Fault-Tolerant Real-Time Multiprocessor Scheduling

Yingfeng Oh,
Sang Son

Computer Science Report No. TR-92-09
April 9, 1992

Fault-Tolerant Real-Time Multiprocessor Scheduling

Yingfeng Oh and Sang H. Son
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
USA

Abstract

Multiprocessors are increasingly used to support hard real-time systems. The increase in the number of processors in a system makes a system susceptible to processor failures. Fault-tolerant mechanisms as well as real-time scheduling techniques should be used together to ensure that hard real-time systems continue to operate correctly even in the presence of processor failures, since deadline missing in a hard real-time system may result in catastrophic consequences. In this paper, we present approaches to achieving fault-tolerance in hard real-time multiprocessor systems. Using the primary-backup copy approach, we propose two efficient scheduling algorithms to solve a special case of the general scheduling problem. The scheduling algorithms have the property of generating near-optimal solutions to the problem as well as determining the amount of redundancy required to achieve the desired level of fault-tolerance. Experimental results are obtained to evaluate the performance of the heuristics.

Keywords: multiprocessor, fault-tolerance, real-time, scheduling, processor failure

I. Introduction

Hard real-time systems are defined as those systems in which the correctness of the system depends not only on the logical results of computation, but also on the time at which the results are produced. Missing of a hard deadline in such a system may result in catastrophic consequences, such as immediate danger to human life, severe damage to equipments, and waste of expensive resources. As hard real-time systems are being increasingly used in applications which are mission-critical and life-critical, fault tolerance of those systems becomes extremely important.

While extensive research has been done in the areas of fault-tolerant computing [Anders81] [Johnson89] [Lala85] [Pradha86] [Randel78] [Rennel84] [Siewio82] and real-time computing [Stanko88], relatively few research has been carried on in combining the two together to create highly reliable real-time systems. Like any other systems, a computer system is not immune to failure. Various techniques such as fault prevention, fault avoidance and fault-tolerance have been developed to improve the reliability of a system. However, few techniques have been proposed to ensure that the tasks in a system complete in a timely manner. Researchers in real-time computing has been mainly focusing on various scheduling techniques to guarantee that the deadlines of tasks are met [Bettat89]. Few techniques are devised to make the real-time systems fault-tolerant.

Several studies have focused on achieving fault tolerance in real-time systems through the scheduling of redundant resources, such as replicated tasks and redundant processor power [Anders83] [Balaji89] [Bannis83] [Bertos91] [Liestm86] [Krishn86] [Randel78]. Bannister and Trivedi [Bannis83] showed that a set of periodic tasks, each having the same number of clones, can be scheduled on a set of identical processors by the Best-Fit heuristic to achieve a good balance between processor utilization. Their dynamic scheduling algorithm assumes that once a possible missing of a deadline is detected, the task is either aborted or scheduled on additional processors. The number of additional processors are assumed to be enough to accommodate the execution of all real-time tasks. Obviously, their approach can not be used for hard real-time systems. Krishna and Shin [Krishn86] proposed a dynamic programming algorithm for multiprocessor systems to ensure that the backup schedules can be efficiently embedded with the primary schedule. The algorithm assumes the existence of the optimal allocation of tasks to processors and schedules the tasks on each processor in order to minimize a given local cost function. Liestman and Campbell [Liestm86] proposed an algorithm to generate optimal schedule for a single processor system using the recovery-block approach. Bertossi and Mancini[Bertos91] recently showed how to use a scheduling heuristic -- LPT (Largest Processing Time first) [Graham69] to schedule tasks with primary-backup copies on a multiprocessor system to tolerate one arbitrary processor failure, while minimizing the makespan of the schedule. As we shall see later, all these approaches are related to a more general fault-tolerant real-time scheduling problem defined in the next section.

The contribution of this paper is to define a general fault-tolerant hard real-time multiprocessor scheduling problem and propose two efficient scheduling heuristics to solve a specific case of the general problem. Our approach differs from other approaches [Bannis83] [Krishn86] [Bertos91] in that all the hard deadlines of the real-time tasks are guaranteed even in the presence of up to m processor failures in the best case, where m is less than or equal to the number of processors needed to schedule the primary task set. Also, the scheduling heuristics calculate the number of redundant processors and the number of replicated tasks needed to achieve the desired level of fault tolerance.

The rest of the paper is organized as follows: Section II defines the general fault-tolerant real-time multiprocessor scheduling problem. Section III identified a special case of the general scheduling problem and states the scheduling problem more specifically. The two scheduling algorithms are presented in Section IV and V. The analysis and performance evaluation of the scheduling algorithms are described in Section VI. Section VII concludes the paper and suggests future work.

II. Background and Formulation of the General Scheduling Problem

Tasks in a real-time system arrive periodically or aperiodically with hard or soft deadlines associated with them. Periodic tasks appear in a fixed period, while aperiodic tasks arrive at the system with unpredictable intervals. Tasks such as environment monitoring, sampling and fault diagnosis, are periodic tasks. Tasks such as garbage collection, fault isolation and system configuration are aperiodic tasks. Most real-time systems support functions which are periodically executed and the execution of these functions has hard deadlines.

A failure in computer systems is caused by hardware, software or timing faults. Apparently, we want to avoid all of them, especially the timing faults because missing a hard deadline may result in catastrophic consequences. Timing faults are usually avoided through real-time scheduling. Hardware and software faults are generally tolerated through the use of redundant hardware and software. Most hard real-time systems are supported by multiprocessor systems primarily for the following two reasons. First, a multiprocessor system is generally more reliable than a uniprocessor system, because the failure of one processor in a multiprocessor system does not necessarily cause the whole system to fail if some fault tolerance techniques are provided. Second, a multiprocessor system can offer more computational power for hard real-time systems than a uniprocessor system. However, with the employment of more processors also comes the problem of more likelihood of processor failures. A multiprocessor system can be less reliable than a uniprocessor system if one processor failure may cause the whole system to fail. This can happen if no fault-tolerant technique is provided. Thus, a processor failure in a hard real-time multiprocessor system is a very serious problem, which needs to be tolerated. In this paper, we consider the usage of hardware redundancy as well as software redundancy to tolerate processor failures. The approaches taken can be described as the solutions to the following fault-tolerant real-time multiprocessor scheduling problem.

The General Fault-Tolerant Real-Time Multiprocessor Scheduling Problem: The work to be performed by a real-time system is specified as consisting of a set of n tasks $S = \{T_1, T_2, \ldots, T_n\}$. A task i is represented as a quadruple (A_i, C_i, D_i, P_i), where A_i is the arrival time of the task, C_i is the computation time of the task, D_i is the task's deadline and P_i is the task's period. If the task is aperiodic, the value of P_i is undetermined. The fault-tolerant real-time multiprocessor scheduling problem is to find a schedule subject to the following constraints:

(1) All the hard deadlines of the tasks are guaranteed.

(2) Minimize the hardware and software redundancies required.

(3) Maximize the number of processor failures to be tolerated.

(4) Maximize the number of soft deadline tasks to be executed.

Formally, let I be a schedule instance generated by the scheduling algorithm, $N_F(I)$ be the number of processor failures to be tolerated, $D(I)$ be the value function used to measure the success the scheduler has on real-time tasks, and $R_H(I)$ and $R_S(I)$ be the degree of redundancy of hardware and software respectively. The scheduling goal is thus to maximize the following value function:

$$F(I) = \frac{D(I) \times N_F(I)}{R_H(I) R_S(I)}$$

The specification of the scheduling constraints does not limit the possible solutions to one specific hardware or software redundancy approach which may be used to guarantee the deadlines of the tasks even in the presence of hardware or software failures. Software redundancy approaches such as primary-backup copy approach [Randel78] and N-version programming technique [Chen78] can be used separately or together. As we shall see more clearly later, redundant processor power has to be used if processor failure is to be tolerated in a hard real-time system. Among the above requirements, there exists a trade-off between the requirements (2) and (3). Requirement (3) suggests that a task should be replicated as many times as the number of processor failures to be tolerated. However, as we require that no timing faults (i.e., missing deadlines) should occur, more processors are needed to execute the tasks in order to make the deadlines, as the deadlines of the periodic tasks are fixed and the number of tasks increases due to the result of the duplication of tasks. The requirement of more processors not only incurs higher cost, but also increases the probability of processor failures if the reliability of each processor remains the same. Thus, there is a threshold of reliability which can be achieved under the above constraints. This threshold is achieved when the number of processors used is minimized to execute the set of tasks under a certain degree of hardware and software redundancy.

III. Solutions to A Special Case of the General Scheduling Problem

The scheduling problem formulated in the previous section is an extension of the basic multiprocessor scheduling problem considered by Karp[Karp72]. The basic multiprocessor scheduling problem was proved to be NP-complete. In other words, it is widely believed that no optimal solution to minimize the length of schedule can be found in polynomial time complexity. Two efficient multiprocessor scheduling heuristics--LPT [Graham69] and MULTIFIT [Coffma78] have been proposed to obtain approximate solutions to the problem. The scheduling goal of the algorithms is to find a schedule with minimal makespan. The makespan of a schedule is defined as the maximum of the sums of the lengths of the tasks assigned to each processor. An optimal schedule is the one for which the length is the shortest possible for a given number of processors. In both the LPT schedule and the MULTIFIT schedule, the tasks are first sorted into nonincreasing order of computation time. For the LPT schedule the tasks are assigned in decreasing order so that when a task is assigned to a processor, the finishing time of the task on that processor is the earliest. The MULTIFIT algorithm is based on the FFD (First Fit Decreasing) bin packing algorithm. By regarding the processors as bins and the tasks as items having sizes equal to their computation times, the completion of a schedule by time t can be considered as the successful packing of the n items into M bins of size t. In the FFD algorithm for bin packing the bins are numbered from 1 to M and the items, pre-sorted into decreasing order of size, are packed sequentially, each going into the lowest numbered bin in which it will fit. By trying to pack the items using different bin sizes, schedules of different lengths can be obtained. The minimal bin size for which the tasks are packed with the number of bins equal to the number of processors is the makespan of the MULTIFIT schedule. This bin size is found by using a binary search technique. The LPT and MULTIFIT algorithms find schedules whose makespans are within 4/3 and 13/11 of the optimal makespan respectively.

The problem of scheduling a set of periodic tasks on multiprocessor systems are studied in [Dhall78] [Leung82]. There are two basic approaches to schedule a set of real-time periodic tasks on a multiprocessor system: partitioning and non-partitioning approaches. In the partitioning approach considered by Dhall and Liu [Dhall78], the set of tasks are partitioned into groups and assigned to distinct processors. The partitioning is done by applying one of the two bin packing heuristics--Next-Fit and First-Fit. The rate-monotonic priority assignment algorithm [Liu73] is then used to ensure that each group of tasks are guaranteed to meet their deadlines. The rate-monotonic priority assignment algorithm assigns static priorities to tasks according to their periods. Higher priorities are assigned to tasks with shorter periods. The non-partitioning approach [Leung82] is the opposite of the partitioning approach in that tasks are not partitioned into groups. Instead, the processors are treated collectively as one entity with increased computing power to execute the entire task set. Each task is assigned a distinct priority and the processors execute requests of high-priority tasks before requests of low-priority tasks.

Suppose we want to tolerate one arbitrary processor failure. A naive solution to the scheduling problem is readily proposed. First, one of the scheduling heuristics proposed by Dhall and Liu in [Dhall78] is used to obtain the number of processors needed to execute the set of periodic tasks and their schedules. One redundant processor is introduced into the system in order to tolerate one arbitrary processor failure. We further assume that this redundant processor can be immediately informed of the failure of any other processors and it can execute any remaining tasks scheduled on the failed processor. Yet the failure of any processor can not be tolerated in the following case. Suppose that the deadline of a task is tight, i.e., the deadline of the task is equal to the sum of its starting time and the amount of time needed to execute the task. The processor executing this task fails in the middle of its execution. Even if the redundant processor is immediately informed of the failure and starts to execute the task immediately, there is not enough time left to execute the whole task on the redundant processor. The deadline of the task is thus missed.

Another possible solution to the general scheduling problem is to duplicate the tasks and schedule them on a set of processors. Obviously, for fault-tolerant purpose, the duplicated copies of a task should be scheduled on different processors and each of them should be finished before their deadlines. Due to the immense complexity involved in this scheduling algorithm, the feasibility of this solution is not known yet.

In this paper, a special case of the general scheduling problem is considered. We resolve to the primary-backup approach to tolerate one arbitrary processor failure in the worst case and up to $\lfloor m/2 \rfloor$ processor failures in the best case, where m is the number of processors needed to schedule the task set. The two scheduling algorithms are of polynomial time complexity and have the property of obtaining near-optimal solutions to the problem in both cases of failure and non-failure.

Assumptions: We assume that processors fail in the fail-stop manner and the failure of a processor can be detected by other processors. All periodic tasks arrive at the system in one cycle T, i.e., having the same period and are ready to execute any time within each cycle. We further assume that all periodic tasks have hard deadlines and their deadlines have to be met even in the presence of processor failures. We define a task's meeting its deadline as either its primary copy or its backup copy finishes before or at the deadline. Because the failure of processors is unpredictable and there is no optimal dynamic scheduling algorithm for multiprocessor scheduling [Dertou89], we focus on static scheduling algorithms to ensure that the deadlines of tasks are met even if some of the processors might fail. The scheduling problem can be rephrased using the framework of the general scheduling problem as follows:

The Scheduling Problem: A set of n periodic tasks $S = \{T_1, T_2, \ldots, T_n\}$ is to be scheduled on a number of processors. For each task i, there are a primary copy P_i and a backup copy B_i associated with it. The computation time of a primary copy P_i is denoted as C_i, which is the same as the computation time of its backup copy B_i. The tasks are independent of each other. The scheduling requirements are given as follows:

(1) Each task is executed by one processor at a time and each processor executes one task at a time.

(2) All periodic tasks should meet their deadlines. Aperiodic tasks have soft deadlines.

(3) Maximize the number of processor failures to be tolerated.

(4) For each task i, the primary task P_i or the backup B_i is assigned to only one processor for the duration of C_i and once it starts, it runs to its completion unless a failure occurs.

(5) The number of processors used should be minimized.

The deadlines of aperiodic tasks are assumed to be soft. However, as we will show later, the execution of aperiodic tasks are taken into account. Thus, in normal execution situation, aperiodic tasks are able to meet their deadlines. We further assume that all the processors are identical. Requirement (1) specifies that there is no parallelism within a task and within a processor. Requirement (2) dictates that the deadlines of periodic tasks should be met, maybe at the expense of more processors. Requirement (3) is a very strong requirement. The primary and backup tasks should be scheduled on different processors such that any one or more processor failure will not result in the missing of the hard deadlines of the periodic tasks. Furthermore, the primary copy and the backup copy of a task should not overlap each other, as we shall see in Lemma 2. Requirement (4) implies that tasks are not preemptive. A processor is informed the failure of other processors only at the end of the execution of a task. Also, care has to be taken to ensure that exactly one of the two copies of a task is executed during a cycle to minimize the wasted work. Requirement (5) states that the number of processors to be used to execute the tasks should be the smallest possible.

Since no efficient scheduling algorithm exists for the optimal solution of the fault-tolerant real-time multiprocessor scheduling problem as defined above, we resolve to a heuristic approach. Two heuristics based on a bin packing algorithm and LPT, are used to obtain approximate solutions. Before presenting the heuristics, we state the following Lemmas as the basic results upon which the two scheduling heuristics are developed.

Lemma 1: In order to tolerate one or more processor failures and guarantee that the deadline of all the periodic tasks are met using the primary-backup copy approach, the longest computation time of the tasks must satisfy the following condition: $(C_j = \max_{1 \le i \le n} \{C_i\}) \le T/2$, where T is the period of tasks.

Proof: Suppose that the deadline of the task T_j can still be met even if $C_j > T/2$. Suppose the processor which executes T_j fails at the time of $T/2$ and the backup task B_j is immediately started, then the finishing time of T_j is $BF_j = T/2 + C_j$. As $C_j > T/2$, we have $BF_j > T$, i.e., the deadline of the task is missed. This is a contradiction. Δ

Lemma 2: One arbitrary processor failure is tolerated and the deadlines of tasks are met with the minimum number of processors possible, if and only if the primary copy P_i and the Backup copy B_i of task i is scheduled on two different processors and there is no overlapping between them.

Proof: In [Lawler81], it is shown that a set of periodic tasks is schedulable on a multiprocessor if and only if there exists a valid schedule which is cyclic with a period T; i.e., each processor does exactly the same thing at time t as it does at time $t + T$. Therefore it suffices to consider the execution of tasks within a period T only. We first prove the necessary condition. Suppose one arbitrary processor failure is tolerated. It is evident that the primary copy of a task and its backup copy should be scheduled on two different processors. To prove that there is no overlapping between the primary copy of a task and its backup copy, we define BB_i as the beginning time of the backup copy B_i and FP_i as the finishing time of the primary copy P_i. If there is an overlapping between the primary copy of task i and its backup copy, then $FP_i - BB_i > 0$. Suppose the processor k on which the backup copy B_i of task i is assigned has no unused time within a period and the processor j on which the primary copy is executed fails at time $t > BB_i$. Processor k can only be notified of the failure of processor j no earlier than t. Thus the finishing time of the whole schedule of processor k is lengthened by $t - BB_i > 0$, resulting in a missed deadline. To prove the sufficient condition, we have that any pair of primary and backup copies are scheduled on two processors and there is no overlapping between them. Then the failure of any one of the two processors will trigger the execution of the backup tasks on another processor. Thus the deadline of the tasks will be met. Δ

IV. An Efficient Scheduling Algorithm

The basic idea of using primary-backup copy approach to tolerate processor failures is that there are two copies associated with each tasks, i.e., the primary copy and the backup copy. Once the primary copy fails, the backup copy is activated. Since the possible execution of the backup copies should also be finished before the deadline, enough time must be reserved on each processor to execute the backup copies. The reservation of enough time for the execution of backup copies implies that redundant processors have to be used to execute the primary task set earlier enough so that once a processor failure occurs, there will be time to execute the backup copies.

Our first scheduling algorithm is based on the First-Fit Decreasing (FFD) bin packing heuristic. As shown in Lemma 1, the computation times of all tasks should be less than half of the period in order to tolerate at least one arbitrary processor failure. Because the deadline of the tasks are known a priori to be T, T is used as the size of bins for the FFD heuristic. The scheduling algorithm proceeds as follows: First, the primary tasks are arranged in the order of decreasing computation times, denoted as $P_1, P_2, \ldots, P_n$. Second, the FFD heuristic is used to schedule the primary copies of the tasks into bins with size T. More specifically, we begin with one processor. Once the assignment of a task fails for the existing processors, a new processor is added. Tasks are assigned to processors in the order of their decreasing computation time. In other words, task P_i is scheduled before task P_j, where $i < j$. Task P_i is assigned to the lowest-indexed processor on which its finishing time is less than the period T. The schedule thus obtained is called the primary schedule. Let the number of processors required be m. It is apparent that though the tasks are schedulable to finish before the deadline, at least one of the tasks will miss its deadline if there is a failure. Therefore, the following steps are necessary. Third, the primary schedule is duplicated on another set of m processors to form the backup schedule. The tasks in the backup schedule are swapped based on the swapping rules to be defined below. Fourth, the tasks in the two schedules--primary and backup schedules are all renamed according to the following renaming rule, such that the primary schedule uses $2 \times m$ processors and precedes the backup schedule, and there is no overlapping between any pair of primary and backup copies of tasks.

By summarizing what we described above, we state the algorithm as follows.

```
procedure Heuristic1(Set of Tasks, Period T);
        Sort the set of tasks in the order of decreasing computation time and rename them
                $P_1, P_2, \ldots, P_n$;
        Apply FFD (First-Fit Decreasing) to assign the set of tasks into $m$ processors;
        Duplicate the schedule on $m$ backup processors to form the backup schedule;
        Applying swapping rules to the backup schedule;
        Applying the renaming rule to both the primary schedule and the backup schedule;
end Heuristic1;
```

In the following, we define the rules precisely and prove that a schedule produced by applying these rules can tolerate one arbitrary processor failure.

Definition 1: For the schedule on each processor, L_p is defined as the length of schedule less than or equal to half of the period T such that it is the sum of the computation times of those tasks whose finishing times are less than or equal to half of the period. L_q is the length of schedule for a processor. L_r is defined as the $L_q - L_p$. Obviously, $L_q \le T$ and $L_p \le T/2$, as illustrated in Figure A. From now on, where no confusion can be incurred, L_p is also used to denote the time interval whose length is L_p. L_q and L_r are also used in the similar manner.

In Figure 1, for example, L_p and L_q are equal to 5 and 9 respectively for processor 1. For processor 2, L_p is 4, and L_q is 9.

With the definition of L_p as above, the swapping rules for each processor in the backup schedule can be described as follows:

Swapping Rules:

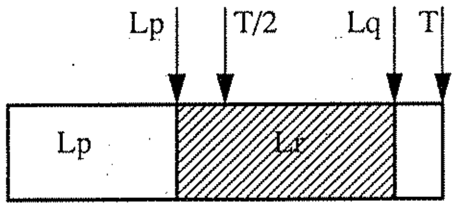


Figure A

(1) Tasks in L_p and tasks in L_r are swapped together.

(2) The first task in L_r starts at time 0. The starting times of the rest of the tasks in L_r follows.

(3) If $L_p > L_r$, the first task in L_p starts at time L_p. Since $L_p \le T/2$, the tasks in L_p can meet their deadlines. If $L_p \le L_r$, the starting time of the first task in L_p is L_r.

In Figure 1, for example, as $L_p = 5 > L_r = 4$, T_1 is swapped with T_2 with its starting time being $L_p = 5$. For processor 2, $L_p = 4 < L_r = 5$, tasks 4 and 5 are swapped with task 3. The starting time of T_3 is $L_r = 4$.

Having described the swapping rules, we are now ready to present the renaming rule.

Renaming Rule: In the primary schedule of each processor, rename all tasks whose finishing times are less than or equal to $T/2$ as primary tasks. The rest of the tasks are renamed as backup tasks. In other words, the tasks in L_p for each processor are renamed as primary tasks and the tasks in L_r are renamed as backup tasks. In the backup schedule of each processor, all tasks in L_r after swapping are renamed primary tasks and the tasks in L_p after swapping are renamed as backup tasks.

As an example, let us apply the scheduling algorithm to the following scenario. Suppose the period T is 10 time units and the set of tasks is given by $S = \{T_1, T_2, T_3, T_4, T_5\}$, with computation time $C = \{C_1, C_2, C_3, C_4, C_5\} = \{5, 4, 4, 3, 2\}$ .Because $C_1 \le T/2$, the set of tasks can be scheduled to meet their deadlines even though one of the processors may fail. Applying the FFD algorithm to the set of primary tasks with bin size equal to T, we obtain the schedule as shown in Figure 1. The number of processors used is 2. We double the number of processors and duplicate the schedule of primary tasks to form the backup schedule. The resulting schedule is shown in Figure 2. The reason the number of processors is doubled is to allocate enough number of processors to execute the primary tasks as well as the backup tasks once a failure occurs. In Figure 2, the schedules on processor 1 and 2 are called primary schedules, while the schedules on processors 3 and 4 are called backup schedules. Notice that the primary schedule and the backup schedule described here are different from the ones after the renaming process. For each duplicated schedule on a processor, tasks are swapped pivoted at L_p as defined in Definition 1. The renaming rule is then applied to the primary schedules as well as the backup schedules. The resulting schedule is shown in Figure 3. As shown by this example, merely swapping the tasks may not result in a valid schedule. Backup task B_1 can not be executed at time $t = 4$, because there is an overlapping between P_1 and B_1. The starting time of B_1 should be 5. This is guaranteed by the swapping rules. After the swapping process is done, the tasks and the primary schedules and backup schedules are all renamed such that the primary schedule now consists of all tasks in the task set and precedes the backup schedule. In Figure 3, the primary copy P_3 was originally the backup copy on the backup schedule on processor 4.

Before proving the following Theorem, we introduce the concept of twin processors.

Definition 2: Two processors are called twin processors if one's backup tasks are appended to the primary schedule of the other. The two schedules on twin processors are called twin schedules. For example, in Figure 3, processor 3 and 4 are twin processors.

Theorem 1: The swapping rules and the renaming rule as described above successfully transform the schedule into a schedule in which an arbitrary processor failure is tolerated and the deadlines of the tasks are guaranteed.

Proof: To prove that an arbitrary processor failure is tolerated, we only need to consider the case of two processors which are twin, because every processor has exactly one twin processor. By the swapping rules and the renaming rule, any pair of primary-backup copies is scheduled to run on two twin processors. For any two schedules which are twin, we shall prove that there is no overlapping between the primary copy of a task and its backup copy. The following two cases are considered.

The meaning of L_p and L_r used in the following is as defined in Definition 1.

Case 1: $L_p > L_r$, as shown in Figures B and C.

The tasks in L_r of the backup schedule are swapped with the tasks in L_p of the backup schedule according to the swapping rules. The starting time of the first task in L_p which becomes backup schedule after the renaming is L_p. Since the finishing time of the last task in L_p of the primary schedule is L_p, there is no overlapping between any pair of the primary and backup copies.

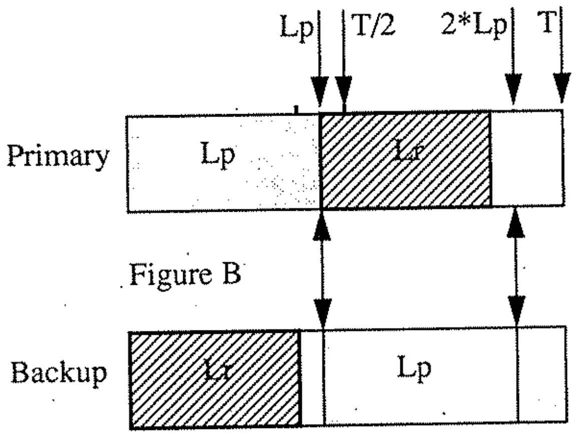


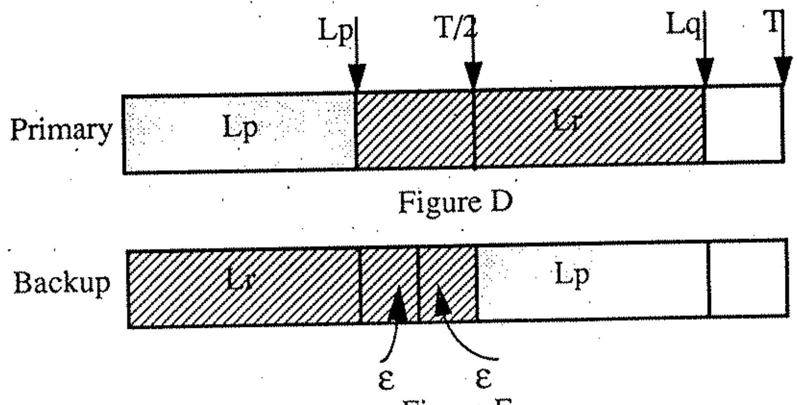
Figure B

Figure C

Figures B and C: Any twin schedules after swapping but before renaming

Case 2: $L_p \leq L_r$ as shown in Figures D and E.

The tasks in L_p are swapped with the tasks in L_r. First we claim that there are at least two tasks in L_r. Suppose there is only one task in L_r. Because $L_p < L_r$, i.e., the computation time of any task in L_p is shorter than the computation time of the only task in L_r, this contradicts the FFD algorithm for assigning tasks in the order of decreasing computation time to processors. Therefore there are at least two tasks in L_r. We further claim that if there is no overlapping between the first primary copy in L_r of Figure E and its backup copy in L_r of Figure D, there is no overlapping between the primary copy of any task and its backup copy on the twin processors. Suppose task w is one of the tasks, but not the first task in L_r of Figure E and its primary copy overlaps with its backup copy in L_r of Figure D. Then the computation time of task w must be longer than that of the first task in L_r. This again contradicts the rules used by FFD to assign tasks in the order of decreasing computation time to processors. Now suppose that the first primary task in L_r of Figure E overlaps with its backup task in L_r of Figure D for the length of $\varepsilon > 0$ time unit, then the computation time of this task is $L_p + \varepsilon > L_p$ which again contradicts the rule used by FFD to assign tasks to processors. We have shown that there is no overlapping between the primary copy of any task in L_r of Figure E and its backup copy in L_r of Figure D. Since $L_p \leq L_r$, the primary copy of any task in L_p of Figure D can not overlap with its backup copy in L_p of Figure E.



Figures D and E: Any twin schedules after swapping but before renaming

From the above two cases, it is clear that for any pair of twin processors, one arbitrary processor failure is tolerated and the deadlines of the tasks are guaranteed. Δ

V. Another Scheduling Algorithm

The number of processors required by the first scheduling algorithm depends on the performance of the FFD bin packing heuristic. In the worst case, the FFD bin packing algorithm finds the number of bins within 11/9 of the optimal number of bins required [Johnson73]. At the duplication step, the same number of processors are used to schedule the same set of tasks. This duplication step may not be optimal because the number of processors used to execute the same set of tasks twice subject to the constraints listed in Section III may be smaller than twice the number of processors used to execute the set of tasks once. Thus, there is some room for further improvement. The second scheduling algorithm is thus devised to minimize the number of processors required. The minimization of the number of processors is necessary as stated in the definition of the scheduling problem.

This scheduling algorithm first uses the FFD bin packing heuristic to find the number of processors necessary for tolerating at least one arbitrary processor failure and for guaranteeing the meeting of the deadlines of the tasks. Then the LPT heuristic is used to minimize the number of processors required until no further reduction in the number of processors is possible. The algorithm is given as follows:

Step1: The set of tasks are ordered according to their decreasing computation times and rename them $T_1, T_2, \ldots, T_n$. The FFD bin packing heuristic is used to schedule the primary set of tasks with bin size equal to $T/2$. The number of processors required is obtained as m.

Step2: Set $k = m - 1$ and apply LPT to schedule the primary set of tasks into k identical processors. The primary schedule is thus obtained.

Step3: Sort the primary schedule in the order of decreasing schedule length. Duplicate the primary schedule to form a backup schedule and append it at the end of the primary schedule.

Step4: Swap the backup schedule according to the swapping rules as defined below.

Step5: Shift the backup schedule according to the shifting rules as defined below to obtain the mixed schedule.

Step6: The maximum schedule length of mixed schedule is found and compared to T. If it is longer than T, then no further reduction in the number of processors required is possible. Therefore, m is the minimum number of processors necessary for the execution of the tasks. The algorithm terminates at this point. Otherwise, $m = m - 1$ and go to step 2.

The duplication and appending process is illustrated by Figure F.

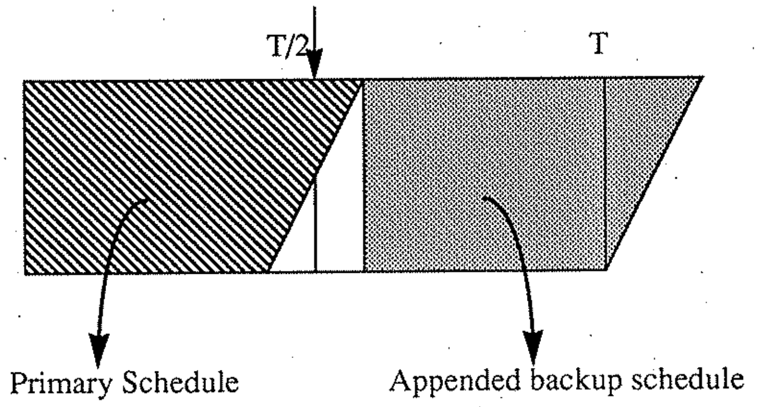


Figure F: Schedule after Appending

Swapping Rules:

(1) If the number of processors is even, the longest backup schedule is appended behind the shortest primary schedule, and the second longest backup schedule is appended behind the second shortest primary schedule and so forth.

(2) If the number of processors is odd, then the backup schedules of the three central processors are cyclically appended. The three central processors are the ones whose position is in the middle and its two adjacent neighbors. The backup schedules of the rest of the processors are swapped by following swapping rule (1).

The swapping process is illustrated as follows.

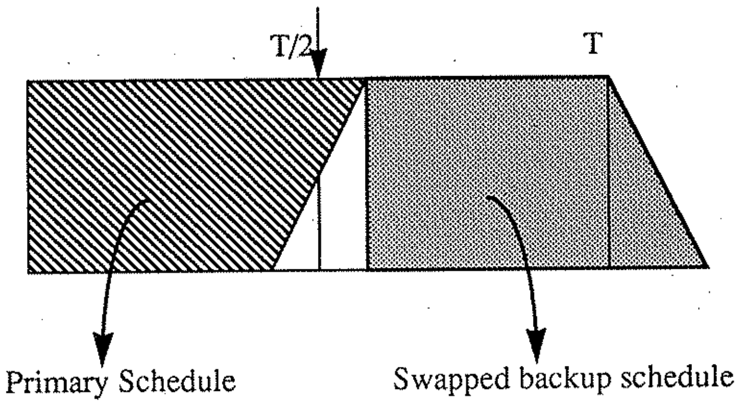


Figure G: Schedule after Swapping

Definition 3: For the primary schedule of a processor i, $L_p(i)$ is defined as its schedule length. $L_q(i)$ is defined as the computation time of the first task in the schedule. Obviously, $L_p(i) \geq L_q(i)$. Where no confusion can occur, we also use $L_p(i)$ to denote the time interval whose length is $L_p(i)$.

Shifting Rules:

Suppose the backup schedule of processor j is appended behind the primary schedule of processor i.

(1) If $L_p(i) \leq T/2$ and $L_p(j) \leq T/2$, then the tasks in $L_p(j)$ are shifted together ahead of time such that the starting time of the first task in $L_p(j)$ is $max\{L_p(i), L_p(j)\}$. Otherwise, the starting time of the first task in $L_p(j)$ is $L_p(i)$.

(2) If $L_p(i) \leq T/2$ and $L_p(j) > T/2$, the tasks in $L_p(j)$ are shifted together ahead of time such that the starting time of the first task in $L_p(j)$ is $L_p(i)$.

(3) If $L_p(i) > T/2$ and $L_p(j) \leq T/2$, the tasks in $L_p(j)$ are shifted together ahead of time such that the starting time of the first task in $L_p(j)$ is $L_p(i)$.

(4) If $L_p(i) > T/2$ and $L_p(j) > T/2$, the tasks in $L_p(j)$ are shifted together ahead of time

such that the starting time of the first task in $L_p(j)$ is $L_p(i)$.

(5) Apply the above rules to every schedule on the processors.

The shifting process of the swapped backup schedule is illustrated in Figure H.

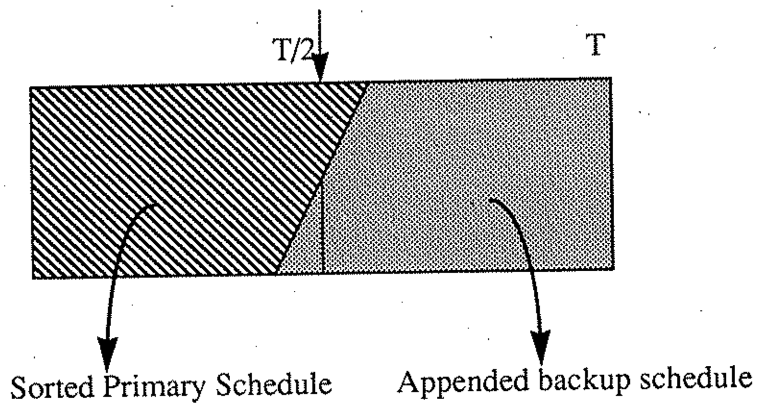


Figure H: Schedule after Shifting

The process of using this scheduling algorithm to schedule a set of three tasks is shown in Figure 4 through Figure 6. Figure 4 shows the result obtained after step 1. The result after Step 4 is shown in Figure 5, while the final result is shown in Figure 6.

After presenting the swapping rules and shifting rules, we are ready to prove the following lemma.

Lemma 3: There is no overlapping between the primary copy of any task and its backup copy after the shifting rules are applied.

Proof: In order to prove that no overlapping occurs between the primary copy of any task and its backup copy, we first prove that there is no overlapping occurring between any pair of primary and backup copies of the tasks on any twin processors as defined in Definition 2.

Suppose processors i and j are twin processors.

Case 1: $L_p(i) \le T/2$ and $L_p(j) \le T/2$. Because all backup copies of the tasks start no earlier than $max\{L_p(i), L_p(j)\}$ by the shifting rules, there is no overlapping between the primary copy of any task and its backup copy.

Case 2: $L_p(i) \le T/2$ and $L_p(j) > T/2$. Suppose that there is an overlapping between the primary copy of a task on processors j and its backup copy on processor i, then the computation time of the task is more than $L_p(i)$. Because $L_p(j) > L_p(i)$ and $L_p(j) > T/2$, there must be at least two tasks in the primary schedule on processor j (the computation time of any task is less than $T/2$ as stated before). This contradicts the rule LPT uses to schedule task, i.e., LPT always schedules the next task on the first available processor. The second task in $L_p(j)$ should have been scheduled after $L_p(i)$ according to the LPT rule. Thus, there is no overlapping between any primary copy on processor j and its corresponding backup copy on processor i.

Case 3: The proof is similar to that of Case 2.

Case 4: $L_p(i) > T/2$ and $L_p(j) > T/2$, the resulting schedule length of either processor i or processor j is longer than T. This can not happen in a valid schedule as guaranteed by Step 6 in the algorithm.

Having shown that no overlapping between any primary copy on processor i and its backup copy on processor j can occur, we need to consider the two cases where the number of processors used is odd and even.

If the number of the processors used is even, every processor has exactly one twin processor as guaranteed by the swapping rule. Because there is no overlapping between any pair of the primary and backup copies of the tasks on any twin processors, there is no overlapping between the primary copy of any task in the system and its backup copy.

If the number of the processors used is odd, except for the three central processors, every processor has exactly one twin processor. For the schedules on the three central processors, it turns out to be that there is no overlapping between the primary copy of a task and its backup copy also. This is readily proved by slightly modifying the proof of Case 2.

Therefore the Lemma is proved. Δ

Theorem 2: The scheduling algorithm as described above successfully finds the schedule in which an arbitrary processor failure is tolerated and the deadlines of the tasks are guaranteed.

Proof: By Lemma 3 we have no overlapping between any primary copy of a task and its backup copy. Since Step 6 in the scheduling algorithm ensures that any backup copy finishes before the deadline T, the theorem follows from Lemma 2. Δ

VI. Analysis and Performance Evaluation

It is apparent that both scheduling algorithms meet the scheduling requirements identified in Section 3, though two algorithms may differ in obtaining the number of processors required for the system to tolerate at least one arbitrary processor failure. In the worst case, only one processor failure can be tolerated. In the best case, up to $\lfloor m/2 \rfloor$ processor failures can be tolerated, where m is the total number of processors used. Though the main focus of our scheduling algorithms is to guarantee tasks with hard deadlines to meet their deadlines even in the presence of processor failures, tasks with soft deadlines still have ample time for execution if there is no processor failure or the number of processor failures is small. This is achieved through the scheduling of primary copies to finish around half of the period. Furthermore, the makespan of the primary schedule is near-optimal, as guaranteed by the guaranteed performance bounds of FFD and LPT. In other words, the primary schedule is the best we can hope for using existing scheduling algorithms of polynomial time complexity.

The time complexity of the first algorithm is $O(max\{nm, n\log n\})$, since the sorting algorithm takes $O(n\log n)$ and the rest of the algorithm takes $O(nm)$. The sorting process can dominate the running time only when $m = o(\log n)$.

The time complexity of second algorithm is $O(n\log n + n\log m + i(n + m\log m))$ where n is the number of tasks, i is the number of iterations taken between Step 2 through Step 6, and m is the number of processors. The number of processors decrements at each iteration, but it is bound by the number of processors first obtained by using FFD. More specifically, sorting the task set takes $O(n\log n)$ time. The FFD takes $O(nm)$ time, which can actually be done in $O(n\log m)$ time. The LPT takes $O(n)$ time. The sorting of the m schedules takes $O(m\log m)$ time. The appending, the swapping and shifting takes $O(m)$ time. As verified by experiments, the number of iterations i never exceeds 2. Thus the time complexity of the algorithm is again dominated by the sorting time.

Because the multiprocessor scheduling problem is known to be *NP*-complete, we are hopeless in finding an optimal solution to the problem even when the number of tasks is small (e.g. 10). Thus, we consider the most ideal case, which we call "best possible". The number of processors used in the most ideal case is the result of taking the ceiling of the result of dividing the sum of computation times of all the tasks (primary and backup) by the cycle. The performance of two scheduling algorithms and the "best possible" case is shown in Figure 9. The computation time of each tasks is randomly generated from the range of [1, 20]. The period T is 90 time units. For the first algorithm, there is only one processor difference in most of the cases. For the second algorithm, we could not find any difference from the best possible case in all the experiments we have done so far.

It seems that any set of tasks which can be scheduled by the first algorithm on m processors can be scheduled by the second algorithm with the same or less number of processors. One example where the first algorithm uses four processors while the second algorithm uses three processors is shown in Figures 7 and 8. Figure 7 is the result after applying the FFD to the set of tasks. The number of processors required for the primary schedule is two, and therefore, to tolerate one arbitrary processor failure, the number of processors required is four.

However, we have also found cases where the first scheduling algorithm performs better than the second scheduling algorithm. An example is given in Figure 10, where the task set consists of 12 tasks and the period T is equal to 39. The first scheduling algorithm finds the number of processors needed is 4 while the second scheduling algorithm finds the number of processors needed is 5. The first scheduling algorithm finds the best possible solution in this case. As this task set is very special and the probability of generating such a task set using a random number generator is very small, it is therefore not surprising that this case is not shown in Figure 9.

Even though we have not been able to obtain the exact guaranteed performance bounds for the two scheduling algorithms, we believe from extensive experiments that as the number of tasks increases, the scheduling algorithms can always find the near-optimal solutions, i.e., with maximum of one processor difference.

VII. Conclusion

In this paper, we have identified the general fault-tolerant real-time multiprocessor scheduling as the key problem in achieving fault-tolerance in hard real-time systems. Two efficient scheduling algorithms are proposed to solve a particular case of the general problem. Experiment results show that the scheduling algorithms find near-optimal solutions. We have also showed that one arbitrary processor failure can be tolerated by the schedules. Several examples are used to compare the two scheduling algorithms.

There are many open questions which needs to be answered in order to design extremely reliable hard real-time systems. The case where tasks have different periods in the general scheduling problem is still an open problem. Another open problem is the same scheduling problem but under the condition that the processors available are all uniform processors (the speeds of the processors have linear relations). These are the topics for our future research.

References

[Anders81] Anderson, T. and Lee, P.A., Fault Tolerance Principles and Practices, Prentice-Hall, International, London, England, 1981.

[Anders83] Anderson, T. and Knight, J.C., A Framework for Software Fault Tolerance in Real-time Systems, IEEE on Software Engineering, SE-9(3), May 1983, pp. 355-364.

[Balaji89] Balaji, S. et al., Workload Redistribution for Fault-Tolerance in a Hard Real-Time Distributed Computing System, FTCS-19, Chicago, Illinois, pp. 366-373, June 1989.

[Bannis83] Bannister, J.A. and K. S. Trivedi, K.S., Task Allocation in Fault-Tolerant Distributed Systems. Acta Informatica, 20, Springer-Verlag, 1983.

[Bettat89] Bettati, R. et al, Recent Results in Real-Time Scheduling, Technical Report, Department of Computer Science, University of Illinois at Urbana-Champaign, October 10 1989.

[Bertos91] Bertossi, A.A. and Mancini, L., Fault-Tolerant Task Scheduling in Multiprocessor Systems, Tech. Report, Universita di Pisa, Italy, 1991.

[Carlow84] Carlow, G.D., Architecture of the Space Shuttle Primary Avionics Software System, CACM, 27(9), September 1984.

[Chen78] Chen, L. and Avizienis, A., N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation, Digest FTCS-8: Eighth International Symposium on Fault Tolerant Computing, Tolouse, France, June, 1978, pp. 3-9.

[Chung87] Chung, J.Y., Liu, J.W.S. and Lin, K.J., Scheduling Periodic Jobs Using Imprecise Results, Tech. Report, University of Illinois, Nov. 1987.

[Coffma78] Coffman, E.G., Jr., Garey, M.R., and Johnson, D.S., An Application of bin-packing to Multiprocessor Scheduling, SIAM J. Computing, 7 (1978), pp. 1-17.

[Dertou89] Dertouzos, M. and Mok, A.K-L, Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks, IEEE Trans. on Computer, 15(12), December 1989, pp. 1497-1506.

[Dhall78] Dhall, S.K. and Liu, C.L., On a Real-Time Scheduling Problem, Operations Research, Vol. 26, 1978, pp. 127-140.

[Graham69] Graham, R.L., Bounds on Multiprocessing Timing Anomalies, SIAM J. Appl. Math., 17 (1969), pp. 416-429.

[Johnson73] Johnson, D.S., Near-Optimal Bin Packing Algorithms, doctoral thesis, MIT, 1973.

[Johnson89] Johnson, B.W., Design and Analysis of Fault Tolerant Digital Systems, Addison-Wesley, 1989.

[Karp72] Karp, R.M., Reducibility among Combinatorial Problems. In Complexity of Computer

Computations, R.E. Miller and J.W. Thatcher, Eds., Plenum Press, New York, 1972, pp. 85-103.

[Krishn86] Krishna, C.M. and Shin, J.C., On Scheduling Tasks with a Quick Recovery from Failure, IEEE Transactions on Computers, C-35(5), May 1986, pp. 448-454.

[Lala85] Lala, P.K., Fault Tolerant and Fault Testable Hardware Design, Prentice-Hall, International, London, England, 1985.

[Lawler83] Lawler, E.L. and Martel, C.U., Scheduling Periodically Occurring Tasks on Multiple Processors, Information Processing Letters, 12(1), 1981, pp. 9-12.

[Leung82] Leung, J.Y.T. and Whitehead, J., On the complexity of fixed-priority scheduling of periodic, real-time tasks, Performance Evaluation, Vol. 2, pp. 237-250, 1982.

[Liestm86] Liestman, A.L. and Campbell, R.H., A Fault Tolerant Scheduling Problem, IEEE Transactions on Software Engineering, SE-12(11), November 1986, pp. 1089-1095.

[Pradha86] Pradhan, D.K., Fault-Tolerant Computing -- Theory and Techniques, Volumes I and II, Prentice-Hall, Englewood Cliffs, N.J., 1986.

[Randel78] Randell, B., Lee, P.A., and Treleavan, P.C., Reliability Issues in Computing System Design", ACM Computing Surveys, Vol. 10, No. 2, 1978, pp. 123-166.

[Rennel84] Rennels, D.A., Fault-Tolerant Computing -- Concepts and Examples, IEEE Trans. on Computers, Vol. C-33, No. 12, December 1984, pp. 44-49.

[Siewio82] Siewiorek, D.P. and Swarz, R.S., THe Theory and Practice of Reliable System Design, Digital Press, Bedford, Mass. 1982.

[Stanko88] Stankovic,J.A., Misconception of Real-Time Computing, IEEE Computer, Oct. 1988, pp. 10-19.

[Ullman76] Ullman, J.D., Complexity of Sequencing Problems, Computer and Job/Shop Scheduling Theory, E.G. Coffman, Ed., John Wiley, New York, 1976, Chap. 4

[Zhao87] Zhao, W., Ramamritham, K., and Stankovic, J.A., Scheduling Tasks with Resource Requirements in Hard Real-Time Systems, IEEE Trans.on Software Engineering, Vol. SE-13, No. 5, May 1987, pp. 564-577.

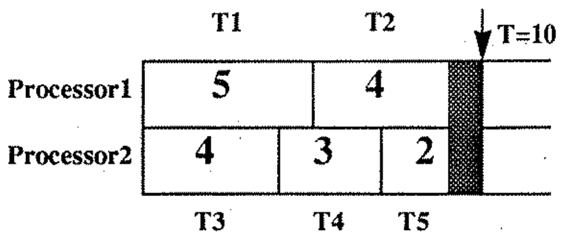


Figure 1

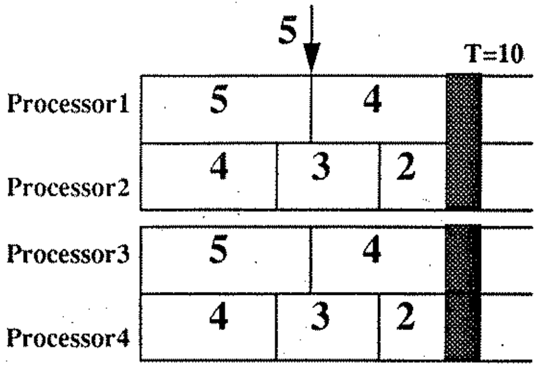


Figure 2

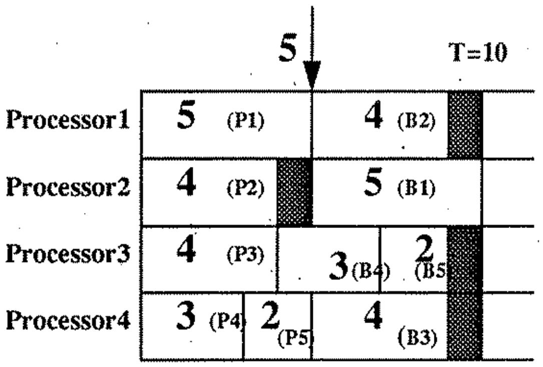


Figure 3

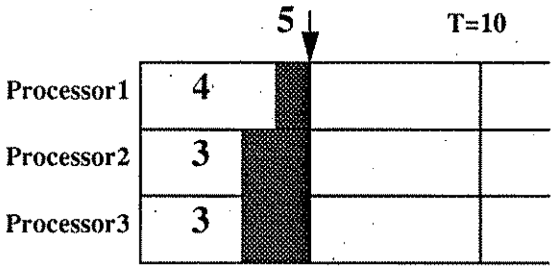


Figure 4

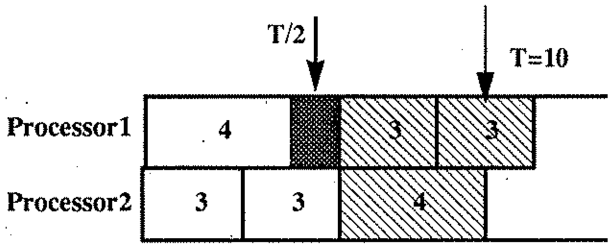


Figure 5

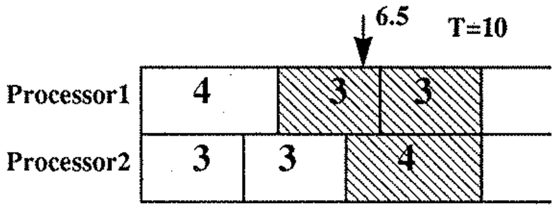


Figure 6

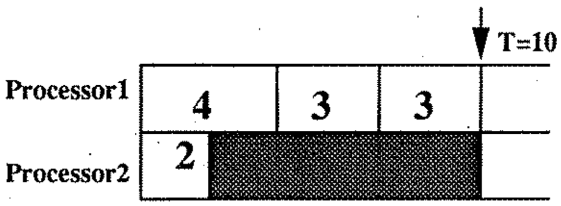


Figure 7

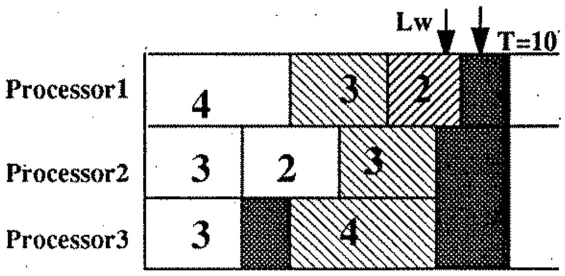


Figure 8

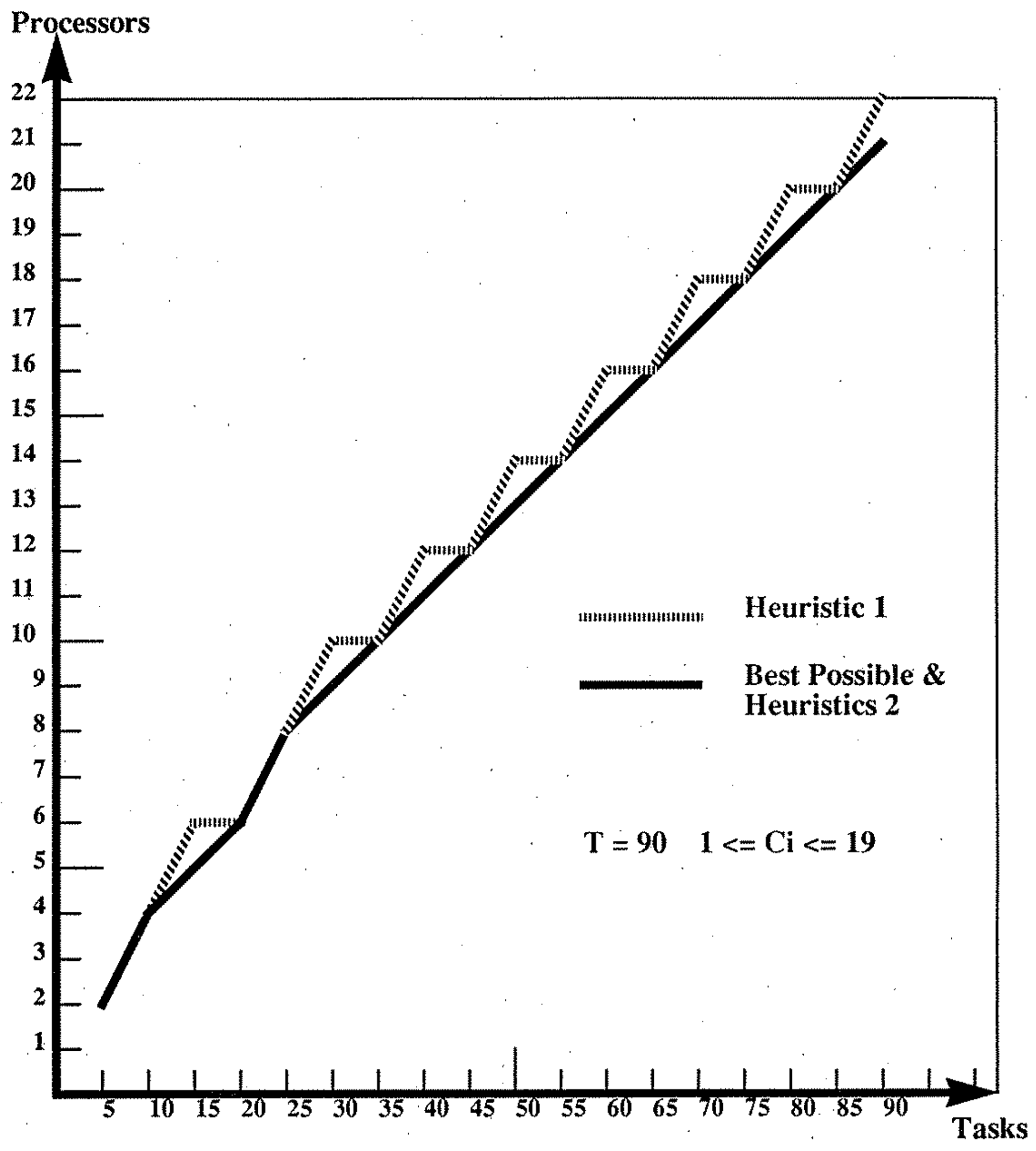


Figure 9: Comparison of Two Heuristics