tb are scheduled.

Definition 3: $\Psi_{nf} = \{t_{n+1}, t_{n+2}, ... ,t_{n+m}\}$, where $t \in \Psi_{nf}$ is a tuple, t = (p, d, c, s, pcs), where p, d, c and s are the period, the deadline, the computation time, and start time of task t respectively, pcs indicates that the task t is scheduled to processor pcs.

Definition 4: An aperiodic real-time task is defined as a tuple, apt = (st, a, c), where st is the status of task, st = WAIT | SERVE. a is the arrival tim and c is computation time.

Definition 5: Ψ_{ap} is a set of aperiodic real-time tasks, $\Psi_{ap} = \Psi_{wt} \cup \Psi_{sv}$, where Ψ_{wt} is a waiting set, Ψ_{sv} is a serving set, $\Psi_{wt} = \{ apt \mid apt.st = WAIT, apt \in \Psi_{ap} \}$, $\Psi_{sv} = \{ apt \mid apt.st = SERVE, apt \in \Psi_{ap} \}$.

Definition 6: Assume there are k processors, the set of processors is defined as: $\Omega = \{P_1, P_2, ..., P_k\}$, each P in Ω is a tuple, $P = (\Delta, \xi_s, \xi_p, \varpi, \tau)$, where Δ is a set of tasks which are allocated to this processor, and ξ_s is the scheduling length of this processor, ξ_p is the scheduling length of the primary copies and tasks without FTR, ϖ is the status of the process, τ is an aperiodic task allocated to this processor. If $\tau = \varepsilon$, no aperiodic task is allocated. ϖ = FR | PT | AP, FR indicates that process is free, PT means processor is for periodic real-time task, and AP means that processor is used by aperiodic task.

Property 1: Assume that all the tasks have the same deadline and the deadline coincides with the period, thus $\forall t_i, t_j \in (\Psi_{ft} \cup \Psi_{nf})(t_i \neq t_j) \rightarrow (t_i.d = t_j.d = t_i.p = t_j.p = DL)$. In order to tolerant one processor failure, the computation time of tasks in Ψ_{ft} must be less than or equal to half of the period, formally described as: $\forall t \in \Psi_{ft}$ (t.tp.c + t.tb.c $\leq$ t.p).

Property 2: One processor failure is tolerant, if and only if all backup copy of each task in Ψ_{ft} is schedule on different processor and there is no overlapping in time between their executions, thus: (One processor failure is tolerated) $\leftrightarrow \forall t \in \Psi_{ft}$ (t.tp.pcs $\neq$ t.tb.pcs) $\wedge$ (t.tp.s + t.tp.c $\leq$ t.tb.s).

Property3: Set Ψ_{wt} and set Ψ_{sv} have no intersection, $\Psi_{wt} \cap \Psi_{sv} = \varnothing$.

Property 4: If the status of processor P is AP, then the aperiodic task which served by P must in set Ψ_{sv}, thus: $\forall P \in \Omega$ (P.st = AP $\rightarrow$ P.$\tau \in \Psi_{sv}$).

Property 5: If the status of processor P is FP, then no aperiodic task is served by P, thus: $\forall P \in \Omega$ (P.st = FR $\rightarrow$ P.$\tau = \varepsilon$).

Property 6: Two different processors can not serve the same aperiodic task. In formal way, it is: $\forall P_i, P_j \in \Omega$ ($P_i.\tau \neq \varepsilon, P_j.\tau \neq \varepsilon, P_i \neq P_j \rightarrow P_i.\tau \neq P_j.\tau$).

The model assumes that, for each task in Ψ_{ft}, the computation times of its primary copy and its backup copy are exactly the same, in other words, $\forall t \in \Psi_{ft}$ (t.tp.c = t.tb.c). As a results, the definition of tasks in Ψ_{ft} becomes to a tuple, t = (p, d, c, tp, tb). tp and tb are primary copy and backup copy of the task. tp and tb are defined by a tuple, tp = tb = (s, pcs), where s and pcs is the same as Definition 2.

3 FAULT-TOLERANT SCHEDULING

In this section, an efficient fault-tolerant scheduling algorithm, based on the assumption that there are overlaps between backup copies, will be proposed first. Like algorithm BKCL[18], efficient BKCL(EBKCL) first sorts the primary copies of the tasks in set Ψ_{ft} and the tasks in set $\Psi_{nf.}$ Then, it sorts and schedules the backup copies, allowing the backup copies to be overlapped with each other. When scheduling backup copies, EBKCL starts by scheduling the backup copies of the tasks that are scheduled to execute on processor 1, then schedule the backup copies of the tasks that are allocated to processor 2, and so on. Before the scheduling of backup copies, the scheduling length of each processor ($P_i.\xi_s$) is saved into a buffer ($PB_i.\xi$), and before the scheduling of backup copies for each processor, the scheduling length of each processor will be restored from the buffer. So, the backup copies are allowed to be overlapped in this way.

Algorithm EBKCL (Input: Task set Ψ, k; Output: sch_success, Ω)

Step1 FOR i = 1 TO k DO $P_i.\Delta = \varnothing$; $P_i.\xi_s = 0$;

Step2 IF $\forall t_i \in \Psi_{ft}, \forall t_j \in \Psi_{nf} (\sum_{i=1}^{n} t_i.c + \sum_{j=1}^{m} t_j.c \geq k * DL)$ or $\exists \forall t \in \Psi_{ft}$(t.c > DL/2)

THEN sch_success = fail, exit; ELSE goto step 3;

Step3 Sort the tasks in set Ψ_{ft} and the tasks in set Ψ_{nf}, and rename as $t_1, t_2, ..., t_{n+m}$;

Step4 FOR i = 1 TO n + m DO

4.1 Select P_j which satisfies: $\forall P \in \Omega$ ($P_j.\xi_s \leq P.\xi_s$);

4.2 IF $t_i \in \Psi_{ft}$ THEN $t_i.tp.pcs = j$; $t_i.tp.s = P_j.\xi_s$; $P_j.\Delta = P_j.\Delta + \{t_i.tp\}$;

ELSE $t_i.pcs = j$; $t_i.s = P_j.\xi_s$; $P_j.\Delta = P_j.\Delta + \{t_i\}$;

4.3; $P_j.\xi_s = P_j.\xi_s + t_i.c$;

Step5 IF $\exists P \in \Omega$ (P.ξ > DL) THEN sch_success = fail; exit; ELSE goto step6;
Step6 FOR i = 1 TO n DO $PB_i.\xi = P_i.\xi_s$; /* Save the schedule length of each processor */
Step7 FOR i = 1 TO n DO /* Schedule backup copies */
7.1 FOR j = 1 TO n DO $P_j.\xi_s = PB_j.\xi$; /* Restore the schedule length of each processor */
7.2 Let $y_1, y_2, ..., y_{ki}$ be the k_i tasks in Ψ_{ft} on processor i; FOR j = 1 TO k_i DO
7.2.1 Select P_v which satisfies: $\forall P \in (\Omega - \{P_i\})$ ($P_v \neq P_i \wedge P_v.\xi \leq P.\xi$);
7.2.2 max_len = MAX(y_j.tp.s + y_i.c, $P_v.\xi$);
7.2.3 y_j.tb.pcs = v; y_j.tb.s = max_len; $P_v.\xi$ = max_len + y_j.c; $P_v.\Delta = P_v.\Delta + \{y_j.tb\}$;
7.2.4 IF $P_v.\xi_s$ > DL THEN sch_success = fail; exit;
Step8 sch_success = Success;

The time complexity of EBKCL is $O[(n+m)(\log(n+m)+\log k)]$, where n and m are the numbers of periodic real-time tasks in set Ψ_{ft} and Ψ_{nf} respectively, k is the number of processors. The sorting of tasks in set Ψ_{ft} and the tasks in set Ψ_{nf} takes $O((n+m)\log(n+m))$ time, and scheduling these tasks takes $O((n+m)\log k)$ time. Scheduling the backup copies of the tasks in set Ψ_{ft} takes $O(n\log(k-1))$ time. It is obvious that the time complexity of EBKCL is less than BKCL, because the algorithm EBKCL uses less time to schedule backup copies than that of algorithm BKCL.

The scheduling process of EBKCL is illustrated by an example. Assume the set of processors is: $\Omega = \{P_1, P_2, P_3, P_4\}$, two sets of periodic real-time tasks are: $\Psi_{ft} = \{t_1, t_2, t_3, ..., t_6\}$, $\Psi_{nf} = \{t_7, t_8, t_9, t_{10}\}$, $\{t_i.c \mid i = 1, 2, ..., 6\} = \{12, 10, 9, 6, 5, 3\}$, $\{t_i.c \mid i = 7, 8, 9, 10\} = \{11, 7, 4, 2\}$, DL = 32. EBKCL schedules 16 tasks including the backup copies of the tasks in Ψ_{ft} on four processors. Here all the tasks are defined simply as a tuple: t = (c, s, pcs), where c is the computation time, s is starting time and pcs is the index of the processor respectively. The scheduling result is illustrated formally as follows:
$P_1.\Delta = \{t_1.tp, t_9, t_6.tp\} = \{(12, 0, 1), (4, 12, 1), (3, 16, 1)\}$;
$P_2.\Delta = \{t_7, t_5.tp, t_{10}, t_4.tb\} = \{(11, 0, 2), (5, 11, 2), (2, 16, 2), (6, 18, 2)\}$;
$P_3.\Delta = \{t_2.tp, t_4.tp, t_1.tb, t_3.tb, t_5.tb\} = \{(10, 0, 3), (6, 10, 3), (12, 16, 3), (9, 16, 3), (5,16, 3)\}$;
$P_4.\Delta = \{t_3.tp, t_8, t_2.tb, t_6.tb\} = \{(9, 0, 4), (7, 9, 4), (10, 16, 4), (3, 19, 4)\}$.

The second way to enhance the performance of algorithm BKCL is scheduling aperiodic tasks with periodic real-time tasks. In BKCL, if the primary copy of a task is complete successfully, its backup copy will not run in the system, since many experiments proved that, in most situations, primary copies will not fail. In this case, time in the interval [$P.\xi_p$, DL], which is booked by backup copies will be waste. A dynamic scheduling is designed here to schedule aperiodic tasks in the interval [$P.\xi_p$, DL]. So, running aperiodic tasks when backup copies need not to execute will enhance processor utilization. If one primary copy has errors during its execution, its backup copy can preempt aperiodic task that is using the processor. Dynamic scheduling contains three parts, insert, dispose and process. Insert algorithm deal with a newly arrival task, if there is a processor in free status, this processor could start serving aperiodic task right now, otherwise put this task into set Ψ_{wt}. Insert algorithm is described as follows:

INSERT（Input: aperiodic task apt, Ψ_{wt}, Ψ_{sv}, Ω; Output: Ψ_{wt}, Ψ_{sv}, Ω）
Step 1 IF ($\forall P \in \Omega$ ($P.\tau \neq \varepsilon$)) THEN apt.st = WAIT; put apt into waiting setΨ_{wt}; goto step (6);
Step 2 FOR i = 1 to k DO IF ($P_i.\tau = \varepsilon$) THEN do (3); /* Find a proper processor to serve aperiodic task */
Step 3 $P_i.\tau$ = apt; apt.st = SERVE;
Step 4 Put apt into waiting setΨ_{sv};
Step 5 IF (P.st = FR) THEN P.st = AP; Start execute apt;
Step 6 End of INSERT;

When the system complete an aperiodic task, dispose algorithm will be invoked to delete this task from set Ψ_{sv}. If there is no task in set Ψ_{wt}, then the status of the processor should be modified, otherwise select a task from set Ψ_{wt} according to first come first serve strategy, and allocates this task to the processor.

DISPOSE（Input: Processor P）
Step 1 Delete task $P.\tau$ from serving set Ψ_{sv};
Step 2 IF ($\Psi_{wt}=\varnothing$) THEN DO P.st = FR; $P.\tau = \varepsilon$; goto (6);
Step 3 Select apt $\in \Psi_{wt}$, which satisfies: $\forall apt' \in \Psi_{wt}$ (apt.a $\leq$ apt′.a); /* According to first come first serve */
Step 4 apt.st = SERVE; Move apt from Ψ_{wt} into Ψ_{sv};
$P.\tau$ = apt;
Step 5 IF (P.st = AP) THEN Start execute apt;
Step 6 End of DISPOSE;

The third part of the dynamic scheduling algorithm is used to manage the status of the processors, and decide which task has right to use processor. It is given as follows:

DYNSCH（Input: Ω，current time cur_t; Output: Ω）
Step 1 t = cur_t MOD PR;
Step 2 FOR i = 1 TO k DO
2.1 IF (t = 0) THEN DO P_i.st = PT; Start serving periodic real-time tasks in set $P_i.\Delta$;
2.2 IF (t = $P_i.\xi_p$) THEN goto (2.3);

2.3 IF ($P_i.\tau \neq \varepsilon$) THEN P_i.st = AP;
Start serving aperiodic task $P_i.\tau$;
ELSE P_i.st = FR;
Step 3 End of DYNSCH;

4 SIMULATION AND DISCUSSION

Given sets of periodic real-time tasks and the number of processor (k), BKCL and EBKCL determine whether k processors can complete these tasks before the deadline. An algorithm, which is called Find Minimum Number of Processors(FMNP) is presented to find the minimum number of processors (k) on which all tasks in Ψ_{pd} can be scheduled to finish before the specified deadlines. The input parameter Sch_Alg indicates which algorithm FMNP should invoke to find the minimum number of processors. FMNP is described as follows:

FMNP (Input: Task set Ψ, Sch_Alg; Output: k, Ω)
Step1 Lower = $\forall t_i \in \Psi_{ft}, \forall t_j \in \Psi_{nf}$
$(\lfloor (\sum_{i=1}^{n} t_i.c + \sum_{j=1}^{m} t_j.c) / DL \rfloor)$; Upper = n + m;
Step2 k = $\lfloor$ (Lower + Upper) / 2 $\rfloor$; IF (Lower = k)
THEN m = m + 1; exit;
Step3 IF (Sch_Alg = BKCL)
THEN BKCL(Ψ, k; sch_success, Ω);
ELSE EBKCL(Ψ, k; sch_success, Ω)
Step4 IF (sch_success = success) THEN Upper = k;
ELSE Lower = k; goto step 2;

Performance evaluation for BKCL and EBKC is presented firstly. The simulation experiments are carried out in steps as follows. A deadline DL is selected; then a range of values CR for the computation time of tasks is chosen. The computation times of the tasks in Ψ_{pd} are generated randomly, according to the uniform distribution. Finally, the algorithm FMNP is running, applying BKCL and EBKCL respectively. The number of processors computed by BKCL and by EBKCL will be compared. Here, the ratio between the deadline and the range of values CR is defined as: R = DL / CR. We determine the parameter FTP that denotes the ratio between the number of the tasks in Ψ_{ft} and the number of tasks in Ψ_{pd}. FTP = $N(\Psi_{ft}) / N(\Psi_{pd}) = N(\Psi_{ft}) / (N(\Psi_{ft}) + N(\Psi_{nf}))$, where the function $N(\Psi)$ is used to find the number of tasks in set Ψ.

(1) Experiment 1

This section analysis the comparison of BKCL and EBKCL. The parameters in the simulation are R= 4, DL= 160 and FTP is set to be 80% and 50% respectively. There are 1000 sets of task (Ψ_{pd}) generated independently to run on algorithm FMNP. When FTP = 50%, the performance of EBKCL are quite similar as the performance of it when FTP = 80%, so the date for EBKCL on the condition that FTP = 50% is omitted from figure 1. It is obvious that MNP increases with the rise of the number of tasks. Because the more tasks in the system, the more processors need to use to complete the tasks before the timing constraint. It is also concluded from figure 3 that the number of processors created by EBKCL is less than the one created by BKCL. Simulaton has proven that algorithm EBKCL significantly improve the performance of algorithm FMNP.

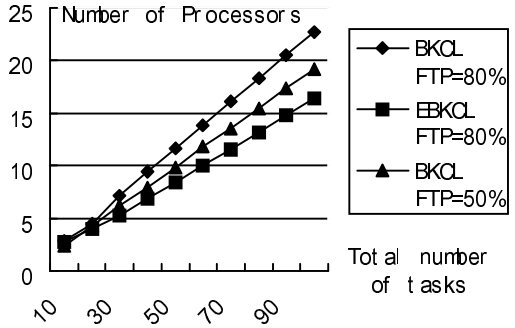

Fig. 1 Relation between the number of processors and the number of tasks

(2) Experiment 2

Only when R and FTP are high enough does the difference between BKCL and EBKCL become significant. This conclusion is showed by two sets of experiment data. Table 1 shows that when R is small, EBKCL only enhance the performance of algorithm FMNP a little bit. Table 2 proves that when FTP is small, the difference between the performance of BKCL and EBKCL is also insignificant.

Table 1 Comparison between BKCL and EBKCL. (DL = 210, R = 2, FTP = 80%)

N(Ψ)	10	30	50	70	90
BKCL	5.3	14.5	23.9	33.3	42.7
EBKCL	4.7	13.8	23.1	32.5	41.9

Table 2 Comparison between BKCL and EBKCL. (DL = 210, R = 10, FTP = 10%)

N(Ψ)	10	30	50	70	90
BKCL	1.0	2.0	3.1	4.2	5.2
EBKCL	1.0	2.0	3.0	4.1	5.1

(3) Experiment 3

This experiment concern about the performance of

dynamic scheduling. The average respond time(ART) of aperiodic tasks can illustrate the performance of dynamic scheduling. The relationship between ART and the period of tasks in Ψ_{pd} is displayed in figure 2. k and CR are set to be 4 and 30. Value ξ_p for P_1, P_2, P_3 and P_4 are 40, 50 60 and 70 respectively. Under the same overload, the longer the period is, the smaller the ART is. Because when value ξ_p for each processor is given, the longer the period is, the more time will be left for aperiodic tasks, so APT becomes smaller.

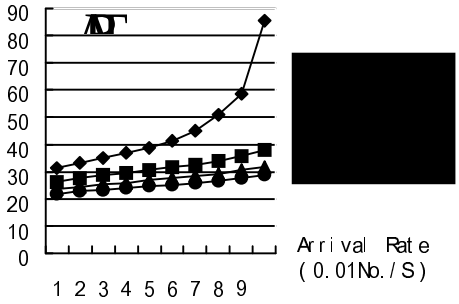

Fig. 2 Relation between period and ART, k = 4, CR = 30

5 CONCLUSION

This paper presents an efficient algorithm for BKCL and calculates time complexity of EBKCL. The simulation experiments show that the performance of EBKCL is better than that of BKCL, especially when R and FTP are high. Based on BKCL, this paper also studies a dynamic scheduling which could schedule aperiodic tasks when processors have no periodic real-time tasks to process. This dynamic scheduling algorithm can enhance the utilization of processors. Performance evaluation for this dynamic scheduling is given.

As the aperiodic tasks scheduled by dynamic scheduling algorithm are non-real-time, our next work is trying to study another algorithm that can scheduling aperiodic real-time tasks dynamically. The future research also includes extending EBKCL to schedule tasks with different deadlines.

REFERENCES

[1] Kieckhafer, R. M., C. J. Walter, A. M. Finn, and P. M. Thambidurai. The MAFT Architecture for distributed fault tolerance. IEEE Transactions on Computers, 1988, 37(4) : 398-405.

[2] G. T. Wilfong. Guest Editor's Introduction: Robotics and Automation. IEEE Computer, 1989, 22(3): 6-7.

[3] M. Factor. Real-Time Data Fusion in the Intensive Care Unit. IEEE Computer, 1991, 24(11) : 45-54.

[4] G.Miremadi and J.Torin. Evaluating Processor Behavior and Three Error Detection Mechanisms Using Physical Fault-Injection. IEEE Transactions on Reliability, 1995, 44(3) : 441-453.

[5] Xu. Lihao and Bruck J. Deterministic Voting in distributed Systems Using Error-Correcting Codes. IEEE Trans. on Parallel and Distributed Systems, 1998, 9(8) : 813-824.

[6] Lin T.H. and Shin K.G. Damage Assessment for Optimal Rollback Recovery. IEEE Trans. on Computers, 1998, 47(5) : 603-613.

[7] Chang G. L., Joosun H., Yang M. S. et. al. Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling. IEEE Trans. on Computers, 1998, 47(6) : 700-713.

[8] G. Manimaran and C. Siva Ram Murthy. An Efficient Dynamic Scheduling Algorithms for Multiprocessor Real-Time Systems. IEEE Trans. On Parallel and Distributed Systems, 1998, 9(3) : 312-319.

[9] M. H. Klein, J. P. Lehoczky and R. Rajkumar. Rate Monotonic Analysis for Real-Time Industrial Computing. IEEE Computer, 27(1): 24-33, Jan. 1994.

[10]A.Avizienis and J.Laprie. eds, Dependable Computing for Critical Applications. 1991, Springer Verlag.

[11]Shin, K.G., G.Koob, and F.Jahanian. Fault-tolerant in real-time systems. IEEE Real-time Systems Newsletter, 1991, 7(3) : 28-34.

[12]K.H.Kim and H.O.Welch. Distributed execution of recovery blocks: An approach for uniform treatment of hardware and software faults in real-time application. IEEE Transactions on Computers, 1989, 38(5) : 626-636.

[13]W.K.Shih, J.W.Liu, and J.Y.Chung. Fast algorithms for scheduling imprecise computations. Real-Time System Symposium, 1989, 12-19.

[14]C. M. Krishna and K. G. Shin. On scheduling tasks with a quick recovery from failure. IEEE Transactions on Computers, 1986, 35(4) : 448-454.

[15]S.Balaji, L.Jenkins, L.M.Patnaik, and P.S.Goel. Workload redistribution for fault-tolerant in a hard real-time distributed computing system. FTCS, Chicago, Illinois, 1989, 19 : 366-373.

[15]Y.Oh and S.H.Son. Multiprocessor support for real-time fault-tolerant scheduling. IEEE Workshop on Architecture Aspects of Real-Time System, San Antonio, Texas, 1991, 76-80.

[17]Y.Oh and S.H.Son. An algorithm for real-time fault-tolerant scheduling in multiprocessor systems. 4th Euromicro Workshop on Real-Time Systems, Athens, Greece, 1992, 190-195.

[18]Han Zongfen and QinXiao. Fault-tolerant scheduling algorithm for real-time tasks in distributed systems.

Journal of Huazhong University of Science and Technology, 1999, 27(4) : 12-14. (in Chinese)