An Effective Dynamic Task Scheduling Algorithm for Real-time Heterogeneous Multiprocessor Systems

*Wen-Pin Liu, Yi-Hsuan Lee, and Cheng Chen**

Department of Computer Science and Information Engineering
National Chiao-Tung University
1001 Ta Hsueh Road, Hsinchu, Taiwan, 300, Republic of China
E-mail: {wpliu, yslee, cchen}@csie.nctu.edu.tw

Abstract

Real-time systems require both functionally correct executions and the results that are produced in time. Fault-tolerance is an important requirement of such systems, due to the catastrophic consequences of not tolerating faults. In this paper, we propose an effective Load-driven Adaptive Scheduling Algorithm (LASA) to dynamically schedule real-time tasks with fault-tolerance used in heterogeneous multiprocessor systems. In LASA, it has an adaptive mechanism to monitor the processor utilization and determine the number of backup copies been scheduled. Besides, we also design a task deferment mechanism to improve the utilization of reclaimed computing resources from redundant copies. According to our simulations, LASA makes a trade-off between the guarantee ratio and the reliability of fault-tolerance, and obviously outperforms other related methods.

Keywords: *Heterogeneous multiprocessor, Real-time, Task scheduling, Adaptive, Fault-tolerant*

1 Introduction

Real-time systems are defined as those systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced [1]. In such systems, the real-time task scheduling can be performed either statically or dynamically. Since there does not exist an optimal scheduling algorithm for dynamically arrival tasks, many heuristic approaches have been evolved [1, 4, 8, 11-12].

Multiprocessor systems have emerged as a powerful computing means because of their capability for high performance and reliability for the real-time applications [8, 13-14]. Due to the nature of real-time tasks, several techniques have evolved for fault-tolerant scheduling [2, 4, 10]. In multiprocessor systems, fault-tolerance can be provided by scheduling multiple versions of tasks on different processors. Among different schemes for fault-tolerant scheduling, we choose the *Primary/Backup* (*PB*) scheme which is the most popular one.

In recent years, the adaptation mechanism opens up many avenues for further research in the dynamic scheduling problem [14]. The concept of adaptation mechanism is to allow the scheduler dynamically adjust its scheduling strategy, which can flexibly satisfy the different requirements under different circumstances. In this paper, we propose *Loading-driven Adaptive Scheduling Algorithm* (*LASA*), which will adjust the number of backup copies been scheduled based on current processor utilization. Clearly, LASA introduces a trade-off between rejecting fewer tasks and risking the fault-tolerance. Besides, we add a waiting queue to collect unschedulable tasks instead of directly reject them. When the computing resources are reclaimed after deallocating backup copies, tasks in the waiting queue can be rescheduled to improve the overall schedulability. From the simulation results, our LASA outperforms other related algorithms.

The remainder of this paper is organized as follows. Section 2 describes the system model and related work.

Design issues and principles of LASA are introduced in section 3. In section 4, some performance evaluations are given. Finally, we give some conclusions in section 5.

2 Fundamental Background

2.1 System, Task, and Fault Models

The *heterogeneous multiprocessor system* consists of m application processors $P_1 \ldots P_m$ connected by a network and one dedicated *scheduler*. The communication between the scheduler and application processors is through *dispatch queues*. Real-time tasks arrive at the scheduler and executed separately on all application processors. The *Spring* system is such an example [3].

Because [17] have proven that precedence constraints can be actually removed, real-time tasks are usually assumed non-preemptive, non-parallelizable, aperiodic, and independent [1, 4, 8, 11-15]. Every task T_i has following attributes: *arrival time* (a_i), *deadline* (d_i), and *computation time* on processor P_j (c_{ij}) [11-12]. These attributes are not known *a priori* until T_i arrives at the system. Each task T_i has *primary* (Pr_i) and *backup* (Bk_i) copies with identical attributes. Since tasks are not parallelizable, $d_i - r_i$ should be long enough to schedule both primary and backup copies of T_i [4, 8].

Assume that each task encounters at most one failure either due to processor or software. That is, if Pr_i fails, Bk_i will always be completed successfully. This also implies that there is at most one failure in the system at a time. The faults are independent, and can be transient or permanent. Simply, we assume the scheduler is fault free.

2.2 Basic Terminologies [11, 12]

In the following, we list some definitions which will be used in our proposed algorithm. For each task T_i, we don't allow its two copies Pr_i and Bk_i been scheduled at overlapped time intervals. In addition, Pr_i and Bk_i must be executed on different application processors to tolerant permanent processor failure.

Definition 2.1 For a task T_i, its *Latest Finish time of Primary* (*LFP*) is defined as

$$LFP(T_i) = d_i - \mathbf{min}\{c_{ij}\},\ \forall\ P_j$$

Definition 2.2 For a task T_i, $EFT_j(T_i)$ indicates its *Earliest Finish Time* on P_j. If T_i cannot be completed before d_i on P_j, $EFT_j(T_i)$ is set as *infinite*.

Definition 2.3 For a task T_i, its *Earliest Finish Time* (*EFT*) is defined as

$$EFT(T_i) = \mathbf{min}\{EFT_j(T_i)\},\ \forall\ P_j$$

Definition 2.4 For a task T_i, its *Latest Start Time of primary* (*LST*) is defined as

$$LST(T_i) = d_i - \mathbf{max}\{c_{ij}\} - \mathbf{2^{nd}max}\{c_{ij}\},\ \forall\ P_j$$

Definition 2.5 $H(T_i)$ is the *heuristic* function defined as

$$H(T_i) = EFT(T_i) + d_i, \text{ if } EFT(T_i) \text{ is } \textit{not infinite}$$

Definition 2.6 For a task T_i, $BLST_j(T_i)$ indicates its *Latest Start Time of backup* on P_j. If Bk_i cannot be completed before d_i on P_j, $BLST_j(T_i)$ is set as *zero*.

Definition 2.7 For a task T_i, its *Latest Start Time of backup* (*BLST*) is defined as

$$BLST(T_i) = \mathbf{max}\{BLST_j(T_i)\},\ \forall\ P_j \text{ except the one that executes } Pr_i$$

2.3 Related Work

Because PB scheme schedules two copies of each task on different processors, the entire schedulability is obviously decreased. Therefore, two techniques *BB-overloading* and *backup deallocation* are designed to reduce the negative influence [4]. *Guarantee Ratio* (*GR*), which means the percentage of tasks whose deadlines are met, is a common objective for real-time task scheduling algorithms. In this paper we use the same definition of *GR* as in [5, 11-14].

$$GR = \frac{\textit{number of tasks whose deadlines are met}}{\textit{total number of tasks arrived in the system}} \times 100\%$$

Distance Myopic Algorithm (*DMA*) is a heuristic search algorithm that schedules real-time tasks on homogeneous multiprocessor with fault-tolerance [5, 11]. It uses an integrated heuristic function to prioritize tasks, and a *feasibility check window* to achieve look-ahead nature. *Fault Tolerant Myopic Algorithm* (*FTMA*) is extended from DMA to be used on heterogeneous multiprocessor [11]. It further changes the mechanism of

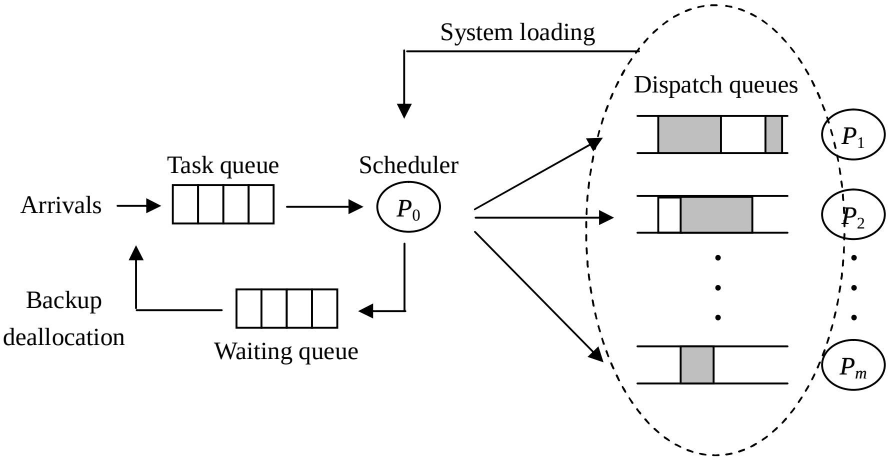

Figure 1. The loading-driven scheduler.

task queue construction to improve the schedulability. *Density first with minimum Non-overlap scheduling Algorithm* (*DNA*) is another effective algorithm [12]. It proposes the *density* function to select the most urgent task, and the *Minimum Non-Overlap* (*MNO*) strategy to minimize the reserved time slots for backup copies. All these three scheduling algorithms are quite efficient but never consider the adaptation mechanism.

In the following we introduce two adapted scheduling algorithms. [13] is an algorithm that can adjust the number of copies of each task been scheduled. Each task is given the redundancy level and fault probability, where the redundancy level is the maximum number of copies it can be scheduled. A task contributes a positive value to the *Performance Index* (*PI*) if completed successfully. Conversely, it incurs a small *PI* penalty if rejected and a large *PI* penalty if all its copies are failed. By evaluating the expected value of *PI*, the scheduler will decide the number of copies that each task must be scheduled.

[14] is a feedback-based algorithm that can adjust the degree of overlapping between the primary and backup copies of the same task. Its adapting strategy is based on an estimation of the primary fault probability and laxities of tasks. However, this algorithm is impractical because to decide the degree of overlapping is difficult. Besides, backup deallocation technique is unfit for this algorithm, because only part of backup copies is reclaimed.

3 Loading-driven Adaptive Scheduling Algorithm (LASA)

In Section 3.1, we give an overview of our proposed *Loading-driven Adaptive Scheduling Algorithm* (*LASA*). Two main mechanisms of LASA, including the action of waiting queue and the loading-driven adaptation strategy, are described in Section 3.2 and 3.3 respectively.

3.1 Overview

Before introducing proposed algorithm, we introduce the system model as shown in Figure 1. This architecture is similar as *Spring* system [3], with an additional *waiting queue* and the *feedback* from dispatch queues to scheduler. Unschedulable tasks will be collected in the waiting queue and tried to be rescheduled later, where other related methods usually reject them directly. The information of system loading will be responded back to the scheduler. According to this feedback, the scheduler will decide the backup copy of a task should be scheduled or not. Both these two mechanisms will be introduced in detail later.

Our proposed LASA mainly contains three phases: to select a task from the task queue, to allocate the selected task to application processors, and to reject unfitted tasks from the waiting queue. In this subsection we describe the task selection and allocation without adaptation strategy. The action of waiting queue and adaptation strategy will be introduced in Section 3.2 and 3.3.

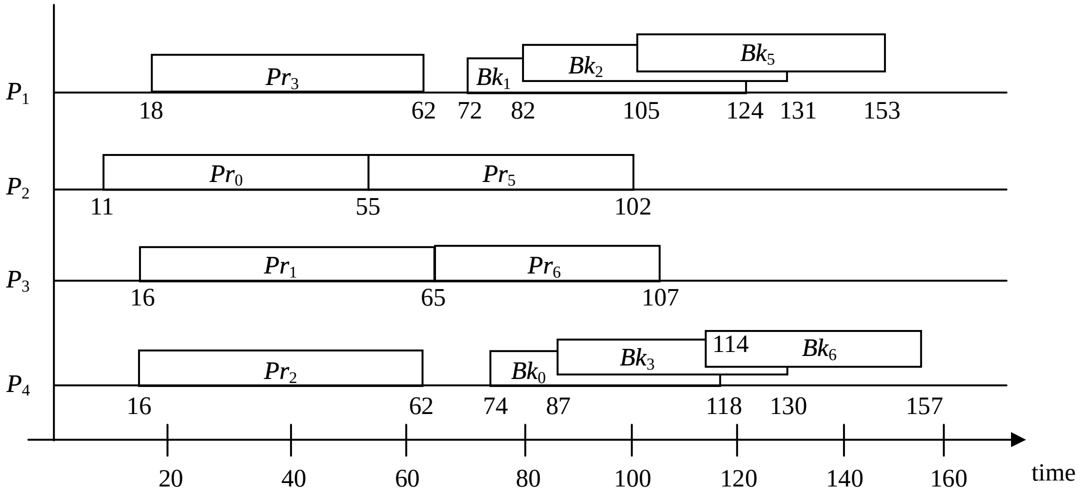

Figure 3. The complete scheduling result of task set in Figure 2.

	T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9
r_i	11	16	16	18	29	45	48	53	54	70
d_i	118	124	131	130	137	153	157	173	165	165
c_{i1}	52	52	49	44	46	48	53	45	43	47
c_{i2}	44	52	54	48	47	47	52	54	45	46
c_{i3}	53	49	56	56	58	48	42	57	48	46
c_{i4}	44	53	46	43	44	43	43	59	46	44

Figure 2. A task set and its attributes.

In the first phase, the heuristic values $H(T_i)$ of all tasks in the task queue are calculated based on Definition 2.5. During the calculation, if we find a task T_i with infinite $EFT(T_i)$, which means T_i cannot be successfully scheduled currently, this task will be moved into the waiting queue. After that, T_i with smallest $H(T_i)$ value is selected to be scheduled.

In the second phase, we try to schedule both primary and backup copies of T_i to application processors. Notice that from the definition of $EFT(T_i)$, only one task copy of T_i is considered. That is, for the selected task T_i, Pr_i can always be successfully scheduled but Bk_i may not. Therefore, we first calculate $BLST(T_i)$ defined above before scheduling T_i. If $BLST(T_i)$ equals to zero, T_i is moved to the waiting queue because there is no available time slot for Bk_i on any application processor. Otherwise, both Pr_i and Bk_i can be successfully scheduled. We simply use ASAP and ALAP strategies to schedule Pr_i and Bk_i respectively in LASA. In order to increase the overall schedulability, *BB-overloading* technique is also applied. The first two phases of LASA will be executed repeatedly until the task queue is empty.

For example, Figure 2 lists a task set and its attributes. Suppose there are four application processors, the complete scheduling result of this task set is shown in Figure 3. From this result, we can find that tasks T_4, T_7, T_8, and T_9 are moved into the waiting queue and the GR equals to 60%.

3.2 Task Deferment and Rejection in Waiting Queue

From related works we have surveyed, a task has only one chance to be scheduled. If it cannot be successfully scheduled at that time, it will be rejected directly. In fact, because Bk_i will be executed only when the corresponding Pr_i fails, a task still has other chances to be rescheduled. This situation is more obvious when the *backup deallocation* technique is applied. Therefore, in LASA, we add a waiting queue to collect unschedulable tasks and try to reschedule them when every backup copy is deallocated.

Meanwhile, we also need a mechanism to reject unfitted tasks from the waiting queue that cannot be successfully rescheduled any more. Hence, before rescheduling, we calculate $LST(T_i)$ values of tasks in the

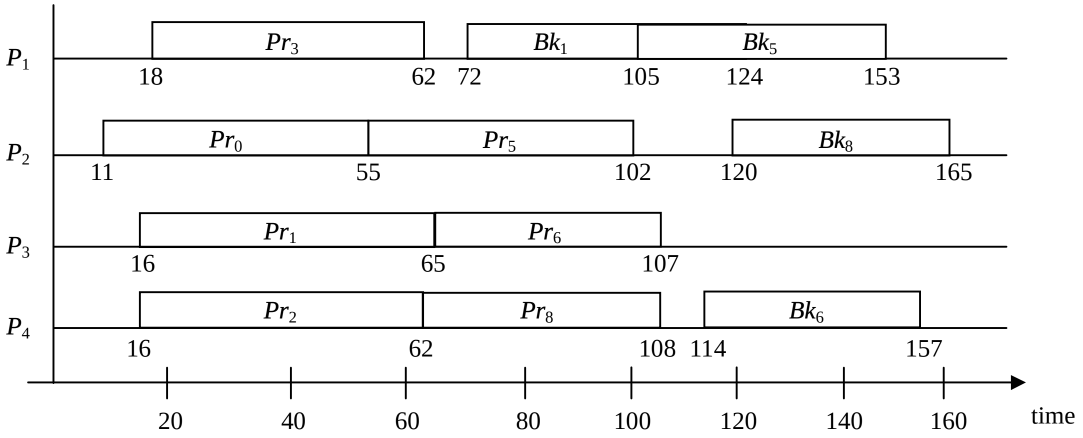

Figure 4. The scheduling result of task set in Figure 2 (with the waiting queue).

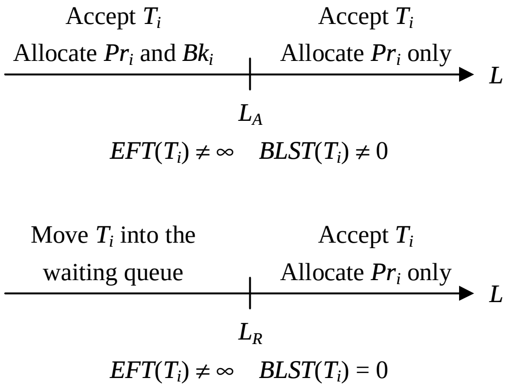

Figure 5. Adaptation strategy used in LASA.

waiting queue. All $LST(T_i)$ values are compared with the time that the backup deallocation just happens. If any $LST(T_i)$ is smaller, which means T_i has missed the latest time been successfully scheduled already, that task will be rejected from the waiting queue.

Let us consider the previous example. Suppose no failure happens, Bk_0 will be deallocated at time 55. At that time, three tasks T_4, T_7, and T_8 in the waiting queue are with $LST(T_i)$ values 32, 57, and 63 respectively. Clearly that T_4 is rejected. Because both T_7 and T_8 cannot be rescheduled at that time, they are resided in the waiting queue. Then, at time 62, Bk_2 and Bk_3 are deallocated simultaneously. At this time, T_7 is rejected and T_8 is successfully rescheduled on P_4 and P_2. The modified scheduling result is shown in Figure 4. We can see that *GR* is improved from 60% to 70%.

3.3 Loading-driven Adaptation Strategy

When too many tasks arrive at a small time interval, the system is overloaded and some tasks will be rejected by the scheduler. Since rejecting tasks degrades the overall *GR* (or schedulability), it is reasonable to intentionally stop scheduling backup copies to accept more tasks. This approach apparently takes a trade-off between the *GR* and the degree of reliability. Hence, in LASA, we propose a loading-driven adaptation strategy, which aims to improve the *GR* without sacrificing too much reliability.

Definition 3.1 For a real-time system with *m* application processors, *L* denotes the *system loading* defined as:

$$L = \frac{1}{m}\sum_{i}\frac{avg(c_{ij})}{d_i - a_i} \quad ,$$

for all tasks currently resided in dispatched queues and $avg(c_{ij})$ indicates the average execution time of T_i on $P_1 \ldots P_m$

As shown in above definition, we use the processor utilization in a small time interval to indicate the *system loading*. *L* is dynamically calculated by the scheduler. In the beginning, all dispatch queues are empty and *L* equals to zero. Then, *L* increases when the scheduler dispatches a new task, and decreases when a dispatched task is finished either successfully or faultily.

Our adaptation strategy is appended to the task allocation phase in LASA. Two thresholds L_A and L_R are

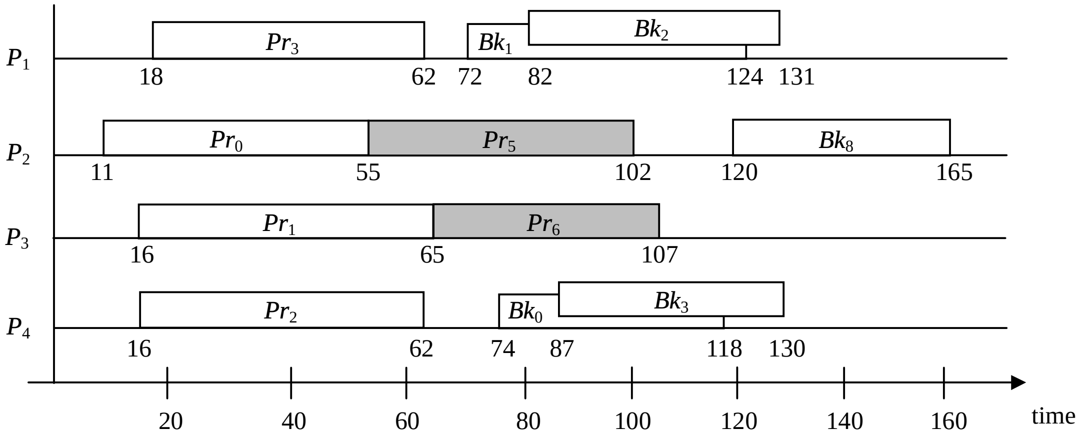

Figure 6. The scheduling result of task set in Figure 2 (at time step 54).

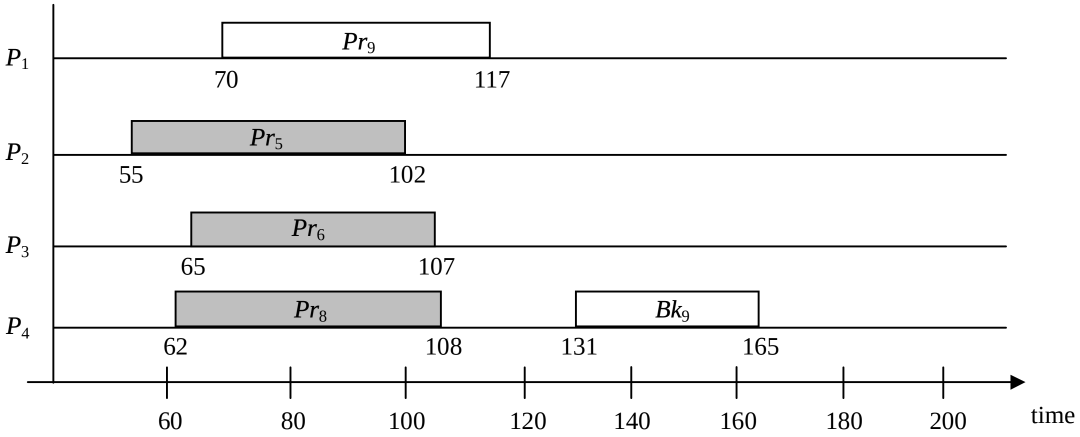

Figure 7. The scheduling result of task set in Figure 2 (at time step 70).

given in advance. As the variation of L, we adaptively apply three mechanisms as follows to allocate the selected task T_i.

Case 1: If both feasible time slots of Pr_i and Bk_i can be found, T_i is accepted. Pr_i is allocated directly, but Bk_i is only allocated if $L \leq L_A$.

Case 2: If only the feasible time slot of Pr_i can be found, T_i is accepted with condition $L > L_R$. Otherwise, T_i is moved to the waiting queue.

Case 3: If the feasible time slot of Pr_i cannot be found, T_i is moved to the waiting queue directly.

Above allocation mechanisms are illustrated in Figure 5. Notice that the quantitative relation between L_A and L_R is uncertain. Considering the previous example, suppose that L_A and L_R equal to 0.4 and 0.5 respectively. When T_5 arrives at time step 45, T_0~T_3 have been scheduled with both two copies and T_4 has been moved into the waiting queue. At that time, even through both feasible time slots for Pr_5 and Bk_5 can be found, only Pr_5 is allocated because $L = 0.451 > L_A$. Similarly, T_6 is accepted with only Pr_6 been allocated. After the scheduler moves T_7 and T_8 into the waiting queue, the partial scheduling result is shown in Figure 6 (time step 54).

At time step 55, the scheduler deallocates Bk_0 and rejects T_4. When Bk_2 and Bk_3 are both deallocated at time step 62, T_7 is rejected and only Pr_8 is allocated because $L = 0.448 > L_A$. Then, at time step 70, T_9 is accepted with both two copies because L becomes 0.319. Figure 7 shows the complete scheduling result at time step 70. We can see that with the loading-driven adaptation strategy, GR is further improved to 80%. Finally, the overall algorithm of LASA is listed in Figure 8.

```
1. Calculate $H(T_i)$ for all tasks in the task queue
     if ($EFT(T_i)$ is infinite)
         Move $T_i$ into the waiting queue
2. Select $T_i$ with the smallest $H(T_i)$
3. Find feasible time slots for the selected $T_i$
4. if ($BLST(T_i)$ is not zero)
       if ($L > L_A$)   Allocate only $Pr_i$
       else   Allocate $Pr_i$ and $Bk_i$
   else if ($L > L_R$)   Allocate only $Pr_i$
   else   Move $T_i$ into the waiting queue
5. Repeat steps 1~4 until the task queue is empty
6. if backup deallocations happened at time step $t$
     Calculate $LST(T_i)$ for all tasks in waiting queue
     if ($LST(T_i) < t$)   Reject $T_i$
```

Figure 8. The overall algorithm of LASA.

Parameter	Explanation	Values
MIN_C	Min. execution time	10
MAX_C	Max. execution time	80
l	Task arrival rate	{0.4, 0.5, …, 1.2}
R	Laxity	{2, 3, …, 10}
m	Number of application processors	{3, 4, …, 10}

Figure 9. Parameters used in the task generator.

Parameter	Explanation	Values
FP	Probability of a primary copy failure	[0, 0.1]
SoftFP	Probability of software failure	0.2
HardFP	Probability of hardware failure	0.8
PermHardFP	Probability of a permanent hardware failure	10^{-6}
MAX_Recovery	Maximum recovery time after a transient hardware fault	50

Figure 10. Parameters used in the dynamic simulator.

4 Performance Evaluations

After designing the algorithm of LASA, we construct a simulation environment to evaluate it. Our environment and experimental results are described in this section.

4.1 Simulation Environment

Our environment contains two parts named the task generator and the dynamic simulator. The task generator generates a real-time task set in the non-decreasing order of arrival times. Figure 9 lists all used parameters, which can generate task set with any characteristic [12]. For a task T_i, its worst case execution times c_{ij} are uniformly distributed in interval [*MIN_C*, *MAX_C*]. The inter-arrival times between tasks is exponentially distributed with mean $(MIN\_C + MAX\_C) / 2lm$ [5]. In order to make sure that both copies of T_i can be successfully scheduled, its deadline d_i is chosen uniformly between $(a_i + \mathbf{max}\ c_{ij} + 2^{nd}\mathbf{max}\ c_{ij}, a_i + R \times \mathbf{max}\ c_{ij})$.

The dynamic simulator simulates events including task arrivals, task finishes, backup deallocations, and failure occurs. We consider three failure types: software failure, permanent hardware failure, and transient hardware failure. A software failure immediately terminates the task that causes the fault. The failed processor with permanent hardware failure will never be available. Contrarily, if the hardware failure is transient, that processor will be available after *MAX_Recovery*. Probabilities for failures and related parameters are listed in Figure 10 [12].

4.2 Experimental Results

In this subsection, we evaluate performances of FTMA, DNA, n_DNA (DNA with waiting queue), LASA, and m_LASA (LASA with MNO strategy). For each set of parameters, we generate 20 task sets and each one contains 20000 independent tasks. Moreover, because FTMA requires additional parameters, we evaluate it with various parameter combinations and select the best result. We directly use the *GR* defined above as the objective.

Figures 11~13 shows the *GR* of different scheduling algorithm with various task arrival rates (l), task laxity (R), and the number of processors (m). In these evaluations we simply assume all application processors are fault-free. Interestingly, results in these figures are quite similar. FTMA has the lowest *GR* because it schedules backup copies by ASAP strategy, which is hard to take advantages from backup deallocation. Next, DNA performs better than that of FTMA, since it highly

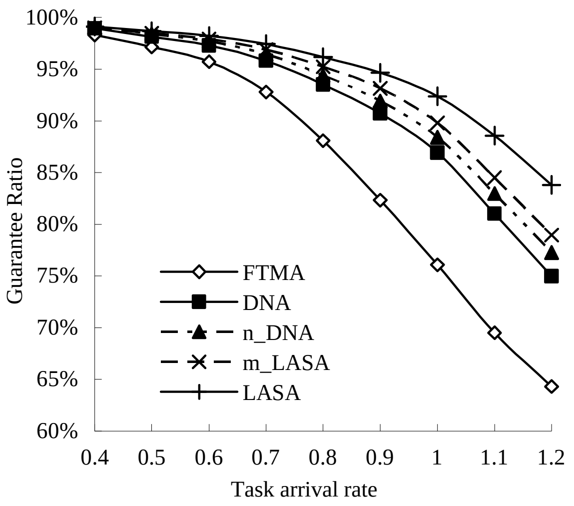

Figure 11. Effect of the task arrival rate (l).
($R = 3$, $m = 8$, $FP = 0$, $L_A = 0.95$, $L_R = 1$)

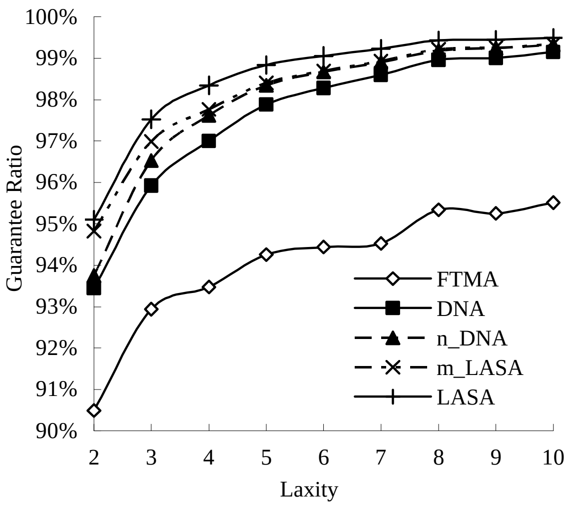

Figure 12. Effect of the laxity (R).
($l = 0.7$, $m = 8$, $FP = 0$, $L_A = 0.95$, $L_R = 1$)

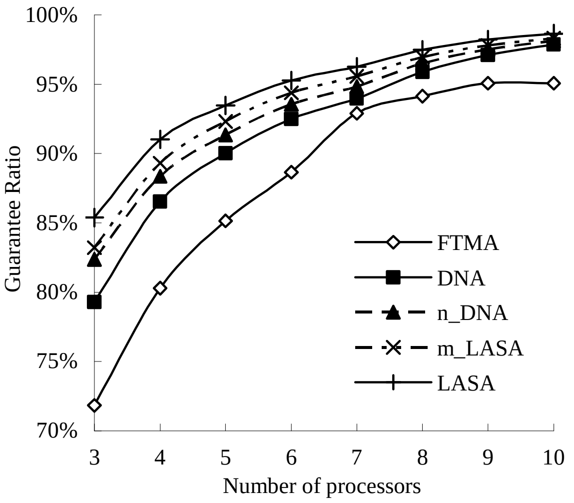

Figure 13. Effect of the number of processors (m).
($R = 3$, $l = 0.7$, $FP = 0$, $L_A = 0.95$, $L_R = 1$)

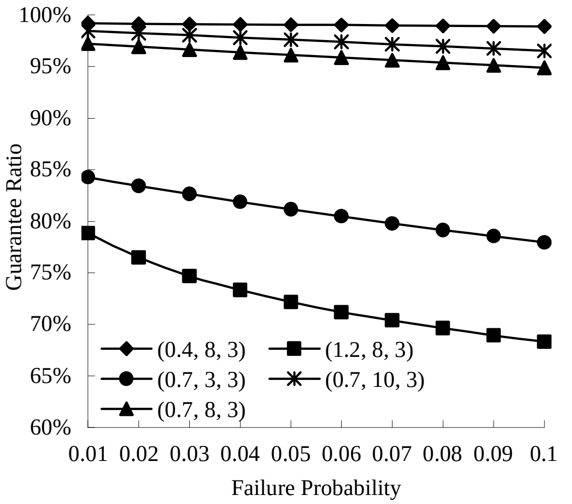

Figure 14. Effect of the failure probability.
The 3-tuple indicates (l, m, R)

exploits properties of backup overloading and deallocation. From performances of DNA and n_DNA, we find that adding the waiting queue can cause the positive influence. In summary, LASA obviously has the highest *GR* in most circumstances. According to curves of LASA and m_LASA, we further conclude that MNO strategy is unfit for LASA.

Figure 14 shows the *GR* of our LASA with various failure probabilities (*FP*). We find that the *GR* decreases with the *FP* increasing in all cases, especially when the workload is heavy. However, the decrease of *GR* is actually not significant, which means the performance of LASA is quite stable.

Next, in Figure 15, we evaluate the influence of *GR* between L_A and L_R. When L_A varies from 0.5 to 1.0, the decrease of *GR* is about 5% in different L_R values. Contrarily, for any constant L_A, the difference of *GR* is less than 0.5% when L_R varies from 0.7 to 1.0. It is obvious that the value of L_A causes more influence of *GR* than that of L_R.

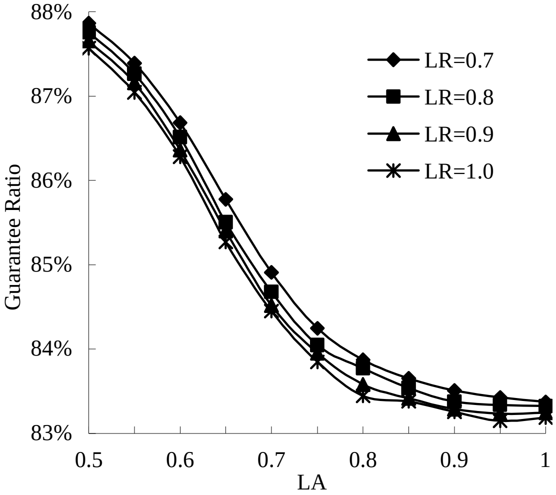

Figure 15. Effect of the threshold values.
(R = 3, m = 8, l = 1.2, FP = 0)

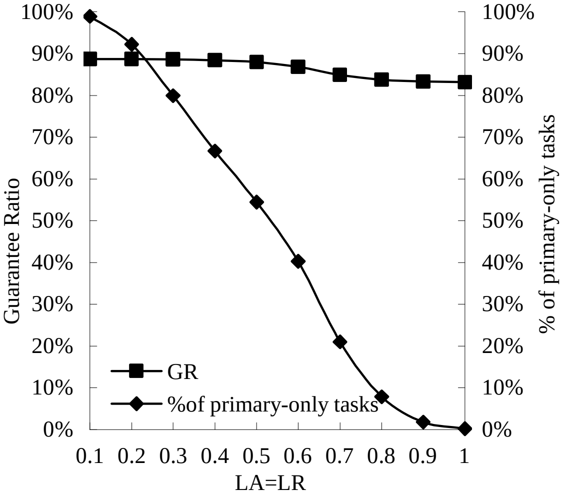

Figure 16. Effect of the threshold values.
(R = 3, m = 8, l = 1.2, FP = 0)

Finally, Figure 16 simultaneously shows the *GR* and the percentage of primary-only tasks been scheduled with various L_A. Since the value of L_R has slightly effects of *GR*, we set L_R equals to L_A for convenience in this evaluation. In this figure, we find that when L_A varies from 1.0 to 0.1, the proportion of scheduled primary-only tasks increases from 0 to 100% but the improvement of *GR* is less than 8%. Actually, the proportion of scheduled primary-only tasks can imply the fault-tolerant capability of the system. Therefore, if we don't want to sacrifice too much reliability, L_A should be set larger.

5 Conclusions

In this paper, we propose an effective *Loading-driven Adaptive Scheduling Algorithm* (*LASA*) to dynamically schedule real-time tasks with fault-tolerance. LASA mainly contains two features. First, an additional waiting queue is added to collect unschedulabe tasks instead of reject them directly. These tasks are tried to be rescheduled at proper time. Second, based on the information of system loading responded back to the scheduler, we intentionally schedule only one copy of a task to accept more tasks when the system is overloaded. A simulation environment is also constructed to evaluate LASA. From experimental results, these two features actually can improve the overall schedulability. Besides, although the system cannot tolerant failures when it is overloaded, the sacrifice of reliability is quite minor.

References

[1] K. Ramamritham, J. A. Stankovic, and Perng-fei Shiah, "Efficient Scheduling Algorithms for Real-time Multiprocessor Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, pp 184-194, April 1990.

[2] K. G. Shin and P. Ramanathan, "Real-Time Computing: A New Discipline of Computer Science and Engineering", *Proc. of IEEE*, Vol. 82, No. 1, pp 6-24, Jan. 1994.

[3] J. A. Stankovic, K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Systems", *IEEE Transactions on Software Engineering*, Vol. 8, Issue 3, pp 62-72, May 1991.

[4] S. Ghosh, R. Melhem, and D. Mosse, "Fault-Tolerance Through Scheduling of Aperiodic Tasks in Hard Real-Time Multiprocessor Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 3, pp 272-284, March 1997.

[5] G. Manimaran and C. S. R. Murthy, "A Fault-Tolerant Dynamic Scheduling Algorithm for

Multiprocessor Real-Time Systems and Its Analysis", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 9, No. 11, pp 1137-1152, Nov. 1998.

[6] R. Al-Omari, G. Manimaran, and A. K. Somani, "An Efficient Backup-overloading for Fault-tolerant Scheduling of Real-time Tasks", *Proc. of IEEE Workshop on Fault-tolerant Parallel and Distributed Systems*, pp 1291-1295, 2000.

[7] R. Al-Omari, A. K. Somani, and G. Manimaran, "A New Fault-tolerant Technique for Improving Schedulability in Multiprocessor Real-time systems", *Proc. of International Parallel and Distributed Processing Symposium*, April 2001.

[8] R. Al-Omari, A. K. Somani, and G. Manimaran, "Efficient Overloading Techniques for Primary-Backup Scheduling in Real-Time Systems", *Journal of Parallel and Distributed Computing*, Vol. 64, No. 1, pp 629-648, Jan. 2004.

[9] C. Shen , K. Ramamritham , and J. A. Stankovic, "Resource Reclaiming in Multi- processor Real-Time Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 4, pp 382-397, April 1993.

[10] L. V. Mancini, "Modular Redundancy in a Message Passing System", *IEEE Transactions on Software Engineering*, Vol.12, No. 1, pp 79-86, Jan. 1986.

[11] Y. H. Lee, M. D. Chang, and C. Chen, "Effective Fault-tolerant Scheduling Algorithms for Real-time Tasks on Heterogeneous Systems", *Proc. of National Computer Symposium*, Dec. 2003.

[12] M. D. Chang, **A Fault-tolerant Dynamic Scheduling Algorithm for Real-time Systems on Heterogeneous Multiprocessor**, Master Thesis, National Chiao-Tung University, June 2004.

[13] S. Swaminathan and G. Manimaran, "A Value-based Scheduler Capturing Schedulability Reliability Tradeoff in Multi- processor Read-time Systems", *Journal of Parallel and Distributed Computing*, Vol. 64, No. 5, pp 629-648, May 2004.

[14] R. Al-Omari, A. K. Somani, and G. Manimaran, "An Adaptive Scheme for Fault-Tolerant Scheduling of Soft Real- Time Tasks in Multiprocessor Systems", *Proc. of International Conference on High Performance Computing*, Dec. 2001.

[15] T. Tsuchiya, Y. Kakuda, and T. Kikuno, "A New Fault-Tolerant Scheduling Technique for Real-Time Multiprocessor Systems", *Proc. of International Workshop on Real-Time Computing Systems and Applications*, pp 197-202, 1995.

[16] M. L. Dertouzos and A. K. Mok, "Multiprocessor On-Line Scheduling of Hard Real-Time Tasks", *IEEE Transactions on Software Engineering*, Vol. 15, No. 12, pp1479-1506, Dec. 1989.

[17] J. W. S. Liu, W. K. Shih, K. J. Lin, R. Bettati, and J. Y. Chung, "Imprecise Computations", *Proc. of IEEE*, Vol. 82, No. 1, pp. 83-94, Jan. 1994.