Fault-Tolerant Real-Time Multiprocessor Scheduling

by

Anand Srinivasan
B.Sc (Hons), University of Delhi, New Delhi, 1986
MCA, Jawaharlal Nehru University, New Delhi, 1989
M.Sc, University of Victoria, Victoria, 1991

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY
in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. G. C. Shoja, Supervisor, Dept. of Computer Science.

Dr. E. G. Manning, Departmental Member, Dept. of Computer Science

---

Dr. M. Serra, Departmental Member, Dept. of Computer Science

Dr. F. El Guibaly, Outside Member, Dept. of Elect. & Comp. Eng.

Dr. V. Nagarajan, External Examiner, Systran Corp., Dayton, Ohio, USA.

Supervisor: Dr. G. C. Shoja

ABSTRACT

Real-Time systems are specialized computer systems where a set of tasks need to be executed within a specified deadline. The imprecise computation approach is a technique for robustness where the accuracy of tasks' computations are traded for meeting the timing constraints. Under this technique, every task is partitioned into two subtasks, namely mandatory and optional. In order to obtain an acceptable result, the mandatory subtask of every such imprecise task in the task set must be executed. The optional subtask, if executed, refines and improves the result produced by the mandatory part.

We first consider the problem of scheduling a set of independent tasks with identical ready times, arbitrary execution times and identical deadlines, for limited time available and fully available multiprocessor real-time systems. We extend our result for imprecise tasks by providing an optimal scheduler that schedules optionals in the maximum possible number of time slots. We propose an adaptive preemptive scheduler for scheduling external sporadic tasks in a multiprocessor real-time system, where the imprecise tasks have been scheduled at pre-run-time. During run-time we optimally schedule the external tasks using the proposed adaptive scheduler. The scheduler removes the minimum number of optionals from the pre-run-time schedule to accommodate the external tasks.

We present a model for representing fault-tolerant parallel programs in a real-time shared memory system. We model parallel programs, and derive the condition under which a program's deadline can be met. We apply a mechanism based on imprecise computation technique and provide an algorithm for deciding when to sacrifice result accuracy in order to meet the timing constraints of parallel programs.

Finally we address the problem of achieving fault-tolerant real-time service in a network of fault-tolerant servers. We use the primary site approach

where every service is replicated in several sites, one of which is selected as the primary. The other sites on hot standby are backups. When a primary fails, one of the backups takes over as primary while the failed primary joins the repair queue. Two types of repair services are considered, namely the delayed repair service and the immediate repair service. We analyze the average response time of the system when the repair server uses LCFS service discipline to service the failed sites. We compare the derived response time with the average response time of a system with a repair server that uses FCFS service discipline.

Examiners:

Dr. G. C. Shoja, Supervisor, Dept. of Computer Science

Dr. E. G. Manning, Departmental Member, Dept. of Computer Science

---

Dr. M. Serra, Departmental Member, Dept. of Computer Science

Dr. F. El Guibaly, Outside Member. Dept. of Elect. & Comp. Eng

---

Dr. V. Nagarajan, External Examiner, Systran Corp., Dayton, Ohio, USA.

Table of Contents

List of Figures

Acknowledgment

I would like to thank my supervisor, Dr. G. C. Shoda of the Department of Computer Science, for his encouragement, patience, and advice during the course of this research and during the preparation of this manuscript. He was always ready to discuss all sorts of problems, work or otherwise, and helped me to solve them. I would also like to thank the supervisory committee for their suggestions to improve the presentation of this thesis.

Financial assistance, received in the form of a fellowship from the University of Victoria, is gratefully acknowledged. I would like to thank Dr. Mike Miller for providing me with an opportunity to work as a lab instructor in the department.

Since the beginning of my graduate studies at UVic, I had made several friends. It is impossible to list all their names, but in particular, I would like to thank Pavan, Robert, Shankar and Vivek for always being there with me during difficult times.

Finally, I would like to thank my wife Smitha for appreciating my work and always being there with me during crucial stages of this thesis. I would also like to thank my family members for their sacrifice and encouragement.

To my parents

Venkataramana Srinivasan & Nirmala Srinivasan

and to

Guru Sri Raghavendra Swamy

Chapter 1

Introduction

1.1 Real-Time Systems

Real-time systems are generally defined as systems that must respond to external requests in a timely manner. Real-time systems are characterized by the fact that the acceptability of output depends not only on the logical correctness of the result, but also on the time at which it is available. Hence real-time systems are used in time-critical applications, such as command and control systems [1], nuclear power plants, process and flight control systems [2], space shuttle and aircraft avionics [3], and robotics [4]. Real-time computing is a wide open research area of challenging problems. Articles on the current state of the art in real-time computing can be found in [5,6,7,8].

A deadline of a real-time task can be defined as a timing constraint (specified in terms of time units) within which the task must be executed. Rajkumar [9] classifies real-time tasks into three categories depending on its arrival pattern and its deadline. They are as follows:

- a task deadline is classified as a *hard deadline* if it is critical to system functionality and must be met at all cost. For example, an exception task in a space shuttle or a nuclear power plant must be executed within a specified deadline, or the consequences might be catastrophic.

- In contrast a task deadline is classified as a *soft deadline* if it could be missed occasionally without leading to a catastrophic consequence. For example, in a telecommunication system, if a call does not go through the user needs to dial again. Similarly in a multimedia system, if video signals cannot be delivered within a deadline, jitter results. Therefore, the system does not fail but performance is degraded.

- The last category of tasks which may be executed in a real-time system is one with *no* specific deadline. External and sporadic background tasks that perform maintenance and testing activities typically fall into this category. We notice that the arrival times of maintenance tasks are most often not known in advance and they tend to execute at the same, lower or higher priority depending on their criticality. For example, maintenance tasks in telecommunication systems to reconfigure or repair remote systems could be as important as other tasks that are executing, but the arrival time of the maintenance tasks may not be known in advance.

In addition to the timing constraints, a task may also possess, among others, resource constraints, precedence constraints, concurrency constraints and communication constraints. A task may have resource constraints if it requires access to certain resources other than the CPU, such as I/O devices, data structures, files and databases. A complex task may be broken into multiple subtasks related by precedence constraints. Similarly, a real-time parallel program may be subdivided into multiple tasks where the tasks are represented in terms of a precedence graph. A task may have concurrency (mutual exclusion) constraints if several other tasks access the same set of resources that it accesses. A task might be using resources from various nodes or might be obtaining a result from another task, in which case the task is constrained by the communication delay.

With the advent of new technology and the availability of sophisticated processors, multiprocessor systems are often used to implement complex real-time applications. Although the available computation power has increased, the complexity of various problems related to real-time systems has not decreased. One of the major problems in real-time systems is to guarantee the execution of tasks within a specified deadline. Real-time schedulers play an important role in providing such a guarantee. It is extremely difficult, and often intractable, to obtain a general optimal scheduling algorithm for tasks with several constraints. Hence research on obtaining optimal scheduling algorithms is focused on sets of tasks with limited constraints. In this thesis, we focus on the task scheduling problem for multiprocessor real-time systems. In the following section we provide more insight into real-time scheduling problems.

1.2 Real-Time Scheduling

Research in scheduling has been well established in the areas of production engineering, operations research and management science [10,11,12]. Scheduling algorithms in these areas typically optimize metrics such as the total completion time, schedule length, and maximum lateness. Scheduling algorithms that minimize the schedule length can be useful for real-time systems if the tasks have a common deadline. Scheduling algorithms from other disciplines, that optimize metrics such as total completion time and maximum lateness, are in general not useful for real-time systems since they do not take into account the deadlines of tasks.

An optimal scheduling algorithm for a real-time system is one which guarantees a feasible schedule for any task set, if such a schedule exists. In other words, if an optimal scheduler fails to meet the deadline for any task set, then there cannot exist a scheduling algorithm that would meet the deadline for

that task set. Stankovic *et al.* [13] elaborate on some classical results in general scheduling theory that hold true for real-time scheduling. A selection of papers that focus on the state-of-the-art in real-time scheduling can be found in [14]. Casavant and Kuhl [15] provide a comprehensive literature survey on various real-time scheduling algorithms. Lawler *et al.* [12] provide an excellent survey on the complexity of various scheduling algorithms. We discuss in detail various issues pertaining to pre-run-time and run-time scheduling algorithms for multiprocessor real-time systems in the next chapter.

Three important characteristics for a real-time task are its *ready time*, *computation time* and *deadline*. Task scheduling in real-time systems can either be *static* or *dynamic*. A *static* approach calculates schedules for tasks at pre-run-time (before run time), and therefore requires complete prior knowledge of tasks' characteristics. Hence, algorithms for this approach are often classified as pre-run-time scheduling algorithms. A *dynamic* approach determines schedules for tasks at run-time and allows tasks to be dynamically invoked. Hence a dynamic scheduling algorithm has complete knowledge of currently active tasks only, and the characteristics of tasks that may arrive in future are known only upon their arrival. Such algorithms are often classified as run-time scheduling algorithms.

1.3 Fault-Tolerance in Real-Time Systems

The increasing application of computers to real-time control functions has created situations in which a computer failure could result in unacceptably high costs, either in terms of life or property. Such systems therefore, require a high degree of fault-tolerance. Fault-tolerance for real-time systems is provided either at node level or at task level. Node level fault-tolerance is provided by replicating hardware on which the application is implemented and defends against hardware malfunctions, whereas task level fault-tolerance

is provided by replicating tasks that are executed in the system and defends against some task malfunctions.

1.3.1 Architecture and Hardware

A critical design issue for any fault-tolerant system is redundancy management, i.e., the control of resources for fault-tolerance. Fault-tolerance in a real-time system is based on masking errors, either for the duration of the mission (static redundancy), or until the error can be isolated and the system reconfigured (dynamic redundancy). Static redundancy requires massive physical redundancy, but needs only simple redundancy management. Dynamic redundancy minimizes extra hardware and allows re-scheduling and reallocation of tasks. It also provides graceful reconfiguration, by excluding nodes as they fail and readmitting them as they are repaired.

To increase the reliability or availability of a system without sacrificing response time, fault tolerance techniques are employed for real-time systems. The primary site approach is a technique used for achieving fault-tolerance without sacrificing average response time of the system [16]. In the primary site approach, each of the systems to be made fault-tolerant is replicated and called a "site". One of the sites is elected as the primary and services all the task requests while the others remain on hot-standby as backups. The primary periodically checkpoints its state information on backups. If the primary fails, a backup takes over as the new primary and starts executing from the latest checkpoint while the failed primary joins a repair queue. Huang and Jalote [17] studied the availability and reliability of a primary site system. They also analyzed the average response time and its effect on checkpointing frequency in [18].

A landmark development in the search for extreme reliability in real-time control systems was the Software Implemented Fault-Tolerance (SIFT) com-

puter system [2] [19]. The SIFT project pioneered both theoretically provable fault-tolerance and system consistency. SIFT has a large system overhead since the fault-tolerance and executive functions are implemented in software on the same processor which performs the application tasks [20]. Scheduling for all tasks in SIFT is static [20] [21]. A contemporary alternative to SIFT was the Fault-Tolerant Multiprocessor (FTMP) architecture [22]. FTMP has hardware triple modular redundancy (TMR), priority based scheduling, exact voting and tight synchronization. A Multicomputer Architecture for Fault-Tolerance (MAFT) [23] is another system that provides extreme reliability without sacrificing performance, flexibility, or programmability. In MAFT, resource availability to application tasks is improved by separating system overhead from application functions.

1.3.2 Software Fault-Tolerance

Real-time applications require continuity of correctly computed output, and that implies correct performance of both hardware and software. Software fault-tolerance is achieved by incorporating redundancy in the tasks that run in the system. The Recovery Block Scheme pioneered by Horning *et al.* [24] and Randell [25] meets these requirements. In recovery block method, tasks are replicated and an acceptance test is conducted after execution of a task. If the task fails to satisfy the acceptance test then an alternate task is executed. Finally when no alternate tasks are available, an error routine is executed. Hecht [26] developed a technique for reliability analysis of a software system, illustrating the application of the recovery block to real-time programs by adding a timer module to monitor the execution of a task within its deadline. Deadline Mechanism [27] is a variation of the recovery block scheme where every task has two variations namely primary and alternate. The primary task, prone to miss deadlines provides an accurate result,

whereas the alternate task does not miss deadlines and provides an approximate result. N-Version Programming (NVP) is another well known software fault-tolerance technique where N versions of the tasks are executed and a voting is done on the results [28].

Imprecise computation technique was introduced as a part of real-time system named Concord, to provide task level fault-tolerance [29]. Two approaches namely, milestone approach and sieve approach were introduced in [30] to implement imprecise computation. The milestone approach periodically records intermediate results and accepts the latest recorded result when a deadline is reached. The sieve approach skips certain predefined sections of code, thereby trading precision for time.

Liu, Lin and Natarajan [31] decomposed the tasks as mandatory and optional subtasks to formalize the *imprecise computation approach.* In order to obtain an acceptable result the mandatory subtask of every task has to be executed. Optional subtasks may then be executed in full or in part if extra time remains. The optional subtask refines and improves the result produced by the mandatory subtask. If the optional subtask is fully executed, then we obtain an accurate result. A partial amount of an optional subtask can be executed to obtain an acceptable partially accurate result. A system that allows the implementation of imprecise computation approach to a task set is known as *imprecise computation system.* The imprecise computation technique finds applications in telecommunication [32], data base system [33, 34] and image and speech processing [35].

1.4 Fault-Tolerant Scheduling

In a hard real-time system, a timing fault is said to occur when a real-time process delivers its result too late. When redundancy is introduced into the software system to achieve fault-tolerance, the number of tasks to be executed

increases dramatically. An efficient algorithm is needed to schedule tasks in such a way that usage of additional resources is minimized. A fault-tolerant scheduler achieves a good balance between satisfying the timing constraints and achieving result accuracy.

Pre-run-time fault-tolerant scheduling is done with complete *a priori* knowledge of the task characteristics. The scheduler in MAFT follows the frame work for fault-tolerant software proposed by Anderson and Knight [36], where tasks are allocated to processors at pre-run-time based on their periodicity, precedence relationships and task priorities. Multiple versions of the tasks called *clones* were used by Krishna *et al.* [37] to achieve fault-tolerance. There are two types of clones, namely *primary* and *ghost*. A *primary* clone is executed normally, whereas the *ghost* clone is a backup copy which lies dormant until it is activated to take the place of a corresponding primary. A fault-tolerant scheduler based on this method was introduced to achieve quick recovery from failure. Liestman and Campbell [38] introduced an algorithm to schedule optimally a set of periodic tasks in a simply periodic system where the periodicity of tasks with lower period is an integral multiple of the periodicity of tasks with larger period.

Liu, Lin and Natarajan [31] introduced heuristics that minimized average error, to schedule a set of periodic tasks in a single processor imprecise computation system. Leung *et al.* introduced algorithms to minimize flow time [39] and the number of late tasks [40] in a single processor imprecise computation system. Chung *et al.* [41] introduced algorithms based on a rate monotonic scheduler to schedule a set of periodic tasks in a single processor imprecise computation system and compared the performance with other known algorithms optimal for general systems. A comprehensive survey of all the results on imprecise computation systems were provided in [42]. A scheduling algorithm for single processor imprecise computation sys-

tem based on *earliest deadline first* algorithm was introduced by Shih *et al.* in [43] and extended for on-line scheduling in [44]. Yu and Lin [45] proposed a scheduler for imprecise computation multiprocessor system to increase concurrency. Hull *et al.* [46] studied the problem of enhancing performance and dependability when imprecise computation technique was used. We will provide more insight into various results pertaining to multiprocessor systems in the next chapter.

1.5 Objectives and Methods

In this thesis we address the problems of achieving optimal real-time scheduling in imprecise computation multiprocessor systems. We use imprecise computation techniques to achieve fault-tolerance. Real-time systems generally consist of tasks that we have *apriori* information about, as well as tasks that may have to be invoked at run-time. Of particular interest to our study is the problem of scheduling sporadic external tasks that arrive during run-time.

We will attempt to extend already proven scheduling algorithms for the new models we define, or introduce and prove new ones. We will consider both pre-run-time as well as run-time scheduling problems keeping in mind the practicality and applicability of our work.

1.6 Structure of the Thesis

The remainder of the thesis is organized as follows. Chapter 2 gives background on multiprocessor scheduling. Chapter 3 introduces our proposed pre-run-time scheduling algorithm for general and imprecise computation systems. Chapter 4 introduces the run-time scheduling algorithms. Chapter 5 deals with adaptive run-time scheduling. Chapter 6 considers the scheduling problem where tasks have precedence constraints. Chapter 7 considers

a similar problem for parallel program. Chapter 8 provides the response time analysis for a system that has primary site approach as fault-tolerance technique and in chapter 9 we conclude the thesis.

Chapter 2

Related Work

A real-time task is specified by its ready time, execution time and deadline. Finding efficient schedulers for a set of tasks with timing constraints has been an important field of research [13]. A pre-run-time scheduler constructs a schedule with complete *a priori* knowledge of ready times, execution times and deadlines of all tasks that need to be scheduled. A run-time scheduler schedules a set of tasks whose ready times and execution times are known only on their arrival.

Two approaches exist to constructing a schedule for a set of tasks in a multiprocessor real-time system. They are the *bin packing* algorithms and the *level* algorithms. The *bin packing* approach assigns as many tasks as possible to processor P_1 before considering processor P_2, and so on. The *level* approach incrementally, time unit by time unit, builds up a schedule on all processors. *Level* algorithms are in general better since the *bin packing* approach fails when the processor speeds are not identical [47].

In this chapter, we describe some of the work which is closely related to the problems addressed in this dissertation. We present results pertaining to pre-run-time scheduling, run-time scheduling and scheduling a set of tasks with precedence constraints. In this thesis we confine ourselves to scheduling algorithms that allow preemption.

2.1 Pre-Run-Time Scheduling

Pre-run-time schedulers for a set of independent tasks in a single processor real-time system have been exhaustively researched [13]. In the field of operations research, similar problems have been attacked by minimizing the schedule length. Jackson in his seminal work [48], now known as Jackson's rule, introduced an optimal scheduler for a set of tasks with identical ready times, where the tasks are scheduled in the order of increasing deadlines. Horn [49] extended Jackson's rule for tasks with arbitrary ready times but with unit execution times by scheduling at any time an available job with the smallest deadline. Horn also provided an algorithm to schedule a set of tasks with arbitrary ready times allowing preemptions. Liu and Layland [50] introduced two scheduling algorithms, namely the *rate monotonic* scheduler and *earliest deadline first* scheduler, for tasks that are periodic and tasks that have arbitrary deadlines, respectively. Since then, efforts have been made to find efficient algorithms with various combinations of deadlines, ready times and execution times [12].

Pre-run-time schedulers for multiprocessor real-time systems are in general more complex than schedulers for single processor real-time systems [13]. In order to schedule a set of tasks during pre-run-time, the ready time, execution time and the deadline of the tasks need to be known *apriori*.

2.1.1 Bin-Packing Approach

The *bin packing* approach has been successfully used by McNaughton [51] and Rothkopf [52] to construct shortest preemptive schedules for a set of independent tasks on multiprocessor systems. Liu and Liu [53] and Liu and Yang [54] provide algorithms to schedule a set of independent tasks on a multiprocessor system with $m - 1$ processors with identical speed and one

processor with faster speed.

McNaughton [51] provides an algorithm for finding the shortest preemptive schedule for a set of independent tasks on m processors, where all tasks have identical ready times, arbitrary execution times and identical deadlines. McNaughton provided a lower bound on the length of an optimal schedule and then constructed a schedule that matched this bound. The maximum completion time of any schedule is said to be at least $max\{\overset{max}{i} c_i, \frac{\sum_{i=1}^{n} c_i}{m}\}$ where m is the number of processors and c_i is the execution time of the ith task.

In McNaughton's rule, tasks are assigned to the processors successively by scheduling the tasks in any order and splitting the task whenever the above time bound is met. Consider a three processor computer system consisting of five tasks $J = \{J_1, \cdots, J_5\}$ with computation times $\{c_1, \cdots, c_5\}$, where c_1=8, c_2=7, c_3=6, c_4=5, and c_5=4. The maximum completion time, derived from the bound mentioned above for the tasks is at least $max\{8, \frac{30}{3}\}$ which is 10. Figure 2.1 illustrates a schedule constructed using McNaughton's rule by allocating the tasks into one processor at a time and preempting a task if the bound is reached.

Schmidt [55] considered the problem of scheduling a set of independent tasks on a multiprocessor system using the *bin packing* approach, where the processors are only available in certain given time intervals. Figure 2.2 illustrates a multiprocessor system with three processors P_1, P_2 and P_3. Processor P_1 is not available in the 3rd, 4th and 10th time units, Processor P_2 is not available at the 1st, 6th, 7th and 9th time units and Processor P_3 is not available at the 1st, 5th, 6th, 7th, 8th and 9th time units, respectively.

To solve the feasibility problem, Schmidt generated an equivalent problem from the original problem by moving all the unavailable time slots to the right hand side as shown in Figure 2.3. The equivalent problem may have fewer

$C_1 = 8$

$C_2 = 7$

$C_3 = 6$

$C_4 = 5$

$C_5 = 4$

	P1	P2	P3
1	J1	J2	J3
2	J1	J2	J4
3	J1	J2	J4
4	J1	J2	J4
5	J1	J2	J4
6	J1	J3	J4
7	J1	J3	J5
8	J1	J3	J5
9	J2	J3	J5
10	J2	J3	J5

Figure 2.1: Scheduling using McNaughton's rule

	P1	P2	P3
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			

Figure 2.2: Three processors with partial time slots available in 10 time units

available processors as compared to the original problem since all the time slots of some of the processors in the right hand side could be unavailable. Schmidt scheduled the tasks using the following algorithm by comparing the computation time of the task to the remaining processing capability (RPC) in the processors. If there exists a processor with RPC matching the computation time of the task, then the task is scheduled in that processor. If there does not exist such a processor, then the task is scheduled in the processor that has the largest RPC among those processors with RPC less than the computation time of the task. The remaining computation time of the task is scheduled in the processor that has the smallest RPC among those processors with RPC greater than the remaining computation time of the task. Since the RPC of the processor is greater than the computation time, the task can be scheduled in some time interval within the available time slots. Schmidt does not address the problem of where in the available time slots of the processor should the task be scheduled.

Schmidt's algorithm is as follows:

1. If $c_i > RPC(\overline{P_j}), \forall j$ then no feasible solution exists for the task set.

2. If $c_i \leq RPC(\overline{P_j}), \forall j$ then schedule task J_i on processor $\overline{P_k}$, with

$$RPC(\overline{P_k}) = \min_j \{RPC(\overline{P_j})\}$$

3 If neither holds, schedule task J_i on processor $\overline{P_k}$ with

$$RPC(\overline{P_k}) = \max_{j|RPC(\overline{P_j}) \leq c_i} \{RPC(\overline{P_j})\}$$

and on processor $\overline{P_l}$, if it exists, with

$$RPC(\overline{P_l}) = \min_{j|RPC(\overline{P_j}) > remainingtimeofc_i} \{RPC(\overline{P_j})\}$$

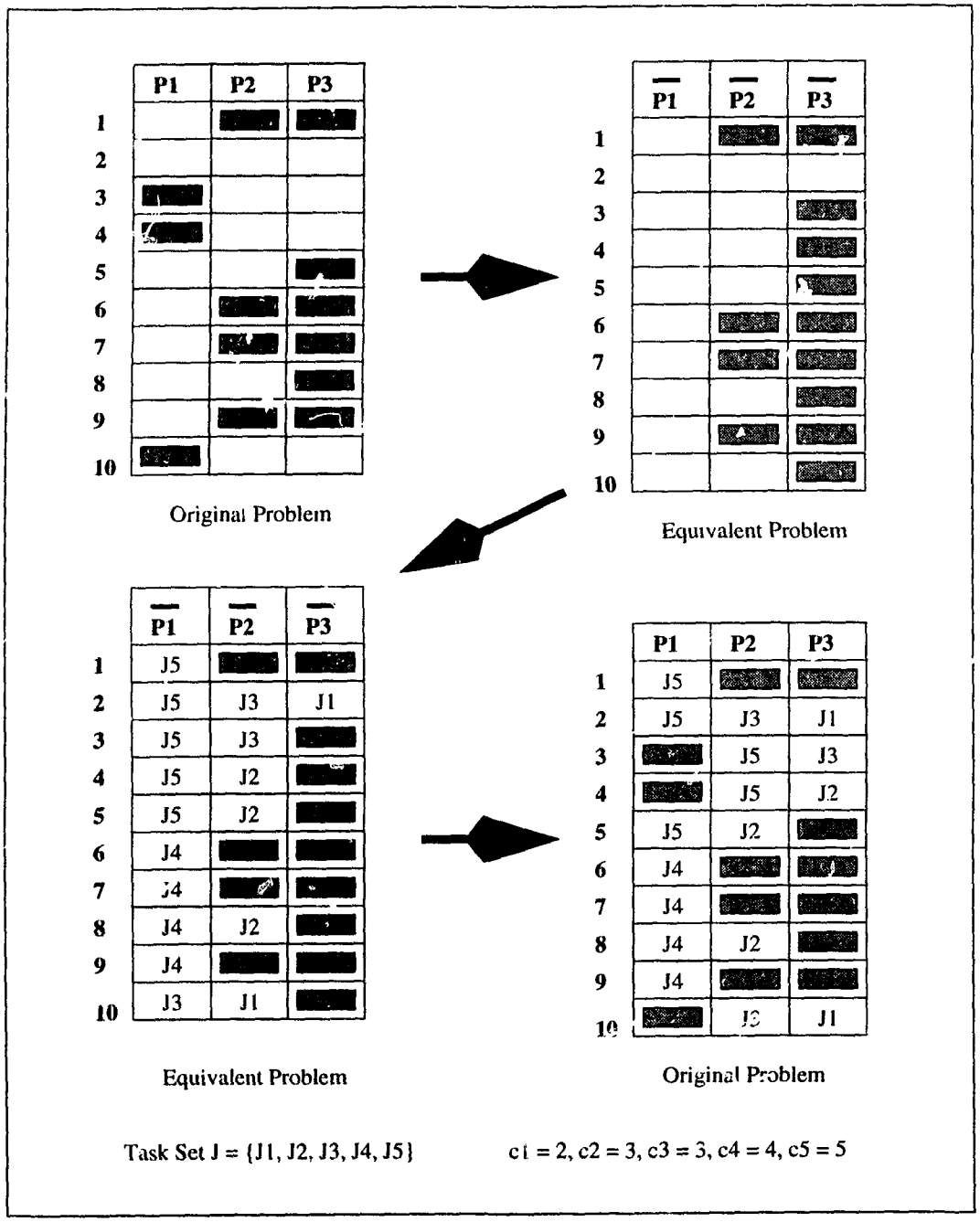


Figure 2.3: Equivalent problem of original problem

Figure 2.3 illustrates the transformation of the original problem into an equivalent problem and the application of the algorithm to a system with five tasks. Consider for example the three processor system shown in Figure 2.4. We illustrate two different equivalent problems with one time slot interchanged. We see that for the first equivalent problem, scheduling bottom first works where as the top first fails. For the second equivalent problem the bottom first fails and the top first succeeds. Hence, in order to schedule the tasks in a particular time slot, all the time slots of other processors in the same time unit need to be checked to ensure that the same task is not scheduled. This is due to the fact that a task cannot be scheduled in a time slot, if it is already scheduled in the same time unit on a different processor. Thus the complexity of the algorithm increases. The second drawback with Schmidt's algorithm is that one must construct an equivalent problem from the original problem, then schedule the equivalent problem and transform it back. We show later in this thesis that such a situation does not arise in our proposed algorithm.

In some real-time applications such as telecommunication, image and speech processing, a partial result of a task is acceptable if the task's execution could not be completed in its entirety [35]. The *imprecise computation approach* has been proposed as a means to partition the task into two subtasks, namely mandatory and optional [30,31]. In order to obtain an acceptable result the mandatory subtask of every task has to be executed. Optional subtasks may then be executed in full or in part if extra time remains. The optional subtask refines and improves the result produced by the mandatory subtask. If the optional subtask is fully executed, then we obtain an accurate result. Partial execution of each optional subtask can be done to obtain an acceptable partially accurate result. A system that allows the application of imprecise computation approach to its task set is known as an *imprecise computation system.*

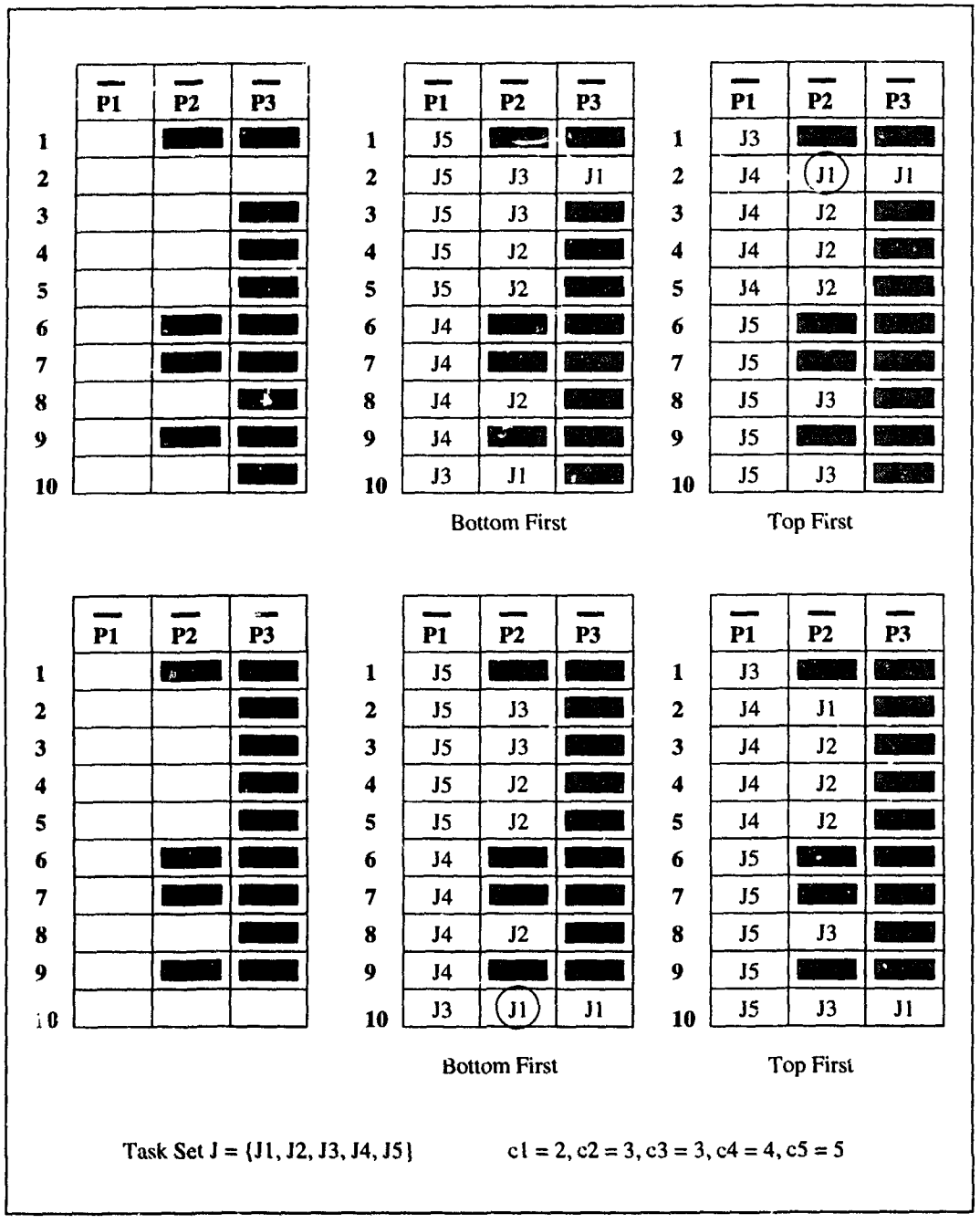


Figure 2.4: Conflict while ordering tasks within a processor

Scheduling algorithms for a single processor imprecise computation system typically minimize the total [42], and the average [56] execution time of the unexecuted portion of the optionals. Shih *et al.* [56] also considered the problem of scheduling a set of independent tasks to minimize average execution time of unexecuted portion of optionals in a multiprocessor system. Chung and Liu [57] proposed several heuristic algorithms to schedule periodic tasks and minimize the average execution time of unexecuted portion of the optionals. Leung and Wong [40] proposed an algorithm to schedule a set of tasks in a multiprocessor imprecise computation system to minimize the total number of tasks that miss the deadline. Khemka *et.al.* [58] introduced a scheduler based on network flow technique for a multiprocessor imprecise computation systems where tasks are periodic.

In this thesis, we propose pre-run-time and run-time multiprocessor scheduling algorithms for a set of independent tasks that have identical ready times, arbitrary execution times and identical deadline. We extend our algorithm for imprecise computation system by minimizing the total execution time of the unexecuted portions of the optionals.

2.1.2 Level Algorithms

Scheduling algorithms that incrementally construct a schedule on all processors are defined as *level* algorithms. Muntz and Coffman [59] provide an algorithm based on the *level* approach to construct an optimal preemptive schedule for tasks with tree precedence constraints. If there are m identical processors, the Muntz-Coffman algorithm assigns for every time unit, one processor to each of the m tasks farthest from the root of tree. If there is a tie among b tasks for the last a machines then $\frac{a}{b}$ of a processor is assigned to each of b tasks.

The extension of the Muntz-Coffman algorithm to m processors of dif-

ferent speeds was considered by Horvath, Lam and Sethi in [47]. Gonzalez and Sahni [60] provide an algorithm to achieve the same result with a minimum number of preemptions. Lam and Sethi [61] consider the extension of Muntz-Coffman algorithm for arbitrary precedence constraints and prove that the ratio of the schedule lengths of Muntz-Coffman algorithm and an optimal schedule is bounded by $2 - \frac{2}{m}$.

2.2 Run-Time Scheduling

A run-time scheduling algorithm is used to schedule a set of tasks, if the ready times of some of the tasks in the task set are available only during run-time. Such tasks are in general known as *sporadic tasks* or *external tasks.* To find an optimal on-line scheduler for a set of tasks is in general considered to be difficult due to unpredictable ready times.

Dertouzos and Mok [62] addressed the problem of dynamically scheduling hard-real-time tasks in a multiprocessor system and proved that it is impractical to design optimal run-time schedulers if there exist preemption due to resource sharing or general precedence constraints between tasks. Mok [63] proves that well known optimal on-line scheduling algorithms for single processors such as the *earliest deadline first* algorithm [48], *rate monotonic scheduler* [50] and the *least laxity first* algorithm [64], do not remain optimal when extended for multiprocessor systems. Dhall and Liu [65] provide some heuristic algorithms for scheduling a set of tasks that are periodic in a multiprocessor system.

Sahni and Cho [66] introduced an optimal nearly on-line scheduler where the ready time of the next task is known when the present task is scheduled. Sahni [67] showed that no optimal nearly on-line scheduler can exist if tasks in the task set have different deadlines. Hong and Leung [68] extended Mc-

Naughton's rule for a set of tasks with varied ready times but one common deadline. They proved that if the set of tasks have more than one distinct deadlines then no optimal scheduling algorithm exists. In Hong-Leung algorithm, tasks that are ready are scheduled initially using McNaughton's rule. Tasks that arrive during run-time are scheduled using a *re-schedule* algorithm that re-schedules the existing schedule to accommodate run-time tasks. Hence this algorithm cannot be extended for tasks with precedence constraints since it is difficult while re-scheduling to maintain the precedence relation of the tasks in existing schedule.

2.3 Scheduling Tasks with Precedence Constraints

The problem of scheduling tasks with precedence constraints on a multiprocessor system is known to be NP-complete in its general form [69,70]. Polynomial time algorithms exist only for a few restricted cases [12]. Hu [71] provided a linear time algorithm to schedule a tree structured task graph with all tasks having unit execution times. Coffman and Graham [72] introduced a polynomial time algorithm that provides an optimal schedule length for a two processor system. Muntz and Coffman [59] described a polynomial time scheduling algorithm that minimizes the completion time for a tree structured task graph allowing preemption. Gonzalez and Johnson [73] improved the performance and provided an $O(n^2)$ algorithm.

The intractability of obtaining an optimal solution for the general scheduling problem has led to the introduction of a large number of scheduling heuristics. One class of scheduling heuristics in which many schedulers are classified is *list scheduling.* In list scheduling, whenever a processor is available, a task that is ready with the highest priority is selected from the list of unscheduled tasks. Lam and Sethi [61] adapted Muntz-Coffman algorithm

for a general task graph in identical processors and proved that

$$\frac{SL(MC)}{SL^*} \leq 2 - \frac{2}{m}, \quad m \geq 2$$

where $SL(MC)$ is the schedule length obtained due to Muntz-Coffman algorithm and SL^* is the optimal schedule length. Horvath, Lam and Sethi [47] extended the Muntz-Coffman algorithm for uniform processors and proved that the algorithm guarantees the following relation:

$$\frac{SL(MC)}{SL^*} \leq \sqrt{\frac{3m}{2}}$$

2.4 Summary

We prove that the largest remaining time first algorithm is optimal for a set of tasks with identical ready time, arbitrary execution time and identical deadline. Level algorithms have not been actively researched for imprecise computation system. We propose a *level* algorithm for scheduling a set of tasks in a multiprocessor imprecise computation system and minimize the total execution time of unexecuted portion of optionals. We also consider the problem of scheduling of a set of tasks with precedence relation at pre-run-time and propose a heuristic algorithm.

Our method schedules tasks that arrive during run-time over the existing pre-run-time schedule. Hence scheduling of tasks that arrive during run-time is done in the time intervals that are not occupied by tasks in the pre-run-time schedule. We derive the necessary and sufficient condition for scheduling a set of tasks during run-time when the system is available only in a certain given time intervals. We propose a *level* algorithm for scheduling a set of tasks during run-time and prove its optimality. We also propose an adaptive run-time scheduler that schedules a set of tasks during run-time by removing

	Multiprocessor Preemptive Scheduling (Identical Deadline) Independent Tasks		
	Pre-run-time	**Run-time**	**Run-time-adaptive**
Bin-packing	**McNaughton [51]**	**Schmidt [55]**	**Hong & Leung [68]**
Level	**Muntz-Coffman [59]**	[redacted]	[redacted]
Imprecise Computation Technique	[redacted]	[redacted]	[redacted]

Figure 2.5: Areas that are addressed in this dissertation

optionals from pre-run-time schedule, such that the total execution time of portion of optionals that were removed is minimum. We extend our result to schedule optimally a set of tasks during run-time in a multiprocessor imprecise computation system, by minimizing the total execution time of unexecuted portion of optionals.

Finally, Figure 2.5 summarizes the previous work in this area and identifies the specific areas that are addressed in this thesis.

Chapter 3

Pre-Run-Time Scheduling

Pre-run-time schedulers construct feasible schedule for a set of tasks with complete knowledge of their task characteristics. Task characteristics such as ready time and deadline are known prior to start of execution of real-time system.

In this chapter, we first consider the problem of scheduling a set of independent tasks with same ready times on multiprocessor systems. We derive the necessary and sufficient conditions to obtain a feasible schedule for a set of tasks within a given deadline. We then introduce an algorithm based on *level* approach and prove its optimality. The problem of scheduling a set of tasks that are implemented using imprecise computation technique is considered. We extend McNaughton's rule to provide a *bin-packing* based scheduling algorithm that maximizes the total computation time of optionals. We then introduce a *level* based scheduling algorithm to maximize the total computation time of optionals.

3.1 Task Characteristics

We consider a multiprocessor system with m identical processors of the same speed. We denote the set of processors as $P = \{P_1, P_2, \cdots, P_m\}$. Let $J = \{J_1, J_2, \cdots, J_n\}$ be the n independent tasks with $\{c_1, c_2, \cdots, c_n\}$ being their

respective computation times. Let D be the deadline within which the tasks in J need to be executed. A schedule is nothing but a time line divided into time units, where tasks are assigned. *Ready time* of a task J_i, namely $RT(J_i)$, is defined as the earliest time unit in which a task can be scheduled. Similarly, *deadline* of a task set J namely D, is defined as the latest time unit in which a task in the task set can be executed, assuming that all tasks have the same deadline. Since this chapter deals with pre-run-time scheduling, we assume that all the tasks are ready to be scheduled at the start. Hence $RT(J_i) = 0, \forall i, \quad 1 \leq i \leq n$. We also assume that two tasks cannot be executed in the same time slot and a task cannot be executed by more than one processor in the same time unit. We assume a task can be stopped on one processor and resumed on another.

Every task in task set J that has been implemented using imprecise computation approach has been decomposed into two subtasks, namely mandatory subtask and optional subtask. Let $\{M_1, \cdots, M_n\}$ be the mandatory subtasks and $\{O_1, \cdots, O_n\}$ be the optional subtasks of $\{J_1, \cdots, J_n\}$. Let $\{m_1, \cdots, m_n\}$ and $\{o_1, \cdots, o_n\}$ be the computation times of $\{M_1, \cdots, M_n\}$ and $\{O_1, \cdots, O_n\}$ respectively. Hence the computation time of a task J_i, namely c_i, is given by $m_i + o_i$. We assume that an optional subtask O_i cannot be scheduled in the same time unit as mandatory subtask M_i. We also assume that mandatory subtasks have to be scheduled in their entirety to obtain an acceptable result, where as optional subtasks can be partially scheduled to obtain a better quality result.

3.2 Scheduling Independent Tasks

In the previous chapter we described in detail the work by McNaughton [51] to obtain a feasible schedule for J with the shortest completion time. McNaughton used ***bin-packing*** technique for obtaining a bound for the shortest

completion time for a task set J. In this section we first derive the necessary and sufficient conditions for scheduling a task set J within a deadline D. We then introduce an algorithm based on *level* approach, namely Largest Remaining Time First (LRTF) and prove its optimality.

We state the following well known theorems directly from [51].

Theorem 1 *The maximum completion time for a task set J, with computation times $\{c_1, \cdots, c_n\}$ in a multiprocessor system with m processors is*

$$max\{\max_i c_i, \frac{\sum_{i=1}^{n} c_i}{m}\}$$

Theorem 2 *McNaughton's rule is optimal.*

We now establish the necessary and sufficient conditions for scheduling J within a deadline D.

Lemma 1 *Let $\{J_1, \cdots, J_n\}$ be the task set with $\{c_1, \cdots, c_n\}$ being their respective computation times. J can be scheduled within D in m processors if and only if*

$$c_i \leq D, \quad \forall i, \quad 1 \leq i \leq n \tag{3.1}$$

$$\sum_{i=1}^{n} c_i \leq mD \tag{3.2}$$

Proof. In order to schedule J within D, the individual computation time of all the tasks in J should be less than or equal to D. Otherwise, the assumption that a task cannot be scheduled in two different time slots in the same time unit, will be violated. Similarly $\sum_{i=1}^{n} c_i \leq mD$, since mD is the total number of time slots available and the cumulative execution times of all the tasks should be less than or equal to mD. Hence conditions 3.1 and 3.2 are necessary.

The shortest completion time for J with computation time $\{c_1, \cdots, c_n\}$ by McNaughton's rule is given by $max\{\overset{max}{i} c_i, \frac{\sum_{i=1}^{n} c_i}{m}\}$. If J has to be executed within D then,

$$max\{\overset{max}{i} c_i, \frac{\sum_{i=1}^{n} c_i}{m}\} \leq D$$

We can derive condition 3.1 from

$$\{\overset{max}{i} c_i\} \leq D$$

We can also derive condition 3.2

$$\frac{\sum_{i=1}^{n} c_i}{m} \leq D \quad \Rightarrow \quad \sum_{i=1}^{n} c_i \leq mD.$$

Hence the conditions are sufficient.

□

We have thus proved that if the computation times of the individual tasks do not exceed the deadline and the total computation times of all tasks do not exceed the available time slots then a feasible schedule for the task set exists.

Consider the LRTF algorithm illustrated in Figure 3.1. In every time unit, m tasks with largest remaining computation times are scheduled in m processors. Unlike McNaughton's algorithm which follows *bin-packing* approach, LRTF algorithm follows the *level* approach where tasks are scheduled in all the processors at any given time. We now prove that LRTF algorithm constructs a feasible schedule for a task set J within D, if J satisfies the necessary and sufficient conditions.

Theorem 3 *Let $J = \{J_1, \cdots, J_n\}$ be the set of tasks with computation times $\{c_1, \cdots, c_n\}$. If J satisfies the necessary and sufficient conditions 3.1 and 3.2, then LRTF algorithm feasibly schedules J within D.*

```
Algorithm LRTF;
// m ← Number of processors;

for (every time unit t  |  1 ≤ t ≤ D )
begin
    Schedule m tasks with largest remaining computation times;
    Decrement their respective task computation times;
end;
```

Figure 3.1: Largest Remaining Time First algorithm

Proof. Let $\{c_1, \cdots, c_n\}$ satisfy the following necessary and sufficient conditions

$$c_i \leq D, \quad \forall i, \quad 1 \leq i \leq n \tag{3.3}$$

$$\sum_{i=1}^{n} c_i \leq mD \tag{3.4}$$

We prove the theorem using induction. Assume that $c_1 \geq c_2 \cdots \geq c_n$. Let the first time unit be scheduled using the largest remaining computation times of $\{c_1, \cdots, c_n\}$. We will show that the remaining computation times of J satisfy the conditions for [2..D]. The first time unit consists of m time slots, one slot per processor, and hence $\{J_1, \cdots, J_m\}$ are scheduled. After scheduling the first time unit, the remaining computation times of $\{J_1, \cdots, J_n\}$ are $\{c_1 - 1, c_2 - 1, \cdots, c_m - 1, c_{m+1}, \cdots, c_n\}$. The remaining computation times need to be allocated within $D - 1$. It is easy to see from condition 3.3 that

$$c_i - 1 \leq D - 1, \quad \forall i, \quad 1 \leq i \leq m$$

We wish to prove that

$$c_i \leq D - 1, \quad \forall i, \quad m + 1 \leq i \leq n$$

We know from condition 3.3 that

$$c_i \leq D, \quad \forall i, \quad m+1 \leq i \leq n$$

We claim that

$$c_i \neq D, \quad \forall i, \quad m+1 \leq i \leq n$$

If any c_i's, where $m+1 \leq i \leq n$ had been equal to D, then our assumption that $c_1 \geq c_2 \cdots \geq c_n$ would require that $c_1, \cdots, c_r$ all be equal to D. This violates condition 3.4. Hence our claim is valid. Therefore we have proved the base case, that the remaining computation times of J satisfy the necessary and sufficient conditions for [2..D], if the first time slot is scheduled using LRTF algorithm.

Let us assume that time units in [1..t-1] time interval have been scheduled using LRTF algorithm. Let $\{c_1^t, c_2^t, \cdots, c_n^t\}$ be the remaining computation times of tasks in J such that $c_1^t \geq c_2^t \geq \cdots \geq c_n^t$. We assume that the remaining computation times of J satisfies the following necessary and sufficient conditions for the time interval [t..D].

$$c_i^t \leq D-(t-1), \quad \forall i, \quad 1 \leq i \leq n \tag{3.5}$$

$$\sum_{i=1}^{n} c_i^t \leq m\{D-(t-1)\} \tag{3.6}$$

Our goal is to show that the remaining computation times of J satisfies the conditions for [t+1..D] after scheduling the tth time unit. Before scheduling the tth time unit, the remaining computation times of J in descending order are

$$\{c_1^t, c_2^t, \cdots, c_m^t, c_{m+1}^t, \cdots, c_n^t\}$$

After scheduling the m time slots in tth time unit, the remaining computation times are

$$\{c_1^t - 1, c_2^t - 1, \cdots, c_m^t - 1, c_{m+1}^t, \cdots, c_n^t\}$$

From condition 3.5 we can derive that

$$c_i^t - 1 \leq D - t, \quad \forall i, \quad 1 \leq i \leq m$$

We need to show that

$$c_i^t \leq D - t, \quad \forall i, \quad m+1 \leq i \leq n$$

From condition 3.5 we know that

$$c_i^t \leq D - (t-1), \quad \forall i, \quad m+1 \leq i \leq n$$

Hence we need to show that

$$c_i^t \neq D - (t-1), \quad \forall i, \quad m+1 \leq i \leq n$$

If $c_i^t = D - (t-1)$ for some $i, m+1 \leq i \leq n$, then $\{c_1^t, c_2^t, \cdots, c_m^t\}$ will all be $D - (t-1)$ time units since $c_1^t \geq c_2^t \geq \cdots \geq c_n^t$. This leads to the following inequality

$$\sum_{i=1}^{n} c_i^t > m\{D - (t-1)\}$$

since more than m tasks have $D - (t-1)$ as remaining computation time. This contradicts condition 3.6 and hence $c_i^t \neq D - (t-1)$. Therefore the first condition holds true for the task set J with remaining computation times in time interval [t+1..D].

For proving the second condition, we know from condition 3.6 that

$$\sum_{i=1}^{n} c_i^t \leq m(D-t) + m$$

Hence it is easy to see that

$$\sum_{i=1}^{m}(c_i^t - 1) + \sum_{i=m+1}^{n} c_i^t \leq m(D-t)$$

Therefore the LRTF algorithm feasibly schedules any task set that satisfies the necessary condition 3.3 and sufficient condition 3.4.

□

Figure 3.2 illustrates the application of LRTF algorithm to a three processor system. We consider a set of five tasks $J = \{J_1, \cdots, J_5\}$ with computation times $\{8, 7, 6, 5, 4\}$. We note that the deadline $D = 10$. It can be seen that while scheduling the 2nd, 4th, 6th, 7th, 8th and 9th time units tasks were interchanged to order them in descending order.

Theorem 4 *The maximum completion time for task set J when scheduled using LRTF algorithm is* $max\{\overset{max}{i} c_i, \frac{\sum_{i=1}^{n} c_i}{m}\}$

Proof. The total time slots available for scheduling is mD, since there are m processors and D time slots per processor. From the previous theorem, we know that $c_i \leq D$ and hence, $\sum_{i=1}^{n} c_i \leq n.D$. Therefore

$$\frac{\sum_{i=1}^{n} c_i}{m} \leq D \tag{3.7}$$

$$c_i \leq D \tag{3.8}$$

Hence, if LRTF algorithm is used, the time taken to complete the task set J, will be at least $max\{\overset{max}{i} c_i, \frac{\sum_{i=1}^{n} c_i}{m}\}$.

□

Theorem 5 *LRTF algorithm is optimal,* i.e., *LRTF algorithm produces a feasible schedule for J within D if such a schedule exists.*

Proof. We know that McNaughton's wrap around rule is optimal. The completion time for a task set J in m processors for McNaughton's wrap around rule is given by $max\{\overset{max}{i} c_i, \frac{\sum_{i=1}^{n} c_i}{m}\}$. Hence from the previous theorem we can note that if McNaughton's rule schedules J so will LRTF algorithm.

□

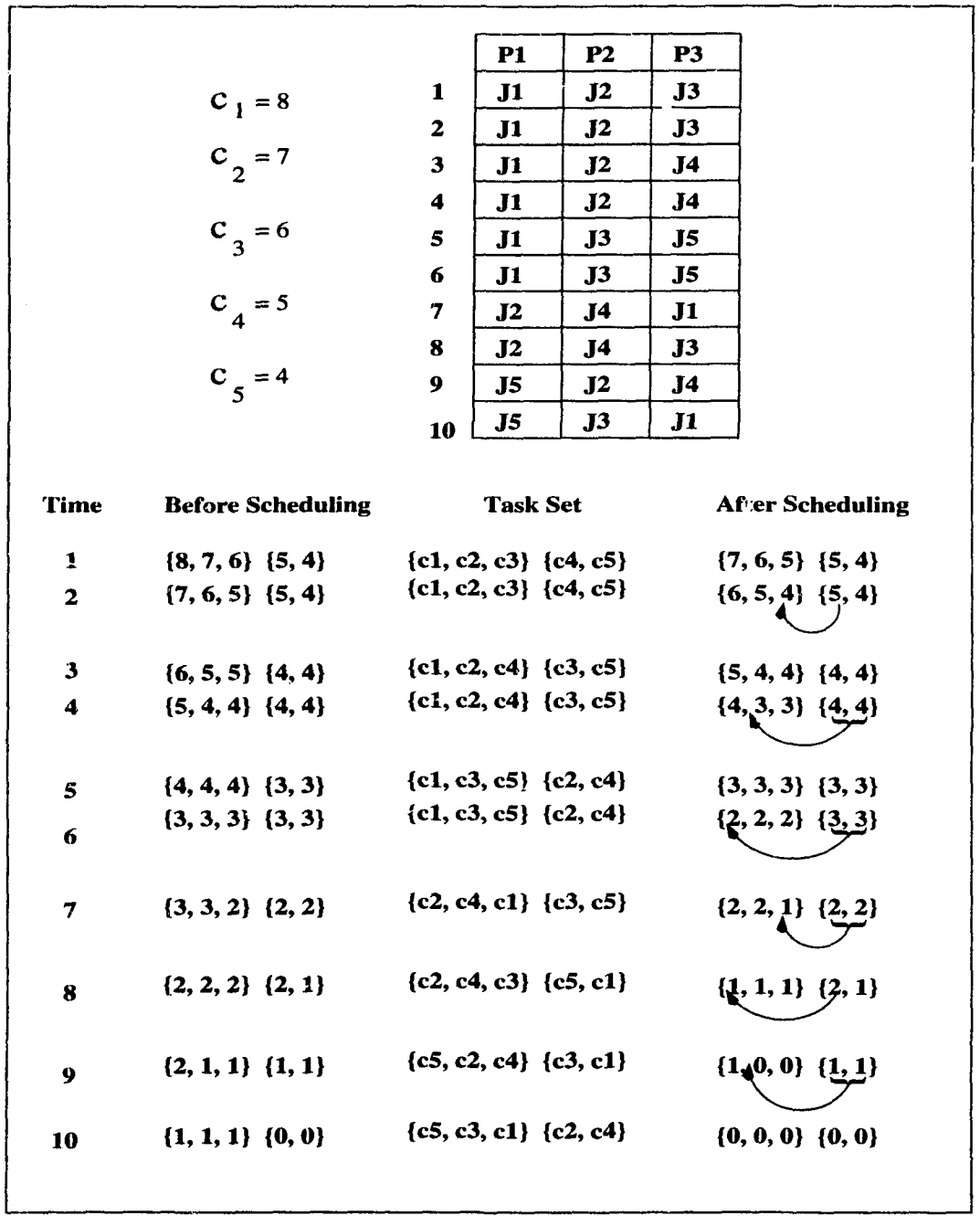


Figure 3.2: Illustration of LRTF algorithm

Therefore LRTF algorithm optimally schedules a set of independent tasks using *level* approach. In the next section we consider the problem of scheduling a set of tasks with imprecise computation technique for fault-tolerance.

3.3 Fault-Tolerant Scheduling of Independent Tasks

3.3.1 Bin-Packing Approach

We extend McNaughton's algorithm for scheduling a set of tasks with same ready times, arbitrary execution times and same deadlines, to systems using imprecise computation technique. In order to get a feasible schedule we need to schedule $\{M_1, \cdots, M_n\}$ within deadline D. Therefore $\max\{m_1, \cdots, m_n\}$ should be less than or equal to D since a task cannot run in the same time unit in two different processors. We observe that the total time slots available in m processors is mD. Thus the total execution times of all the mandatories, namely $\sum_{i=1}^{n} m_i$ should be less than or equal to mD in order to schedule all the mandatory tasks. Hence $\sum_{i=1}^{n} \frac{m_i}{m} \leq D$. We are now in a position to state the following Lemma.

Lemma 2 *We can obtain an acceptable result in an m processor imprecise computation system if,*

$$max\{\overset{max}{i} m_i, \frac{\sum_{i=1}^{n} m_i}{m}\} \leq D$$

The algorithm to optimally schedule $\{J_1, \cdots, J_n\}$ is given in Figure 3.3. We first schedule the mandatory subtasks in any order using McNaughton's rule, *i.e.,* tasks are scheduled in the first processor and then scheduled in the second and so on. Once all the mandatory subtasks are scheduled, optional subtasks are scheduled after the corresponding mandatories. This can be done until either all slots are scheduled, in which case some optional subtasks

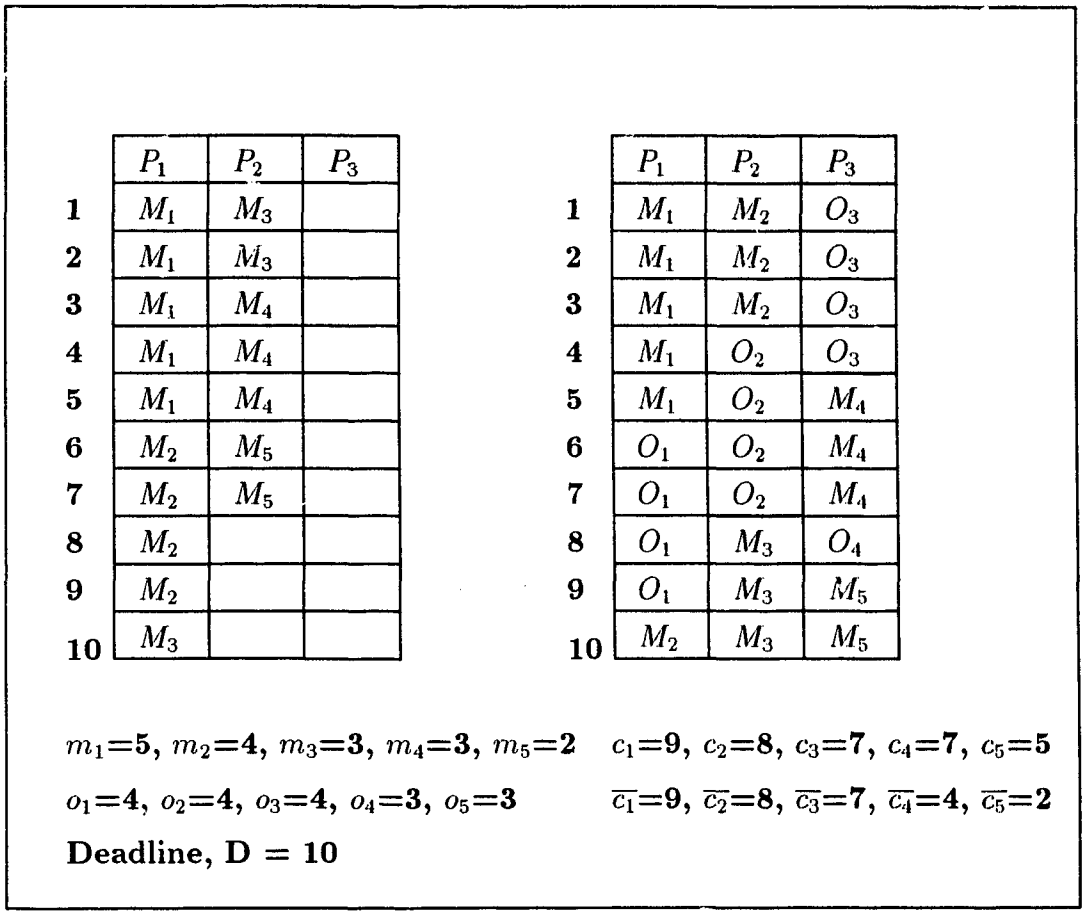
```
Algorithm Bin-Packing-Fault-Tolerant-Scheduler;
begin
    /* Assumption m_i + o_i ≤ D */
    if (max{ max_i m_i, (Σ_{i=1}^{n} m_i)/m } ≤ D)

    begin
        Schedule all mandatories using McNaughton's rule;
        Schedule optionals after corresponding Mandatories until
            either (1) all the slots are filled;
            or (2) all the optionals are scheduled;
    end;
    else "No Optimal Schedule" ;
end.
```

Figure 3.3: Bin packing fault-tolerant scheduler

may remain unscheduled or all optionals are scheduled, in which case some time slots may remain unoccupied. We claim that the algorithm is optimal.

Theorem 6 *Extended McNaughton's rule is optimal for imprecise computation system.*

To prove the correctness, we construct a single time line of mD time slots and schedule the mandatory subtasks in the time slots $[1..mD]$ in any order. The maximum number of empty slots available after scheduling all the mandatory subtasks is given by $mD - \sum_{i=1}^{n} m_i$. We schedule the optional subtasks of the corresponding mandatory subtasks after the mandatory subtasks are finished. If M_k is scheduled from [t..t+m_k], then we start scheduling



	P_1	P_2	P_3
1	M_1	M_3	
2	M_1	M_3	
3	M_1	M_4	
4	M_1	M_4	
5	M_1	M_4	
6	M_2	M_5	
7	M_2	M_5	
8	M_2		
9	M_2		
10	M_3		

	P_1	P_2	P_3
1	M_1	M_2	O_3
2	M_1	M_2	O_3
3	M_1	M_2	O_3
4	M_1	O_2	O_3
5	M_1	O_2	M_4
6	O_1	O_2	M_4
7	O_1	O_2	M_4
8	O_1	M_3	O_4
9	O_1	M_3	M_5
10	M_2	M_3	M_5

m_1=5, m_2=4, m_3=3, m_4=3, m_5=2 c_1=9, c_2=8, c_3=7, c_4=7, c_5=5

o_1=4, o_2=4, o_3=4, o_4=3, o_5=3 $\overline{c_1}$=9, $\overline{c_2}$=8, $\overline{c_3}$=7, $\overline{c_4}$=4, $\overline{c_5}$=2

Deadline, D = 10

Figure 3.4: Example of bin packing fault-tolerant scheduler

O_k in the time interval

$$[t + m_k + 1, \quad t + m_k + min\{o_k, mD - (\sum_{i=1}^{n} m_i + \sum_{i=1}^{k-1} o_i)\}],$$

by moving all the mandatory subtasks scheduled after M_k, $\min\{o_k, mD - (\sum_{i=1}^{n} m_i + \sum_{i=1}^{k-1} o_i)\}$ time slots ahead. Note that $mD - (\sum_{i=1}^{n} m_i + \sum_{i=1}^{k-1} o_i)$ is the number of empty slots available after scheduling $\{O_1, \cdots, O_{k-1}\}$. This is continued until either all the optional subtasks are scheduled or all the empty time slots are occupied. Then we partition the $[0..mD]$ time slots into m time intervals [0..D],[D+1..2D],$\cdots$,$[(m-1)D..mD]$ and use it as a

schedule for m processors. Note that any task in the task set is not scheduled at two time slots in the same time unit since we divide the time units exactly into intervals of length D time slots and $m_i + o_i \leq D$.

We see that the algorithm schedules

$$min\{mD - \sum_{i=1}^{n} m_i, \sum_{i=1}^{n} o_i\}$$

time slots for optional subtasks. If

$$mD - \sum_{i=1}^{n} m_i < \sum_{i=1}^{n} o_i,$$

then the algorithm schedules the first r optional subtasks, entirely after their corresponding mandatory subtasks and then schedules partial amount of $(r+1)$st optional subtask in the empty slots available, namely $mD - (\sum_{i=1}^{n} m_i + \sum_{i=1}^{r} o_i)$. Therefore,

$$\sum_{i=1}^{n} min\{o_i, mD - (\sum_{r=1}^{n} m_r + \sum_{r=1}^{i-1} o_r)\} = mD - \sum_{i=1}^{n} m_i$$

If $mD - \sum_{i=1}^{n} m_i \geq \sum_{i=1}^{n} o_i$ then the algorithm schedules $\{O_1, \cdots, O_n\}$, since the number of empty slots available are more than the total execution times of all optional subtasks. Hence we arrive at the following equation.

$$\sum_{i=1}^{n} min\{o_i, mD - (\sum_{r=1}^{n} m_r + \sum_{r=1}^{i-1} o_r)\} = \sum_{i=1}^{n} o_i$$

It can be easily seen that the algorithm constructs a schedule in a time, linear to the number of tasks. Hence we have proved the following theorem.

Figure 3.4 illustrates an example of imprecise computation system for three processors, where the mandatories are scheduled using our extension of McNaughton's rule. Figure 3.5 shows the various stages of the algorithm while scheduling the optionals. The final schedule after partitioning the time intervals and reassigning to processors, can also be seen in Figure 3.4.

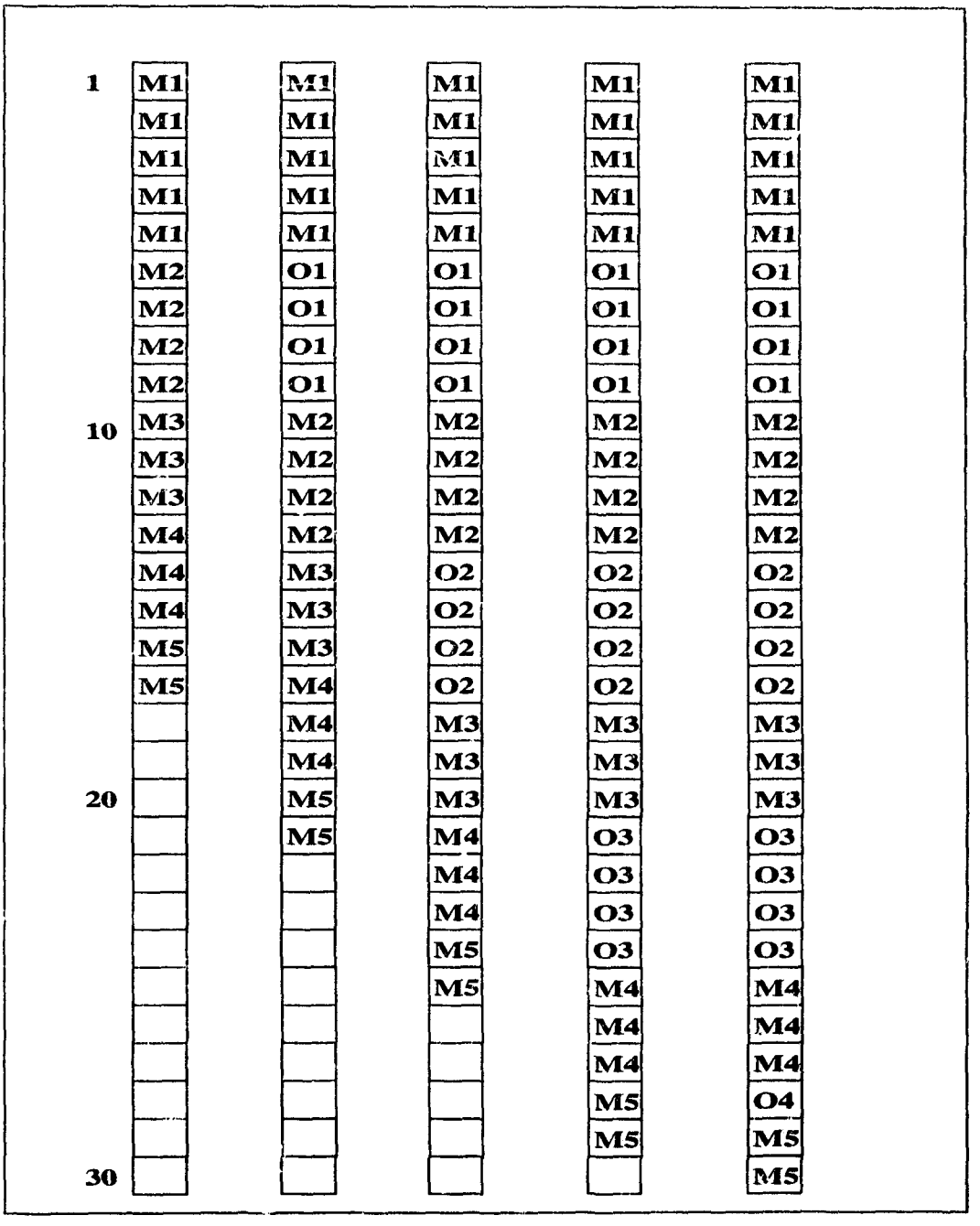


Figure 3.5: Extended McNaughton's rule for imprecise computation systems

3.3.2 Level Approach

We introduce a *level* approach based algorithm for obtaining a schedule that maximizes the number of optionals that can be scheduled. Let $J = \{J_1, \cdots, J_n\}$ be the tasks with $\{M_1, \cdots, M_n\}$ as mandatory subtasks and $\{O_1, \cdots, O_n\}$ as optional subtasks. Let $\{m_1, \cdots, m_n\}$ be the respective computation times of mandatory subtasks and $\{o_1, \cdots, o_n\}$ be the computation times of optional subtasks. In order to obtain an acceptable result, all the mandatory subtasks need to be executed within deadline D. Hence the necessary condition to obtain a feasible schedule for $\{J_1, \cdots, J_n\}$ is given by

$$m_i \leq D, \forall i, \quad 1 \leq i \leq n \tag{3.9}$$

$$\sum_{i=1}^{n} m_i \leq mD \tag{3.10}$$

If a feasible schedule is guaranteed for J then our goal is to maximize the total execution times of all optional subtasks that can be scheduled within D. Note that we are allowed to schedule part of the optionals. Let $\{\overline{o_1}, \cdots, \overline{o_n}\}$ be the computation times of the part of the optionals that get scheduled along with $\{m_1, \cdots, m_n\}$. Our goal is to maximize $\sum_{i=1}^{n} \overline{o_i}$ satisfying the following constraints.

$$m_i + \overline{o}_i \leq D, \forall i, \quad 1 \leq i \leq n \tag{3.11}$$

$$\sum_{i=1}^{n} (m_i + \overline{o_i}) \leq mD \tag{3.12}$$

Figure 3.6 illustrates a simple algorithm that calculates $\{\overline{o_1}, \cdots, \overline{o_n}\}$. The algorithm provides maximum possible computation time for an optional by allocating min {available empty slots, (D-m_i), o_i}. The algorithm is optimal since it fails to allocate any time slot to an optional only if

1. empty_slots = 0. In which case there are no time slots left since providing any more will violate condition 3.12.

```
Algorithm Find-Optionals;
begin
    i=1;
    empty_slots = mD - Σ_{i=1}^{n} m_i;
    while ((i ≤ n) and (empty_slots > 0)) do
    begin
        o_i' = min {empty_slots, (D − m_i), o_i};
        empty_slots = empty_slots - o_i';
        i=i+1;
    end;
end.
```

Figure 3.6: Algorithm to calculate $\{\overline{o_1}, \cdots, \overline{o_n}\}$

2. m_i = D. In which case, providing any time slot for optional will violate condition 3.11.

Figure 3.7 illustrates an example where the algorithm is applied to obtain $\{\overline{o_1}, \cdots, \overline{o_n}\}$. The resulting tasks with computation times $\{m_1 + \overline{o_1}, \cdots, m_n + \overline{o_n}\}$ is scheduled using LRTF algorithm.

3.4 Summary

In this chapter we considered the problem of scheduling a set of tasks with task characteristics known *apriori*. We derived the necessary and sufficient conditions for scheduling a set of independent tasks within a deadline D. We also introduced an optimal scheduling algorithm based on *bin-packing* approach and *level* approach for scheduling a set of tasks implemented using imprecise computation technique. We note that the *bin-packing* approach provides a schedule where the number of pre-emption is minimum but extending it to run-time environment is cumbersome. On the other hand, *level*

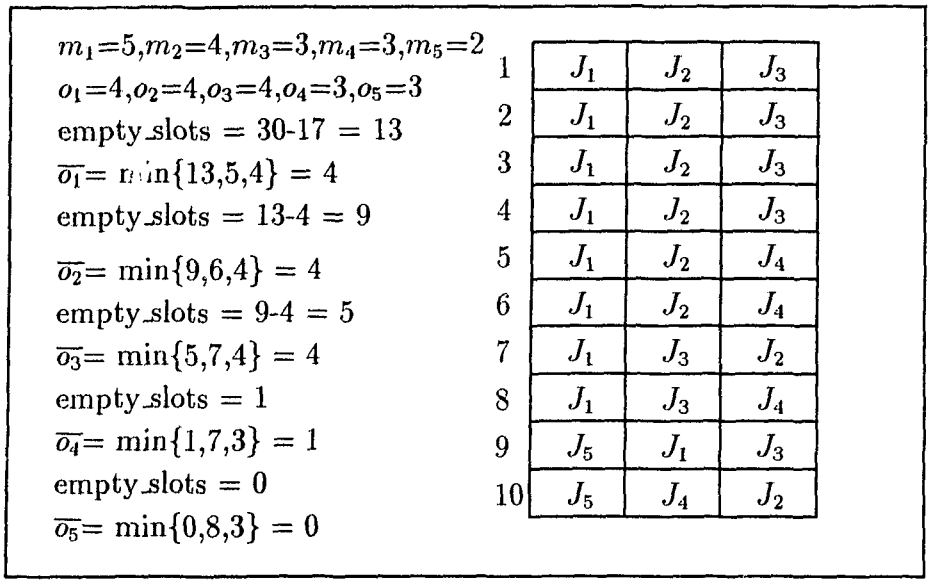


Figure 3.7: An example illustrating algorithm Find-Optionals

algorithm is more conducive for run-time scheduling since task allocation is done time unit by time unit. We maximized the time slots in which optionals are scheduled after scheduling all the mandatory subtasks.

It is not always possible to have the complete knowledge of task characteristics to obtain a feasible schedule. Tasks such as maintenence tasks fall into this catagory where their ready times are not known *apriori* and their computation times are known only when they are ready. Such tasks need to be scheduled during run-time, where the system is not available in certain time intervals due to pre-run-time schedule. In the next chapter we will introduce the problem of scheduling a set of tasks during run-time where the system is available only in certain time intervals.

Chapter 4

Run-Time Scheduling

In this chapter we consider tasks whose characteristics are not known until they are activated during run-time. We assume a pre-run-time scheduler has already been invoked. Hence tasks that arrive during run-time are scheduled on top of pre-run-time schedule, in the empty slots available. We call a system that is available only in partial time intervals, a *limited time available system.* In this chapter, we introduce a *level* algorithm for optimally scheduling a set of tasks that arrive during run-time in a limited time available system. We also propose an optimal scheduling algorithm for scheduling tasks that have been implemented using imprecise computation technique.

4.1 Task Characteristics

We consider a multiprocessor system with m identical processors of the same speed. We denote the set of processors as $P = \{P_1, P_2, \cdots, P_m\}$. Let $J = \{J_1, J_2, \cdots, J_n\}$ be the n independent tasks with $\{c_1, c_2, \cdots, c_n\}$ being their respective computation times. *Ready time* of a task J_i, namely $RT(J_i)$, is defined as the earliest time unit a task can be scheduled. Similarly, *deadline* of a task set J namely D, is defined as the latest time unit a task in the task set can be executed assuming that all tasks have the same deadline. We define *begin time* of a processor P_j, namely $BT(P_j)$, as the earliest time slot

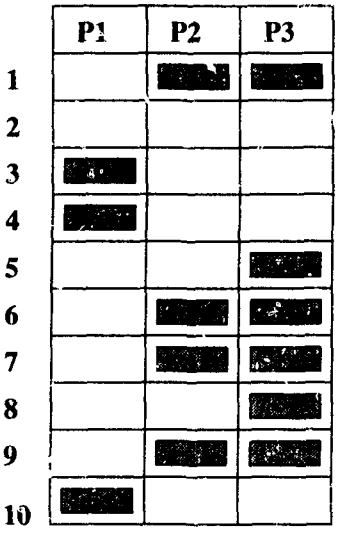


Figure 4.1: Example of a system available only in partial time intervals

a processor is available. Similarly, an *end time* of a processor P_j, namely $EN(P_j)$, is defined as the latest time slot a processor is available. We define *start time* of the system as $min\{BT(P_j)|1 \leq j \leq m\}$ and *finish time* of the system as $max\{EN(P_j)|1 \leq j \leq m\}$. Let a_{tj} be the availability of P_j in tth time slot, which means processor P_j is free to execute one time unit of a task in the tth time slot. For example, in Figure 4.1, a_{11}=1, a_{12}=0, and a_{13}=0. Thus a_{tj} can be represented as follows:

$$a_{tj} = \begin{cases} 0 & if \quad slot \quad not \quad available \\ 1 & if \quad slot \quad available \end{cases}$$

We assume that a task cannot be scheduled in more than one processor in the same time unit. If a task set J has to be executed before the deadline D then $c_i \leq D, \forall i, 1 \leq i \leq n$. If J has to be scheduled within D, there should be enough time slots available to schedule all the tasks in J. The

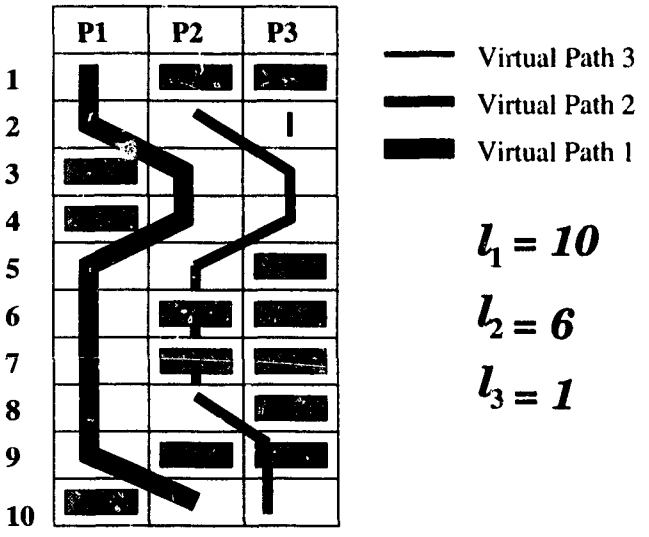


Figure 4.2: Virtual paths

total execution time of tasks in J should satisfy

$$\sum_{i=1}^{n} c_i \leq \sum_{t=1}^{D} \sum_{j=1}^{m} a_{tj},$$

since $\sum_{t=1}^{D} \sum_{j=1}^{m} a_{tj}$ is the total time slots available.

We define a *virtual path* as a collection of empty slots from distinct time units such that if $\{L_1, \cdots, L_k\}$ are the k distinct virtual paths available in P, then for some v, $1 \leq v \leq k$,

$$L_v = \left\{(t,j) | j = \min_{1 \leq w \leq m} (a_{tj} = 1 \wedge (t,w) \notin L_u, 1 \leq u < v)\right\}.$$

Figure 4.2 shows all the virtual paths present in P. Let $\{l_1, \cdots, l_k\}$ be the number of time slots in $\{L_1, \cdots, L_k\}$ where $l_v = cardinality(L_v)$, $1 \leq v \leq k$.

Let ρ_t be the number of empty slots present in the tth time unit. The algorithm for finding all the virtual paths in P is given in Figure 4.3. For any

```
Algorithm Find-Virtual-Path;

for t = start time to finish time
begin
    v = 1; ρ_t = 0; // no.of empty slots
    for j = 1 to m do
        if (a_tj = 1)
            begin
                ρ_t = ρ_t + 1;
                L_v = L_v ∪ {(t, j)};
                l_v = l_v - 1;
                v = v + 1;
            end;
end;
```

Figure 4.3: Algorithm to find virtual paths

time unit t, if ρ_t empty slots are present, then the first ρ_t virtual paths namely, $\{L_1, \cdots, L_{\rho_t}\}$ are allocated one time slot each by the algorithm. We notice that, in the worst case when all the processors are available, the algorithm finds all the virtual paths in $O(mD)$ time. We note that $l_1 = 10, l_2 = 6$, and $l_3 = 1$ for the task set given in Figure 4.2.

4.2 Real-Time Scheduling

4.2.1 Scheduling Tasks with Identical Ready Times

Let $\{J_1, ..., J_n\}$ be the tasks with $\{c_1, ..., c_n\}$ computation times ordered such that $c_1 \geq c_2 \geq ... \geq c_n$. Let $\{L_1, ..., L_k\}$ be the virtual paths with $\{l_1, ..., l_k\}$ time slots respectively, ordered such that $l_1 \geq l_2 ... \geq l_k$. To derive the necessary condition, we assume that $\{J_1, \cdots, J_n\}$ is feasibly scheduled within D. J_1 with computation time c_1 can be scheduled if $c_1 \leq l_1$ since l_1 is the

Algorithm LRTF;
// $\rho_t \leftarrow$ **Number of slots in** t **th time unit;**

for (every time unit t | $1 \leq t \leq D$ **)**
begin
    Schedule ρ_t **tasks with largest remaining computation times;**
    Decrement their respective task computation times;
end;

Figure 4.4: Largest Remaining Time First Algorithm

largest virtual path. J_2 with computation time c_2, has to be scheduled in $min\{c_1, (l_1 - c_1) + l_2\}$ time units. Therefore, $c_2 \leq (l_1 - c_1) + l_2$, and hence $c_1 + c_2 \leq l_1 + l_2$. Assuming that for some r, $\sum_{i=1}^{r} c_i \leq \sum_{v=1}^{r} l_v$, the time available to schedule J_{r+1} with computation time c_{r+1} is given by

$$min\{c_r, \sum_{v=1}^{r} l_v - \sum_{i=1}^{r} c_i + l_{r+1}\}$$

We obtain

$$\sum_{i=1}^{r+1} c_i \leq \sum_{v=1}^{r+1} l_v,$$

since $c_{r+1} \leq c_r$ and $c_{r+1} \leq \sum_{v=1}^{r} l_v - \sum_{i=1}^{r} c_i + l_{r+1}$. We have thus derived the necessary conditions for scheduling task set J within deadline D. We now state the following theorem and prove its sufficiency.

Theorem 7 *J can be scheduled within D if and only if*

$$\sum_{i=1}^{r} c_i \leq \sum_{v=1}^{r} l_v, \quad \forall r | \quad 1 \leq r < k \tag{4.1}$$

and

$$\sum_{i=1}^{n} c_i \leq \sum_{v=1}^{k} l_v \tag{4.2}$$

Proof. We had shown that if a feasible schedule exists for a task set $\{J_1, \cdots, J_n\}$ then the conditions mentioned above are satisfied. For sufficiency we need to show that the conditions are sufficient to produce a feasible schedule. Let us assume that there exists an optimal scheduler which provides a feasible schedule for a task set $\{J_1, \cdots, J_n\}$ with computation times $\{c_1, \cdots, c_n\}$ and that the necessary conditions are not satisfied. For some q,

$$\sum_{i=1}^{r} c_i \leq \sum_{v=1}^{r} l_v, \qquad \forall r | 1 \leq r \leq q \tag{4.3}$$

$$\sum_{i=1}^{q+1} c_i > \sum_{v=1}^{q+1} l_v$$

The maximum contiguous time that could be available after scheduling $\{J_1, \cdots, J_q\}$ is given by

$$\sum_{v=1}^{q} l_v - \sum_{i=1}^{q} c_i + l_{q+1}$$

since $l_1 \geq l_2 \geq l_q \geq l_{q+1}$. Note that $\{L_{q+2}, \cdots, L_k\}$ can have time slots only in time units where L_{q+1} has a time slot. Hence to schedule J_{q+1} with execution time c_{q+1},

$$c_{q+1} \leq \sum_{v=1}^{q} l_v - \sum_{i=1}^{q} c_i + l_{q+1}.$$

Therefore,

$$\sum_{i=1}^{q+1} c_i \leq \sum_{v=1}^{q+1} l_v$$

which is a contradiction to our assumption in Equation 4.3. Hence a task set can be scheduled within D if and only if the necessary and sufficient conditions are met. □

Consider the Largest remaining time first algorithm (LRTF algorithm) provided in Figure 4.4. We define ρ_t as the number of time slots that are available for processing in P at the tth time unit. Note that ρ_t is same as $\sum_{j=1}^{m} a_{tj}$. LRTF algorithm schedules at every time unit t, tasks with largest remaining computation times. Figure 4.5 shows LRTF algorithm applied to P. The figure illustrates the scheduling done, time unit by time unit. The arrow in the figure represents the exchange of elements to order the remaining computation times in descending order.

In order to prove that LRTF algorithm is optimal, we need to show that LRTF algorithm can provide a feasible schedule for the task set that satisfies the necessary and sufficient conditions. We use LRTF algorithm to schedule the first time unit and show that the remaining computation times of the tasks satisfy the necessary and sufficient conditions for the time interval [2..D]. We then assume that the time interval [1..t-1] has been scheduled using LRTF algorithm and the remaining computation times of the tasks satisfy the conditions for the time interval [t..D] before scheduling the tth time unit. By induction, we will show that the remaining computation times of the tasks satisfy the conditions for [t+1..D] after scheduling tasks with largest remaining times in the tth time unit.

It is trivial to see that if the deadline is one time unit, then LRTF algorithm constructs a feasible schedule. This is due to the fact that there are $\sum_{j=1}^{m} a_{1j}$ virtual paths, namely $\{l_1, ..., l_{\rho_1}\}$ each of one time unit. By the condition in Equation 4.1, $\{c_1, ..., c_{\rho_1}\}$ should all be of one time unit each.

Consider the case when deadline D is greater than 1. We define $\{J_1^t, \cdots, J_n^t\}$ as the task set before scheduling the tth time unit. Let $\{c_1^t, \cdots, c_n^t\}$ be the

$c_1 = 6$
$c_2 = 5$
$c_3 = 3$
$c_4 = 2$
$c_5 = 1$

$l_1 = 10$
$l_2 = 6$
$l_3 = 1$

	P1	P2	P3
2	J1	J2	J3
3		J1	J2
4		J1	J2
5	J1	J2	
8	J1	J3	
10		J4	J1
1	J4		
6	J3		
7	J2		
9	J5		

Time Unit	Before Scheduling i th time unit Task Set	ComputationTime	After Scheduling Remaining time
	{ c1, c2, c3, c4, c5 }	{6, 5, 3, 2,1}	{6, 5, 3, 2,1}
2	{c1, c2, c3} {c4, c5}	{6,5,3} {2,1}	{5,4,2} {2,1}
3	{c1, c2} {c3, c4, c5}	{5,4} {2,2,1}	{4,3} {2,2,1}
4	{c1, c2} {c3, c4, c5}	{4,3} {2,2,1}	{3,2} {2,2,1}
5	{c1, c2} {c3, c4, c5}	{3,2} {2,2,1}	{2,1} {2,2,1}
8	{c1, c3} {c4, c2, c5}	{2,2} {2,1,1}	{1,1} {2,1,1}
10	{c4, c1} {c3, c2, c5}	{2,1} {1,1,1}	{1,0} {1,1,1}
1	{c4} {c3, c2, c5, c1}	{1} {1,1,1,0}	{0} {1,1,1,0}
6	{c3} {c2, c5, c4, c1}	{1} {1,1,0,0}	{0} {1,1,0,0}
7	{c2} {c5, c3, c4, c1}	{1} {1,0,0,0}	{0} {1,0,0,0}
9	{c5} {c2.c3, c4, c1}	{1} {0,0,0,0}	{0} {0,0,0,0}

Figure 4.5: Example illustrating application of LRTF algorithm

computation times of $\{J_1^t, \cdots, J_n^t\}$ such that $c_1^t \geq \cdots \geq c_n^t$. We assume that $\{J_1^t, J_2^t, \cdots, J_{\rho_t}^t\}$ is schedulable in the tth time slot. LRTF algorithm schedules $\{J_1^t, \cdots, J_{\rho_t}^t\}$ with computation times $\{c_1^t, \cdots, c_{\rho_t}^t\}$ since ρ_t time slots are available for scheduling in the tth time unit. Let $\{l_1^t, l_2^t, \cdots, l_k^t\}$ be the number of time slots remaining in the virtual paths $\{L_1, \cdots, L_k\}$ respectively. We note that

$$l_1^t \geq l_2^t \geq \cdots \geq l_k^t, \quad \forall t| \quad 1 \leq t \leq D$$

In order to schedule $\{J_1^t, \cdots, J_{\rho_t}^t\}$, the computation times must satisfy the following conditions:

$$\sum_{i=1}^{r} c_i^t \leq \sum_{v=1}^{r} l_v^t, \quad \forall r| \quad 1 \leq r < k$$

and

$$\sum_{i=1}^{n} c_i^t \leq \sum_{v=1}^{k} l_v^t$$

We state and prove the following lemma and show by induction, that LRTF algorithm can schedule

$$\{J_1^{t+1}, J_2^{t+1}, \cdots, J_{\rho_{t+1}}^{t+1}\}$$

with computation times

$$\{c_1^{t+1}, c_2^{t+1}, \cdots, c_{\rho_{t+1}}^{t+1}\}$$

respectively, in the $(t+1)$st time unit and the remaining computation times of tasks satisfy the necessary and sufficient conditions for [t+1..D].

Lemma 3 *If the remaining computation times of tasks in a task set satisfy the necessary and sufficient conditions for [t..D], then the remaining computation times of the tasks satisfy the conditions for [t+1..D], after scheduling the tth time unit using LRTF algorithm.*

Proof. Let the tth time unit contain α empty slots and the $(t+1)$st time unit contain β empty slots. Without loss of generality, we assume that $\alpha \geq \beta$. This can be achieved by rearranging the time units in decreasing order of number of time slots available in each time unit, since we are dealing with offline scheduling. Figure 4.5 illustrates the rearrangement of the limited time available system given in Figure 4.1.

We schedule $J^t = \{J_1^t, \cdots, J_n^t\}$ in the tth time unit using LRTF algorithm. Let $c_1^t \geq c_2^t \geq \cdots \geq c_n^t$ be the computation times of J^t before scheduling the tth time unit. Let $L^t = \{L_1^t, \cdots, L_\alpha^t\}$ be the α virtual paths with $\{l_1^t, \cdots, l_\alpha^t\}$ time slots where $l_1^t \geq l_2^t \geq \cdots \geq l_\alpha^t$. First we schedule $\{J_1^t, \cdots, J_\alpha^t\}$ with computation times $\{c_1^t, \cdots, c_\alpha^t\}$ in the tth time unit. We state the following necessary conditions for scheduling J^t in [t..D].

$$\sum_{i=1}^{r} c_i^t \leq \sum_{v=1}^{r} l_v^t, \qquad \forall r \mid 1 \leq r \leq \alpha - 1 \tag{4.4}$$

$$\sum_{i=1}^{n} c_i^t \leq \sum_{v=1}^{\alpha} l_v^t$$

We prove that the task set J^t with the remaining computation times after scheduling tth time unit namely, $C^{t+1} = \{c_1^t - 1, \cdots, c_\alpha^t - 1, c_{\alpha+1}^t, \cdots, c_n^t\}$, satisfies the necessary conditions for [t+1..D] time interval after scheduling the tth time unit. Note that we do not represent J^{t+1} as yet since C^{t+1} is not ordered. The virtual paths in [t+1..D], given by $L^{t+1} = \{L_1^{t+1}, \cdots, L_\beta^{t+1}\}$ have $\{l_1^t - 1, \cdots, l_\beta^t - 1\}$ time slots respectively. To check if the conditions are satisfied for [t+1..D], we need to order J^t in decreasing order of the remaining computation times C^{t+1}. Let b be the number of tasks that were not scheduled in the tth time unit but are scheduled in the $(t+1)$st time unit. C^{t+1} can be partitioned according to the tasks that were scheduled in

the tth time unit and the tasks that were not scheduled in the tth time unit and it can be represented as shown below.

$$\overbrace{\{c_1^t-1,\cdots,\cdots,c_\beta^t-1,\underbrace{c_{\beta+1}^t-1\cdots,c_\alpha^t-1}_{\alpha-\beta}\}}^{\alpha}\underbrace{\{c_{\alpha+1}^t,\cdots,c_{\alpha+b}^t}_{b},\cdots,c_n^t\}$$

After scheduling the tth time unit, C^{t+1} may not be in descending order, since $\min\{c_{\alpha+1}^t,\cdots,c_n^t\}$ might be greater than $\min\{c_1^t-1,\cdots,c_\alpha^t-1\}$. Note that if $c_{\alpha+1}^t=\cdots=c_n^t=0$ then, the necessary conditions are satisfied for [t+1..D] time interval, since the only non zero remaining computation times are $c_1^t-1\geq\cdots\geq c_\alpha^t-1>0$. Hence we assume that $c_1^t\geq c_2^t\geq\cdots\geq c_n^t>0$ and consider the following cases:

case 1: if $b=0$, then $c_1^t-1\geq c_2^t-1\geq\cdots\geq c_\alpha^t-1\geq c_{\alpha+1}^t\geq\cdots\geq c_n^t$. It is trivial to see that the following conditions are satisfied due to Equation 4.4. Note that $l_{\beta+1}^t=\cdots=l_{\alpha-1}^t=l_\alpha^t=1$ since $\alpha\geq\beta$. This situation arises in the example provided in Figure 4.5, where for the first three time units $b=0$. Hence the conditions for scheduling in [t+1..D] are given by the following equations.

$$\sum_{i=1}^{r}(c_i^t-1)\ \leq\ \sum_{v=1}^{r}(l_v^t-1),\qquad \forall r\ \mid\ 1\leq r<\beta \tag{4.5}$$

$$(\sum_{i=1}^{n}c_i^t)-\alpha\ \leq\ (\sum_{v=1}^{\beta}l_v^t)-\beta$$

case 2: if $b=1$, then $\exists z$ such that $z\leq\beta$, where

$$c_{\alpha+1}^t>max\{c_{z+1}^t-1,\cdots,c_\alpha^t-1\}.$$

Therefore C^{t+1} in decreasing order is

$$\{c_1^t-1,\cdots,c_z^t-1,c_{\alpha+1}^t,c_{z+1}^t-1,\cdots,c_\beta^t-1,\cdots,c_\alpha^t-1\}\{c_{\alpha+2}^t,\cdots,c_n^t\}$$

Note that $J^t_{\alpha+i}$ was not scheduled in the tth time unit and has been included in the set of tasks that will be scheduled in the $(t+1)$st time unit. It can also be seen that, if $J^t_{\alpha+1}$ gets inserted at the $(z+1)$st position, the computation times of all the tasks after $(z+1)$st position until α should be equal, since $c^t_{\alpha+1}$ differs from $\{c^t_{z+1}, \cdots, c^t_\alpha\}$ by one. (If it does not differ by one, then it would have got scheduled in the tth time unit). Therefore

$$c^t_{\alpha+1} = c^t_{z+1} = \cdots = c^t_{\alpha-1} = c^t_\alpha \tag{4.6}$$

For example, $b = 1$ in Figure 4.5, after scheduling the 8th time unit where only one element gets exchanged, *i.e.*, J_4 gets inserted before J_1. We claim that the necessary conditions given by the following equations are satisfied in [t+1..D] time units, where $\{L^{t+1}_1, \cdots, L^{t+1}_\beta\}$ is the remaining virtual paths with $\{l^t_1 - 1, \cdots, l^t_\beta - 1\}$ time slots.

$$\sum_{i=1}^{r}(c^t_i - 1) \leq \sum_{v=1}^{r}(l^t_v - 1), \quad \forall r \mid 1 \leq r \leq z \tag{4.7}$$

$$\sum_{i=1}^{z}(c^t_i - 1) + c^t_{\alpha+1} \leq \sum_{v=1}^{z+1}(l^t_v - 1) \tag{4.8}$$

It can be easily seen from Equations 4.4 and 4.6 that Equation 4.7 is valid. We need to prove that Equation 4.8 is valid. We know that $\sum_{i=1}^{z+1} c^t_i \leq \sum_{v=1}^{z+1} l^t_v$, from the necessary conditions given in Equation 4.4. Equation 4.8 can be simplified into :

$$\sum_{i=1}^{z} c^t_i + c^t_{\alpha+1} \leq \sum_{v=1}^{z+1} l^t_v - 1 \Rightarrow \sum_{i=1}^{z} c^t_i + c^t_{\alpha+1} < \sum_{v=1}^{z+1} l^t_v$$

We need to show

$$\sum_{i=1}^{z+1} c^t_i < \sum_{v=1}^{z+1} l^t_v$$

since $c^t_{\alpha+1} = c^t_{z+1}$.

We know from the necessary condition in Equation 4.4 that for scheduling in [t..D],

$$\sum_{i=1}^{z+1} c_i^t \leq \sum_{v=1}^{z+1} l_v^t$$

Therefore we need to prove that

$$\sum_{i=1}^{z+1} c_i^t \neq \sum_{v=1}^{z+1} l_v^t$$

Let us assume that

$$\sum_{i=1}^{z} c_i^t + c_{\alpha+1}^t = \sum_{v=1}^{z+1} l_v^t$$

Consider the necessary conditions for [1..z] in Equation 4.4. We obtain the following inequality:

$$\sum_{i=1}^{z} c_i^t \leq \sum_{v=1}^{z} l_v^t$$

Adding $c^t_{\alpha+1}$ to both sides we obtain

$$\sum_{i=1}^{z} c_i^t + c_{\alpha+1}^t \leq \sum_{v=1}^{z} l_v^t + c_{\alpha+1}^t$$

We substitute $\sum_{v=1}^{z+1} l_v^t$ for $\sum_{i=1}^{z} c_i^t + c_{\alpha+1}^t$ in the previous equation and obtain the following inequality:

$$\sum_{v=1}^{z+1} l_v^t \leq \sum_{v=1}^{z} l_v^t + c_{\alpha+1}^t$$

Therefore

$$l_{z+1}^t \leq c_{\alpha+1}^t \tag{4.9}$$

Substituting $c^t_{\alpha+1} = c^t_{z+1}$ from Equation 4.6 into our assumption that

$$\sum_{i=1}^{z} c^t_i + c^t_{\alpha+1} = \sum_{v=1}^{z+1} l^t_v \tag{4.10}$$

we obtain the following equation:

$$\sum_{i=1}^{z+1} c^t_i = \sum_{v=1}^{z+1} l^t_v \tag{4.11}$$

Similarly consider [1..z+2] interval in Equation 4.4.

$$\sum_{i=1}^{z+2} c^t_i \leq \sum_{v=1}^{z+2} l^t_v \tag{4.12}$$

By substituting, Equation 4.11 into Equation 4.12 we obtain the following equation.

$$\sum_{v=1}^{z+1} l^t_v + c^t_{z+2} \leq \sum_{v=1}^{z+2} l^t_v \tag{4.13}$$

This can be simplified as

$$c^t_{z+2} \leq l^t_{z+2} \tag{4.14}$$

We obtain $l^t_{z+1} \leq c^t_{\alpha+1} \leq l^t_{z+2}$ from Equation 4.9 and Equation 4.14 and also from the fact that $c^t_{z+2} = c^t_{\alpha+1}$. We can see that $l^t_{z+1} = l^t_{z+2}$ and $c^t_{\alpha+1} = l^t_{z+1} = l^t_{z+2}$ since $l^t_1 \geq l^t_2 \cdots \geq l^t_\alpha$. Therefore from Equation 4.6 we arrive at the following equality

$$c^t_{\alpha+1} = l^t_{z+1} = l^t_{z+2} \tag{4.15}$$

In order to generalize the result for [z+1..α], we assume that

$$c^t_{\alpha+1} = l^t_v, \quad \forall v | \quad z+1 \leq v \leq \alpha - 1 \tag{4.16}$$

and show by induction, that l_α^t should also be equal to $c_{\alpha+1}^t$. We know from Equation 4.4 that

$$\sum_{i=1}^{\alpha} c_i^t \leq \sum_{i=1}^{n} c_i^t \leq \sum_{v=1}^{\alpha} l_v^t$$

From our initial assumption that $\sum_{i=1}^{z} c_i^t = \sum_{v=1}^{z} l_v^t$, and from Equation 4.16, we obtain that $c_\alpha^t \leq l_\alpha^t$. Also since $l_\alpha^t \leq l_{\alpha-1}^t$ and $l_{\alpha-1}^t = c_\alpha^t$, we get $l_\alpha^t \leq c_\alpha^t$. Therefore, we obtain the following equation

$$c_{\alpha+1}^t = l_v^t, \quad \forall v | \quad z+1 \leq v \leq \alpha \tag{4.17}$$

This implies that $\sum_{i=\alpha+1}^{n} c_i^t \leq 0$, due to our initial assumption that $\sum_{i=1}^{z} c_i^t = \sum_{v=1}^{z} l_v^t$. But $\sum_{i=\alpha+1}^{n} c_i^t \leq 0$ is not possible since we have assumed that $c_1^t \geq c_2^t \geq \cdots c_n^t > 0$. Thus we arrive at a contradiction. Hence Equation 4.8 is valid. The argument can be easily extended to show the correctness of the following equations.

$$\sum_{i=1}^{z}(c_i^t - 1) + \sum_{i=z+1}^{z+r+1}(c_i^t - 1) + c_{\alpha+1}^t \leq \sum_{v=1}^{r+z+1}(l_v^t - 1), \forall 1 \leq r \leq \beta - 1 \tag{4.18}$$

$$\sum_{i=1}^{\alpha}(c_i^t - 1) + \sum_{i=\alpha+1}^{n} c_i^t \leq \sum_{v=1}^{\beta} l_v^t \tag{4.19}$$

case 3: General case where there are b tasks that get inserted. We proved that $\sum_{i=1}^{\alpha+1} c_i^t < \sum_{v=1}^{z+1} l_v^t$. Assume that for $b-1$ the following inequality is valid.

$$\sum_{i=1}^{\alpha+b-1} c_i^t < \sum_{v=1}^{z+b-1} l_v^t \tag{4.20}$$

We prove that

$$\sum_{i=1}^{\alpha+b} c_i^t < \sum_{v=1}^{z+b} l_v^t$$

Let us assume that

$$\sum_{i=1}^{\alpha+b} c_i^t = \sum_{v=1}^{z+b} l_v^t$$

then we obtain

$$\sum_{v=1}^{z+b} l_v^t + c_{z+b+1}^t \leq \sum_{v=1}^{z+b} l_v^t + l_{z+b+1}^t$$

since

$$\sum_{i=1}^{z+b} c_i^t = \sum_{v=1}^{z+b} l_v^t$$

Hence

$$c_{z+b+1}^t \leq l_{z+b+1}^t$$

Similarly,

$$\sum_{v=1}^{z+b} l_v^t \leq \sum_{v=1}^{z+b-1} l_v^t + c_{z+b}^t$$

since

$$\sum_{i=1}^{z+b} c_i^t \leq \sum_{v=1}^{z+b-1} l_v^t + c_{z+b}^t$$

Thus $l_{z+b}^t \leq c_{z+b}^t$. Therefore we arrive at the following equation

$$l_{z+b}^t = l_{z+b+1}^t = \cdots = l_\alpha^t = c_{z+1}^t = c_{z+2}^t = \cdots = c_\alpha^t = c_{\alpha+1}^t = \cdots = c_{\alpha+b}^t \tag{4.21}$$

We know that

$$\sum_{i=1}^{z+b} c_i^t = \sum_{v=1}^{z+b} l_v^t,$$

$$\sum_{i=z+b+1}^{\alpha} c_i^t = \sum_{v=z+b+1}^{\alpha} l_v^t$$

and

$$\sum_{i=1}^{n} c_i^t \leq \sum_{v=1}^{\alpha} l_v^t$$

Therefore

$$\sum_{i=\alpha+1}^{n} c_i^t \leq 0,$$

which is a contradiction.

Thus if the remaining computation times of tasks in a task set satisfy the necessary and sufficient conditions for [t..D], then the remaining computation times satisfy the conditions for [t+1..D], after scheduling the tth time unit using LRTF algorithm. Therefore we have shown by induction that LRTF algorithm provides a feasible schedule for a task set that satisfies the necessary and sufficient conditions. Hence LRTF algorithm is optimal. □

4.2.2 Scheduling Tasks with Non-Identical Ready Times

Let $\{J_1, \cdots, J_n\}$ be the set of tasks that are ready at the start time of the processor, with ready times $\{0, \cdots, 0\}$. Let $\{c_1, \cdots, c_n\}$ be the corresponding computation time of the tasks. Let D be the deadline within which $\{J_1, \cdots, J_n\}$ has to be executed regardless of the ready time of the tasks. Let $\{l_1, \cdots, l_k\}$ be the number of slots in $\{L_1, \cdots, L_k\}$ virtual paths. We

know from the necessary and sufficient condition given by Theorem 4.2 that $\{J_1, \cdots, J_n\}$ can be scheduled within D if and only if

$$\sum_{i=1}^{r} c_i \leq \sum_{v=1}^{r} l_v, \quad \forall r | \quad 1 \leq r < k$$

$$\sum_{i=1}^{n} c_i \leq \sum_{v=1}^{k} l_v$$

Let $\{J_{n+1}, \cdots, J_{n+z}\}$ be the z jobs that arrive at tth time unit with computation times $\{c_{n+1}, \cdots, c_{n+z}\}$. Let $\{c_1^t, \cdots, c_n^t\}$ be the remaining computation times of $\{J_1, \cdots, J_n\}$ before scheduling the tth time unit. Let $\{\overline{c_1}, \cdots, \overline{c_n}, \overline{c_{n+1}}, \cdots, \overline{c_{n+z}}\}$ be the remaining computation times of all the tasks in sorted order. Let $\{\overline{l_1}, \cdots, \overline{l_k}\}$ be the time slots available in $\{L_1, \cdots, L_k\}$. We now state the necessary and sufficient condition for executing $\{J_1, \cdots, J_{n+z}\}$ within deadline D. The proof is obvious from Lemma 3 since this is a special case.

Lemma 4 *$\{J_1, \cdots, J_{n+z}\}$ can be executed in [t..D] time interval if and only if*

$$\sum_{i=1}^{r} \overline{c_i} \leq \sum_{v=1}^{r} \overline{l_v}, \quad \forall r | 1 \leq r < k \tag{4.22}$$

$$\sum_{i=1}^{n} \overline{c_i} \leq \sum_{v=1}^{k} \overline{l_v}$$

This result holds true for tasks that have non zero ready times and have arrived at any time unit before the tth time unit. LRTF algorithm will schedule optimally in [t..D] time interval, as long as all the remaining computation times of the unfinished tasks and the computation times of tasks that have ready time t, satisfy the condition mentioned in Equation 4.22. Since the proof is obvious, we simply state the following theorem.

Theorem 8 *LRTF algorithm is an optimal on-line scheduling algorithm.*

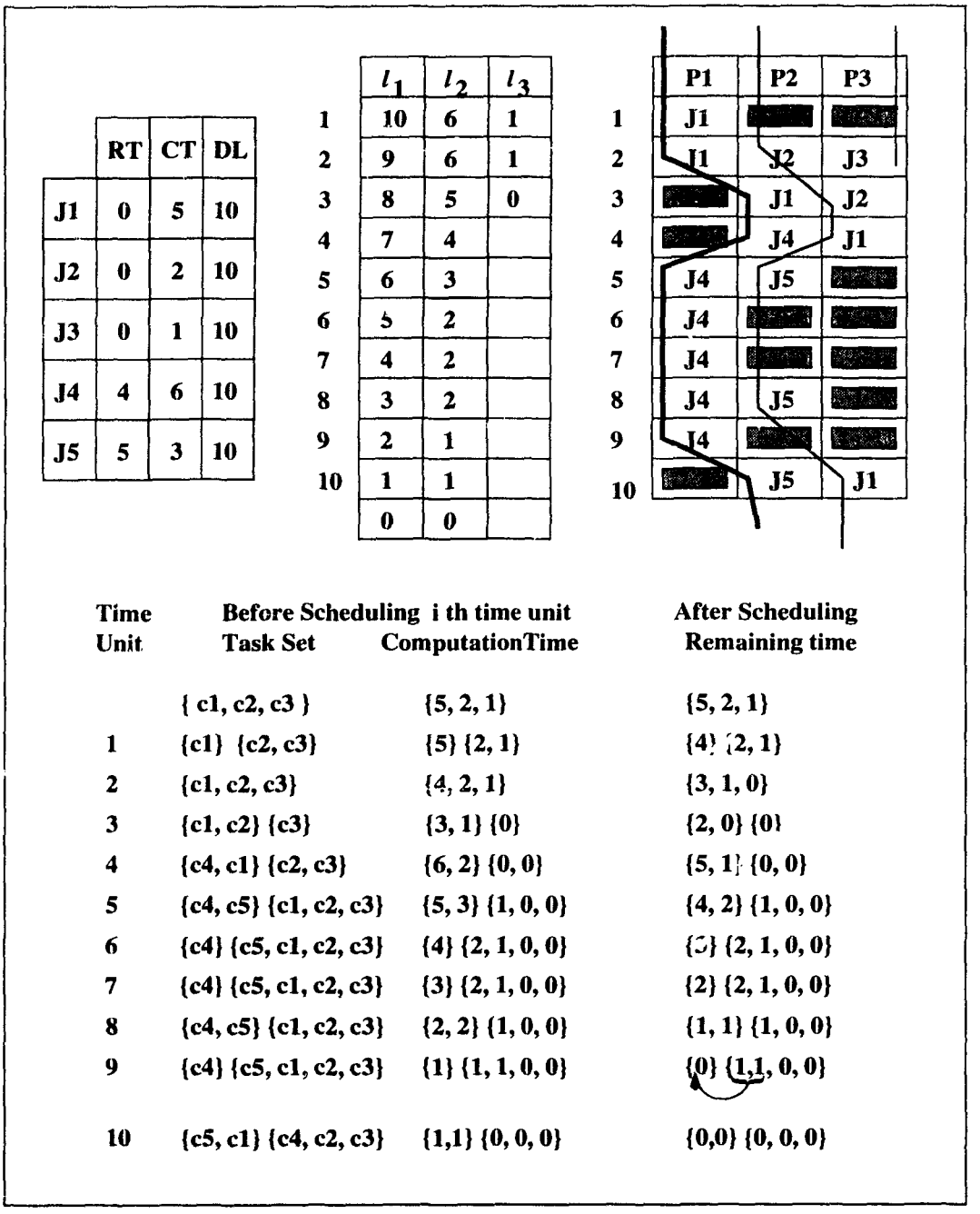
Figure 4.6 illustrates a task system of three tasks that are ready for processing at the start time of processor. Note that the remaining times in the virtual paths at various time units are also provided in Figure 4.6. We notice in the figure that the necessary and sufficient conditions are satisfied for J_1, J_2, and J_3. Task J_4 arrives at the fourth time unit, and we can see that the remaining computation times of J_1, J_2, and J_3 satisfy the conditions for [4..10], with the remaining time slots in the virtual paths being $l_1 = 7$, and $l_2 = 4$. Similarly, J_5 is ready for execution from 5th time unit and the conditions are satisfied for [5..10]. Thus we guarantee $\{J_1, \cdots, J_5\}$ to execute by the deadline 10 using LRTF algorithm.

4.3 Fault-Tolerant Scheduling

4.3.1 Scheduling Tasks with Identical Ready Times

We present an algorithm to schedule a set of tasks $\{J_1, \cdots, J_n\}$ using *level* approach for an imprecise computation system. Individual tasks in an imprecise computation system can be partitioned into two subtasks, namely mandatory subtask and optional subtask. Let $\{M_1, \cdots, M_n\}$ be the mandatory subtasks with computation times of $\{m_1, \cdots, m_n\}$ and $\{O_1, \cdots, O_n\}$ be the optional subtasks with computation times of $\{o_1, \cdots, o_n\}$. Let D be the deadline and $\{L_1, \cdots, L_k\}$ be the k virtual paths with $\{l_1, \cdots, l_k\}$ time slots. Our goal is to schedule the mandatory subtasks of all the tasks in their entirety to ensure that we obtain an acceptable result within D. We then, maximize the total execution time of the optional subtasks to increase the accuracy of the result.

To schedule $\{M_1, \cdots, M_n\}$ within D, the following feasibility conditions



	RT	CT	DL
J1	0	5	10
J2	0	2	10
J3	0	1	10
J4	4	6	10
J5	5	3	10

	l_1	l_2	l_3
1	10	6	1
2	9	6	1
3	8	5	0
4	7	4	
5	6	3	
6	5	2	
7	4	2	
8	3	2	
9	2	1	
10	1	1	
	0	0	

	P1	P2	P3
1	J1		
2	J1	J2	J3
3		J1	J2
4		J4	J1
5	J4	J5	
6	J4		
7	J4		
8	J4	J5	
9	J4		
10		J5	J1

Time Unit	Before Scheduling i th time unit Task Set	ComputationTime	After Scheduling Remaining time
	{ c1, c2, c3 }	{5, 2, 1}	{5, 2, 1}
1	{c1} {c2, c3}	{5} {2, 1}	{4} {2, 1}
2	{c1, c2, c3}	{4, 2, 1}	{3, 1, 0}
3	{c1, c2} {c3}	{3, 1} {0}	{2, 0} {0}
4	{c4, c1} {c2, c3}	{6, 2} {0, 0}	{5, 1} {0, 0}
5	{c4, c5} {c1, c2, c3}	{5, 3} {1, 0, 0}	{4, 2} {1, 0, 0}
6	{c4} {c5, c1, c2, c3}	{4} {2, 1, 0, 0}	{3} {2, 1, 0, 0}
7	{c4} {c5, c1, c2, c3}	{3} {2, 1, 0, 0}	{2} {2, 1, 0, 0}
8	{c4, c5} {c1, c2, c3}	{2, 2} {1, 0, 0}	{1, 1} {1, 0, 0}
9	{c4} {c5, c1, c2, c3}	{1} {1, 1, 0, 0}	{0} {1,1, 0, 0}
10	{c5, c1} {c4, c2, c3}	{1,1} {0, 0, 0}	{0,0} {0, 0, 0}

Figure 4.6: On-line LRTF algorithm

from Theorem 4.2 has to be satisfied.

$$\sum_{i=1}^{r} m_i \leq \sum_{v=1}^{r} l_v, \qquad \forall r | 1 \leq r \leq k-1 \tag{4.23}$$

$$\sum_{i=1}^{n} m_i \leq \sum_{v=1}^{k} l_v$$

If the conditions mentioned above are not satisfied, then the task set cannot be scheduled within D. If the conditions are satisfied, then it is guaranteed that we will obtain an acceptable result by deadline D. Our goal then, would be to maximize the total execution times of the optional's $\{O_1, \cdots, O_n\}$ that can be scheduled within D. Note that we are allowed to schedule partial amount of any optional.

Let $\{c_1, \cdots, c_n\}$ be the execution times of $\{J_1, \cdots, J_n\}$. Therefore $\{c_1, \cdots, c_n\}$ is same as $\{m_1 + o_1, \cdots, m_n + o_n\}$ where $m_1 \geq m_2 \geq \cdots \geq m_n$. In an imprecise computation system, if $\{J_1, \cdots, J_n\}$ cannot be scheduled within D in its entirety, then an acceptable result can be obtained by scheduling all $\{M_1, \cdots, M_n\}$ and partial amount of $\{O_1, \cdots, O_n\}$. Let the execution times of the portion of optional subtasks that can be scheduled be $\{\overline{o_1}, \cdots, \overline{o_n}\}$. Let $\{\overline{c_1}, \cdots, \overline{c_n}\}$ be the execution times of $\{J_1, \cdots, J_n\}$, namely $\{m_1 + \overline{o_1}, \cdots, m_n + \overline{o_n}\}$. In this section, we present an algorithm for calculating $\{\overline{o_1}, \cdots, \overline{o_n}\}$ where $\sum_{i=1}^{n} \overline{o_i}$ is optimal taking into consideration that it satisfies the constraints mentioned in Equation 4.23. Once $\{\overline{o_1}, \cdots, \overline{o_n}\}$ is obtained, the LRTF algorithm could be used to schedule $\{\overline{c_1}, \cdots, \overline{c_n}\}$ optimally in the limited time available system by ensuring that $\{\overline{c_1}, \cdots, \overline{c_n}\}$ satisfies the following conditions.

$$\sum_{i=1}^{r} (m_i + \overline{o_i}) \leq \sum_{v=1}^{r} l_v, \qquad \forall r | \quad 1 \leq r \leq k-1 \tag{4.24}$$

$$\sum_{i=1}^{n}(m_i + \overline{o_i}) \leq \sum_{v=1}^{k} l_v$$

Consider a limited time available system where the total time available to schedule $\{J_1, \cdots, J_n\}$ is given by $\sum_{v=1}^{k} l_v$. The execution time necessary to schedule all the mandatory subtasks are given by $\sum_{i=1}^{n} m_i$. We obtain the amount of time that is available to schedule $\{O_1, \cdots, O_n\}$ from Equation 4.23. Figure 4.7 illustrates the time occupied by mandatories and where optionals could be added without violating the feasibility conditions for scheduling mandatory subtasks, given in Equation 4.23. Let δ_r be the maximum time available for scheduling $\{O_1, \cdots, O_r\}$ without violating the conditions in Equation 4.23. We define $\{\delta_1, \cdots, \delta_k\}$ as follows.

$$\delta_r = \sum_{v=1}^{r} l_v - \sum_{i=1}^{r} m_i, \qquad \forall r | \quad 1 \leq r \leq k-1 \tag{4.25}$$

$$\delta_k = \sum_{v=1}^{k} l_v - \sum_{i=1}^{n} m_i$$

Our goal is to maximize $\sum_{i=1}^{n} \overline{o_i}$ such that Equation 4.24 is satisfied. Let $\{s_1, \cdots, s_p\}$ be the indices of δ's such that $\delta_{s_{r+1}} = \text{Min}\{\delta_{s_r+1}, \cdots, \delta_k\}$ where $\delta_{s_1} = \text{Min}\{\delta_1, \cdots, \delta_k\}$. Figure 4.8 illustrates an example of a limited time available system with $l_1 = 6$, $l_2 = 2$, and $l_3 = 2$. We can see that $\delta_1 = 3$, $\delta_2 = 2$, and $\delta_3 = 3$. We can also notice that δ_{s_1} =Min$\{\delta_1, \delta_2, \delta_3\} = \delta_2 = 2$. δ_{s_2} =Min$\{\delta_3\} = 3$. If a feasible schedule needs to be obtained for the task set, then the conditions mentioned for $\{c_1, c_2, c_3\}$ in Figure 4.8 has to be satisfied.

Figure 4.7 illustrates the maximum time available to schedule optionals. For example, δ_1 is the maximum time available to schedule O_1. Although δ_1 time slots are available for O_1, it could be infeasible to allocate that many time slots, since some of the feasibility conditions might be violated. Con-

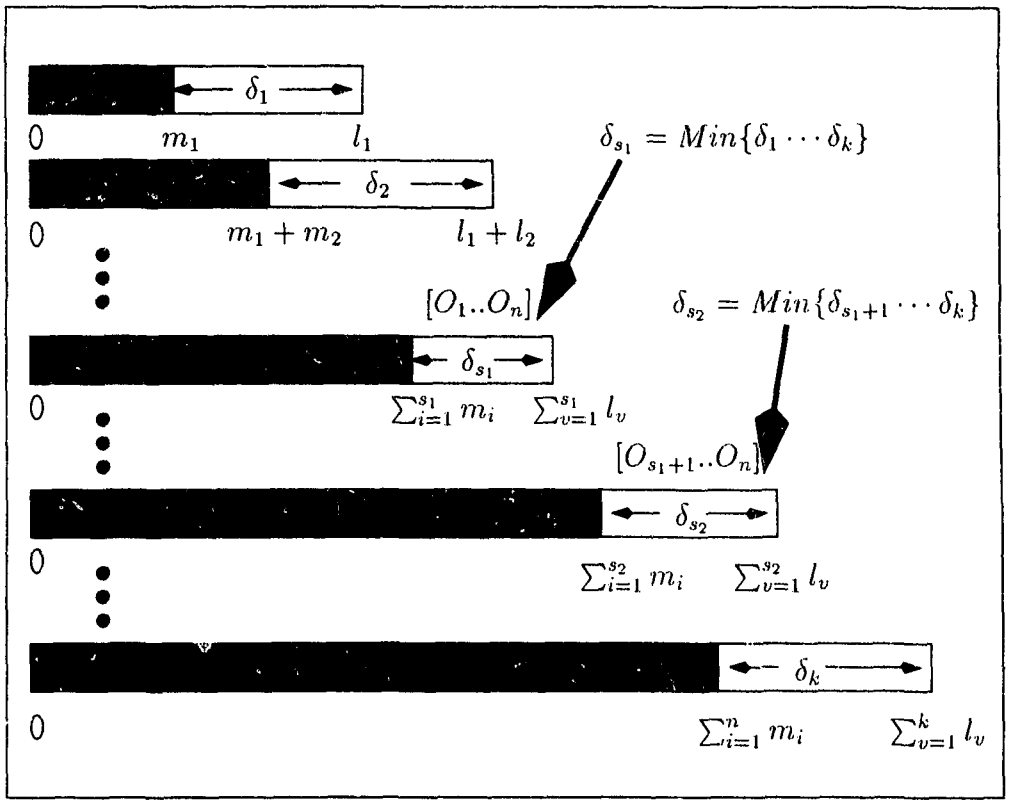
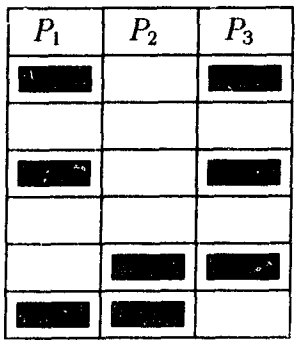


Figure 4.7: Finding maximum time to schedule optionals without violating conditions

sider the task set given in Figure 4.8. Note that $\delta_1(=3)$ time slots are available for scheduling O_1, but only $\delta_{s_1}(=2)$ time slots are provided for O_1 in order to satisfy $(m_1 + \overline{o_1}) + (m_2 + \overline{o_2}) \leq l_1 + l_2$. Consider the situation where $\delta_1 > \delta_{s_1}$ and δ_1 time slots are allocated to O_1. We obtain $\sum_{i=1}^{s_1}(m_i + \overline{o_i}) > \sum_{v=1}^{s_1} l_v$, leading to infeasiblity for scheduling $\{J_1, \cdots, J_n\}$ and violating the necessary and sufficient conditions mentioned in Equation 4.24. Figure 4.9 shows $(m_1 + \overline{o_1}) + (m_2 + \overline{o_2}) > l_1 + l_2$, which is a violation of the necessary conditions when δ_1 time slots are provided for O_1 instead of δ_{s_1} time slots, for the example given in Figure 4.8. Therefore we can allocate maximum possible time slots for optional subtasks by



$m_1 = 3 \quad o_1 = 4 \; l_1 = 6 \quad \delta_1 = 3 \quad \delta_{s_1} = Min\{3,2,3\}$

$m_2 = 3 \quad o_2 = 3 \; l_2 = 2 \quad \delta_2 = 2 \quad \delta_{s_1} = \delta_2$

$m_3 = 1 \quad o_3 = 2 \; l_3 = 2 \quad \delta_3 = 3 \quad \delta_{s_2} = Min\{3\} = \delta_3$

Feasibility Condition for Mandatory subtasks:

$$\begin{array}{rcl} m_1 \le l_1 & & 3 \le 6 \\ \sum_{i=1}^{2} m_i \le \sum_{v=1}^{2} l_v & \longrightarrow & 3+3 \le 6+2 \\ \sum_{i=1}^{3} m_i \le \sum_{v=1}^{3} l_v & & 3+3+1 \le 6+2+2 \end{array}$$

Feasibility Condition for $\{c_1, c_2, c_3\}$

$$\begin{array}{rcl} (m_1 + \overline{o_1}) \le l_1 & & (3+\overline{o_1}) \le 6 \\ \sum_{i=1}^{2}(m_i + \overline{o_i}) \le \sum_{v=1}^{2} l_v & \longrightarrow & (3+\overline{o_1}) + (3+\overline{o_2}) \le 6+2 \\ \sum_{i=1}^{3}(m_i + \overline{o_i}) \le \sum_{v=1}^{3} l_v & & (3+\overline{o_1}) + (3+\overline{o_2}) + (1+\overline{o_3}) \le 6+2+2 \end{array}$$

Figure 4.8: Example illustrating necessary and sufficient conditions

finding $\{\delta_{s_1}, \cdots, \delta_{s_p}\}$ as shown in Figure 4.10. Note that the maximum possible time slots for $\{O_1, \cdots, O_{s_1}\}, \{O_{1+s_1}, \cdots, O_{s_2}\}, \cdots, \{O_{1+s_{p-1}}, \cdots, O_{s_p}\}$ are $\delta_{s_1}, \delta_{s_2}, \cdots, \delta_{s_p}$ respectively.

The major steps of the algorithm to schedule $\{O_1, \cdots, O_n\}$ such that their execution time is maximized is given below. The detailed algorithm, named as Identical ready times (IRT), is provided in Figure 4.11

Algorithm Maximize_Optionals :

1. Compute $\{\delta_{s_1}, \cdots, \delta_{s_p}\}$ recursively such that

   $\delta_{s_1} = min\{\delta_1, \cdots, \delta_k\}$ and

   $\delta_{s_{r+1}} = min\{\delta_{1+s_r}, \cdots, \delta_k\}$

2. Divide $[0..\delta_{s_p}]$ into p intervals namely

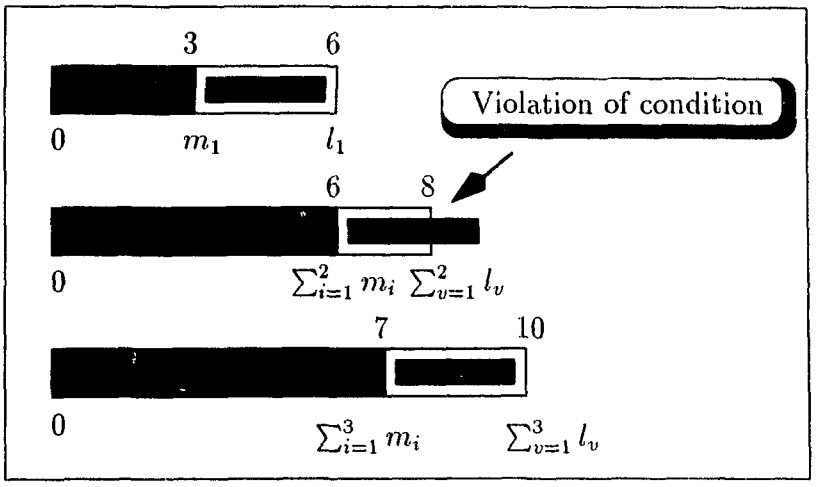


Figure 4.9: Violation of necessary condition when $\overline{o_1} = 3$

$[0..\delta_{s_1}][\delta_{1+s_1}..\delta_{s_2}] \cdots [\delta_{1+s_{p-1}}..\delta_{s_p}]$

3. For every interval $[\delta_{1+s_{r-1}}..\delta_{s_r}]$
   Schedule $\{O_{1+s_{r-1}}, \cdots, O_n\}$ where $s_0 = 0$
   such that $m_i + \overline{o_i} \leq l_1 \qquad \forall i.$

In step 1 of the algorithm, $\{\delta_{s_1}, \cdots, \delta_{s_p}\}$ is calculated recursively. δ_{s_r} can be defined as the maximum time available to schedule $\{O_{1+s_{r-1}}, \cdots, O_{s_r}\}$. For example, δ_{s_1} is the maximum time available to schedule $\{O_1, \cdots, O_{s_1}\}$. The cumulative execution times of $\{O_1, \cdots, O_{s_1}\}$, namely $\sum_{i=1}^{s_1} \overline{o_i}$, cannot be greater than δ_{s_1}, or else $\sum_{i=1}^{s_1}(m_i + \overline{o_i})$ will become greater than $\sum_{v=1}^{s_1} l_v$ leading to infeasibility by violating Equation 4.24. Note that $\{O_{1+s_1}, \cdots, O_n\}$ can also be scheduled in any available slots in $[0..\delta_{s_1}]$ without violating the feasibility conditions. Hence in step 2 we divide the interval $[0..\delta_{s_p}]$ into p parts namely, $[0..\delta_{s_1}][\delta_{1+s_1}..\delta_{s_2}] \cdots [\delta_{1+s_{p-1}}..\delta_{s_p}]$. In step 3, for every interval $[\delta_{1+s_{r-1}}..\delta_{s_r}]$, we schedule $\{O_{1+s_{r-1}}, \cdots, O_n\}$ such that $m_i + \overline{o_i} \leq l_1 \qquad \forall i, 1 \leq i \leq n$. We claim that the algorithm maximizes the total execution time of

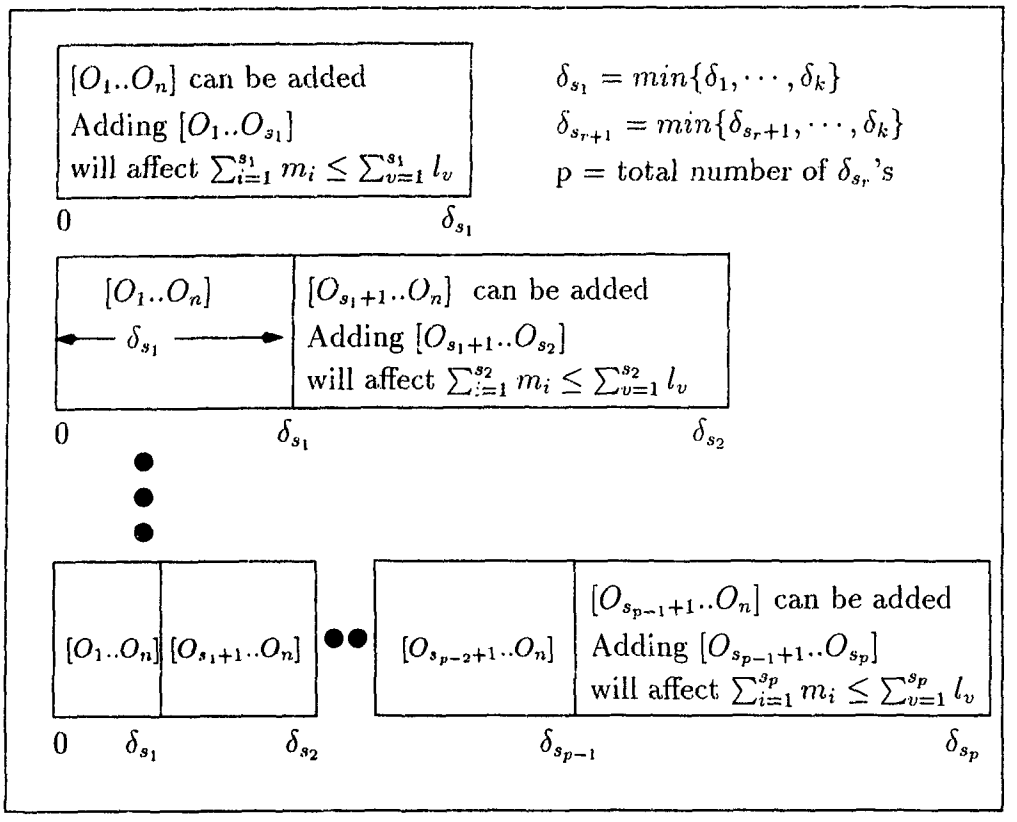


Figure 4.10: Optimal execution times of optionals satisfying the feasibility conditions for mandatory subtasks

all the optionals.

Theorem 9 *Algorithm Maximize_Optionals is optimal.*

Proof. It is easy to see that the algorithm guarantees the execution of mandatory subtasks since the calculation of δ's are based on the conditions mentioned in Equation 4.23. We now prove that the algorithm allocates maximum possible execution times to optional subtasks in $[0..\delta_{s_p}]$. The following cases need to be considered.

case 1: There does not exist any unscheduled empty slot in $[0..\delta_{s_p}]$. In which case, the total execution time of all optionals, namely $\sum_{i=1}^{n} o_i$ is op-

timal, since it is the maximum possible execution time achievable as all the empty time slots available are utilized.

case 2: There exist some unscheduled empty slots in $[0..\delta_{s_p}]$, but portions of some of $\{O_1, \cdots, O_n\}$ are not scheduled. Let us assume O_z to be the first optional that was not scheduled completely, i.e., $\overline{o_z} < o_z$. We will show that $\overline{o_z}$ is the maximum possible execution time for O_z. Although our proof is for the case when one optional is incompletely scheduled, our argument holds true when more than one optional is not scheduled completely.

If $m_z + \overline{o_z} = l_1$, then O_z was not scheduled completely because the algorithm allocates not more than l_1 for any task since l_1 is the number of time slots available in the longest virtual path. If more than l_1 is allocated to any task, then the basic assumption that a task cannot run in more than one processor in the same time unit, is violated. Therefore the algorithm allocates the maximum possible cumulative execution times for O_z.

If $m_z + \overline{o_z} < l_1$, and there exist some unscheduled slots in $[0..\delta_{s_p}]$ then we will show that $\overline{o_z}$ is the maximum possible execution time for O_z. Let $s_x = \max\{s_r \leq z \mid 1 \leq r \leq p\}$. It is easy to see that the partial allocation of O_z was provided by step 3 of the algorithm since $[0..\delta_{s_x}]$ was fully allocated. Therefore $\sum_{i=1}^{s_x}(m_i + \overline{o_i}) = \sum_{v=1}^{s_x} l_v$. The remaining tasks namely, $\{O_{s_x+1}, \cdots, O_n\}$ were scheduled in $[\delta_{s_x} + 1..\delta_{s_p}]$. Some slots in $[\delta_{s_x} + 1..\delta_{s_p}]$ were unscheduled because $o_{s_x+1} + \cdots + o_{s_p} < \delta_{s_p} - \delta_{s_x}$. The unscheduled slots cannot be used to obtain a greater execution time for O_z due to the following reasons.

1. If the execution time of O_z is increased by adding more slots to $\overline{o_z}$ then, $\sum_{i=1}^{s_x}(m_i + \overline{o_i}) > \sum_{v=1}^{s_x} l_v$ violating the feasibility conditions. Hence O_z cannot be scheduled in $[\delta_{s_x+1}..\delta_{s_p}]$.

2. $Min\{m_i + \overline{o_i} | 1 \leq i \leq s_x\} \geq Max\{m_i + \overline{o_i} | s_x + 1 \leq i \leq n\}$. If not, $\exists g, h$ such that, $s_x + 1 \leq g \leq n$ and $1 \leq h \leq s_x$, where $m_g + \overline{o_g} > m_h + \overline{o_h}$. Since

$\{\overline{c_1}, \cdots, \overline{c_{s_x}}\}$ satisfies the condition, $\sum_{i=1}^{s_x}(m_i + \overline{o_i}) = \sum_{v=1}^{s_x} l_v$, there cannot exist a $m_g + \overline{o_g}$ that is greater than $m_h + \overline{o_h}$ since it will violate the feasibility condition. Therefore it is futile to exchange any of $\{O_{s_x+1}, \cdots, O_n\}$ with any of $\{O_1, \cdots, O_{s_x}\}$ to obtain a larger $\sum_{i=1}^{n} \overline{o_i}$.

Hence we have proved that the algorithm provides the maximum possible execution times for the optional subtasks and guarantees the execution of mandatory subtasks. □

The detailed algorithm for calculating the maximum execution times of optional subtasks for identical ready times is given in Figure 4.11. An example illustrating the algorithm for limited time available imprecise computation system is given in Figure 4.12. Note that $C = 5$ and the number of processors, $m = 4$. It can be seen that the algorithm schedules optimal number of optionals in the limited time available system even though there are still four empty slots left. If more optionals are scheduled, then the assumption that a task cannot execute in two processors in the same time unit is violated.

4.3.2 Scheduling Tasks with Non-Identical Ready Times

Let $\{J_1, \cdots, J_n\}$ be the tasks that are ready at the start time of processors. Let $\{M_1, \cdots, M_n\}$ and $\{O_1, \cdots, O_n\}$ be the mandatory and optional subtasks of $\{J_1, \cdots, J_n\}$. We can obtain an acceptable approximate result if,

$$\sum_{i=1}^{r} m_i \leq \sum_{v=1}^{r} l_v, \quad \forall r | 1 \leq r \leq k$$

$$\sum_{i=1}^{n} m_i \leq \sum_{v=1}^{k} l_v$$

Let $\{\overline{c_1}, \cdots, \overline{c_n}\}$ be the maximum execution times calculated using algorithm IRT for tasks $\{J_1, \cdots, J_n\}$. Algorithm IRT is provided in Fig-

Algorithm IRT;

for r = 1 to k-1 do $\delta_r = \sum_{v=1}^{r} l_v - \sum_{i=1}^{r} m_i$;

$\delta_k = \sum_{v=1}^{k} l_v - \sum_{i=1}^{n} m_i$;

δ_s = **Min** $\{\delta_1, \delta_2, \cdots, \delta_k\}$;

for i = 1 to n do $\overline{c_i} = m_i$;

i = 1; z = 1;

while $(i \leq n)$

begin

X = **Min**$\{o_i, l_z - \overline{c_i}\}$;

if $(X \leq \delta_s)$

begin

$\overline{c_i} = \overline{c_i} + X$;

if $(\overline{c_i} = l_z)$ **z = z + 1;**

$o_i = o_i - X$;

for r = s to k do $\delta_r = \delta_r - X$;

i = i + 1;

end else begin

$\overline{c_i} = \overline{c_i} + \delta_s$;

$o_i = o_i - \delta_s$;

for r = s+1 to k do $\delta_r = \delta_r - \delta_s$;

if $(i \leq s)$ **i = s + 1;**

if $(s < k)$ δ_s = **Min** $\{\delta_{s+1}, \cdots, \delta_k\}$;

else exit();

end;

end;

Figure 4.11: Algorithm to find execution times of optimal optionals (IRT)

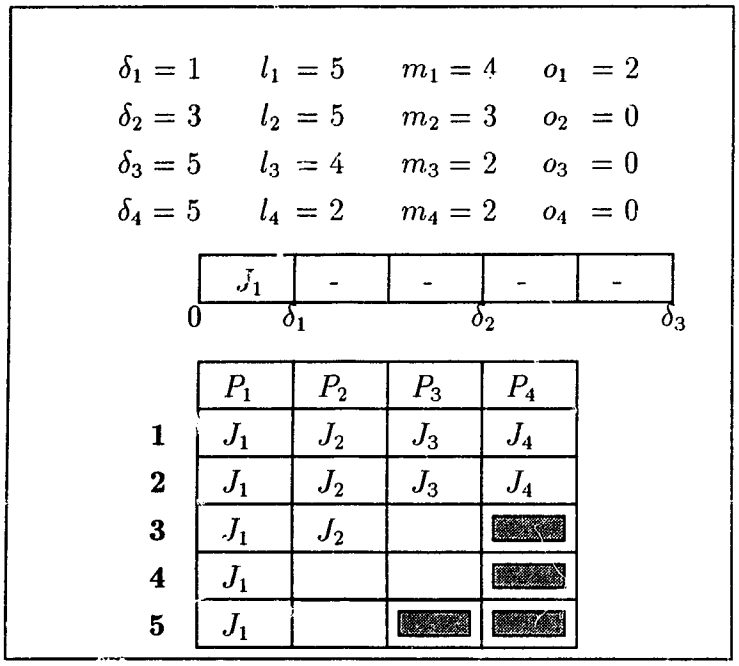


Figure 4.12: Illustration of fault-tolerant LRTF scheduler

ure 4.11. Let $\{J_{n+1}, \cdots, J_{n+z}\}$ be z tasks that arri at time slot t. Let $\{\overline{m_1}, \cdots, \overline{m_n}\}$ be the remaining computation times of the mandatory subtasks of $\{J_1, \cdots, J_n\}$ and $\{m_{n+1}, \cdots, m_{n+z}\}$ be the execution times of the mandatory subtasks that have the ready time t. For convenience, we name the execution times as remaining execution times namely $\{\overline{m_{n+1}}, \cdots, \overline{m_{n+r}}\}$. Let $\{\overline{l_1}, \cdots, \overline{l_k}\}$ be the remaining time slots of the virtual paths $\{L_1, \cdots, L_k\}$. The necessary and sufficient conditions to obtain an acceptable approximate result in [t..D] is given by,

$$\sum_{i=1}^{r} \overline{m_i} \leq \sum_{v=1}^{r} \overline{l_v}, \quad \forall r \quad |1 \leq r < k \tag{4.26}$$

$$\sum_{i=1}^{n+z} \overline{m_i} \leq \sum_{v=1}^{k} \overline{l_v}$$

Algorithm NIRT;

for r = 1 to k-1 do $\delta_j = \sum_{v=1}^{r} \overline{l_v} - \sum_{i=1}^{r} \overline{m_i}$;

$\delta_k = \sum_{v=1}^{k} \overline{l_v} - \sum_{i=1}^{n} \overline{m_i}$;

$\delta_s =$ **Min** $\{\delta_1, \delta_2, \cdots, \delta_k\}$;

for i = 1 to n do $\overline{c_i} = \overline{m_i}$; $\overline{o_i} =$ **remaining ex.time of** o_i;

i = 1;

while (i ≤ n)

begin

  if (Min ($\delta_s, \overline{o_i}) = \overline{o_i}$)

  begin

    $\overline{c_i} = \overline{c_i} + \overline{o_i}$;

    for j = s to k do $\delta_j = \delta_j - \overline{o_i}$;

    i = i + 1;

  end else if (Min $(\delta_s, \overline{o_i}) = \delta_s$)

      begin

        $\overline{c_i} = \overline{c_i} + \delta_s$;

        $\overline{o_i} = \overline{o_i} - \delta_s$;

        for r = s+1 to k do $\delta_r = \delta_r - \delta_s$;

        i = s + 1;

        if (i ≤ k) $\delta_s =$ **Min** $\{\delta_i, \delta_{i+1}, \cdots, \delta_k\}$;

        else exit();

      end;

end;

Figure 4.13: On-line algorithm to find optimal optionals (NIRT)

If the conditions in Equation 4.26 are satisfied, then the tasks can be scheduled in [t..D] and an approximate result will be guaranteed for every task. In order to maximize the number of optionals that can be scheduled, so that the result is enhanced, algorithm NIRT in Figure 4.13 is used to calculate optimal number of optionals that can be scheduled. Let $\{c_1^t, \cdots, c_n^t\}$ be the portion of computation times of $\{J_1, \cdots, J_n\}$ that has been executed in [1..t-1]. The remaining execution times of the optionals can be given by, $\{c_1^t - m_1, \cdots, c_n^t - m_n\}$. Some of the remaining execution times could be less than zero for those mandatories that are still being executed. The remaining execution times of the corresponding optionals for those mandatories are taken to be zero.

4.4 Summary

In this chapter we considered scheduling a set of tasks in a system that is available only in certain time intervals. Run-time scheduling is applicable in areas such as communication systems and process control where tasks are dynamically invoked. Schmidt [55] considered the problem of scheduling a set of tasks in a limited time available system using *bin-packing* approach. We presented a simpler algorithm based on *level* approach and proved its optimality.

We derived the necessary and sufficient condition for the tasks to be scheduled within D. We introduced LRTF scheduling algorithm for limited time available systems. We proved that LRTF algorithm is optimal for scheduling independent tasks with identical ready times and deadlines. We assumed that the tasks were implemented using imprecise computation system with each task containing a mandatory and optional subtask. We provided an algorithm for obtaining the maximum execution time possible for optional subtasks after guaranteeing the execution of mandatory subtasks. We proved

that our algorithm is optimal.

If the necessary and sufficient conditions are not satisfied, then there do not exist a feasible schedule for the task set and the deadline will be missed. If the tasks in pre-run-time schedule had been implemented using imprecise computation approach then we could have scheduled the task set by removing the optionals in pre-run-time schedule. In the next chapter we assume that the tasks in pre-run-time schedule have been implemented using imprecise computation system. We propose an adaptive scheduler to schedule the set of run-time tasks by removing minimum number of optionals from the pre-run-time scheduler.

Chapter 5

Run-Time Adaptive Scheduling

In the previous chapter we had considered the problem of scheduling a set of tasks whose task characteristics are known only at the time of arrival. We had assumed that the tasks that were scheduled in the pre-run-time schedule cannot be removed, and the tasks that arrive during run-time can only be scheduled in the time slots not occupied by the pre-run-time schedule. In this chapter, we assume that the pre-run-time schedule consists of imprecise computation tasks that have mandatory subtasks and optional subtasks. We derive the necessary and sufficient conditions to schedule a set of tasks within D that arrive during run-time by removing optimal number of optional subtasks from the pre-run-time schedule. We provide an adaptive scheduler that dynamically removes minimum number of time slots occupied by optional subtasks to schedule a set of tasks that arrive during run-time. We also prove that the algorithm is optimal and provide an example.

5.1 Task Characteristics

Let $\{J_1, \cdots, J_n\}$ be the set of external tasks that has to be scheduled within D in m processors, namely $\{P_1, \cdots, P_m\}$. Let $\{c_1, \cdots, c_n\}$ be the computation

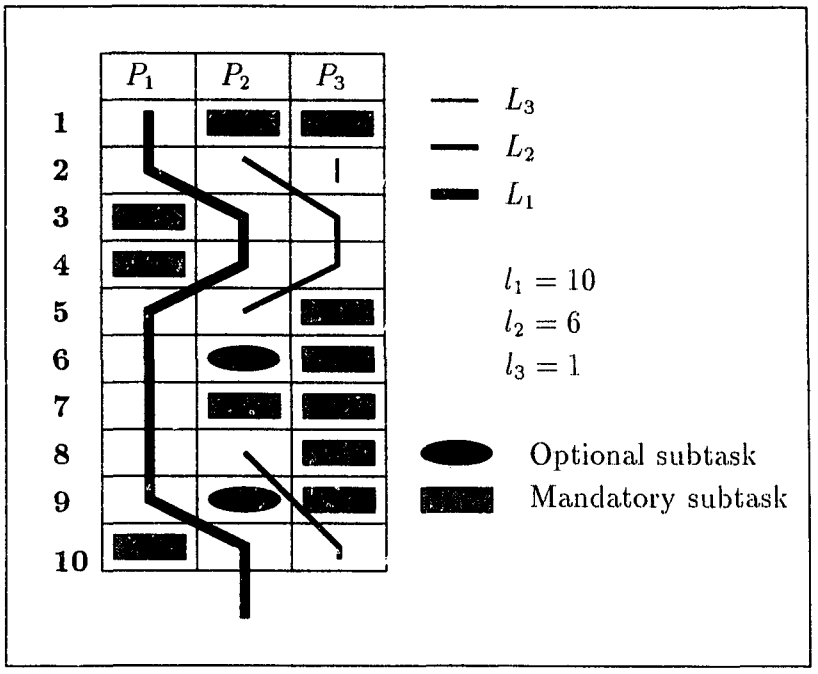


Figure 5.1: Virtual paths

time of $\{J_1, \cdots, J_n\}$ respectively. We assume that a pre-run-time schedule exists for $\{P_1, \cdots, P_m\}$ and the tasks that have been scheduled during pre-run-time are fault-tolerant. Hence each task in pre-run-time schedule has two subtasks, namely mandatory subtask and optional subtask. During run-time, without skipping the deadline, a portion of optional subtasks can be removed if necessary to schedule run-time tasks.

Let $\{L_1, \cdots, L_k\}$ be the virtual paths constructed using the empty slots present in pre-run-time schedule. Let $\{l_1, \cdots, l_k\}$ be the number of slots in the k virtual paths, namely $\{L_1, \cdots, L_k\}$. Figure 5.1 provides an example of virtual paths in $\{P_1, P_2, P_3\}$ where pre-run-time tasks are fault-tolerant. We note that 6th and 9th time slots occupied by optionals in processor P_2 are not considered for constructing virtual paths and $l_1 = 10, l_2 = 6$, and $l_3 = 1$.

We define an *alternate path* as a collection of time slots that are either

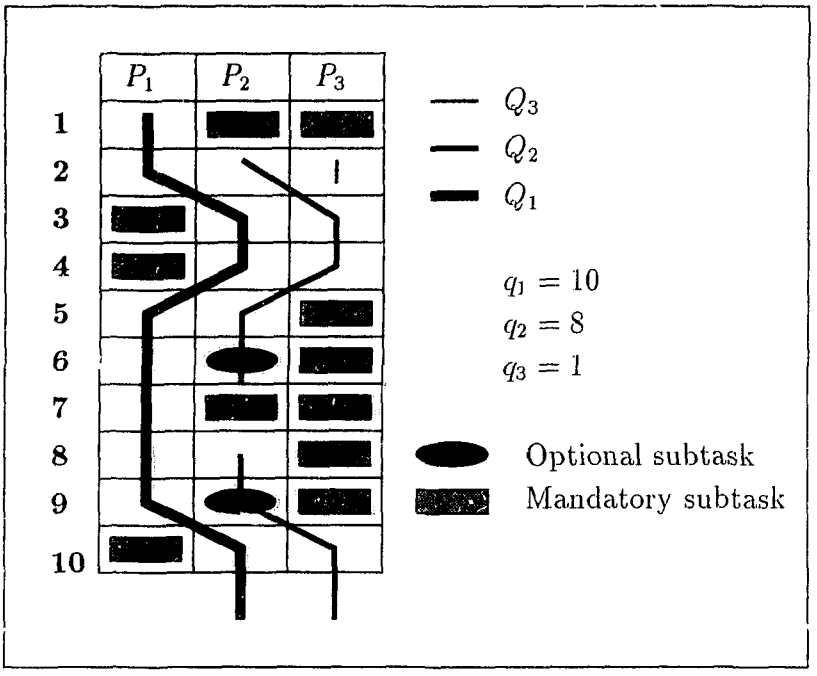


Figure 5.2: Alternate paths

unoccupied or occupied by an optional subtask. While constructing alternate paths, the time slots occupied by optional subtasks in a pre-run-time schedule is also considered as empty slots. Let $\{Q_1, \cdots, Q_x\}$ be the x distinct alternate paths available in $\{P_1, \cdots, P_m\}$ such that for some v, $1 \leq v \leq x$,

$$Q_v = \left\{(t,j) | j = \min_{1 \leq w \leq m} ((optional - in - a_{tj} \vee a_{tj} = 1) \wedge (t,w) \notin Q_u, 1 \leq u < v)\right\}$$

Figure 5.2 illustrates all the alternate paths present in a system with three processors, namely $\{P_1, P_2, P_3\}$. Note in Figure 5.2 that 6th and 9th time unit of P_2 are assumed to be occupied by optional subtasks, where as other time slots of all the processors are either empty slots or occupied by mandatory subtasks. Let $\{q_1, \cdots, q_x\}$ be the number of time slots in $\{Q_1, \cdots, Q_x\}$ where $q_v = cardinality(Q_v)$, $1 \leq v \leq x$. The algorithm for

```
Algorithm Find-Alternate-Paths;
for t = start time to finish time do
begin
    v = 1; ψ_t = 0;
    for j = 1 to m do
      if ((optional scheduled in a_tj ) ∨ ( a_tj = 1 ))
      begin
          ψ_t = ψ_t + 1;
          Q_v = Q_v ∪ {(t, j)};
          q_v = q_v + 1;
          v = v + 1;
      end;
end;
```

Figure 5.3: Algorithm to find alternate paths

finding all the alternate paths in $\{P_1, \cdots P_m\}$ is given in Figure 5.3. We notice that time slots occupied by the optional subtasks are included in alternate paths, if empty slots are not available.

In the next section, we introduce an adaptive scheduler to schedule $\{J_1, \cdots, J_n\}$ within D. If $\{J_1, \cdots, J_n\}$ is not schedulable in the empty slots within D then the algorithm removes minimum number of optionals from the pre-run-time schedule to allocate $\{J_1, \cdots, J_n\}$.

5.2 Adaptive Scheduler

Let $\{L_1, \cdots, L_k\}$ and $\{Q_1, \cdots, Q_x\}$ be the virtual paths and alternate paths with $\{l_1, \cdots, l_k\}$ and $\{q_1, \cdots, q_x\}$ time slots respectively. Let $\{c_1, \cdots, c_n\}$ be the computation times of $\{J_1, \cdots, J_n\}$. Our goal is to schedule $\{J_1, \cdots, J_n\}$ within D by removing minimum number of optionals from $\{Q_1, \cdots, Q_x\}$.

The necessary and sufficient conditions for obtaining a feasible schedule for $\{J_1, \cdots, J_n\}$ within D is given by the following equations.

$$\sum_{i=1}^{r} c_i \leq \sum_{v=1}^{r} q_v, \qquad 1 \leq r < x \tag{5.1}$$

$$\sum_{i=1}^{n} c_i \leq \sum_{v=1}^{x} q_v$$

If conditions 5.1 are not satisfied, then it is not possible to obtain a feasible schedule since $\sum_{v=1}^{x} q_v$ is the total number of time slots available (inclusive of all pre scheduled optional subtasks) to schedule $\{J_1, \cdots, J_n\}$. If $\{J_1, \cdots, J_n\}$ has to be scheduled without removing any optionals from $\{Q_1, \cdots, Q_x\}$, then $\{J_1, \cdots, J_n\}$ has to be scheduled within the empty slots that are available, namely $\{L_1, \cdots, L_k\}$. Hence the following necessary and sufficient conditions need to be satisfied if $\{J_1, \cdots, J_n\}$ has to be scheduled in $\{L_1, \cdots, L_k\}$.

$$\sum_{i=1}^{r} c_i \leq \sum_{v=1}^{r} l_v, \qquad 1 \leq r < k \tag{5.2}$$

$$\sum_{i=1}^{n} c_i \leq \sum_{v=1}^{k} l_v$$

If condition 5.2 is satisfied then there is no need to remove any optional subtask from the pre-run-time schedule since we can use LRTF algorithm to schedule $\{J_1, \cdots, J_n\}$ optimally. If condition 5.1 is satisfied and condition 5.2 is not satisfied then our goal is to schedule $\{J_1, \cdots, J_n\}$ within D by removing minimum number of time slots where optionals are scheduled in pre-run-time schedule. We use all the empty slots and remove only the least number of optionals since removing optionals will decrease the quality of the tasks that are scheduled in pre-run-time scheduler. Let Δ be the minimum number of

time slots needed additional to empty time slots to obtain a feasible schedule. We now present an algorithm that would remove exactly Δ optionals from the pre-run-time scheduler to schedule $\{J_1, \cdots, J_n\}$.

We first calculate the time slots necessary in addition to the empty time slots to schedule $\{J_1, \cdots, J_n\}$ if condition 5.2 is not satisfied. The extra time slots that are required in order to obtain a feasible schedule are given by

$$\{c_1 - l_1, \cdots, \sum_{i=1}^{r} c_i - \sum_{v=1}^{r} l_v, \cdots, \sum_{i=1}^{n} c_i - \sum_{v=1}^{k} l_v\}.$$

Since condition 5.2 is not satisfied, some of the entries are positive. If a feasible schedule has to be obtained for $\{J_1, \cdots, J_n\}$ then none of the entries should be positive. Hence to obtain a feasible schedule for $\{J_1, \cdots, J_n\}$, Δ time slots of optional subtasks need to be removed from the pre-run-time schedule, where Δ is given by

$$\Delta = max \left\{ \begin{array}{c} max \\ 1 \leq r < k \end{array} \left\{ \sum_{i=1}^{r} c_i - \sum_{v=1}^{r} l_v \right\}, \sum_{i=1}^{n} c_i - \sum_{v=1}^{k} l_v \right\}$$

Figure 5.4 illustrates a situation where condition 5.2 is violated. Let, for some z,

$$\Delta = \sum_{i=1}^{z} c_i - \sum_{v=1}^{z} l_v.$$

Therefore by adding Δ empty slots,

$$\sum_{i=1}^{z} c_i = \sum_{v=1}^{z} l_v + \Delta$$

Hence

$$\sum_{i=1}^{r} c_i \leq \sum_{v=1}^{r} l_v + \Delta, \qquad z+1 \leq r \leq k$$

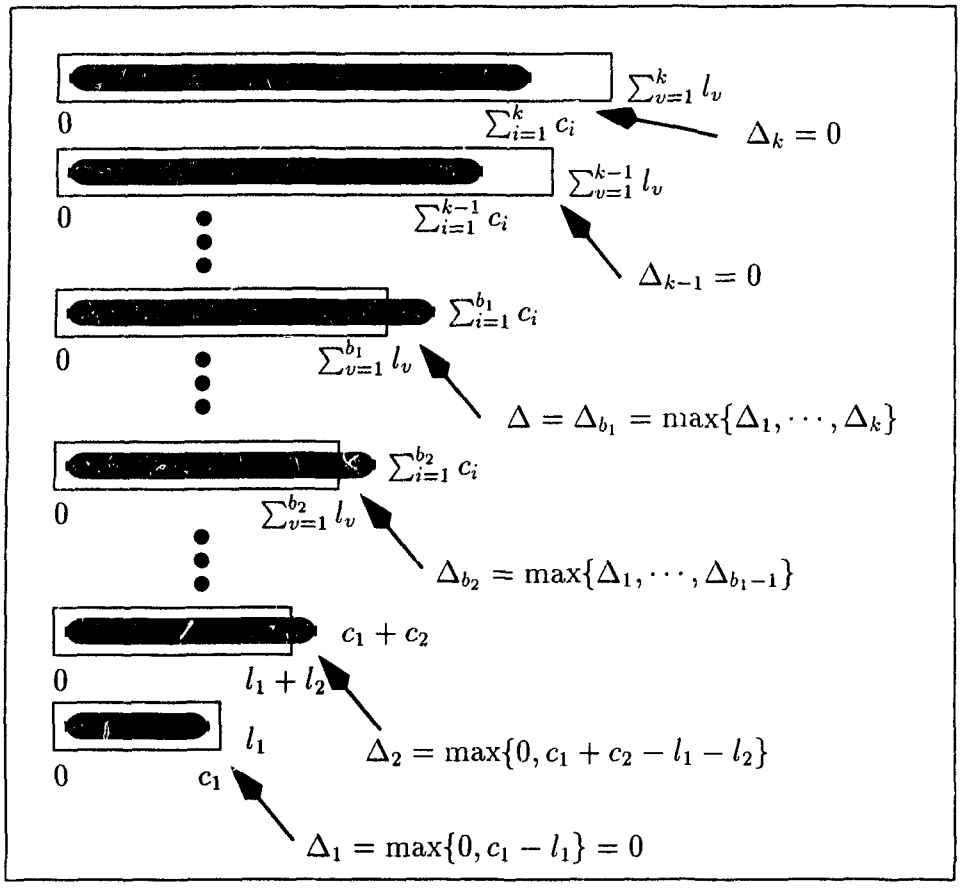


Figure 5.4: Calculation of Δ due to violation of schedulability condition

since Δ is the maximum additional time slots needed. Hence $\{J_z, \cdots, J_k\}$ satisfy the necessary and sufficient conditions if Δ additional time slots are added. We state the following Lemma and prove that the $\{J_1, \cdots, J_{z-1}\}$ also satisfy the necessary and sufficient conditions.

Lemma 5 *$\{J_1, \cdots, J_n\}$ can be scheduled without skipping the deadline if*

$$\Delta \leq \sum_{v=1}^{x} q_v - \sum_{v=1}^{k} l_v$$

where $\Delta = max\left\{ \max_{1 \leq r < k} \left\{ \sum_{i=1}^{r} c_i - \sum_{v=1}^{r} l_v \right\}, \sum_{i=1}^{n} c_i - \sum_{v=1}^{k} l_v \right\}$

Proof. We define $\{\Delta_1, \cdots, \Delta_k\}$ such that

$$\Delta_r = max\{0, \sum_{i=1}^{r} c_i - \sum_{v=1}^{r} l_v\}$$

Note that Δ_r denotes the amount of additional time slots needed to satisfy the conditions. Let $\{b_1, \cdots b_u\}$ be the indices of $\Delta's$ such that $\Delta_{b_{r+1}} = max\{\Delta_1, \cdots, \Delta_{b_r-1}\}$ where $\Delta_{b_1} = max\{\Delta_1, \cdots, \Delta_k\}$. Note that the index b_1 is the same as z. We will prove that the conditions are satisfied for $1 \leq r < b_1$.

We see that by adding Δ_{b_u} to $l_1 + \cdots + l_{b_u}$ the following equation is satisfied.

$$\sum_{i=1}^{b_u} c_i = \sum_{v=1}^{b_u} l_v + \Delta_{b_u}$$

Similarly, we can see that by adding $(\Delta_{b_{u-1}} - \Delta_{b_u})$ to $l_{1+b_u} + \cdots + l_{b_{u-1}}$ (and Δ_{b_u} already added to $l_1 + \cdots + l_{b_u}$) the following equation is satisfied.

$$\sum_{i=1}^{b_{u-1}} c_i = \sum_{v=1}^{b_{u-1}} l_v + \Delta_{b_{u-1}}$$

Hence by extending our argument, adding $(\Delta_{b_1} - \Delta_{b_2})$ to $l_{1+b_2} + \cdots + l_{b_1}$ the following equation is satisfied.

$$\sum_{i=1}^{b_1} c_i = \sum_{v=1}^{b_1} l_v + \Delta_{b_1}$$

Note that the minimum number of time slots needed to satisfy the necessary and sufficient conditions is $\Delta_{b_1} - \Delta_{b_2} + \Delta_{b_2} - \Delta_{b_3} + \cdots + \Delta_{b_{u-1}} - \Delta_{b_u} + \Delta_{b_u}$ which is Δ_{b_1} which is same as Δ. We claim that the maximum value Δ_{b_1} can have is $\sum_{v=1}^{x} q_v - \sum_{v=1}^{k} l_v$. If not, then

$$\sum_{i=1}^{b_1} c_i - \sum_{v=1}^{b_1} l_v > \sum_{v=1}^{x} q_v - \sum_{v=1}^{k} l_v$$

This would imply that

$$\sum_{i=1}^{b_1} c_i > \sum_{v=1}^{b_1} q_v + \sum_{v=1+b_1}^{x} q_v - \sum_{v=1+b_1}^{k} l_v$$

Since $\sum_{v=1+b_1}^{x} q_v - \sum_{v=1+b_1}^{k} l_v \geq 0$, we obtain

$$\sum_{i=1}^{b_1} c_i > \sum_{v=1}^{b_1} q_v$$

which violates the necessary and sufficient condition 5.1.

Hence we have proved that the minimum number of time slots that optionals occupy that need to be removed is Δ. We also showed that $\{J_1, \cdots, J_n\}$ satisfy the conditions to obtain a feasible schedule if Δ additional slots are available.

□

Figure 5.5 provides an adaptive scheduling algorithm that schedules the external tasks by removing Δ optionals. The following cases are considered by the algorithm.

case $\Delta = 0$: This implies that there is no need to remove any optionals to schedule $\{J_1, \cdots, J_n\}$ within D. Hence the empty slots are enough to schedule the task set, in which case the necessary and sufficient condition 5.2 should be satisfied. If the conditions are satisfied then LRTF algorithm can be used to schedule $\{J_1, \cdots, J_n\}$ within D optimally.

case $\Delta > 0$: A feasible schedule exists as long as condition 5.1 is satisfied. We schedule $\{J_1, \cdots, J_n\}$ using LRTF algorithm such that for every time unit, we schedule the largest remaining computation times of $\{J_1, \cdots, J_n\}$ in the empty time slots and time slots occupied by optionals. If an optional is removed then we decrement Δ. This is continued until we provide the necessary additional time slots, namely Δ to $\{J_1, \cdots, J_n\}$. If Δ optionals

are removed from the pre-run-time schedule then the remaining computation times of $\{J_1, \cdots, J_n\}$ are scheduled using LRTF algorithm, only in the empty time slots without removing any more optionals. As we are only removing Δ optionals and the remaining computation times of $\{J_1, \cdots, J_n\}$ can be optimally scheduled in the remaining time slots of $\{L_1, \cdots, L_k\}$, algorithm adaptive-scheduler is optimal.

We are now in a position to state the following theorem.

Theorem 10 *The Adaptive-Scheduler algorithm schedules* $\{J_1, \cdots J_n\}$ *optimally within D by removing minimum number of pre-scheduled optionals.*

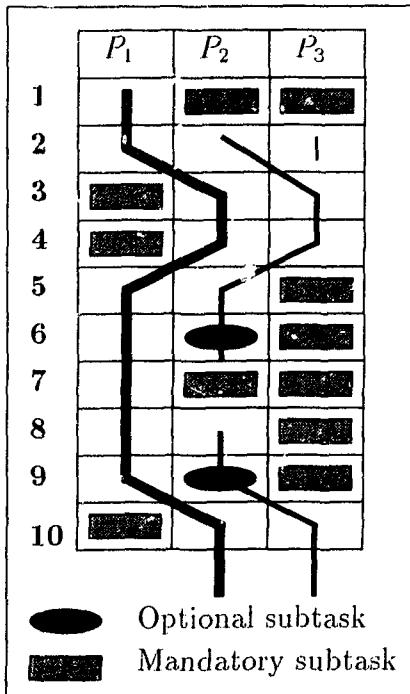
Figure 5.6 illustrates adaptive-scheduler algorithm with an example. Three tasks, $\{J_1, J_2, J_3\}$ are considered with computation times $\{10, 7, 1\}$ respectively. It can be easily seen that the conditions for obtaining a feasible schedule is satisfied if all the empty time slots and all the time slots where optionals are scheduled are considered. It can also be seen that the conditions to obtain a schedule without removing optional subtasks from the pre-run-time scheduler is not satisfied. The calculation shows that $\Delta = 1$. We schedule $\{J_1, J_2, J_3\}$ using LRTF algorithm by considering optionals as empty time slots in [1..6] until $\Delta = 0$. We schedule the remaining computation times of $\{J_1, J_2, J_3\}$ using LRTF algorithm in [7..10] considering only the empty time slots and without removing any pre scheduled optional subtasks.

5.3 Summary

In this chapter, we assumed that the pre-run-time schedule consists of a task set that is fault-tolerant. We derived the necessary and sufficient conditions for scheduling a set of tasks that arrive during run-time with same deadline. We introduced an adaptive scheduler that schedules a set of tasks that arrive during run-time. The scheduler dynamically removes, if necessary, optimal

```
Algorithm Adaptive-Scheduler;
begin

/* Assume c_1 ≥ c_2 ≥ ··· ≥ c_n */
/* Check if optionals need to be removed */

if ((Σ_{i=1}^{r} c_i ≤ Σ_{v=1}^{r} l_v, 1 ≤ r ≤ k − 1) and (Σ_{i=1}^{n} c_i ≤ Σ_{v=1}^{k} l_v))
begin

    Consider Optionals as not available and
    schedule {J_1, ···, J_n} in {L_1, ···, L_k} using LRTF algorithm;

end;
else begin /* some optionals need to be removed */
      if ((Σ_{i=1}^{r} c_i ≤ Σ_{v=1}^{r} q_v, 1 ≤ r ≤ x − 1) and (Σ_{i=1}^{n} c_i ≤ Σ_{v=1}^{x} q_v))
      begin
          Δ = max{ max_{1≤r≤k−1} {Σ_{i=1}^{r} c_i − Σ_{v=1}^{r} l_v}, Σ_{i=1}^{n} c_i − Σ_{v=1}^{k} l_v };
          while (Δ > 0)
          begin
              Schedule remaining computation times of {J_1, ···, J_n}
              in {Q_1, ···, Q_x} using LRTF algorithm.
              Decrement Δ if optionals are removed;

          end;
          /* Δ = 0 */
          Schedule remaining computation times of {J_1, ···, J_n} in empty slots
          using LRTF algorithm without removing any optionals;

      end;
    end;
end.
```

Figure 5.5: Run-time adaptive scheduler



$q_1 = 10 \quad l_1 = 10 \quad c_1 = 10$
$q_2 = 8 \quad l_2 = 6 \quad c_2 = 7$
$q_3 = 1 \quad l_3 = 1 \quad c_3 = 1$

Condition for feasible schedule :-

$$c_1 \leq q_1$$
$$c_1 + c_2 \leq q_1 + q_2$$
$$c_1 + c_2 + c_3 \leq q_1 + q_2 + q_3$$

Calculating Δ :-

$c_1 \leq l_1$
$c_1 + c_2 > l_1 + l_2$
$c_1 + c_2 + c_3 > l_1 + l_2 + l_3$

$\Delta_1 = 0$
$\Delta_2 = max\{0, c_1 + c_2 - l_1 - l_2\} = 1$
$\Delta_3 = max\{0, c_1 + c_2 + c_3 - l_1 - l_2 - l_3\} =$
$\Delta_{b_1} = max\{\Delta_1, \Delta_2, \Delta_3\} = \Delta_2 = 1$
$b_1 = 2; \Delta = 1$

	P_1	P_2	P_3
1	J_1		
2	J_1	J_2	J_3
3		J_1	J_2
4		J_1	J_2
5	J_1	J_2	
6	J_1	J_2	
7	J_1		
8	J_1	J_2	
9	J_1		
10		J_1	J_2

Δ	Comp.times before.	Tasks	Comp.times after
1	{10,7,1}	$\{J_1, J_2, J_3\}$	{9,7,1}
1	{9,7,1}	$\{J_1, J_2, J_3\}$	{8,6,0}
1	{8,6,0}	$\{J_1, J_2\}$	{7,5,0}
1	{7,5,0}	$\{J_1, J_2\}$	{6,4,0}
1	{6,4,0}	$\{J_1, J_2\}$	{5,3,0}
1	{5,3,0}	$\{J_1, J_2\}$	{4,2,0}
0	{4,2,0}	$\{J_1, J_2\}$	{3,2,0}
0	{3,2,0}	$\{J_1, J_2\}$	{2,1,0}
0	{2,1,0}	$\{J_1, J_2\}$	{1,1,0}
0	{1,1,0}	$\{J_1, J_2\}$	{0,0,0}

Figure 5.6: Illustration of adaptive scheduler

number of optional subtasks that have been scheduled in pre-run-time schedule. We proved that the adaptive scheduler is optimal. Adaptive run-time scheduler finds application in areas such as robotics and communication systems, where pre-scheduled optionals can be removed to accomodate tasks arriving at run-time. In the next chapter we consider scheduling a set of tasks that have precedence constraints.

Chapter 6

Scheduling Tasks with Precedence Constraints

In this chapter, we first describe the structure of tasks with precedence constraints using a precedence graph. We then provide an algorithm to decompose the precedence graph into processes, and find the number of tasks that can be executed in parallel and their corresponding execution times. We will introduce a fault-tolerant technique based on imprecise computation approach for tasks with precedence constraints. We will consider the case where tasks with precedence constraints are scheduled during pre-run-time. We then apply the adaptive run-time scheduler to obtain a schedule by removing optimal number of optionals during run-time to accomodate external run-time tasks.

6.1 Task Characteristics

6.1.1 General Precedence Graph

We represent a set of tasks $T = \{T_1, \cdots, T_n\}$ by $(T, <)$ where $<$ is a partial order on T specifying precedence constraints. For example if $T_u < T_v$ then execution of task T_u must be finished before T_v can be started. T_u is called a *predecessor* of T_v and T_v a *successor* of T_u. If there exists no task T_w such

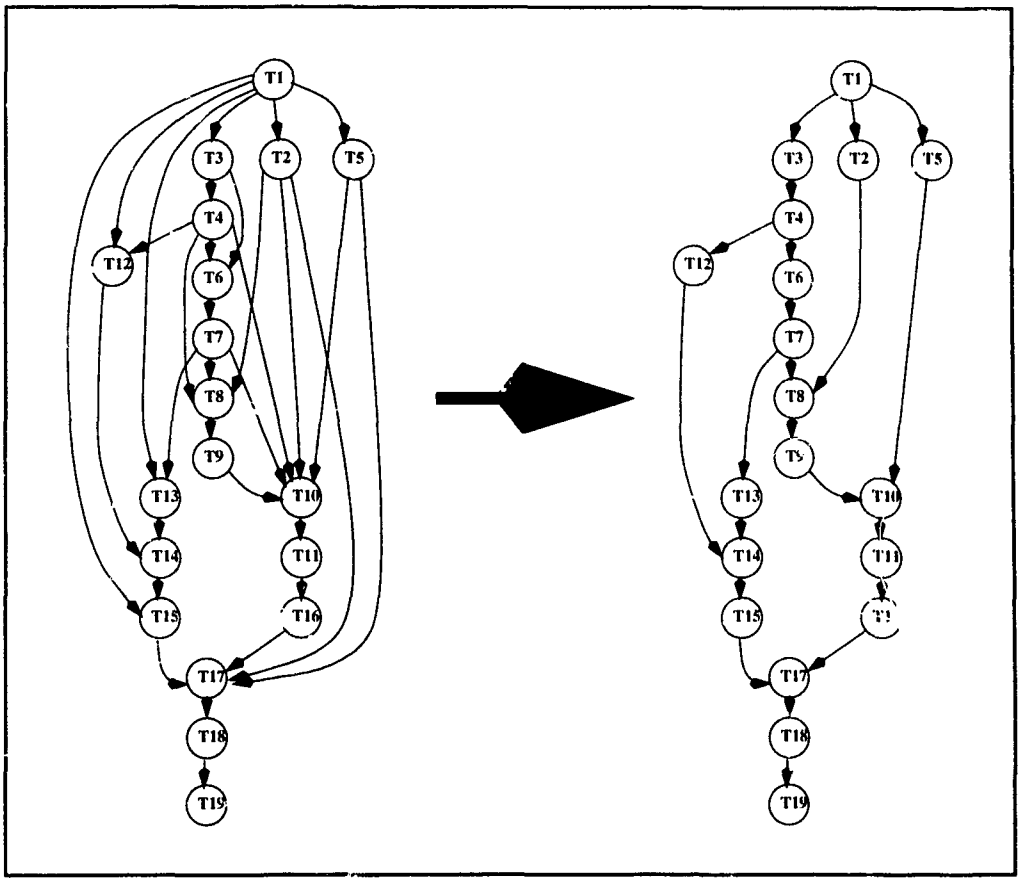


Figure 6.1: Precedence graph and its simplified version

that $T_u < T_w < T_v$ then T_u is called an *immediate predecessor* of T_v and T_v *an immediate successor* of T_u. Examples of such constraints are shown in Figure 6.1. Note that we use T for a set of tasks with precedence constraints where as we used J to represent a set of independent tasks in earlier chapters.

6.1.2 Simplification of Precedence Graph

Precedence graph is formed by introducing edges between nodes to represent various precedence constraints such as control dependency and data dependency. We assume that the resultant precedence graph is acyclic. The

redundant edges in the resultant precedence graph can be removed to obtain a simplified task graph that is represented as a Hesse diagram [74]. For example, in Figure 6.1 there is an edge from T_3 to T_4 and T_4 to T_6. Hence the edge from T_3 to T_6 is redundant since T_6 cannot be executed before T_3 and T_4. This can be done in polynomial time since redundant paths can be identified in polynomial time using Dijkstra's shortest path algorithm by assuming negative weights for edges [75]. Figure 6.1(b) illustrates the simplified task graph of Figure 6.1(a).

6.1.3 Decomposition of Precedence Graph

The tasks in the precedence graph are decomposed and grouped into subsets of tasks such that the subsets can be executed in parallel. In this section, we develop an algorithm to do this decomposition. The precedence relations between the tasks are taken care of while decomposing the precedence graph. Consider the example illustrated in Figure 6.2. We notice that the task set consists of four tasks $\{T_1, \cdots, T_4\}$ with execution times $\{5, 2, 3, 4\}$ respectively. We note that T_1 has to get executed first. Once T_1 is executed, then T_2 and T_3 can be executed in parallel for 2 time units each, after which T_3 is continued for 1 more time unit. T_4 is finally executed after T_2 and T_3 have completed.

A *process change sequence* (PCS) is defined as the sequence of the number of processes that can be executed in parallel. PCS for the example provided in Figure 6.2 equals (1,2,1,1). Similarly, a *process time requirement sequence* (PTRS) is defined as the sequence of computation time required to execute the corresponding processes in the process change sequence. For the example given in Figure 6.2, PTRS is (5,2,1,4). Since the precedence relations between various tasks are known *apriori*, PCS and PTRS can be calculated pre-run-time.

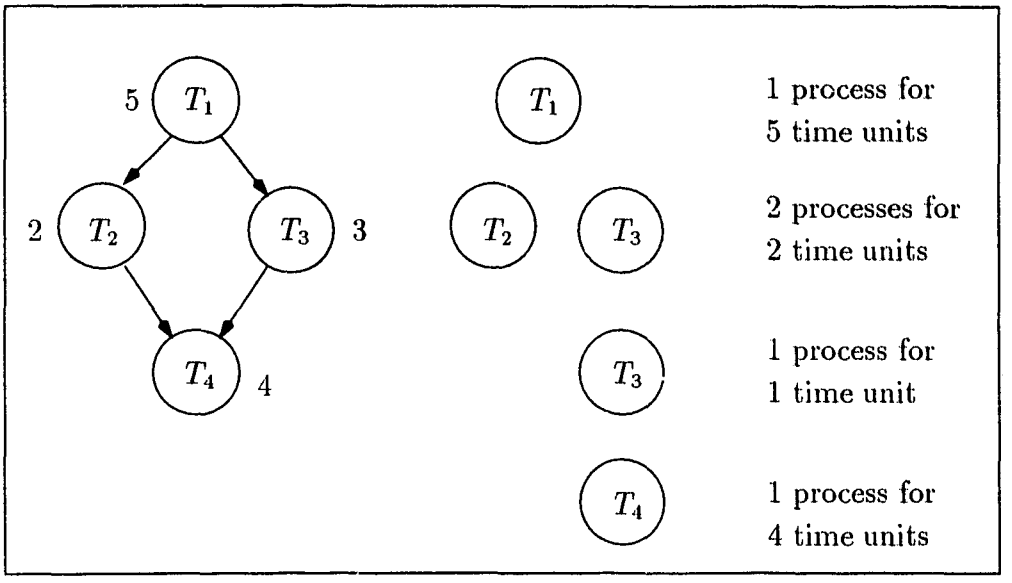


Figure 6.2: Decomposition of a precedence graph

Figure 6.3 illustrates the algorithm Find-PCS-PTRS to compute PCS and PTRS of a general precedence graph. The algorithm in Figure 6.3 does the decomposition. Let T be the set of tasks with precedence constraints represented by a precedence graph. A set of tasks that do not have any predecessors is defined as start nodes. A set of tasks is defined as a successor nodes to a subset of T if the tasks in the set are immediate successors of the subset of T. We begin from the start nodes of the precedence graph. It can be easily seen that the start nodes can be executed in parallel as they do not have any predecessors. Once a task in the start nodes complete its execution, all the successor nodes to that task can be executed in parallel as long as they do not have any precedence constraints with the unfinished tasks in start nodes. In general, we denote FIN to be the set of tasks that have already been considered, and $SUCC$ to be the set of tasks that are successor nodes to the nodes in FIN. We calculate the maximum time for which tasks in $SUCC$ can be executed in parallel. The number of

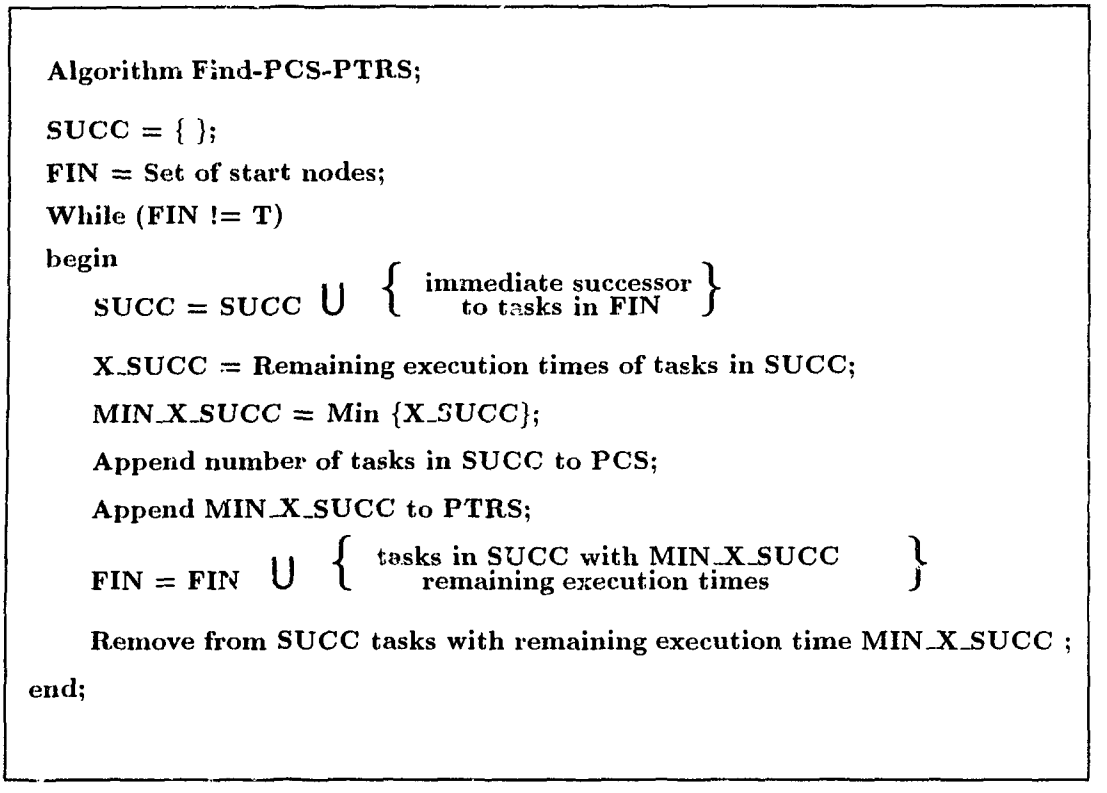


Figure 6.3: Algorithm Find-PCS-PTRS

tasks in $SUCC$ is appended to PCS and the minimum execution time is appended to $PTRS$. Figure 6.4 illustrates the steps of the algorithm Find-PCS-PTRS to find PCS and PTRS of the precedence graph given in Figure 6.1. We assumed that the execution times of the mandatory subtasks are $(4, 12, 4, 8, 16, 12, 8, 8, 14, 4, 3, 8, 4, 4, 4, 2, 8, 2, 3)$ and the execution times of the optional subtasks are $(1, 3, 1, 2, 4, 3, 2, 2, 1, 1, 0, 2, 1, 1, 1, 0, 2, 0, 0)$ respectively. We note that the PCS is given by $(1, 3, 3, 3, 2, 1, 1, 2, 2, 2, 1, 1, 1, 1, 1, 1, 1)$ and the $PTRS$ is given by $(4, 4, 8, 4, 4, 4, 8, 4, 4, 4, 10, 4, 3, 2, 8, 2, 3)$.

SUCC	X_SUCC	MIN	PCTS	PCS	PTRS	FIN
{ }	{ }	-	$\{T_1\}$	{1}	{4}	$\{T_1\}$
$\{T_3, T_2, T_5\}$	{4,12,16}	4	$\{T_3, T_2, T_5\}$	{3}	{4}	$FIN \cup \{T_3\}$
$\{T_2, T_5, T_4\}$	{8,12,8}	8	$\{T_2, T_5, T_4\}$	{3}	{8}	$FIN \cup \{T_2, T_4\}$
$\{T_5, T_{12}, T_6\}$	{4,8,12}	4	$\{T_5, T_{12}, T_6\}$	{3}	{4}	$FIN \cup \{T_5\}$
$\{T_{12}, T_6\}$	{4,8}	4	$\{T_{12}, T_6\}$	{2}	{4}	$FIN \cup \{T_{12}\}$
$\{T_6\}$	{4}	4	$\{T_6\}$	{1}	{4}	$FIN \cup \{T_6\}$
$\{T_7\}$	{8}	8	$\{T_7\}$	{1}	{8}	$FIN \cup \{T_7\}$
$\{T_{13}, T_8\}$	{4,8}	4	$\{T_{13}, T_8\}$	{2}	{4}	$FIN \cup \{T_{13}\}$
$\{T_{14}, T_8\}$	{4,4}	4	$\{T_{14}, T_8\}$	{2}	{4}	$FIN \cup \{T_{14}, T_8\}$
$\{T_{15}, T_9\}$	{4,14}	4	$\{T_{15}, T_9\}$	{2}	{4}	$FIN \cup \{T_{15}\}$
$\{T_9\}$	{10}	10	$\{T_9\}$	{1}	{10}	$FIN \cup \{T_9\}$
$\{T_{10}\}$	{4}	4	$\{T_{10}\}$	{1}	{4}	$FIN \cup \{T_{10}\}$
$\{T_{11}\}$	{3}	3	$\{T_{11}\}$	{1}	{3}	$FIN \cup \{T_{11}\}$
$\{T_{16}\}$	{2}	2	$\{T_{16}\}$	{1}	{2}	$FIN \cup \{T_{16}\}$
$\{T_{17}\}$	{8}	8	$\{T_{17}\}$	{1}	{8}	$FIN \cup \{T_{17}\}$
$\{T_{18}\}$	{2}	2	$\{T_{16}\}$	{1}	{2}	$FIN \cup \{T_{18}\}$
$\{T_{19}\}$	{3}	3	$\{T_{19}\}$	{1}	{3}	$FIN \cup \{T_{19}\}$

Figure 6.4: PCS, PTRS for tasks in a precedence graph

6.2 Pre-Run-Time Scheduling

In this section, we introduce a heuristic algorithm based on McNaughton's rule [51] for scheduling a set of tasks with precedence constraints. We assume that each task has been implemented using imprecise computation technique. Hence each task has been decomposed into two subtasks, namely: mandatory subtasks and optional subtasks. Let $\{M_1, \cdots, M_n\}$ be the set of mandatory subtasks and $\{O_1, \cdots, O_n\}$ be the set of optional subtasks of the task set $\{T_1, \cdots, T_n\}$. Let $\{m_1, \cdots, m_n\}$ be the execution times of the mandatory subtasks and $\{o_1, \cdots, o_n\}$ be the execution times of the optional subtasks of $\{T_1, \cdots, T_n\}$. We define PCS_{acc} and $PTRS_{acc}$ to be the PCS and PTRS of $\{T_1, \cdots, T_n\}$ with execution time $\{m_1 + o_1, \cdots, m_n + o_n\}$. Similarly, we define PCS_{app} and $PTRS_{app}$ to be the PCS and PTRS of $\{T_1, \cdots, T_n\}$ with execution time $\{m_1, \cdots, m_n\}$. Our goal is to schedule T with computation time $\{m_1 + o_1, \cdots, m_n + o_n\}$ to obtain an accurate result. If it is not feasible, then we will try to schedule T with computation time $\{m_1, \cdots, m_n\}$ to obtain an acceptable approximate result.

We calculate PCS_{acc} and $PTRS_{acc}$ using the algorithm provided in Figure 6.3 by considering both mandatory and optional for every task in the precedence graph. Similarly, we calculate PCS_{app} and $PTRS_{app}$ by considering mandatories alone for every task in the precedence graph. Let $\{P_1, \cdots, P_\alpha\}$ and $\{B_1, \cdots, B_\alpha\}$ be the PCS_{acc} and $PTRS_{acc}$ respectively. Similarly, $\{R_1, \cdots, R_\beta\}$ and $\{C_1, \cdots, C_\beta\}$ be the PCS_{app} and $PTRS_{app}$ respectively. Hence by executing PCS_{acc} we obtain an accurate result for the set of tasks with precedence constraints where as, by executing PCS_{app} we obtain an approximate result.

For obtaining an accurate result for every i, we schedule P_i processes with execution time B_i. Let $\{Z_i^1, \cdots, Z_i^{P_i}\}$ be the P_i processes with execution time

B_i. We note that $\{Z_i^1, \cdots, Z_i^{P_i}\}$ has to be scheduled before $\{Z_{i+1}^1, \cdots, Z_{i+1}^{P_{i+1}}\}$. We use McNaughton's rule [51] to schedule $\{Z_i^1, \cdots, Z_i^{P_i}\}$ for B_i time units each in m processors. According to McNaughton's rule tasks are scheduled by *bin packing* approach filling processor by processor. It can be easily seen that the schedule length for $\{Z_i^1, \cdots, Z_i^{P_i}\}$ is $max\{B_i, \frac{B_i P_i}{m}\}$. Hence the schedule length for $\{T_1, \cdots, T_n\}$, denoted as SL_{acc} is given by

$$SL_{acc} = \sum_{i=1}^{\alpha} max\{B_i, \frac{B_i P_i}{m}\}$$

Similarly for obtaining an approximate result, we use McNaughton's rule to schedule R_i processes of C_i time units each in m processors. The schedule length SL_{app} for scheduling $\{T_1, \cdots, T_n\}$ is given by

$$SL_{app} = \sum_{i=1}^{\beta} max\{C_i, \frac{C_i R_i}{m}\}$$

If $SL_{acc} \leq D$, then we schedule both mandatory and optional subtasks for every task in T using the heuristic algorithm given in Figure 6.5. Similarly, If $SL_{app} \leq D \leq SL_{acc}$, then we schedule only the mandatory subtasks using the *bin-packing* algorithm given in Figure 6.5. If $SL_{app} > D$, then it is not possible to schedule P within D.

To analyze the heuristic, we compare the schedule length obtained for any task set using our algorithm to that of the smallest schedule length that is possible. We present the analysis for a task set with tasks containing both mandatory and optional parts. We define a critical path of the precedence graph as the path from the start node to end node with the largest cumulative execution time. The smallest schedule length that is possible for a set of tasks with precedence constraints is given by the total execution time of all nodes in its critical path [76]. Figure 6.6 demonstrates the critical path for the example given in Figure 6.1.

```
Algorithm Pre-Run-Time Heuristic;
begin
    (R_1, R_2, ..., R_β) = PCS_app
    (P_1, P_2, ..., P_α) = PCS_acc
    (B_1, B_2, ..., B_α) = PTRS_acc
    (C_1, C_2, ..., C_β) = PTRS_app
    if (SL_acc ≤ D) then
        for i = 1 to α do
        begin
            z_i = max{B_i, (B_i * P_i)/m};
            Schedule P_i processes in z_i time units using McNaughton Rule;
        end;
    else if (SL_app ≤ D) then
        for i = 1 to β do
        begin
            z_i = max{C_i, (C_i * R_i)/m};
            Schedule R_i processes in z_i time units using McNaughton Rule;
        end;
end.
```

Figure 6.5: Pre-run-time scheduler

Note that for larger values of m, the minimum schedule length possible is $\sum_{i=1}^{\alpha} B_i$ since every task that is ready can be scheduled completely in a processor without being preempted. In this case, the heuristic performs close to optimality since the minimum schedule length for the heuristic is given by

$$\sum_{i=1}^{\alpha} max\{B_i, \frac{B_i P_i}{m}\} = \sum_{i=1}^{\alpha} B_i$$

Note that the schedule length is $\sum_{i=1}^{\alpha} B_i$ since it cannot be less than the execution times of all tasks in the longest path from start node to end node. For smaller values of m we notice that for any B_i and P_i, the schedule length

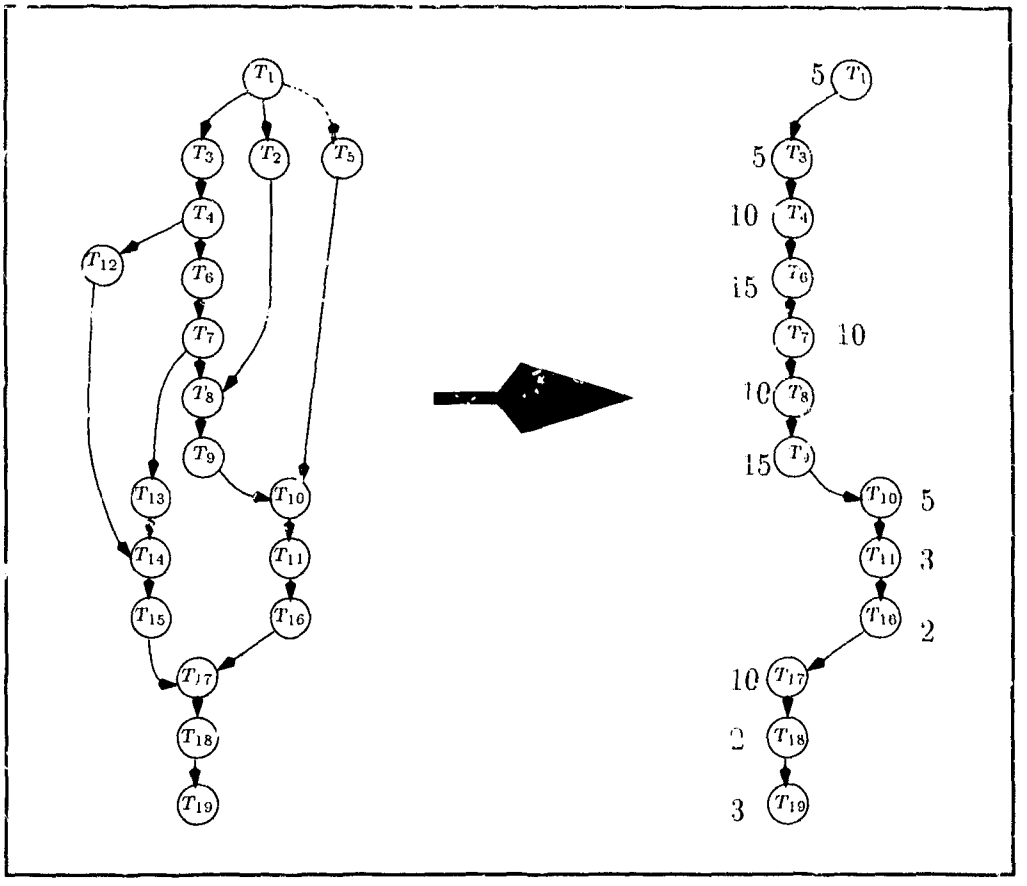


Figure 6.6: Critical Path

is given by $\frac{B_i P_i}{m}$. If B_i is the minimum schedule length possible then the ratio of schedule length of heuristic to the schedule length of optimal schedule length is given by P_i/m.

6.3 Adaptive Scheduling of Run-Time Tasks

In the previous section, we had used McNaughton's rule for scheduling a set of tasks within D. If mandatory subtasks are scheduled, and none of the optional subtasks are scheduled then optimal run-time scheduling algorithms

presented in Chapter 4 can be used to schedule tasks that arrive during run-time. On the contrary, if both mandatory and optional subtasks of every task in the task set are scheduled during pre-run-time then optimal adaptive run-time scheduling algorithms presented in Chapter 5 can be used to schedule tasks that arrive during run-time. In this section, we first apply the pre-run-time scheduling algorithm for an application in robotic systems. We then use the adaptive run-time scheduling algorithm described in Chapter 5 to remove minimum number of optionals that have been scheduled in pre-run-time schedule.

Parallel architectures and scheduling algorithms for multiprocessors are used extensively in real-time control of robotic systems [4,77,78]. With the advancement of robot control theory, there has been increased demand for more sophisticated control of a robot arm facilitating precise motion along a prescribed trajectory for a varying or unknown load. In most cases, advanced control schemes involve the computation of the appropriate input-generalized forces based on measured data of the displacement, velocities of all the joints and the values of the accelerations. There are a number of ways to compute the input-generalized forces among which the Newton-Euler formulation has been shown to be the most efficient [79].

The dynamic coupling of the Newton-Euler equation creates precedence relationships among the computational tasks. Figure 6.1 illustrates the precedence graph for the Newton-Euler equations of motion. The notations followed in the decomposition of the precedence graph is same as the one followed by Yoshikawa in [80]. We introduce fault-tolerance to the tasks with precedence constraints, by incorporating the imprecise computation mechanism for the Newton-Euler formulation.

The assumed execution times for the mandatory and optionals are given in pre-run-time schedule of Figure 6.7. The PCS for the task graph is:
(1,3,3,3,2,1,1,2,2,2,1,1,1,1,1,1,1)

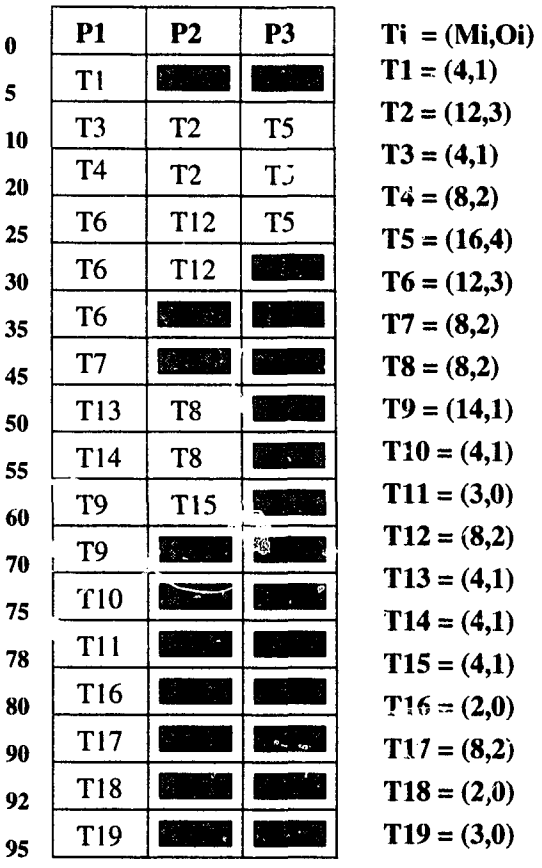


Figure 6.7: Pre-run-time schedule for task graph

and the corresponding $PTRS$ is given by:

(5,5,10,5,5,5,10,5,5,5,10,5,3,2,10,2,3).

Note that the dotted patches in the schedule denotes empty slots and black patches denote the optionals. Figure 6.8 decomposes every task in the pre-run-time schedule into mandatory and optional subtasks.

Figure 6.9 elaborates the slot wise assignment and illustrates the alternate paths. To avoid confusion we haven't illustrated all possible virtual paths. The empty slots are represented by dotted squares and the optionals are represented by black circles. We note that l_1, l_2, and l_3 are 75, 55 and 0

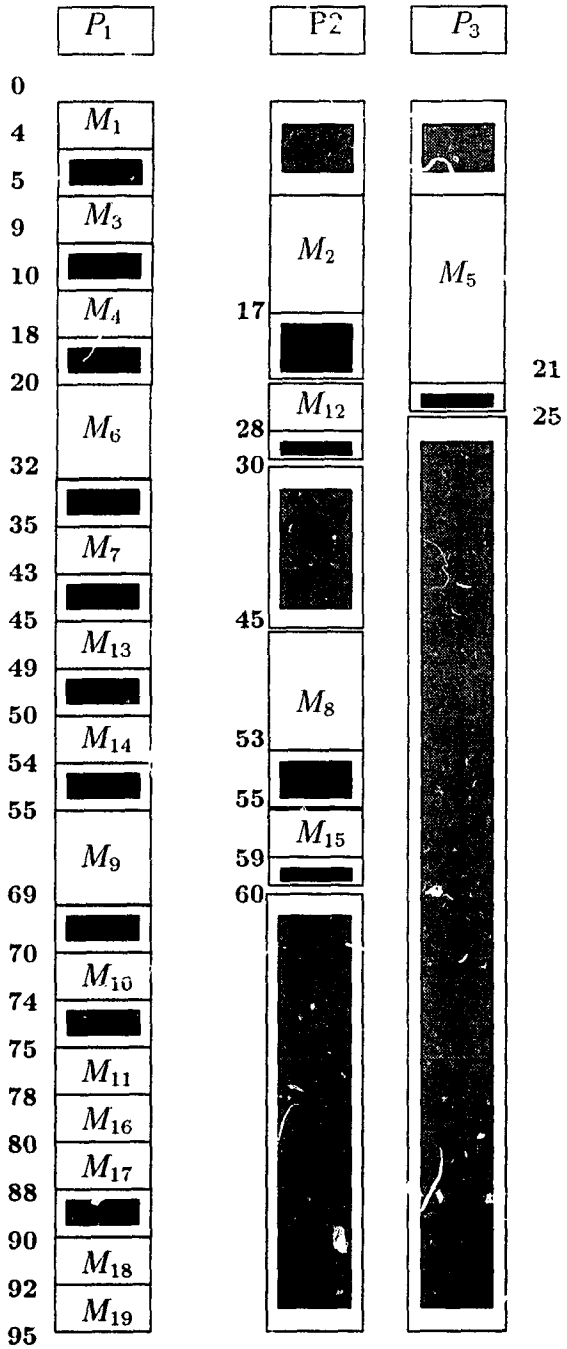


Figure 6.8: Schedule showing mandatory and optional tasks

respectively. Similarly q_1, q_2, and q_3 are 83, 63 and 11 respectively.

Consider the external task set $\{J_1, \cdots, J_{12}\}$ with the execution times $(20, 20, 15, 15, 14, 13, 13, 10, 10, 10, 5, 5)$. The cumulative execution time of all the external tasks exceeds that of the available empty slots. Therefore some optionals need to be removed to accommodate all the external tasks in the final schedule. This is accomplished by calculating Δ which is 20. Thus we need to remove 20 optionals and schedule J_is apart from using the 130 empty slots. Figure 6.10 illustrates the application of algorithm Adaptive-Scheduler provided in Figure 5.5.

We introduce the adaptability factor as a metric for measuring the adaptability, namely $1 - \frac{\sum_{i=1}^{s} m_i}{\sum_{i=1}^{s}(m_i+o_i)}$. Note that as $o_i \to \infty$, $1 - \frac{\sum_{i=1}^{s} m_i}{\sum_{i=1}^{s}(m_i+o_i)} \to 1$. Therefore adaptability of the run-time scheduler is a function of the cumulative execution time of the optionals. If the tasks are without any fault-tolerance then $\sum_{i=1}^{s} o_i$ is 0 and $1 - \frac{\sum_{i=1}^{s} m_i}{\sum_{i=1}^{s}(m_i+o_i)}$ is 0. Figure 6.11 illustrates the adaptability of the run-time scheduler due to incorporation of imprecise computation technique for the example we had considered. We plot $1 - \frac{\sum_{i=1}^{s} m_i}{\sum_{i=1}^{s}(m_i+o_i)}$ against the cumulative execution time of external tasks at run-time. The dark region demonstrates the amount of extra external tasks the proposed scheduler can schedule after utilizing all empty slots without skipping the deadline.

6.4 Summary

In this chapter we represented the set of tasks with precedence constraints as a general precedence graph. We modelled by decomposing the set of tasks into processes and derived the process control sequence and process time requirement sequence. We introduced imprecise computation technique for

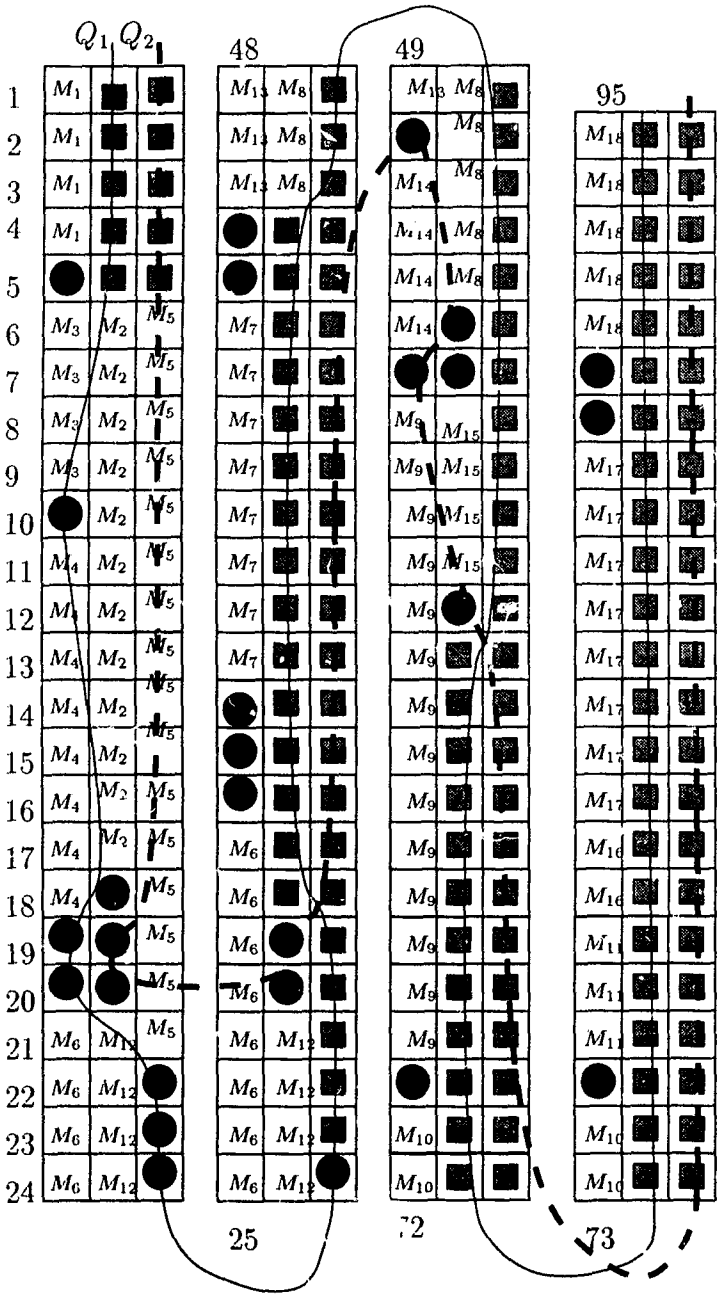


Figure 6.9: Detailed schedule showing alternate paths

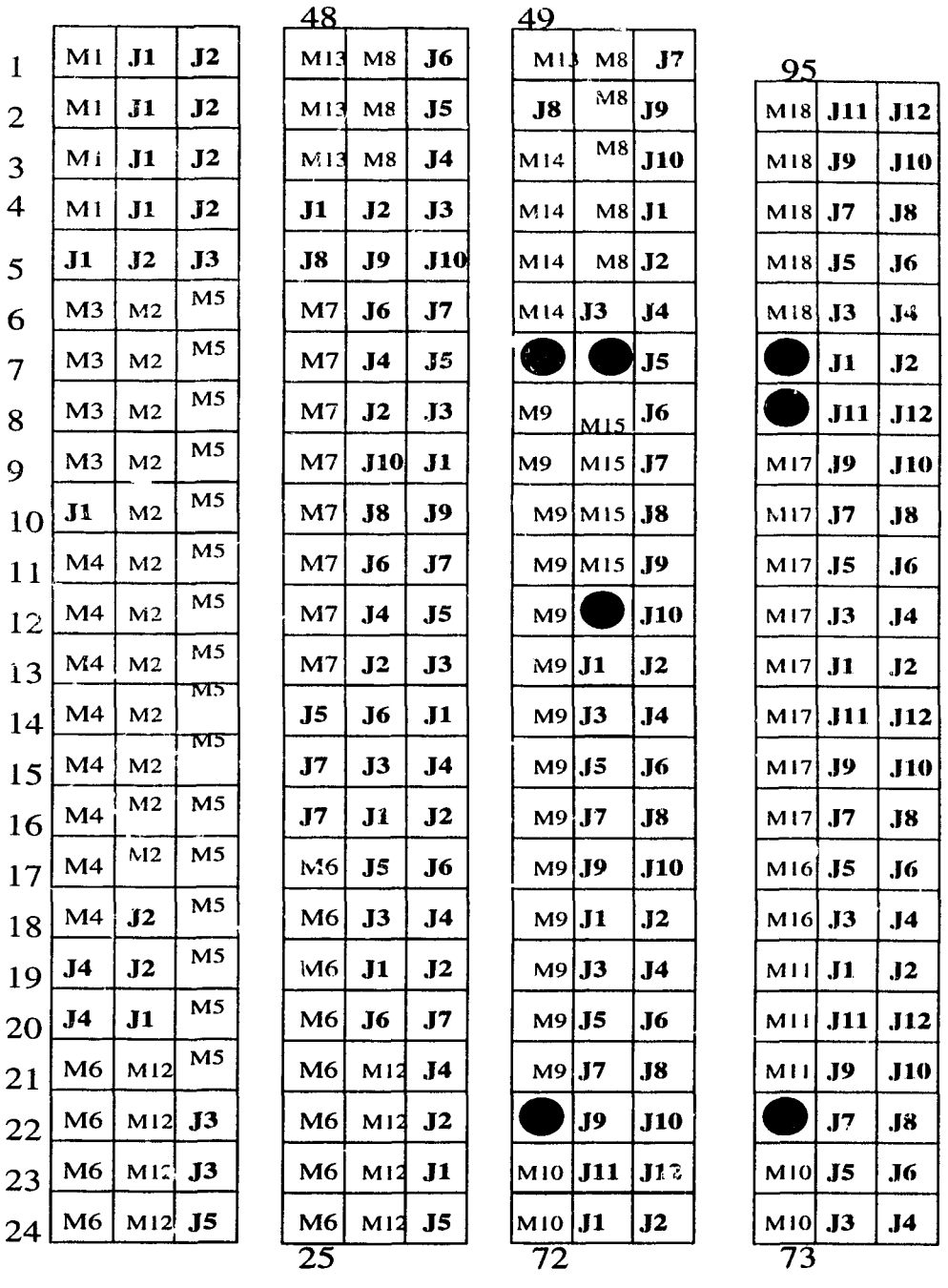


Figure 6.10: Final schedule

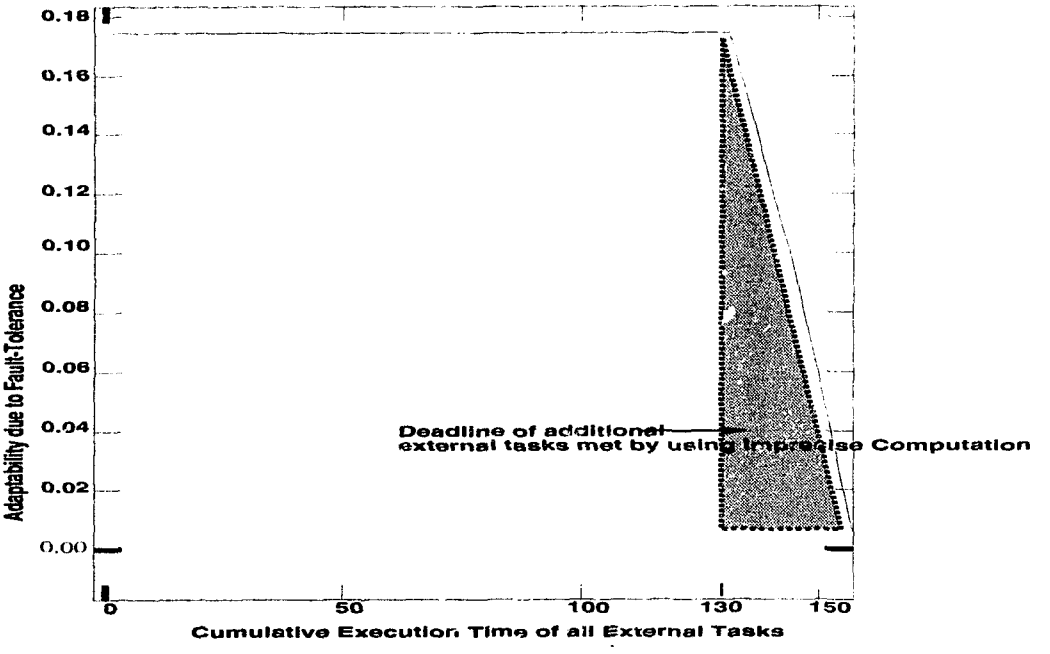


Figure 6.11: Performance of adaptive run-time scheduler for the robotic example

the set of tasks by representing it as an accurate precedence graph and an approximate precedence graph.

We considered the problem of scheduling a set of tasks with precedence constraints at pre-run-time within its deadline. A heuristic based on McNaughton's rule was used to schedule the set of tasks within D. The algorithm schedules all the mandatory and optional subtasks if an accurate result can be obtained within D. If not then the schedulability of mandatory subtasks alone is checked to obtain an acceptable approximate result. We considered the problem of scheduling a set of independent tasks that arrive during run-time using our proposed adaptive scheduler once all the mandatory and optional subtasks of tasks in task set has been scheduled at pre-run-time. An example based on an application in robotics was also provided.

Chapter 7

Run-Time Scheduling of Parallel Program

In the previous chapter, we introduced a pre-run-time fault-tolerant scheduling algorithm to schedule a set of tasks that have arbitrary precedence constraints. We scheduled the external tasks that arrive during run-time on top of the pre-run-time schedule. In this chapter we address a similar problem but consider run-time scheduling in the presence of external tasks. We consider a parallel program represented as a set of tasks with precedence constraints using a control-flow graph. For fault-tolerance the parallel program is decomposed into an accurate parallel program and an approximate parallel program. We assume that the external tasks are queued and a decision of whether to schedule an accurate parallel program or an approximate parallel program is taken dynamically depending on the number of external tasks. Once the execution starts, only the external tasks in the queue are executed along with the tasks with the parallel program and the tasks that arrive during the execution are queued in an external queue. We propose the fault-tolerant scheduling algorithm and derive the schedulability conditions.

7.1 Structure of Parallel Program

Let $\{T_1, \cdots, T_n\}$ be the set of tasks that parallel program contain with various control-flow and data-flow constraints. Let $\{t_1, \cdots, t_n\}$ be the execution times of $\{T_1, \cdots, T_n\}$. We assume that the parallel program is represented using a task control-flow graph [81]. A task control-flow graph represents the control flow of the execution of various tasks of a program. A task control-flow graph is a collection of subgraphs which can be of three different types. Figure 7.1 shows three different subgraphs that are generally used in a task control graph namely: *and-fork* to *and-join* subgraph, *sequential* subgraph, and *loop* subgraph. The *and-fork* to *and-join* subgraph enables several execution paths. Tasks in all the execution paths need to be computed before they join at an *and-join* node. A collection of tasks are represented as a *sequential* subgraph when they need to be executed in series. A *loop* subgraph represents a collection of tasks that need to be executed within a loop on the affirmation of a decision node.

7.2 Task and System Characteristics

We consider a system with m processors that have a shared memory architecture. Let P be the parallel program under consideration with n tasks. Let $\{T_1, \cdots, T_n\}$ be the n tasks of P with $\{M_1, \cdots, M_n\}$ and $\{O_1, \cdots, O_n\}$ being their respective mandatory and optional subtasks. We assume that N independent tasks compete with P for m processors in the system. We also assume a homogeneous set of processors executing at the same speed. Any available processor serves the next task in the ready queue for Δt time units. The task being served leaves the system if its execution is completed before the Δt time units. If the task is unfinished by the end of Δt time units, then it is put back in the queue. Our goal is to schedule P within D in the

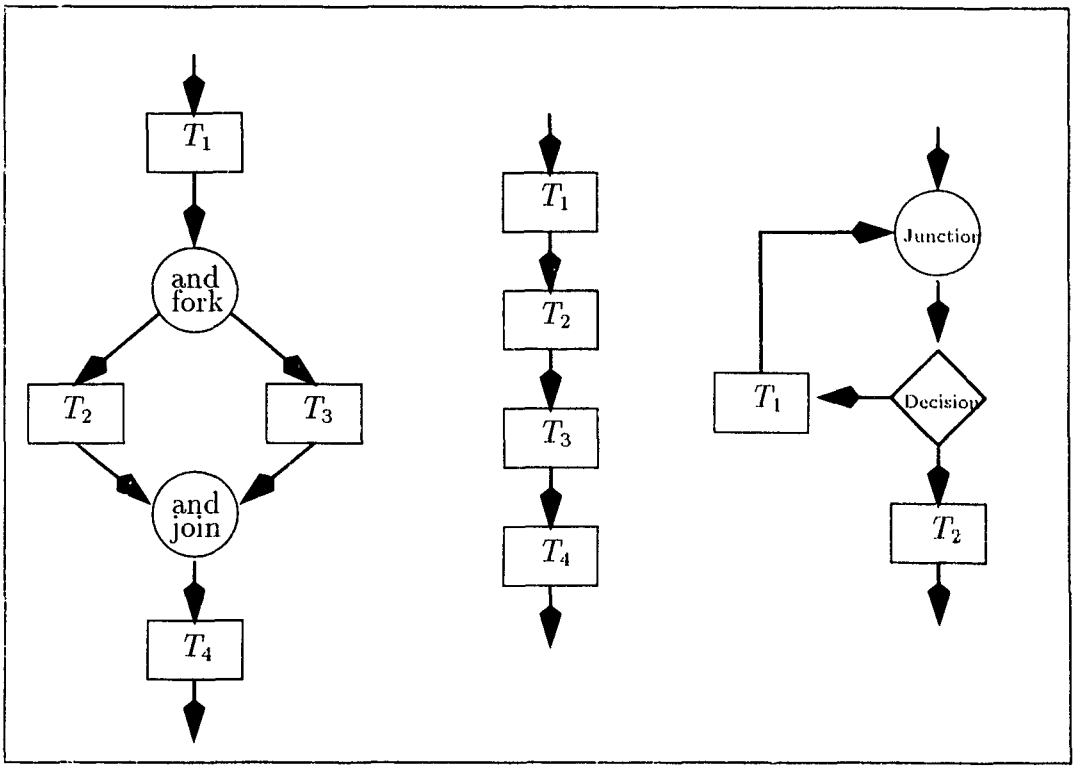


Figure 7.1: Control subgraphs of a parallel program

presence of N external tasks. We consider that N external tasks are ready for execution at the same time as P is ready.

7.3 Schedulability Analysis

Let us consider the schedulability of a single task T_i with execution time t_i. The time taken to execute a task T_i scheduled in the presence of N external tasks is given by $\frac{t_i}{min(1,\frac{m}{N})}$. It can be observed that when the number of processors exceed the number of tasks in the system, the time taken is t_i since there exists at least one processor that is free to execute T_i. On the contrary, if the number of tasks exceed the number of processors in the

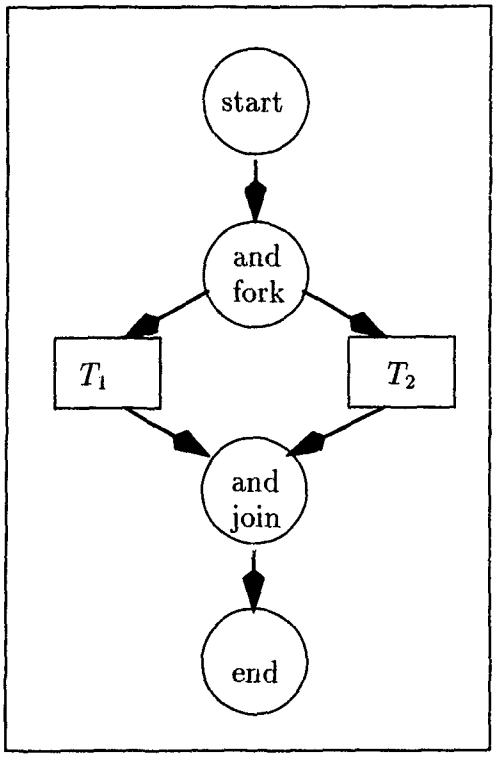


Figure 7.2: Example of a parallel program

system, then the time taken by the system to execute T_i is given by $\frac{t_i N}{m}$. This situation occurs when T_i gets queued for execution at some point. We notice that as $\Delta t \rightarrow 0$, the time taken by the system tends to $\frac{t_i}{min(1, \frac{m}{N})}$.

Now we consider the schedulability of P in presence of N external tasks. Consider a simple task graph of a parallel program, given in Figure 7.2. Let t_1 and t_2 represent the execution time for the tasks T_1 and T_2 respectively. Let t_s, t_{af}, and t_{aj} be the execution times for the control nodes *start, and-fork* and *and-join* respectively. Initially one process is needed for the *start* and *and-fork* control nodes. Fork operation spawns a child process to execute T_2 concurrently. Assuming $t_2 \geq t_1$, two tasks are present for $t_2 - t_1$ time units. Finally, the parent process continues to execute *and-join* and *end*

nodes for the completion of the program. Thus the process time requirement sequence for the example is $(t_s + t_{af}, t_1, t_2 - t_1 + t_{aj})$, assuming that the execution time for the *end* control node is zero. Since the precedence relations between various tasks of a program are known *apriori*, PCS and PTRS can be calculated pre-run-time using the algorithm given in Figure 6.4.

As stated in earlier chapters, a task implemented using the imprecise computation technique typically has two subtasks namely: mandatory and optional. We define an *accurate parallel program* to be one that contains both mandatory and optional subtasks of all the fault-tolerant tasks. Similarly, an *approximate parallel program* is one where all the fault-tolerant tasks contain only the mandatory subtasks. If an accurate parallel program is executed, both mandatory and optional subtasks of all the tasks in the program are scheduled. Figures 7.3(a) and 7.3(b) show the task control flow graph for an approximate parallel program and an accurate parallel program. For the example shown in Figure 7.3, the process change sequence for the accurate parallel program denoted as PCS_{acc}, and approximate parallel program denoted as PCS_{app}, is $(1, 2, 1)$. Let t_{M_1} and t_{M_2} be the execution times of M_1 and M_2 respectively. Let t_{O_1} and t_{O_2} be the execution times of O_1 and O_2 respectively. The process time requirement sequence for the accurate parallel program denoted as $PTRS_{acc}$ is given by $(t_s + t_{af}, t_{M_1} + t_{O_1}, t_{M_2} + t_{O_2} - t_{M_1} - t_{O_1} + t_{aj})$ and for the approximate parallel program denoted as $PTRS_{app}$ is given by $(t_s + t_{af}, t_{M_1}, t_{M_2} - t_{M_1} + t_{aj})$.

Let P_{acc} and P_{app} be the accurate and approximate parallel programs of P respectively. Let SL_{acc} and SL_{app} denote the schedule length of P_{acc} and P_{app} respectively. Let D be the deadline of the parallel program P. Let $(P_1, \cdots, P_\alpha)$ and $(R_1, \cdots, R_\beta)$ be PCS_{acc} and PCS_{app} respectively. Let $(B_1, \cdots, B_\alpha)$ and $(C_1, \cdots, C_\beta)$ be the $PTRS_{acc}$ and $PTRS_{app}$ respectively.

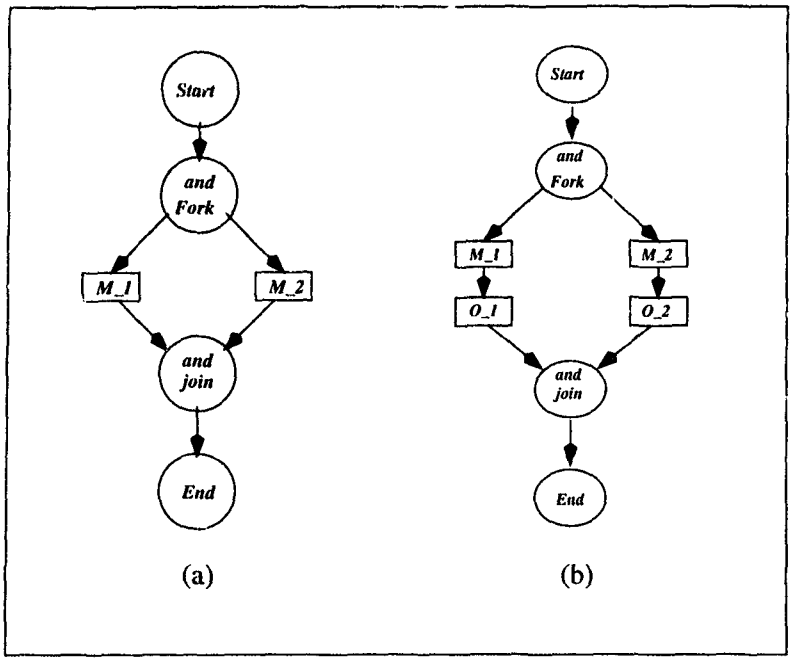


Figure 7.3: Task control flow graph for approximate and accurate parallel programs

Therefore SL_{acc} and SL_{app} are given by

$$SL_{acc} = \sum_{i=1}^{\alpha} \frac{B_i}{min(1, \frac{m}{P_i})} \tag{7.1}$$

$$SL_{app} = \sum_{i=1}^{\beta} \frac{C_i}{min(1, \frac{m}{R_i})} \tag{7.2}$$

The decision to schedule an accurate parallel program P_{acc} or an approximate parallel program P_{app} depends on the number of tasks that compete for the processors along with P during run-time. Hence depending on the timing constraints and the number of external tasks, a decision is made whether to execute P_{app} or P_{acc}. The decision is taken dynamically considering the number of other tasks in the system and the time available for the system to execute the program. We now derive the condition under which an accurate

result can be obtained.

In order to know whether to execute P_{acc} or P_{app}, the schedule length for P_{acc} and P_{app} in the presence of N external tasks needs to be calculated. The schedule lengths are given by

$$SL_{acc} = \sum_{i=1}^{\alpha} \frac{B_i}{min(1, \frac{m}{N+P_i})} \tag{7.3}$$

$$SL_{app} = \sum_{i=1}^{\beta} \frac{C_i}{min(1, \frac{m}{N+R_i})} \tag{7.4}$$

Minimum schedule length, denoted as MSL, for a parallel program occurs when no process corresponding to the program P waits in the queue for a processor at any time i.e., the denominator $min(1, \frac{m}{N+P_i})$ is 1. Let MSL_{app} be the minimum schedule length of the approximate parallel program and MSL_{acc} be the minimum schedule length of the accurate parallel program. Therefore

$$MSL_{acc} = \sum_{i=1}^{\alpha} B_i \tag{7.5}$$

$$MSL_{a,p} = \sum_{i=1}^{\beta} C_i \tag{7.6}$$

The sequential execution schedule length, denoted as SESL for a parallel program is the schedule length if the program is executed sequentially. Let $SESL_{app}$ be the sequential execution schedule length of the approximate parallel program and $SESL_{acc}$ be the sequential execution schedule length of the accurate parallel program. Therefore SESL is given by

$$SESL_{acc} = \sum_{i=1}^{\alpha} P_i B_i \tag{7.7}$$

$$SESL_{app} = \sum_{i=1}^{\beta} R_i C_i \tag{7.8}$$

If we assume that m is less than the minimum number of tasks that can be executed in parallel, then $min(1, \frac{m}{N+P_i})$ is $\frac{m}{N+P_i}$. This case arises when no process in the system gets a unity processing power. Thus the tasks need to be queued at some stage while getting processed. The schedule length for this case is given by

$$SL_{acc} = \sum_{i=1}^{\alpha} \frac{B_i(N+P_i)}{m} \tag{7.9}$$

$$SL_{app} = \sum_{i=1}^{\beta} \frac{C_i(N+R_i)}{m} \tag{7.10}$$

This can be simplified as

$$SL_{acc} = \frac{N * MSL_{acc} + SESL_{acc}}{m} \tag{7.11}$$

$$SL_{app} = \frac{N * MSL_{app} + SESL_{app}}{m} \tag{7.12}$$

In order to receive an accurate result, SL_{acc} should be less than the deadline. Therefore,

$$D \geq \frac{N * MSL_{acc} + SESL_{acc}}{m} \tag{7.13}$$

After rearranging Equation 7.13 we get

$$N \leq \frac{D * m - SESL_{acc}}{MSL_{acc}} \tag{7.14}$$

In order to receive an approximate result, SL_{app} should be less than the deadline. Therefore,

$$D \geq \frac{N * MSL_{app} + SESL_{app}}{m} \tag{7.15}$$

After rearranging Equation 7.15 we obtain

$$N \leq \frac{D * m - SESL_{app}}{MSL_{app}} \tag{7.16}$$

In order to execute P_{acc} and obtain an accurate result, the following bound in terms of the number of tasks in the system should be satisfied. Equation 7.14 can be written as :

$$N \leq \frac{D * m - \sum_{i=1}^{\alpha} B_i P_i}{\sum_{i=1}^{\alpha} B_i} \tag{7.17}$$

If the number of tasks in the system does not satisfy the above inequality, then an accurate result cannot be guaranteed for P. To obtain an approximate result, P_{app} is executed if the bound in terms of the number of tasks satisfies the following inequalities.

$$\frac{D * m - SESL_{acc}}{MSL_{acc}} \leq N \leq \frac{D * m - SESL_{app}}{MSL_{app}} \tag{7.18}$$

$$\frac{D * m - \sum_{i=1}^{\alpha} B_i P_i}{\sum_{i=1}^{\alpha} B_i} \leq N \leq \frac{D * m - \sum_{i=1}^{\beta} C_i F_i}{\sum_{i=1}^{\beta} C_i} \tag{7.19}$$

If the number of tasks exceeds the following inequality, the deadline of the program would be missed.

$$N \geq \frac{D * m - \sum_{i=1}^{\beta} C_i R_i}{\sum_{i=1}^{\beta} C_i} \tag{7.20}$$

7.4 Performance

Consider the task control flow graph of P_{acc} and P_{app} provided in Figure 7.4. The execution times for the control nodes are assumed to be one time unit each. To achieve fault-tolerance, T_1 has two subtasks: mandatory M_1 and optional O_1 with computation times 10 and 5 time units respectively.

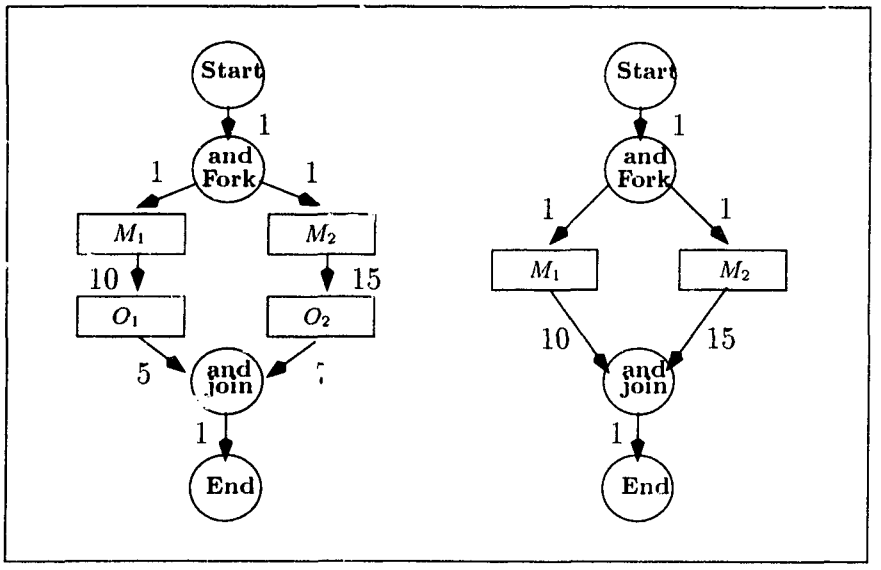


Figure 7.4: Example of control graphs of accurate and approximate parallel programs

Similarly for T_2, M_2 and O_2 need computation times of 15 time units and 7 time units respectively.

PCS_{acc} and PCS_{app} equal $(1,2,1)$. $PTRS_{acc}$ is $(t_s + t_{af}, t_{M_1} + t_{O_1}, t_{M_2} + t_{O_2} - t_{M_1} - t_{O_1} + t_{aj})$ and $PTRS_{app}$ is $(t_s + t_{af}, t_{M_1}, t_{M_2} - t_{M_1} + t_{aj})$. Therefore $PTRS_{acc}$ and $PTRS_{app}$ are $(2,15,8)$ and $(2,10,6)$ respectively. MSL_{acc}, MSL_{app}, $SESL_{acc}$ and $SESL_{app}$ are 25, 18, 40 and 28 time units respectively. By assuming $m \leq N + min(P_1, ...P_\alpha)$, we get the following conditions. If $N \leq \frac{D*m-40}{25}$, then P_{acc} can be executed, otherwise if $\frac{D*m-40}{25} < N \leq \frac{D*m-28}{18}$ is satisfied, P_{app} can be executed. The program will skip the deadline if $N > \frac{D*m-28}{18}$.

We define the *efficiency*, E as the ratio between the schedule lengths of approximate parallel program and accurate parallel program.

Lemma 6 $\lim_{N \to \infty} E = \frac{MSL_{acc}}{MSL_{app}}$

Proof. $\lim_{N\to\infty} E = \lim_{N\to\infty} \frac{N * MSL_{acc} + SESL_{acc}}{N * MSL_{app} + SESL_{app}}$

Since, $\frac{SESL_{acc}}{N} \to 0$ and $\frac{SESL_{app}}{N} \to 0$ as $N \to \infty$, $E \to \frac{MSL_{acc}}{MSL_{app}}$ as $N \to \infty$

□

7.5 Summary

In this chapter we considered a parallel program P and represented P as a task control graph. We then modelled P by breaking the program into tasks and derived the process control sequence and process time requirement sequence. We introduced imprecise computation technique for P by representing it as an accurate parallel program P_{acc} and an approximate parallel program P_{app}. We considered the problem of scheduling P during run-time along with other external tasks that arrive during run-time. We derived schedulability conditions for P_{acc} and P_{app} within a deadline D in the presence of N external tasks.

We have until now considered the problem of scheduling tasks with fault-tolerance in a multiprocessor system. Task level fault-tolerance is an effective technique to obtain an acceptable result within a deadline by replicating or partitioning a task. In the next chapter we consider node level fault-tolerance where a single processor system or a multiprocessor system is replicated. We analyze the average response time of the system in the presence of node failures.

Chapter 8

Response Time Analysis

In the earlier chapters, we concentrated on scheduling a set of tasks within a deadline in a multiprocessing system. We first considered pre-run-time scheduling of a set of tasks whose characteristics were known before run-time. We then considered the problem of scheduling a set of external tasks during run-time when a pre-run-time schedule is already present. We also considered the situation where the task set may not meet the deadline and incorporated imprecise computation technique for achieving fault-tolerance at task level. Hence, if a task cannot meet the deadline, then we scheduled a portion of that task and obtained an acceptable approximate result.

So far in our analysis we have not considered the situation when a site fails. To complete our analysis of fault-tolerant real-time systems, we now address the problem of hardware fault-tolerance where the system is replicated. A real-time system can be conceptually modelled as a black box called "site", as illustrated in Figure 8.1. In this chapter, we consider site level fault-tolerance where if a site fails then a backup site takes over to continue the processing.

Redundancy has generally been used for achieving fault-tolerance in computer systems [82]. Redundancy is simply the addition of information, resources, or time beyond what is needed for normal system operation. Triple

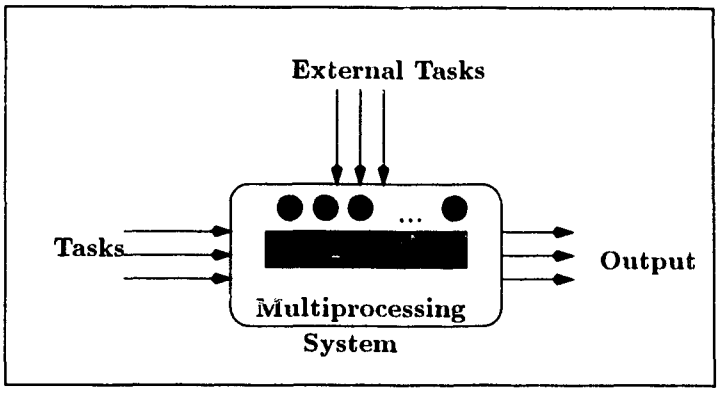


Figure 8.1: Model of a site

modular redundancy, N-modular redundancy, and standby sparing are some of the means to achieve fault-tolerance in computer systems [83]. A well known approach for supporting fault-tolerance against site failures in a system is the primary site approach [16].

In the primary site approach the service to be made fault-tolerant is replicated on many sites. The site which is executing the requests is denoted as primary, and the others which are on hot standby are called backups. The primary periodically checkpoints its state information on the backups. If the primary fails, a backup takes over as the primary. The failed primary joins the queue for the repair server which repairs the failed sites. The primary site approach has been used for supporting fault-tolerant data objects [84,85]. Figure 8.2 illustrates a multi computer system with primary site approach.

An analytic model was developed by Huang and Jalote [17] to study the issues related to a system employing the primary site approach for fault-tolerance. The effect of the degree of replication and the periodicity of the checkpoints on the reliability and availability of the system were studied. Using the analytic model, the average response time of a system employing the primary site approach was determined under the condition in which the

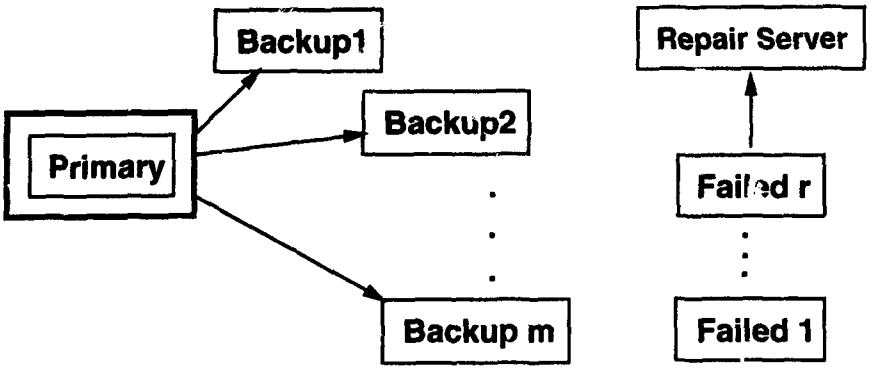


Figure 8.2: A multi computer system with primary site approach

repair server services the failed sites in a FCFS (First Come First Served) service discipline. The optimal checkpoint interval and the optimal degree of replication that minimizes the average response time of the system were determined [18].

The objective of this chapter is to develop an analytic model to determine the average response time of a system employing the primary site approach in which the repair server services the failed sites in LCFS (Last Come First Served) service discipline. We propose that a LCFS service discipline is better suited for servicing the failed sites. We show that the average response time of the system when the repair server services using LCFS discipline is lower than the average response time of the system when the repair server services using FCFS discipline.

8.1 System Model

8.1.1 List of Terms

K	Total number of computing sites
μ	Service rate of the sites
f	Failure rate of the sites
g	Checkpointing rate of the primary on backups
h	Cost of checkpointing on a single site
u	Recovery rate of the sites
V_N	Sojourn time (time the system is in normal state)
ω_N	Average response time for requests during V_N
a	Availability of the system during normal state
λ	Transaction arrival rate during normal state
$E(V_N)$	Expected value of V_N
F_N	Fraction of requests arriving in $E(V_N)$
V_I	Sojourn time of the system in the idle state
ω_I	Average response time for requests during V_I
$E(V_I)$	Expected value of V_I
F_I	Fraction of requests arriving in $E(V_I)$
V_R	Sojourn time of the system in the recovery state
ω_R	Average response time for requests during V_R
$E(V_R)$	Expected value of V_R
F_R	Fraction of requests arriving in $E(V_R)$
λ'	Transaction arrival rate during Idle or Recovery state
ω	Average response time of the system
c	Repair rate of the repair server

8.1.2 Assumptions

We consider a network of K homogeneous computing sites on which the service or data is replicated. The computing sites use the primary site approach for their services. A repair queue is maintained for the service of failed primaries. The service time and the lifetime of the sites are assumed to be exponentially distributed. We consider the inputs to the site as "transactions" and assume that the transaction arrival process follows Poisson distribution. The sites are linearly ordered and, at the start, the first site is

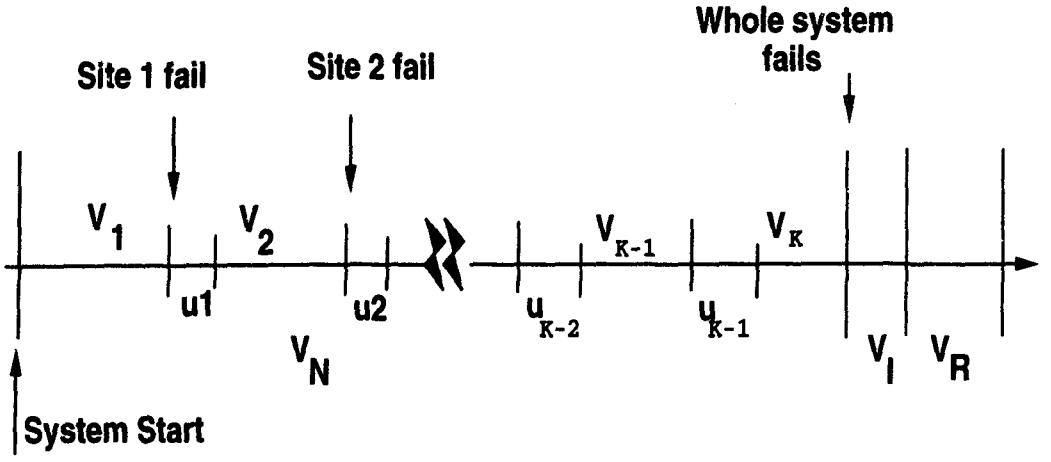


Figure 8.3: Timing diagram of the system

chosen as the primary site; all the other sites are specified as the backups. If the cost of single checkpointing is $1/x$, then the cost of checkpointing on a single site is equal to $1/x(K-1)$. The network communication system uses a point-to-point communication protocol.

The user is assumed to know whenever a primary site fails. All requests after the last checkpoint are sent to the new primary. As the latest results are lost due to failure of the previous primary, the request since the last checkpoint is redone by the new primary site. The average number of transactions needed to be redone by the new primary site when a failure occurs is given by $\lambda/(2g)$. The mean recovery time $(1/u)$ is equal to $\lambda/(2\mu g)$. The requests that arrive during the recovery period of the system are queued and are serviced by the new primary (the first site to recover). It is assumed that the mean lifetime is much larger than the repair cost and checkpointing overhead. Figure 8.3 illustrates how a system might behave from the start until it fails.

A total failure occurs when a primary fails and no backup is operational.

The sites that are failing are assumed to join a queue to be serviced by the repair server. We assume that sites satisfy the *fail-stop* assumption [86]. The backups and the repair server are notified of the failure of a primary. The repair server keeps log of order of site failures. The state information is obtained from stable storage of the failed site. Two types of services are considered for repairing failed sites, namely, FCFS and LCFS. In FCFS [18], the site that fails last would recover last. In LCFS, the site that fails last would recover first. Since only the final failed site has the correct state of the data, service can be restarted only after the final failed site recovers. In other words, only the final failed site can assume the role of the primary; other sites, on recovery, can only be considered as backups. Therefore, the system is idle until the final failed site recovers. The main difference in FCFS and LCFS becomes apparent when considering situations where all sites have failed or delayed repair service is used.

Since the system waits for the final site to recover, using a FCFS service results in considerable idle time. The paper uses a LCFS discipline so that the primary that fails last gets recovered first. This leads to the minimal idle time. All the requests that arrive during the idle period are queued outside the system until the system is repaired. After repair, the primary site processes all queued requests before providing normal services. Since the final failed site contains the latest system status, this site would be the primary after recovery. The other sites that are recovered become backups.

8.2 Repair Server

When a site fails, it joins the repair queue. This queue is assumed to be serviced on the average, in $1/\epsilon$ time units. Two types of repairs are considered, namely, Delayed repair and Immediate repair.

8.2.1 Delayed Repair

The repair queue is not serviced until the final site fails. Once the final site joins the repair queue, all the sites in the queue are recovered at a stretch. This procedure is used when the system cannot be repaired as soon as they join the repair queue due to reasons such as cost and distance. Since the last site to fail contains the latest computation, the system cannot start functioning until the last site is recovered. If the repair server uses the FCFS service discipline, then the last site to fail will be the last site to recover. This leads to a situation where the sites that have been repaired earlier than the last failed site are idle. If the repair server uses the LCFS service discipline, then the last site to fail will be the first one to recover.

8.2.2 Immediate Repair

The repair server monitors continuously, and services the repair queue if it is nonempty. Thus the sites dynamically leave the system when they fail and rejoin the system when they are repaired. Since the failed sites are repaired immediately, the sites join the system as soon as they are serviced. When a total failure occurs, the situation is quite similar to the delayed repair case. If the repair server uses the FCFS service discipline during a total failure, then the sites that get recovered earlier than the last site to fail remain idle. If the repair server uses the LCFS service discipline, then the last site to fail will be the first one to recover. Thus the system is operational while the repair server is serving the backups.

8.3 System State

The system is considered to be in one of the three states : Normal state, Idle state and Recovery state. For the system to be in the normal state at least one primary should be functional. When no primaries are functional,

the system is termed to have totally failed. The system is said to be in an idle state until a primary becomes functional after total failure. After being in idle state, the system recovers by serving all the queued requests that had come during the idle period. During this period of time the system is termed to be in the recovery state.

8.3.1 Normal state

The system is in normal state during the time period between the start of the system and its total failure. There is at least one primary that is functional in a normal state. This primary services the requests to the system and checkpoints the status on the backups. If there is a backup, then it assumes the role of the primary on the failure of the presently functional primary, else a total failure results. A failed primary joins the repair queue of the repair server to be serviced either immediately or after total failure. Let V_N be the random variable representing the sojourn time of the system in the normal state and ω_N be the expected response time during V_N.

8.3.2 Idle state

The transition from normal state to idle state occurs after total failure. During the idle state, the system does not have any site functional to service the requests. The arrival rate of the requests decreases during the idle state since the users refrain from submitting transactions to a failed system. Therefore we assume the arrival rate during the idle state to be λ' which is less than λ. The requests are queued outside the system to facilitate the execution once the system is out of the idle state. The repair server services the failed sites during the idle state. The system is in idle state until, at least one primary becomes functional after being repaired by the repair server. Let V_I be the random variable representing the sojourn time of the system in the idle state and ω_I be the expected response time during V_I.

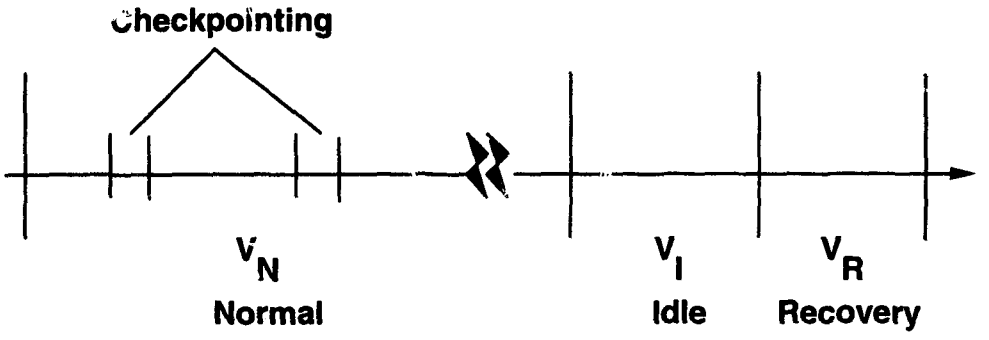


Figure 8.4: The timing diagram for failure-repair period

8.3.3 Recovery state

The system is in recovery state once the repair server services the last site to fail. The system in this state is recovering from a total failure by processing all the queued requests, which include the requests that arrive during the idle state and the unserved requests that had arrived during the normal state. We assume the arrival rate of requests during the recovery state to be λ'. Let V_R be the random variable denoting the sojourn time of the system in the recovery state and ω_R be the expected response time during V_R.

8.4 Average Response Time Analysis

A **failure-repair** period is the interval between two adjacent total failures. The system is cyclic with this period. The timing diagram for a failure-repair period is shown in Figure 8.4. Since ω_N, ω_I, and ω_R are the average response time of the system during Normal, Idle and Recovery states respectively, the average response time of the system is given by

$$\omega = F_N\omega_N + F_I\omega_I + F_R\omega_R \tag{8.1}$$

where F_N,F_I, and F_R are the fraction of requests that arrive during V_N, V_I, and V_R of a failure-repair period, respectively. F_N, F_I, and F_R are calculated as follows :

$$F_N = \frac{\lambda E(V_N)}{\lambda E(V_N) + \lambda'[E(V_I) + E(V_R)]} \tag{8.2}$$

$$F_I = \frac{\lambda' E(V_I)}{\lambda E(V_N) + \lambda'[E(V_I) + E(V_R)]} \tag{8.3}$$

$$F_R = \frac{\lambda' E(V_R)}{\lambda E(V_N) + \lambda'[E(V_I) + E(V_R)]} \tag{8.4}$$

where $\lambda E(V_N), \lambda' E(V_I), \lambda' E(V_R)$ are the average number of requests arriving during various states of a failure-repair period.

The average response time of the system during the normal state can be described by the Checkpoint-Rollback-Recovery model [87]. Therefore, ω_N is approximated by

$$\omega_N \approx \frac{1 + a^2\mu(\frac{f}{u^2} + \frac{c}{x^2})}{a\mu - \lambda} \tag{8.5}$$

If $u >> f$ and $x >> g$ then

$$\omega_N \approx \frac{1}{a\mu - \lambda} \tag{8.6}$$

where the availability of the system, a, is given by [87]

$$a = \frac{1}{1 + \frac{g}{x} + \frac{f}{u}}$$

$$a = \frac{1}{1 + g(K-1)h + \frac{f\lambda}{2\mu g}} \tag{8.7}$$

The average response time of the system for a request that arrives during the idle state is affected by the time in which the system is in the idle state and

the time which the system takes to service the queued requests. The average waiting time for transition to occur from idle state is given by $E(V_I)/2$. Once the system recovers, the request queued during the idle state will wait $\lambda' E(V_I)/(2\mu)$ time units to be served after the primary site starts processing the queued requests. Therefore, the expected response time as derived in [18] is given by:

$$\omega_I = \frac{E(V_I)}{2} + \frac{\lambda' E(V_I)}{2\mu} \tag{8.8}$$

It is assumed that the requests arriving in recovery state will be executed during the next normal state. Therefore, the expected waiting time for a request arriving during the recovery state will be $E(V_R)/2$. Since the arrival rate during the recovery state is λ' the expected response time again as derived in [18] for a request arriving during recovery state is given by:

$$\omega_R = \frac{E(V_R)}{2} + \frac{\lambda' E(V_R)}{2\mu} \tag{8.9}$$

8.5 Average Response Time using LCFS

In order to compute $E(V_N), E(V_I)$, and $E(V_R)$ both the repair cases need to be considered.

8.5.1 Delayed Repair Case

In the delayed repair case the failed sites are repaired only when no primary site is available i.e., when total failure occurs. The repair server starts repairing only when all K sites have failed. The site that failed last contains the latest information regarding the job service. The system cannot be functional until this site is restored. Since Huang and Tslote assume FCFS service in [18] the system is functional only when the last site is repaired. In order

to repair the last site all the other sites in the repair queue needed to be repaired. But we assume a LCFS service, where we need only to repair the last site for the system to be functional. After the last site to fail has become functional again other sites could be repaired and added as backups. Since the repair rate for the repair server is $1/\epsilon$, the total time for the system to be functional is given by

$$E(V_I) = \frac{1}{\epsilon} \tag{8.10}$$

Since the requests that arrived during the idle state (i.e., $\lambda' E(V_I)$) are serviced by the system during recovery state, the expected value of V_R is given by

$$E(V_R) = \frac{\lambda'}{\mu\epsilon} \tag{8.11}$$

V_N, the expected time to a total failure can be modeled as shown in Figure 8.5. The state of the system is the total number of working sites. Thus, the expected time the system spends in normal state is equal to the time taken by the system to transit from state K to state 0. The expected sojourn time of the system at the state i is $1/(i.f)$, where f is the failure rate of a site. Therefore, the expected value of V_N as derived in [18]

$$E(V_N) = \frac{1}{f}\sum_{i=1}^{K}\frac{1}{i} \tag{8.12}$$

8.5.1.1 Response Time of the System

Since $E(V_N), E(V_I)$, and $E(V_R)$ are known, the average response time of the system is given by

$$\omega = F_N\omega_N + F_I\omega_I + F_R\omega_R$$

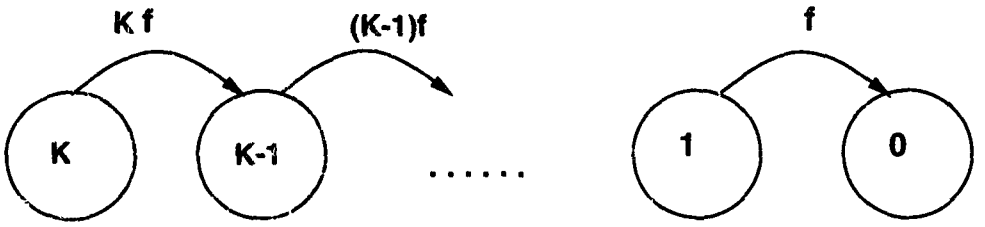


Figure 8.5: State diagram of the delayed-repair system

where F_N, F_I, and F_R are given as:

$$F_N = \frac{\frac{\lambda}{f}(\sum_{i=1}^{K}\frac{1}{i})}{\frac{\lambda}{f}(\sum_{i=1}^{K}\frac{1}{i}) + \frac{\lambda'}{\epsilon} + \frac{\lambda'^2}{\mu\epsilon}} \tag{8.13}$$

$$F_I = \frac{\frac{\lambda'}{\epsilon}}{\frac{\lambda}{f}(\sum_{i=1}^{K}\frac{1}{i}) + \frac{\lambda'}{\epsilon} + \frac{\lambda'^2}{\mu\epsilon}} \tag{8.14}$$

$$F_R = \frac{\frac{\lambda'^2}{\mu\epsilon}}{\frac{\lambda}{f}(\sum_{i=1}^{K}\frac{1}{i}) + \frac{\lambda'}{\epsilon} + \frac{\lambda'^2}{\mu\epsilon}} \tag{8.15}$$

Therefore the average response time of the system is given by

$$w = \frac{\frac{\lambda}{f}(\sum_{i=1}^{K}\frac{1}{i}).\omega_N + \frac{\lambda'}{\epsilon}.\omega_I + \frac{\lambda'^2}{\mu\epsilon}.\omega_R}{\frac{\lambda}{f}(\sum_{i=1}^{K}\frac{1}{i}) + \frac{\lambda'}{\epsilon} + \frac{\lambda'^2}{\mu\epsilon}} \tag{8.16}$$

where

$$\omega_N = \frac{1}{a\mu - \lambda} \tag{8.17}$$

$$\omega_I = \frac{1}{2\epsilon} + \frac{\lambda'}{2\mu\epsilon} \tag{8.18}$$

$$\omega_R = \frac{\lambda'}{2\mu\epsilon} + \frac{\lambda'^2}{2\mu^2\epsilon} \tag{8.19}$$

8.5.2 Immediate Repair Case

The repair server monitors continuously, and services the repair queue if it is non empty. Huang and Jalote [18] assumed FCFS where the last site to fail has the latest information and the system cannot be functional until all the sites are repaired. In our proposed LCFS discipline, the last site to fail on its arrival will have to wait on an average $1/2\epsilon$ for the failed site which is currently being repaired. Since preemption is not assumed, the last site to fail has to wait for the repair server to complete the service. In addition $1/\epsilon$ time units are spent on repairing the last site to fail which would be serviced next. Therefore, the expected value of V_I is given by

$$E(V_I) = \frac{3}{2\epsilon} \tag{8.20}$$

Since the expected value of V_R is $(\lambda'/\mu)E(V_I)$, we substitute for $E(V_I)$. Therefore,

$$E(V_R) = \frac{3\lambda'}{2\mu\epsilon} \tag{8.21}$$

Figure 8.6 shows the system state using the machine-repairman model [88]. The state of the system is the number of sites that are working. When a failure occurs (failure rate = f) the system moves to the neighboring state on its right (one less working site). This failed site is queued for repair. When a site is repaired (repair rate = ϵ), the system transits to the neighboring state on its left (one more working site). The system basically moves from state K to state 0 in order to fail completely. But unlike the delayed repair case, the system in the immediate repair case could visit an intermediate state $i, (0 < i \leq K)$ more than once.

Since the calculation of $E(V_N)$ is independent of the service discipline that is used by the repair server to service the failed sites, we refer the readers to

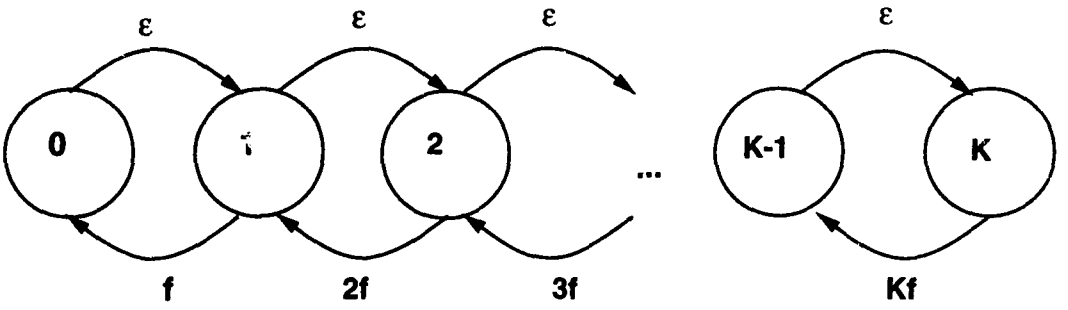


Figure 8.6: State diagram of the immediate-repair system

the expression and the recurrence equations derived in [18].

The expected value of V_N is given by

$$E(V_N) = \frac{m_N(K)}{Kf} + \sum_{i=1}^{K-1} \frac{m_N(i)}{i.f + \epsilon} \tag{8.22}$$

where $m_N(i)$ is the expected number of visits to state i before visiting state 0. $m_N(i)$ is calculated using a set of recurrence equations.

8.5.2.1 Response Time of the System

The average response time of the system is given by:

$$w = F_N\omega_N + F_I\omega_I + F_R\omega_R$$

where,

$$F_N = \frac{\lambda E(V_N)}{\lambda E(V_N) + \frac{3\lambda'}{2\epsilon} + \frac{3\lambda'^2}{2\mu\epsilon}} \tag{8.23}$$

$$F_I = \frac{\frac{3\lambda'}{2\epsilon}}{\lambda E(V_N) + \frac{3\lambda'}{2\epsilon} + \frac{3\lambda'^2}{2\mu\epsilon}} \tag{8.24}$$

$$F_R = \frac{\frac{3\lambda'^2}{2\mu\epsilon}}{\lambda E(V_N) + \frac{3\lambda'}{2\epsilon} + \frac{3\lambda'^2}{2\mu\epsilon}} \tag{8.25}$$

Therefore,

$$w = \frac{\lambda E(V_N).\omega_N + \frac{3\lambda'}{2\epsilon}.\omega_I + \frac{3\lambda'^2}{2\mu\epsilon}.\omega_R}{\lambda E(V_N) + \frac{3\lambda'}{2\epsilon} + \frac{3\lambda'^2}{2\mu\epsilon}} \tag{8.26}$$

where,

$$\omega_N = \frac{1}{a\mu - \lambda} \tag{8.27}$$

$$\omega_I = \frac{3}{4\epsilon} + \frac{3\lambda'}{4\mu\epsilon} \tag{8.28}$$

$$\omega_R = \frac{3\lambda'}{4\mu\epsilon} + \frac{3\lambda'^2}{4\mu^2\epsilon} \tag{8.29}$$

8.5.3 Example

For comparison, we consider the same example as used in [18]. A primary site system is assumed with three sites where the mean life of each site is 700000 seconds (approximately 8 days). The arrival rate during the normal state is 6 requests per second ($\lambda = 6$). The request arrival rate during other states is 0.1 ($\lambda' = 0.1$). The service rate of a site is 8 ($\mu = 8$). The cost of checkpointing to a backup is 0.5 seconds ($h = 0.5$) and the checkpointing interval is 1000 seconds (approximately 16 minutes). Let the average repair time be 4000 seconds.

8.5.3.1 Delayed Repair Case using FCFS

Failed sites are repaired only after all sites have failed. The expected times are, $E(V_N) = 1283333s$, $E(V_I) = 12000s$, (approximately 3.3 hours); $E(V_R) = 150.0s$ (approximately 2.5 min). The expected response times of ω_N, ω_I, and ω_R are 0.50309, 6075, and 75.9 seconds, respectively, and the probabilities

of F_N, F_I, and F_R are 0.99984, $1.5X10^{-4}$, and $1.95X10^{-6}$, respectively. The average response time is 1.44975 seconds.

8.5.3.2 Delayed Repair Case using LCFS

Failed sites are repaired only after all sites have failed. The expected times are, $E(V_N) = 1283333$ seconds, $E(V_I) = 4000$ seconds, (approximately 1.3 hours); $E(V_R) = 50.0$ seconds. The expected response times of ω_N, ω_I, and ω_R are 0.50309, 2025, and 25.35 seconds, respectively, and the probabilities of F_N, F_I, and F_R are 0.99984, $5.192X10^{-5}$, and $6.49X10^{-7}$, respectively. The average response time is 0.6082692 seconds as compared to 1.44975 seconds for FCFS.

8.5.3.3 Immediate Repair Case using FCFS

Failed sites are repaired as and when they arrive to queue. The expected times are, $E(V_N) = 3655866666s$, $E(V_I) = 12000s$, (approximately 3.3 hours); $E(V_R) = 150.0s$ (approximately 2.5 min). The expected response times of ω_N, ω_I, and ω_R are 0.50309, 6075, and 75.9 seconds, respectively, and the probabilities of F_N, F_I, and F_R are 1, $5.47X10^{-8}$, and $7X10^{-10}$, respectively. The average response time is 0.50341 seconds.

8.5.3.4 Immediate Repair Case using LCFS

Failed sites are repaired as and when they arrive to queue. The expected times are, $E(V_N) = 3655866666$ seconds, $E(V_I) = 6000$ seconds, (approximately 1.85 hours); $E(V_R) = 75.0$ seconds (approximately 1.25 minutes). The expected response times of ω_N, ω_I, and ω_R are 0.50309, 3037.5, and 37.95 seconds, respectively, and the probabilities of F_N, F_I, and F_R are 1, $2.7X10^{-8}$, and $3.5X10^{-10}$, respectively. The average response time is 0.503172 seconds as compared to 0.50341 seconds for FCFS.

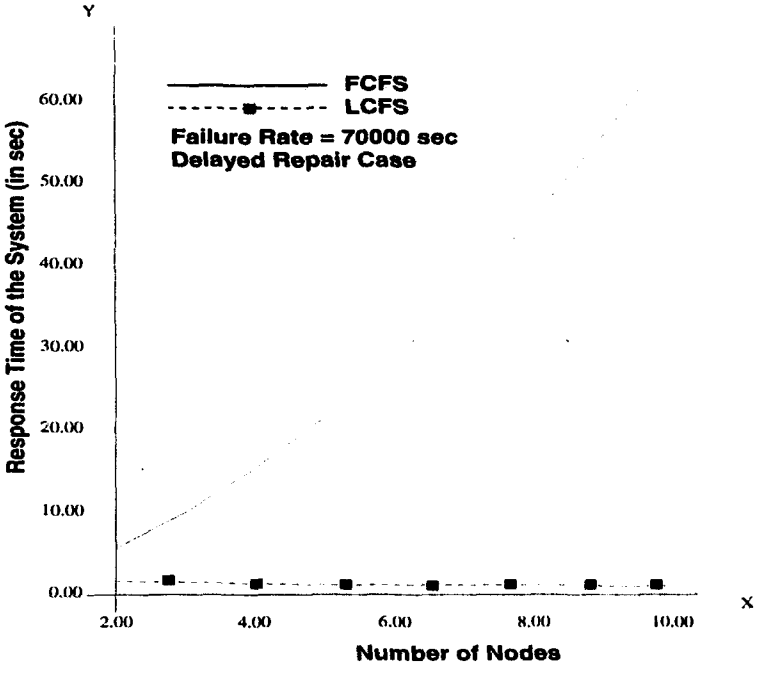


Figure 8.7: Delayed repair case

8.6 Performance Analysis

Let w^K_{FCFS} and w^K_{LCFS} be the average response time of the system that uses FCFS and LCFS discipline to repair the failed sites respectively.

Theorem 11 *$w^K_{FCFS} > w^K_{LCFS}$ for delayed repair case as $K \to \infty$.*

Proof. Since a decreases as K increases, we can deduce that ω_N increases as K increases. Denoting ω_N as $f(K)$ for convenience, the average response time can be written as

$$w^K_{FCFS} = \frac{A\sum_{i=1}^{K}\frac{1}{i}f(K) + BK^2 + CK^2}{A\sum_{i=1}^{K}\frac{1}{i} + DK + EK} \tag{8.30}$$

where $A = \frac{\lambda}{f}, B = \frac{\lambda'}{2\epsilon^2} + \frac{\lambda'^2}{2\mu\epsilon^2}, C = \frac{\lambda^2\lambda'}{2\mu^2\epsilon^2} + \frac{\lambda^2\lambda'^2}{2\mu^3\epsilon^2}, D = \frac{\lambda'}{\epsilon}$, and $E = \frac{\lambda'^2}{\mu\epsilon}$.

$$w^K_{FCFS} = \frac{A + \frac{Bf(K)}{\sum_{i=1}^{K}\frac{1}{i}} + \frac{Cf(K)}{\sum_{i=1}^{K}\frac{1}{i}}}{\frac{A}{K} + \frac{D}{\sum_{i=1}^{K}\frac{1}{i}} + \frac{E}{\sum_{i=1}^{K}\frac{1}{i}}} \tag{8.31}$$

since $K^3 > K^2 > K > logK > \sum_{i=1}^{K}\frac{1}{i} > 1$ is true for $K > 1$ and $\frac{f(K)}{\sum_{i=1}^{K}\frac{1}{i}}$ diverges for increasing K, in equation 8.31 the denominator decreases and the numerator increases as K increases. Therefore, w^K_{FCFS} diverges as K increases. Similarly, w^K_{LCFS} is given by

$$w^K_{LCFS} = \frac{A + \frac{B}{K\sum_{i=1}^{K}\frac{1}{i}} + \frac{C}{K\sum_{i=1}^{K}\frac{1}{i}}}{\frac{A}{K} + \frac{D}{K\sum_{i=1}^{K}\frac{1}{i}} + \frac{E}{K\sum_{i=1}^{K}\frac{1}{i}}} \tag{8.32}$$

The response time given in equation 8.32, converges as K increases. Therefore $w^K_{FCFS} > w^K_{LCFS}$ for delayed repair case as $K \to \infty$.

□

A similar analysis can be done for immediate repair case.

Figure 8.7 illustrates the response time of the system with delayed repair for increasing degree of replication (K). Figure 8.8 illustrates the response time of the system with immediate repair for increasing K. The failure time of the system is assumed to be 70000 seconds. The other data are similar to the one provided in the example earlier. When a FCFS service is considered for delayed repair, as the degree of replication (K) increases, the time spent by the system in the idle state ($E(V_I)$), and the recovery state ($E(V_R)$) increases. But when a LCFS service is considered for delayed repair, the time spent by the system in the idle state and the recovery state decreases as K increases. Thus the system is functional for a longer time when LCFS service is used and therefore the availability of the system is greater in the proposed scheme. It can also be noticed that for the same assumption on

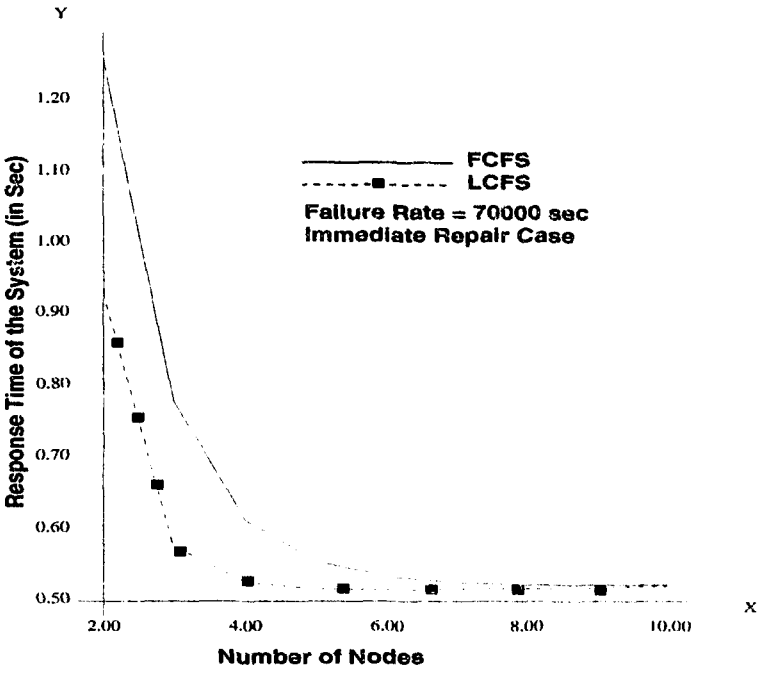


Figure 8.8: Immediate repair case

various rates and degree of replication, the response time of the system for the proposed method using LCFS discipline is less than the response time of the system using FCFS discipline.

8.7 Summary

In this chapter, we considered node level fault-tolerance using primary site approach where sites were replicated to tolerate site failures. One of the sites was designated as primary site to execute the arriving tasks while the others remained as backup sites. The primary site periodically checkpointed its result to other backup sites. When a primary site failed, a predetermined backup became the new primary and the failed site entered a repair queue. The repair queue serviced the sites using either delayed repair service or immediate repair service. Two different queuing disciplines to service the repair

queue were considered, namely LCFS and FCFS. We derived the average response time for the system when the repair server used the LCFS and FCFS queuing disciplines. We showed that the average response time of the system for LCFS queuing discipline was lower than the average response time for FCFS queuing discipline.

Chapter 9

Conclusions

The main objective of this thesis has been to explore the problems of achieving fault-tolerant real-time scheduling using imprecise computation methods in multiprocessor systems. The main contributions of this thesis are as follows:

- We introduced a *bin packing* algorithm based on McNaughton's rule for multiprocessor systems for tasks with real-time constraints. The problem of scheduling a set of imprecise computation tasks within a deadline has not been well researched. We extended our algorithm based on McNaughton's rule for tasks formulated using imprecise computation technique.

- We proposed a *level* algorithm for scheduling a set of tasks at pre-run-time and proved its optimality. Also the algorithm was extended to include tasks formulated using imprecise computation technique.

- We introduced a new approach, using *virtual paths* to describe the run-time status of a multiprocessor systems. We also derived the necessary and sufficient conditions for scheduling a set of tasks that arrive during run-time within a given deadline.

- Although algorithms using *bin-packing* approach had been introduced to schedule tasks at run-time, *level* algorithms for the same problem had not been presented before. We considered a *level* algorithm for optimally scheduling tasks with identical and non identical ready times that arrive during run-time.

- The problem of scheduling a set of imprecise computation tasks during run-time had not been researched before. In this dissertation, we proposed a *level* algorithm to schedule a set of imprecise computation tasks with identical and non-identical ready times. We derived the necessary and sufficient conditions for scheduling the mandatory subtasks of all the tasks. We then provided an algorithm that maximized the total number of time slots in which optionals could be scheduled, after scheduling all the mandatory subtasks.

- We proposed an adaptive run-time scheduling algorithm for scheduling optimally a set of tasks on top of pre-run-time schedule, by removing minimum number of optional parts of the tasks that are present in the pre-run-time schedule. We demonstrated the applicability of this algorithm in some practical areas such as robotics.

- Imprecise computation technique was extended to include tasks with precedence constraints. A heuristic algorithm based on McNaughton's rule for deciding when to schedule tasks in accurate precedence graph and when to schedule tasks in approximate precedence graph was developed. A metric to calculate the adaptability of the run-time scheduler was also introduced.

- We introduced fault-tolerance based on imprecise computation technique for the tasks in parallel program and analyzed it using a round-robin scheduler to schedule the tasks in parallel program along with the

external tasks. We provided the schedulability conditions under which an accurate parallel program should be scheduled and the conditions under which an approximate parallel should be scheduled in order to meet the deadline.

- Average response time of the system with node level fault-tolerance using primary site approach was derived assuming LCFS queuing discipline for the repair server queue. We compared the average response time obtained using LCFS repair queue with that of FCFS repair queue and proved that LCFS is a better queuing discipline for repair server.

We can conclude that *level* algorithms are more conducive for run-time scheduling than *bin-packing* algorithms. *Level* algorithm for imprecise computation had not been explored previously and our work opens a new field of research in the areas of fault-tolerant scheduling.

9.1 Future Work

The next logical step is to extend this work to distributed systems. Hence active research is needed to obtain approximate algorithms for situations where tasks have communication constraints and resource constraints. Tasks in distributed systems typically have these constraints and heuristic algorithms are needed to schedule tasks under such conditions.

In this thesis we assumed that the tasks have identical deadlines. Scheduling algorithms for a set of tasks with non-identical deadlines can be explored. Similarly, the thesis assumes that the processors are identical. It will be interesting to explore the fault-tolerant run-time scheduling problem for uniform processors and unrelated processors. The work can also be extended for tasks that are periodic.

Bibliography

[1] J. D. Schoeffler, "Distributed Computer Systems for Industrial Process Control," *IEEE Computer*, 17, no. 2, pp. 11–18, Feb., 1984.

[2] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak and C. B. Weinstock, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE*, 66, no. 11, pp. 1240–1255, Oct., 1978.

[3] G. D. Carlow, "Architecture of the Space Shuttle Primary Avionics Software System," *Communications of ACM*, 27, no. 9, pp. 926–936, Sept., 1984.

[4] H. Kasahara and S. Narita, "Parallel Processing of Robot-Arm Control Computation on a MultimicroprocessorSystem," *IEEE Journal of Robotics and Automation*, RA-1, no. 2, pp. 3–6, June,1985.

[5] J. A. Stankovic and K. Ramamritham, *Tutorial: Hard Real-Time Systems.* IEEE Computer Society Press, 1988.

[6] A. M. vanTilborg, *Foundations of Real-Time Computing: Formal Specifications and Methods.* Kluwer Academic Publishers, 1991.

[7] Y. H. Lee and C. M. Krishna, *Readings in Real-Time Systems.* IEEE Computer Society Press, 1993.

[8] J. A. Stankovic and K. Ramamritham, *Advances in Real-Time Systems.* IEEE Computer Society Press, 1993.

[9] R. Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers, 1991.

[10] S. French, "Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop," in *Ellis Horwood Series in Mathematics and its Applications.* Ellis Horwood, 1982.

[11] E. L. Lawler, "Recent Results in the Theory of Machine Scheduling," in *Mathematical Programming: The State of the Art (Ed: A. Bachem et al.).* New York: Springer-Verlag, pp. 202–233, 1983.

[12] E. L. Lawler, J. K. Lenstra, A. H. G. R. Kan and D. B. Shmoys, Sequencing and Scheduling: Algorithms and Complexity," in *Handbooks in OR and MS (Editors: S.C. Graves et al.,)*, vol. 4 (Chapter 9). Elsevier Science Publishers, pp. 445–522, 1993.

[13] J. A. Stankovic, M. Spuri, M. D. Natale and G. C. Buttazzo, "Implications of Classical Scheduling Results for Real-Time Systems," *IEEE Computer*, 28, no. 6, pp. 16–25, June, 1995.

[14] A. M. vanTilborg and G. M. Koob, *Foundations of Real-Time Computing: Scheduling and Resource Management.* Kluwer Academic Publishers, 1991.

[15] T. L. Casavant and J. G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Systems," *IEEE Transactions on Software Engineering*, 14, no. 2 , pp. 141–154, Feb., 1988.

[16] P. A. Alsberg and J. D. Day, "A Principle for Resilient Sharing of Distributed Resources," in *Proceedings of the 2nd International Conference on Software Engineering.* pp. 562–570, Oct., 1976.

[17] Y. Huang and P. Jalote, "Analytic Models for the Primary Site Approach to Fault-Tolerance," *Acta Informatica*, 26, pp. 543–557, 1989.

[18] Y. Huang and P. Jalote, "Effect of Fault Tolerance on Response Time Analysis of the Primary Site Approach," *IEEE Transactions on Computers*, C-41, no. 4, pp. 420–428, Apr., 1992.

[19] R. Butler, "An analysis of the real-time application capabilities of the SIFT Computer System," NASA, Tech. Memo. TM-84482, Apr., 1982.

[20] D. Palumbo, D. Palumbo and R. Butler, "SIFT - A Preliminary Evaluation," in *Proceedings of Digital Avionics Systems Conference.* Seattle, WA: pp. 21.4.1–21.4.6, Oct., 1983.

[21] S. J. Larimer and S. L. Maher, "Reconfiguring Multi-Microprocessor Flight Control System," Air Force Wright Aeronautical Laboratories, Tech. Report AFWAL-TR-81-3070, May, 1981.

[22] A. Hopkins, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proceedings of IEEE*, 66, no. 10, pp. 1221–1239, Oct., 1978.

[23] C. J. Walter, R. M. Kieckhafer and A. M. Finn, "MAFT: A Multicomputer Architecture for Fault-Tolerance in Real-Time Control Systems," in *IEEE Proceedings of the Real-Time Systems Symposium*, vol. 6 . pp. 133–140, Dec., 1985.

[24] J. J. Horning, "A program structure for error detection and recovery," in *Lecture Notes in Computer Science 16.* New York- Heidelberg–Berlin: Springer-Verlag, pp. 171–187, 1974.

[25] B. Randell, "System Structure for Software Fault-Tolerance," in *Proceedings, International Conference on Reliable Software.* pp. 437–439, 1975.

[26] H. Hecht, "Fault-Tolerant Software for real-time applications," *ACM Computing Surveys*, 8, no. 4, pp. 391–406, Dec., 1976.

[27] R. H. Campbell, K. H. Horton and G. C. Belford, "Simulations of a Fault-Tolerant Deadline Mechanism," *9th IEEE International Conference on Fault-Tolerant Computing*, 9, pp. 95–101, 1979.

[28] L. Chen and A. Avizienis, "N-Version Programming : A Fault-Tolerance approach to reliability of Software operation," *8th annual international conference on Fault-Tolerant Computing*, 8, pp. 3–9, June, 1978.

[29] K. J. Lin, S. Natarajan, J. W. S. Liu and T. Krauskopf, "Concord : A system of imprecise computations," in *IEEE Proceedings on Compsac.* Japan: Oct., 1987.

[30] K. J. Lin, S. Natarajan and J. W. S. Liu, "Imprecise results : Utilizing partial computations in real-time systems," *IEEE Proceedings on Real-Time System Symposium*, 8, pp. 210–217, 1987.

[31] J. W. S. Liu, K. J. Lin and S. Natarajan, "Scheduling real-time, periodic jobs using imprecise results," in *IEEE Proceedings on Real-Time System Symposium*, vol. 8. San Jose: pp. 252–260, 1987.

[32] V. Millan-Lopez, W. Feng and J. W. S. Liu, "Using the Imprecise Computation Technique for Congestion Control on a Real-Time Traffic Switching Element," in *Proceedings of the International Conference on Parallel and Distributed Systems.* Taiwan: Dec., 1994.

[33] S. V. Vrbsky, J. W. S. Liu and K. P. Smith, "An Object-Oriented Query Processor that Returns Monotonically Improving Approximate Answers," University of Illinois, Urbana-Champaign, UIUCDCS-R-90-1568, 1990.

[34] S. V. Vrbsky and K-J. Lin, "Recovering Imprecise Transactions with Real-Time Constraints," in *Proceedings of the Seventh International Symposium on Reliable Distributed Systems.* Columbus, Ohio: pp. 185–193, Oct., 1988.

[35] W. Feng and J. W. S. Liu, "An Extended Imprecise Computation Model for Time-Constrained Speech Processing and Generation," in *Proceedings of the IEEE Workshop on Real-Time Applications.* New York: pp. 76–80, May,1993.

[36] T. Anderson and J. C. Knight, "A framework for software fault tolerance in real-time systems," *IEEE Transactions on Software Engineering*, 9, pp. 355–364, May 1983.

[37] C. M. Krishna and K. G. Shin, "On Scheduling Tasks with a Quick Recovery from Failure," *IEEE Transactions on Computers*, 35, no. 5, pp. 448–455, May, 1986.

[38] A. L. Liestman and R. H. Campbell, "A Fault-Tolerant Scheduling problem," *IEEE Transactions on Software Engineering*, 12, no. 11, pp. 1089–95, Nov., 1986.

[39] J. Y-T. Leung, T. W. Tam, C. S. Wong and G. H. Young, "Minimizing Mean Flow Time with Error Constraint," in *Proceedings of Real-Time Symposium.* Santa Monica, California: pp. 2–11, Dec., 1989.

[40] J. Y-T. Leung and C. S. Wong, "Minimizing the Number of Late Tasks with Error Constraint," in *Proceedings of the Eleventh International Symposium on Real-Time Systems.* Orlando, Florida, USA: pp. 32–40, Dec., 1990.

[41] J. Y. Chung, J. W. S. Liu and K. J. Lin, "Scheduling Periodic jobs that allow imprecise results ," *IEEE Transactions on Computers*, 39, pp. 1156–1173, Sept., 1990.

[42] J. W. S. Liu, K-J. Lin, W-K. Shih, A. C-S. Yu, J-Y. Chung and W. Zhao, "Algorithms for Scheduling Imprecise Computations," *Computer*, 24, no. 5, pp. 58–68, May, 1991.

[43] W-K. Shih, J. W. S. Liu and J-Y. Chung, "Algorithms for Scheduling Imprecise Computations with Timing Constraints," *SIAM Journal on Computing*, 20, no. 3, 1991.

[44] W-K. Shih and J. W. S. Liu, "On-line Scheduling of Imprecise Computations to Minimize Error," in *Proceedings of Real-Time Systems Symposium*. Phoenix, Arizona: pp. 280–289, Dec., 1992.

[45] A. C. Yu and K-J. Lin, "Scheduling Parallelizable Imprecise Computations on Multiprocessors," in *Proceedings of the Fifth International Parallel Processing Symposium*. Anaheim, California: pp. 531–536, Apr., 1991.

[46] D. L. Hull, W. Feng and J. W. S. Liu, "Enhancing the Performance and Dependability of Real-Time Systems," in *International Symposium on Performance and Dependability* . Erlangen, Germany: Apr., 1995.

[47] E. C. Horvath, S. Lam and R. Sethi, "A Level Algorithm for Preemptive Scheduling ," *Journal of the Association for Computing Machinery*, 24, no. 1, pp. 32–43, Jan., 1977.

[48] J. R. Jackson, "Scheduling a Production Line to Minimize Maximum Tardiness," University of California, Los Angeles, Technical Research Report 43, Management Science Research Project, 1955.

[49] W. A. Horn, "Some Simple Scheduling Algorithms," *Naval Research Logistics Quarterly*, 21, pp. 177–185, 1974.

[50] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the Association for Computing Machinery*, 20, no. 1, pp. 46–61, Jan., 1973.

[51] R. McNaughton, "Scheduling with Deadlines and Loss Functions," *Management Science*, 6, no. 1, pp. 1–12, Oct., 1959.

[52] M. H. Rothkopf, "Scheduling Independent Tasks on Parallel Processors," *Management Science*, 12, no. 5, pp. 437–447, Jan., 1966.

[53] J. W. S. Liu and C. L. Liu, "Performance Analysis of Heterogeneous Multiprocessor Computing Systems," in *Computer Architectures and Networks (Editors: E. Gelenbe and R. Mahl).* Amsterdam: North-Holland Publishing Company, pp. 331–343, 1974.

[54] J. W. S. Liu and A-T. Yang, "Optimal Scheduling of Independent Tasks on Heterogeneous Computing Systems," in *Proceedings of the ACM Annual Conference* . San Diego: pp. 38–45, Nov., 1974.

[55] G. Schmidt, "Preemptive Scheduling on Identical Processors with Time Dependent Availabilities," Technische Universitat, Institut fur Quantitative Methoden, Berlin, Bericht 83-4, Fachberieich 20 Informatik (Fachgebiet Operations Research), Feb., 1983.

[56] W-K. Shih, J. W. S. Liu, J-Y. Chung and D. W. Gillies, "Scheduling Tasks with Ready Times and Deadlines to Minimize Average Error," presented at ACM Operating Systems Review, July, 1989.

[57] J-Y. Chung and J. W. S. Liu, "Algorithms for Scheduling Periodic Jobs to Minimize Average Error," in *Proceedings of the Ninth International Symposium on Real-Time Systems.* Alabama: pp. 142–151, Dec., 1988.

[58] A. Khemka, K. V. Subrahmanyam and R. K. Shyamasundar, "Multiprocessors Scheduling for Imprecise Computations in a Hard Real-Time Environment," in *Proceedings of the 7th International Parallel Processing Symposium.* New Port, California: pp. 374–378, Apr., 1993.

[59] R. R. Muntz and E. G. Coffman, "Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems," *Journal of the Association for Computing Machinery*, 17, no. 2, pp. 324–338, Apr., 1970.

[60] T. Gonzalez and S. Sahni, "Preemptive Scheduling of Uniform Processor Systems," *Journal of the Association for Computing Machinery*, 25, no. 1, pp. 92–101, Jan., 1978.

[61] S. Lam and R. Sethi, "Worst Case Analysis of Two Scheduling Algorithms," *SIAM Journal of Computing*, 6, no. 3, pp. 518–536, Sept., 1977.

[62] M. L. Dertouzos and A. K-L. Mok, "Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks," *IEEE Transactions on Software Engineering*, 15, no. 12, pp. 1497–1506, Dec., 1989.

[63] A. K-L. Mok, "Task Scheduling in the Control Robotics Environment," Laboratory of Computer Science, Massachusetts Institute of Technology, Tech. Report TM-77, Sept., 1976.

[64] C. L. Liu, J. W. S. Liu and A. L. Liestman, "Scheduling with Slack Time," *Acta Informatica*, 17, pp. 31–41, 1982.

[65] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, 26, no. 1, pp. 127–140, 1978.

[66] S. Sahni and Y. Cho, "Nearly On-Line Scheduling of a Uniform Processor System with Release Times," *SIAM Journal of Computing*, 8, pp. 275–285, 1979.

[67] S. Sahni, "Preemptive Scheduling with Due Dates," *Operations Research* , 27, pp. 925–934, 1979.

[68] K. S. Hong and J. Y-T. Leung, "On-Line Scheduling of Real-Time Tasks," *IEEE Transactions on Computers*, 41, no. 10, pp. 1326–1331, Oct., 1992.

[69] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco, Freeman, 1979.

[70] J. D. Ullman, "NP-Complete Scheduling Problem," *Journal of Computer and System Sciences*, 10, pp. 384–393, 1975.

[71] T. C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, 9, pp. 841–848, 1961.

[72] E. G. Coffman and R. L. Graham, "Optimal Scheduling for Two-Processor Systems," *Acta Informatica*, 1, pp. 200–213, 1972.

[73] T. Gonzalez and D. B. Johnson, "A New A'gorithm for Preemptive Scheduling of Trees," *Journal of the Association for Computing Machinery*, 27, pp. 287–312, 1980.

[74] J. P. Trembley and R. Manohar, *Discrete Mathematical Structures with Applications to Computer Science.* McGraw-Hill Inernational Editions, Computer Science Series, 1987.

[75] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids.* Holt, Rinehart and Winston, 1976.

[76] R. L. Graham, "Bounds on the Performance of Scheduling Algorithms," in *Computer and Job-Shop Scheduling Theory (Editor : E. G. Coffman Jr.).* John Wiley and Sons, Wiley Interscience Publication, Chapter 5, pp. 190–194, 1976.

[77] A. Dey and R. J. Caudill, "Scheduling Algorithms for Real-Time Multiprocessor Robot Arm Control," The Winter Annual Meeting of the American Society of Mechanical Engineers , 14, pp. 289–296, Dec., 1989.

[78] J. Y. S. Luh and S. C. Lin, "Scheduling of parallel computation for a computer-controlled mechanical manipulators," *IEEE Transactions on Systems, Man and Cybernetics*, 12, pp. 214–234, 1982.

[79] J. M. Hollerbach, "A Recursive Lagrangian Formulation of Manipulator Dynamics and a Comparative Study of Dynamics Formulation Complexity," *IEEE Transactions on Systems, Man and Cybernetics*, SMC-10, no. 11, pp. 730–736, 1980.

[80] T. Yoshikawa, *Foundations of Robotics : Analysis and Control.* Cambridge, Massachusetts, MIT Press, 1990.

[81] H. Sholl and T. Booth, "Software Performance Modeling using Computation Structures," *IEEE Transactions on Software Engineering*, SE-1, no. 4, pp. 414–420, Dec., 1975.

[82] A. Avizienis, "Fault-Tolerant Systems," *IEEE Transactions on Computers*, 25, pp. 1304–1311, 1976.

[83] B. W. Johnson, *Design and Analysis of Fault-Tolerant Digital Systems*. Reading, MA, Addison Wesley, 1989.

[84] K. P. Birman, T. A. Joseph, T. Raeuchle and A. E. Abbadi, "Implementing Fault-Tolerant Distributed Objects," *IEEE Transactions on Software Engineering*, 11, pp. 502–508, Jan., 1985.

[85] B. Walker, G. Popek, R. English, C. Kline and G. Thiel, "The Locus Distributed Operating System," in *Proceedings of the 9th Symposium on Operating System Principles*. Bretton Woods, NH: pp. 49–70, Oct., 1983.

[86] R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computer Systems*, 1 , no. 3 , pp. 222–238, Aug., 1983.

[87] E. Gelenbe and D. Derochette, "Performance of Rollback Recovery Systems under Intermittent Failures," *Communications of ACM*, 21, no. 6, pp. 493–499, June, 1978.

[88] K. S. Trivedi, *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Englewood Cliffs, NJ, Prentice-Hall, 1988.

VITA

Surname: Srinivasan Given Names: Anand

Educational Institutions Attended:

University of Victoria	1989 to 1995
Jawaharlal Nehru University, India	1986 to 1989
University of Delhi, India	1983 to 1986

Degrees Awarded:

MSc	University of Victoria	1991
MCA	Jawaharlal Nehru University	1989
BSc (Hons)	University of Delhi	1986

Awards :

University of Victoria Fellowship 1989 to 1994

Publications:

1. A. Srinivasan and G. C. Shoja, "Run-Time Scheduler for Fault-Tolerant Multiprocessor Systems", *in preparation for IEEE Transactions on Parallel and Distributed Systems*

2. A. Srinivasan and G. C. Shoja, "Adaptive Scheduling of Fault-Tolerant Parallel Program with Timing Constraints", *IEEE Conference on System, Man & Cybernetics*, Oct 95

3. A. Srinivasan and G. C. Shoja, "Time-Cost Analysis of Fault-Tolerant Parallel Programs with Timing Constraints",*IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, 343-346, May 1995.

4. A. Srinivasan and G. C. Shoja, "Fault-Tolerance in Limited Time Available Multiprocessing System", *Tech-Report TR-DCS-IR-237*, April 1995.

5. A. Srinivasan and G. C. Shoja,"Response Time Analysis of Fault-Tolerant System using Primary Site Approach with LCFS Repair Server", *IEEE-TENCON, Singapore*, August 1994.

6. A. Srinivasan and G. C. Shoja, "A Fault-Tolerant Dynamic Scheduler for Distributed Hard Real-Time Systems", *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, Canada, vol. 2, 268-271, May 1993.

7. A. Srinivasan, "Fault-Tolerant Distributed Real-Time Scheduling", *M.Sc Thesis, University of Victoria, Canada, August 1991.*

8. A. Srinivasan and G. C. Shoja, "A Fault-Tolerant Scheduler for Distributed Real-Time Systems" *Proceedings of the 1991 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, Victoria, Canada, vol. 1, May 1991, pp. 219 - 222 .

PARTIAL COPYRIGHT LICENSE

Title of Thesis: Fault-Tolerant Real-Time Multiprocessor Scheduling

Author: _____

Signature

Anand Srinivasan

(Name in Block Letters)

December 7, 1995

(Date)