

Scheduling Algorithms Exploiting Spare Capacity and Tasks' Laxities for Fault Detection and Location in Real-Time Multiprocessor Systems

K. Mahesh G. Manimaran C. Siva Ram Murthy
Dept. Computer Science and Engg.
Indian Institute of Technology
Madras 600 036, INDIA
Email: {gmani@bronto.,murthy@}iitm.ernet.in

Arun K. Somani
Dept. Electrical and Computer Engg.
Iowa State University
USA
Email: arun@iastate.edu

Abstract

Several schemes for detecting and locating faulty processors through self-diagnosis in multiprocessor systems have been discussed in the past. These schemes attempt to start multiple copies (versions) of the tasks on available idle processors simultaneously and compare the results generated by the copies to detect or locate faulty processors. These schemes are based on FCFS scheduling strategy. But, they cannot be applied directly to real-time multiprocessor systems where tasks have timing constraints. In this paper, we present a new scheduling algorithm that not only schedules real-time tasks, but also attempts to perform self-diagnosis if the system is not heavily loaded. We define load as a function of tasks' laxities. We have carried out extensive simulations and compared the results of our algorithm with that of the myopic algorithm, a real-time task scheduler. Simulation results show that our algorithm that exploits both tasks' laxity and spare capacity (unused processors) offers the same performance (guarantee ratio) as that of the myopic algorithm in addition to achieving fault detection and location.

1 Introduction

Real-time systems are defined as those systems in which correctness of the system depends not only on the logical result of computation, but also on the time at which the results are generated. Air traffic control system, process control system, and nuclear plant control system are some examples of such real-time systems. Such systems are life critical and the outcome could be catastrophic, if results are not generated within certain specified time intervals. Multiprocessor systems are being employed for satisfying such requirements due to their potential for high performance and reliability. Scheduling of tasks in a real-time multiprocessor system involves deciding when and on which processor the given tasks have to execute. This can be done either statically or dynamically. In static scheduling, the as-

signment of tasks to processors and also the time at which the tasks start their execution are determined *a priori*. On the other hand, if the characteristics (e.g. deadline) of tasks are known only on their arrivals (and not in advance), scheduling decisions have to be made dynamically. In dynamic scheduling, when a task arrives, the scheduler dynamically determines the feasibility of the task. The scheduler checks if the new task can be guaranteed without jeopardizing the guarantees provided to the previous tasks. Thus for predictable executions, schedulability analysis must be done before a task's execution is begun. A feasible schedule is generated if the requirements (timing, resource, etc.) of the tasks can be satisfied. The tasks are dispatched, at run-time, according to this feasible schedule. The general problem of optimal scheduling of tasks in a multiprocessor system is *NP-complete*. In dynamic scheduling, since the scheduling decisions have to be made at run-time, employing any optimal feasible scheduling algorithm is ruled out. Therefore, most of the dynamic scheduling algorithms resort to heuristic techniques.

Generally, in real-time multiprocessor systems, the demand for the system resources varies with time. The systems are usually provided with enough spare capacity (processors) to meet tasks' timing constraints even when the system is heavily loaded. Therefore, except at peak load, not all processors will be busy. Such unused processors are called as spare capacity. The presence of a large number of processors increases the probability of failure of one of the processors. Hence some mechanism has to be employed to constantly check the health of the system. Instead of employing any additional hardware, the tasks to be scheduled themselves can be used to perform health checking. Such an approach is called as self-diagnosis. Several approaches to perform self-diagnosis in non real-time multiprocessor systems [1, 2] have been discussed in the past. The main objective of this work is to perform self-diagnosis to detect and locate faulty processors in real-time multiprocessor systems.

1.1 Related Work

Fault detection and location in non-real-time multiprocessor systems using self-diagnosis have been discussed in [1] and [2]. [1] describes a scheme in which a task is started (primary version) on any available idle processor. Another copy of the task (secondary version) is started simultaneously if there is an idle processor. The results generated are then compared to detect if one of the two processors is defective. The performance metric used is

$$\Pi(n, \rho) = \frac{\text{number of secondary tasks completed}}{\text{number of primary tasks completed}} * 100$$

where, n is the number of processors and ρ is the average system load. It is to be noted that only fault detection is possible in this approach. [2] extends this by proposing three schemes by which faulty processors can be located directly. The basic idea is to start more than one secondary version whenever possible. The performance metrics used are Fault Detection Capability (FDC) and Fault Location Capability (FLC). FDC gives the average amount of time a processor is checked by two or more processors, whereas FLC gives the average amount of the time a processor is checked by three or more processors. They are defined as

$$FDC(n, \rho) = \left(\frac{1}{n}\right) \sum_{i=2}^n \frac{i * \gamma_i}{\mu_i} * 100$$

$$FLC(n, \rho) = \left(\frac{1}{n}\right) \sum_{i=3}^n \frac{i * \gamma_i}{\mu_i} * 100$$

where γ_i is the average number of tasks that complete with i versions and μ_i is the average service time (execution time) of the tasks that complete with γ_i versions. The following example shows the weakness of these algorithms when applied to real-time tasks, where tasks have timing constraints.

Example 1 : Consider a real-time multiprocessor system with 4 processors. Let four tasks, each having a computation time of 6 time units and deadline of 30 time units, arrive at the same time. If we apply the schemes in [1] and [2] directly, all four tasks will be scheduled with only one version and hence fault detection or location is not possible. However, since the tasks have enough laxities (defined in Section 2.1), each one of them can be started with 4 copies resulting in performing both fault detection and location. Thus, the schemes proposed in [1] and [2], if applied directly to real-time systems, fail to take advantage of tasks' laxities. \square

2 Task Model and Definitions

• Each Task (T_i) is characterized by Ready time (r_i), Worst case computation time (c_i), and Deadline (d_i) - the time by which T_i must finish its computation.

- Tasks are non-preemptable. A task may require some resources during its execution. The resource itself can be accessed in shared or exclusive mode.
- The scheduler fixes the start time (s_i) for each task - the time at which T_i is scheduled to start its execution. And finish time (f_i) - the time at which T_i will finish its execution ($= s_i + c_i$).
- Laxity (l_i) of task T_i denotes the latest time by which the task must start its execution, defined as $l_i = d_i - c_i$.

2.1 Notations and Definitions

\mathcal{T} = Set of all tasks; \mathcal{S} = Set of scheduled tasks; \mathcal{U} = Set of unscheduled tasks ($= \mathcal{T} - \mathcal{S}$); δ_i - Number of versions (copies) of T_i that are scheduled; L - the largest deadline of the unscheduled tasks; p - the number of processors in the real-time multiprocessor system; $R = \{ R_1, R_2, \dots, R_r \}$ be the set of resources in the system; $|R_i|$ - the number of instances of resource type R_i . Each resource type R_i can be accessed either in Shared (S) or Exclusive (X) mode. Let $R(T_i)$ be the set of resources requested by T_i ; EAT_{ij}^m - the j^{th} earliest available time of the resource R_i in mode m. if $j > |R_i|$ then $EAT_{ij}^m = L$; EST_{ij} - the earliest time j versions of T_i may be started.

Definition 1: A task T_i is said to be feasible (schedulable) iff $EST_{i1} + c_i \leq d_i$.

Definition 2: A partial schedule is one which does not contain all the tasks. A partial schedule is said to be *strongly feasible* if all the schedules obtained by extending the current schedule by any one of the yet unscheduled (primary) tasks are also feasible [4]. By definition, an empty schedule is strongly feasible.

Definition 3: The performance metric *guarantee ratio* is defined as the fraction of total tasks arrived in the system that are found to be schedulable by an algorithm.

2.2 The Proposed Scheduling Algorithm

In this section, we present a dynamic scheduling algorithm that schedules real-time tasks and also attempts to perform self-diagnosis. The proposed algorithm is a variation of myopic algorithm [4]. Myopic algorithm is a heuristic algorithm that schedules dynamically arriving tasks which have resource constraints. It starts with an empty partial schedule (\mathcal{P}) and constructs the full feasible schedule by extending \mathcal{P} with one task at a time. Before extending \mathcal{P} with a task, it first checks if the current \mathcal{P} is strongly feasible or not. Strong feasibility check is performed by considering only the first k tasks (called the feasibility check window) instead of all the remaining tasks in the list. If \mathcal{P} is strongly feasible, a heuristic function $h()$ is applied to the first k tasks in the list. The schedule is then extended with the task that has the smallest $h()$ value. It has been shown in [4] that the integrated heuristic function $EST_{i1} + d_i$ performs better than simple heuristics such as earliest deadline first and shortest laxity first.

Any algorithm that attempts to perform self-diagnosis using real-time tasks must tackle two problems. The first is to select the correct task to extend the schedule and second is to schedule the right number of versions (δ_i) to the selected task so that tasks' timing requirements are met. The proposed scheduling algorithm solves the first problem by using the same heuristic function $h()$ as that of the myopic algorithm. To determine δ_i , it uses another heuristic function $\mathcal{R}(T_i)$. The proposed algorithm is given below:

Input : A task set ordered in non-decreasing order of deadlines.

Output : TRUE if the task set is feasible.
FALSE otherwise.

$U \leftarrow T$

While ($U \neq \phi$)

 If (current schedule is strongly feasible)

 Pick the task with minimum h value

 Let T_i be the selected task to extend schedule.

$\delta_i \leftarrow \mathcal{R}(T_i)$

 Extend schedule with δ_i versions of T_i

 each having start time $EST_{i\delta_i}$,

$U \leftarrow U - T_i; \quad \mathcal{S} \leftarrow \mathcal{S} \cup T_i$

 Else

 Backtrack to the previous level

 Let T_j be the task last scheduled

 If ($\delta_j > 1$) $\delta_j \leftarrow \delta_j / 2$

 Else

 Increment the backtrack count

 If (maximum backtracks reached)

 return (FALSE)

$U \leftarrow U \cup T_j; \quad \mathcal{S} \leftarrow \mathcal{S} - T_j$

return (TRUE)

In addition to the guarantee ratio, which is the primary metric, we define two more metrics, viz; Time spent on Fault Detection (TFD) and Time spent on Fault Location (TFL) as follows :

$$TFD = \frac{\sum_{T_i \in \mathcal{S} \text{ and } \delta_i \geq 2} \delta_i * c_i}{\text{Total time the system was operated}} * 100$$

$$TFL = \frac{\sum_{T_i \in \mathcal{S} \text{ and } \delta_i \geq 3} \delta_i * c_i}{\text{Total time the system was operated}} * 100$$

It is obvious that if δ_i is always set to 1, then the algorithm behaves like the myopic algorithm. *The main objective is to achieve as much TFD and TFL as possible while achieving the same guarantee ratio as that of myopic algorithm.* In the rest of this section, we present three different algorithms (heuristics), that employ different $\mathcal{R}(T_i)$ to determine the number of versions.

2.3 Greedy Algorithm

The greedy algorithm attempts to schedule as many versions as possible for a task. The heuristic function

$\mathcal{R}(T_i)$ is defined as

$$\delta_i \leftarrow \text{Max } j \text{ such that } EST_{ij} + c_i \leq d_i \quad (j = 1 \dots p)$$

Thus the number of versions scheduled is limited only by the availability of the resources required by T_i . The greedy algorithm takes $\mathcal{O}(kr)$ time to perform strong feasibility check and $\mathcal{O}(pr)$ time to compute δ_i . Further, the schedule is extended in $\mathcal{O}(pr)$ time. Hence the total complexity of the greedy algorithm is $\mathcal{O}(n(kr + pr + pr)) = \mathcal{O}(n)$.

Example 2 : Consider a real-time system with 4 processors. Assume that 5 tasks $\{T_1, T_2, \dots, T_5\}$ are to be scheduled and they all have $c_i = 5$ time units and $d_i = 10$ time units. The greedy algorithm will assign all the four processors to the first 2 tasks. But after T_2 is scheduled, the partial schedule will no longer be strongly feasible. The scheduler backtracks and reduces the number of versions for T_2 (from 4) to 2. Now after T_3 is scheduled, the partial schedule once again is no longer be strongly feasible. It is quite easy to see that the greedy scheduler has to backtrack a large number of times to come up with a feasible schedule. \square

2.4 Look Ahead Algorithm

The basic problem with greedy approach is that it does not consider the timing requirements of the unscheduled tasks. It blindly introduces secondary versions and corrects the error entirely with backtracks. The look ahead algorithm attempts to overcome this by examining the laxities of the tasks within the feasibility check window before deciding δ_i . The heuristic function $\mathcal{R}(T_i)$ first scans the feasibility check window and determines the number of tasks whose laxities are smaller than that of f_i . Let t be the number of such tasks. It is clear that these t tasks have to be scheduled before T_i finishes. Since these t tasks have to share the p processors,

$$\delta_i = \begin{cases} \lfloor p/t \rfloor & \text{if } p > t \\ 1 & \text{otherwise} \end{cases}$$

$\mathcal{R}(T_i)$ takes $\mathcal{O}(kr)$ time to compute δ_i . Further, once a task is selected, it takes $\mathcal{O}(pr)$ time to extend the schedule. Hence the time complexity of the look ahead algorithm is $\mathcal{O}(n(kr + kr + pr)) = \mathcal{O}(n)$.

Example 3 : Consider the same problem mentioned in example 2. Let the size of the feasibility check window be 4.

The look ahead scheduler, after scanning the window, finds out that T_2, T_3 and T_4 have enough laxities and hence schedules 4 versions for T_1 . Next, after T_2 is selected for extending the schedule, it scans the window to find out that, all the remaining tasks do not have enough laxities. Hence, including T_2 , 4 tasks have to share the

4 processors. Hence δ_i is 1. Thus, no backtrack is required at all due to look ahead. It is now clear that look ahead offers better guarantee ratio particularly when the maximum number of backtracks allowed is small. \square

2.5 Spare Capacity Algorithm

The problem with the look ahead algorithm is that it blindly introduces a large number of secondary versions for the first few tasks, and hence is left with a large number of tasks to be scheduled in a short time span. Both greedy and look ahead approaches achieve a low guarantee ratio because they fail to take into account the resource requirements of the yet unscheduled tasks. The spare capacity algorithm presented in this section overcomes this problem. The basic idea employed by this algorithm is to determine the spare capacity at a given point of time. The heuristic function $\mathcal{R}(T_i)$ calculates the spare capacity and using this knowledge determines δ_i . We describe the $\mathcal{R}(T_i)$ function below.

Let \mathcal{U} be the set of unscheduled tasks and let T_i be the task selected to extend the schedule. $\mathcal{R}(T_i)$ determines the spare capacity based on the processing and resource requirements of the tasks in \mathcal{U} . The total resource usage time of R_j by the tasks in \mathcal{U} is given by

$$\sigma_j = \sum_{T_x \in \mathcal{U} \text{ and } T_x \text{ uses } R_j} C_x$$

Now, the time k^{th} instance of R_j is available for processing up to d_i is given by

$$\alpha_{jk}(d_i) = \begin{cases} d_i - EAT_{jk}^m & \text{if } EAT_{jk}^m > EST_{i1} \\ d_i - EST_{i1} & \text{otherwise} \end{cases}$$

Therefore, the total time the resource R_j is available up to time d_i is

$$\alpha_j(d_i) = \sum_{k=1}^{\lfloor R_j \rfloor} \alpha_{jk}(d_i)$$

and the utilization of resource R_j up to d_i by tasks in \mathcal{U} is given by

$$\eta_j = \frac{\sigma_j}{\alpha_j(d_i)} * 100\%$$

Intuitively, it can be seen that η_j itself gives a measure of number of instances of R_j to be employed to meet the requirements of the tasks in \mathcal{U} . Thus, a utilization η_j also implies that η_j % of $\lfloor R_j \rfloor$ are enough to handle the tasks in \mathcal{U} . In other words,

$$\beta_j = \lceil \frac{\eta_j * \lfloor R_j \rfloor}{100} \rceil$$

number of resources will be required to handle the requirements of the primary versions of the tasks in \mathcal{U} .

Obviously, the spare capacity is $\lfloor R_j \rfloor - \beta_j$. This spare capacity has to be shared by the β_j primary versions. Therefore, the number of secondary versions that can be safely run per primary task, based on the future requirement for R_j is given by

$$\lfloor \frac{\lfloor R_j \rfloor - \beta_j}{\beta_j} \rfloor$$

Finally, taking into account the requirements of all the resources, the heuristic function $\mathcal{R}(T_i)$ returns the number of versions (primary + secondary), δ_i , for task T_i

$$\delta_i \leftarrow \text{Min}(1 + \lfloor \frac{\lfloor R_j \rfloor - \beta_j}{\beta_j} \rfloor) \quad (\forall j \text{ such that } T_i \text{ uses } R_j)$$

It can be easily shown that the complexity of the spare capacity algorithm is $\mathcal{O}(\text{nr}(k+p+1))$ which is $\mathcal{O}(n)$. It is important to note that the spare capacity algorithm reduces to myopic algorithm if $\mathcal{R}(T_i)$ always sets δ_i to 1. The algorithm achieves the same guarantee ratio as that of the myopic algorithm if the utilization of tasks in \mathcal{U} is above 50%. Also, any $\delta_i > 1$ implies that all the tasks in \mathcal{U} are also likely to be scheduled with same δ_i . However, this may not be always possible as some of the tasks in \mathcal{U} may have some resource conflicts leading to holes in the schedules. Holes result in reduction in available processing time and this indirectly results in reduction in δ_i . Both greedy and look ahead algorithms are unaware of such holes and still attempt to introduce as many secondary versions as possible. The presence of hole is detected by the spare capacity algorithm by calculating $\alpha_{ij}(d_i)$ based on EST_i and not just on EAT_j . By calculating the spare capacity at every stage of the schedule, the spare capacity algorithm controls δ_i to offset the effect of holes. The algorithm takes a very pessimistic view by trying to find spare capacity within d_i . Since meeting task deadlines is the primary objective, we feel that making such a pessimistic assumption is justified.

Example 4 : Consider the same problem mentioned in example 3. For task T_1 :

$$\begin{aligned} \eta_1 &= \frac{5+5+5+5}{4*10} = 50 \\ \beta_1 &= \lceil \frac{50*4}{100} \rceil = 2 \\ \delta_1 &= 1 + \lfloor \frac{4-2}{2} \rfloor = 2 \end{aligned}$$

After T_1 is scheduled with 2 versions, for T_2

$$\begin{aligned} \eta_2 &= \frac{5+5+5}{(2*5)+(2*10)} = 50 \\ \beta_2 &= \lceil \frac{50*4}{100} \rceil = 2 \\ \delta_2 &= 1 + \lfloor \frac{4-2}{2} \rfloor = 2 \end{aligned}$$

It is easy to see that, all tasks will be started with 2 versions thus offering 100% TFD. Moreover, no backtracking is necessary. \square

3 Simulation Studies

To study the effectiveness of the proposed heuristics, we have conducted extensive simulation studies. Here, we are interested in whether or not all the tasks in a task set can finish before their deadlines. Therefore, the most appropriate metric is the schedulability of task sets [4], called *success ratio*, which is defined as the percentage of total number of task sets which are found to be schedulable by a scheduling algorithm. In addition to success ratio, TFD and TFL are also studied. Schedulable task sets are generated for simulation using the following approach.

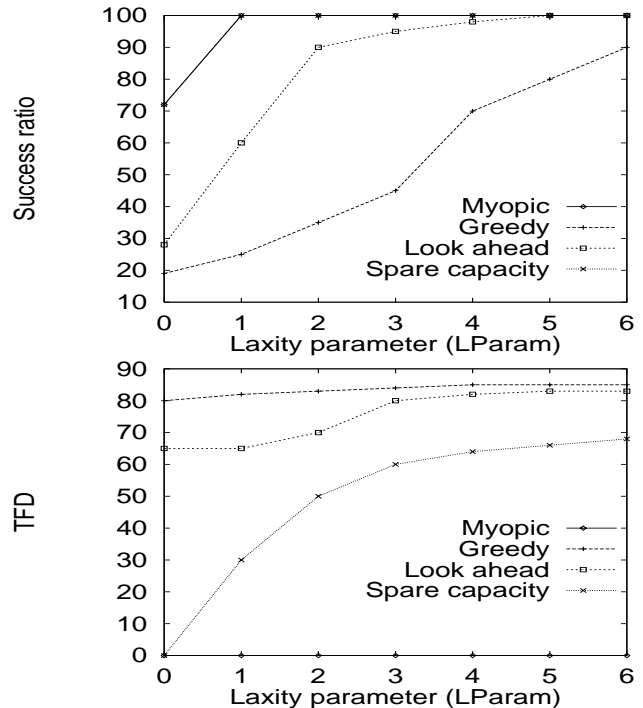
- A task set, consisting only primary versions, is generated up to SC (schedule length which is an input parameter, taken to be 800) with no processors left idle at any time [4].
- The computation time (c_i) of tasks are uniformly distributed between minimum (40) and maximum (60) computation times.
- The deadlines of the tasks are uniformly distributed between SC and $(1 + LParam)*SC$, where $LParam$ is an input parameter.
- The heuristic function $h()$ is defined as $EST_{i1} + d_i$ and the same $h()$ is used for all the above algorithms.

The important point to be noted is that *the guarantee ratio (success ratio) offered by the spare capacity algorithm is the same as that of the myopic algorithm for all parameter variations*. The TFD and TFL offered by the myopic algorithm is always 0 since it does not incorporate fault detection and location capabilities. Here, due to space limitations, we present only a few results.

Fig.1 shows the effect of $LParam$ on success ratio. The success ratio increases with increasing $LParam$ for all the four algorithms. This is because, increasing $LParam$ increases the average laxities of tasks. As mentioned earlier, the success ratio offered by the spare capacity algorithm is the same as that of the myopic algorithm, which is better than the other two algorithms.

Fig.2 shows the effect of $LParam$ on TFD. For all the three proposed heuristics, TFD increases with increasing $LParam$. For the greedy and look ahead algorithms, a larger deadline reduces the number of tasks affected by introducing incorrect number of secondary versions, and hence the TFD increases with $LParam$. For the spare capacity algorithm, a larger deadline implies a larger α_j and hence a larger δ_i . Also, note that, the TFD tends to saturate for larger values of $LParam$. This is because at larger values of $LParam$, the number of versions scheduled is limited by the availability of resources (rather than processors) in the system since tasks have resource requirements. The effect of $LParam$ on TFL exhibits similar behaviour for all the algorithms.

Though the greedy algorithm offers the maximum TFD and TFL, its success ratio is lesser than the other



Figs.1-2 Effect of $LParam$ on success ratio and TFD

algorithms. Since the guarantee ratio (success ratio) is a crucial metric in real-time systems, the algorithm which offers the best guarantee ratio with a capability for fault detection and location is preferable. From our studies, spare capacity algorithm is one such algorithm.

4 Conclusions

In this paper, we have proposed three heuristics for scheduling real-time tasks with fault detection and location capabilities in multiprocessor systems. Our simulation studies show that the spare capacity heuristic, which exploits both tasks' laxities and processor spare capacity, performs better than the other two, and offers the same guarantee ratio as that of the myopic algorithm in addition to achieving fault detection and location.

References

- [1] A. Dabhbura, K. Sabnani, and W. Hery, "Spare capacity as a means of fault detection and diagnosis in multiprocessor systems," *IEEE Trans. Computers*, vol.38, no.6, pp.881-891, June 1989.
- [2] S. Tridandapani, A.K. Somani, and U.R. Sandadi, "Low overhead multiprocessor allocation strategies exploiting system spare capacity for fault detection and location," *IEEE Trans. Computers*, vol.44, no.7, pp.865-877, July 1995.
- [3] F. Wang, K. Ramamritham, and J.A. Stankovic, "Determining redundancy levels for fault tolerant real-time systems," *IEEE Trans. Computers*, vol.44, no.2, pp.292-301, Feb. 1995.
- [4] K. Ramamritham, J.A. Stankovic, and P.-F. Shiah, "Efficient scheduling algorithm for real-time multiprocessor systems," *IEEE Trans. Parallel and Distributed Systems*, vol.1, no.2, pp.184-194, Apr. 1990.