Modeling, assignment and scheduling of tasks in distributed real-time systems

Peng, Dar-Tzen, Ph.D.

The University of Michigan, 1990

MODELING, ASSIGNMENT AND SCHEDULING OF TASKS IN DISTRIBUTED REAL-TIME SYSTEMS

by

Dar-Tzen Peng

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1990

To Yu-Jiing

ACKNOWLEDGMENTS

I would like to express my sincere thanks to my advisor, Professor Kang G. Shin, for his guidance, encouragement and, especially, patience throughout the past few years of my graduate study. His technical and mental guidance and support are perhaps the best that a graduate student can hope for from his/her advisor. I would also like to thank my other committee members for their constructive comments on this thesis. And, of course, my deepest gratitude goes to my parents and my wife. They have shown unquestionable confidence in me and shared with me all the joy and frustration that I have experienced through these long years. Finally, many many thanks go to my son, Bernard. His smiling face and innocent acts are always the best solution to a temporary depression.

The work reported in this thesis was supported in part by the Office of Naval Research under contract No. N00014-85-K-0122 and the National Science Foundation under grant No. DMC-8721492. Any opinions, findings, and conclusions or recommendations reported in the thesis are mine and do not necessarily reflect the view of the funding agencies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

CHAPTER 1
INTRODUCTION

1.1. Overview

Real-time control systems, such as aircraft and nuclear power plants, consist of two synergistic components: the *controlled process* and the *controlling computer.* As the use of digital computers for real-time control increases, real-time computing has emerged as an important discipline in computer science and engineering.

Real-time computing is typified by the speedy execution of computational tasks and high reliability of the system. The former implies that tasks in the system must be allocated (assigned and scheduled) in such a way that they can be completed before their deadlines; otherwise, a *dynamic* failure will occur [SKL85]. The latter implies that the computing system must be reliable; loss of a large number of system components will lead to a *static* failure. In real-time systems, both dynamic and static failures could lead to catastrophic consequences.

Since both task flowtime (response or turnaround time) and system reliability can be improved by using multiple CPUs and memories, distributed systems are an attractive candidate for implementing real-time applications [Sta84]. However, there are many difficult problems to be solved before the distributed system is realized, such as determining a physical topology of the system, allocating tasks, selecting communication structures, and incorporating reliability considerations. For non-real-time applications, ad hoc or intuitive solutions are usually acceptable. For applications that have strict timing constraints, any design should be the result of rigorous analysis and validation. The research results on such a system, as reported in the literature, are far from being satisfactory. This is probably because most problems involved in the design and analysis of distributed real-time systems are NP-hard. Therefore, ad hoc or pseudo-optimal approaches are usually adopted to obtain fast, suboptimal solutions. Among the various problems involved with distributed real-time systems, this thesis will consider only those research issues dealing with the efficient execution of tasks.

Unlike other applications, computational tasks in real-time systems are usually either *periodic* or *aperiodic*. Periodic tasks are invoked at fixed time intervals and constitute the normal computation for the processes under control. An aperiodic task can be invoked at any time in response to environmental stimuli, such as abnormal or critical situations. While an aperiodic task is independent of other tasks, most periodic tasks communicate with one another to accomplish a common system goal. These communications impose precedence constraints during the course of their concurrent execution.

In order to take advantage of a distributed implementation of the system, a good system design should allocate both periodic and aperiodic tasks among the processing nodes (PNs) of the distributed system so that the desired performance can be achieved. Ideally, a system must be designed to have the capability of dynamically allocating both types of tasks to take full advantage of the parallelism in the system. However, the increased overhead accompanying this capability on intertask communication and the increased complexity involved in system analysis might outweigh the benefit of such a design. Therefore, the following compromised scheme is used in the thesis:

CP1. Periodic tasks are pre-assigned among PNs and their assignment remains unchanged throughout the mission lifetime provided no PNs fail.

CP2. Aperiodic tasks may be dynamically transported to other PNs for execution to accomplish load sharing and a load sharing algorithm is assumed to be available.

1.2. Problem Statements and Contributions

In this thesis, five research problems are addressed on how to manage system resources, processors in particular, such that the underlying system can perform as desired. As shown in Fig. 1.1, these five research problems form a hierarchical structure. That is, each problem is formulated assuming that the preceding problems have been solved. The solutions to the five individual problems thus constitute a major part of the complete solution to an integrated problem that covers the entire system.

The performance criterion used here is the *system hazard,* or the maximum normalized task flowtime, where the maximum runs over all tasks of the system. The normalized flowtime of a task is the ratio of the task flowtime to its normalization factor, which is typically the allowable deadline of the task following its arrival. This criterion is chosen because it subsumes many other criteria related to meeting task deadlines. Besides, if the task execution times are random, then the system hazard is a good measure for the system's inability to meet task deadlines.

In what follows, the five research problems and their associated contributions are described. Task allocation is the central step in the design of any distributed real-time system (see Figure 1.1). By allocation, we mean *assignment* with the subsequent *scheduling* considered. To derive an optimal task allocation, an optimal schedule of tasks for any given assignment has to be determined with respect to (w.r.t.) the system hazard.

First, optimal scheduling algorithms for independent tasks on a PN with a single processor are studied in detail. Specifically, optimal algorithms for various scheduling schemes are derived and their associated best processor utilization bounds computed. These results are useful for the scheduling, assignment and sharing of independent tasks in a distributed real-time system.

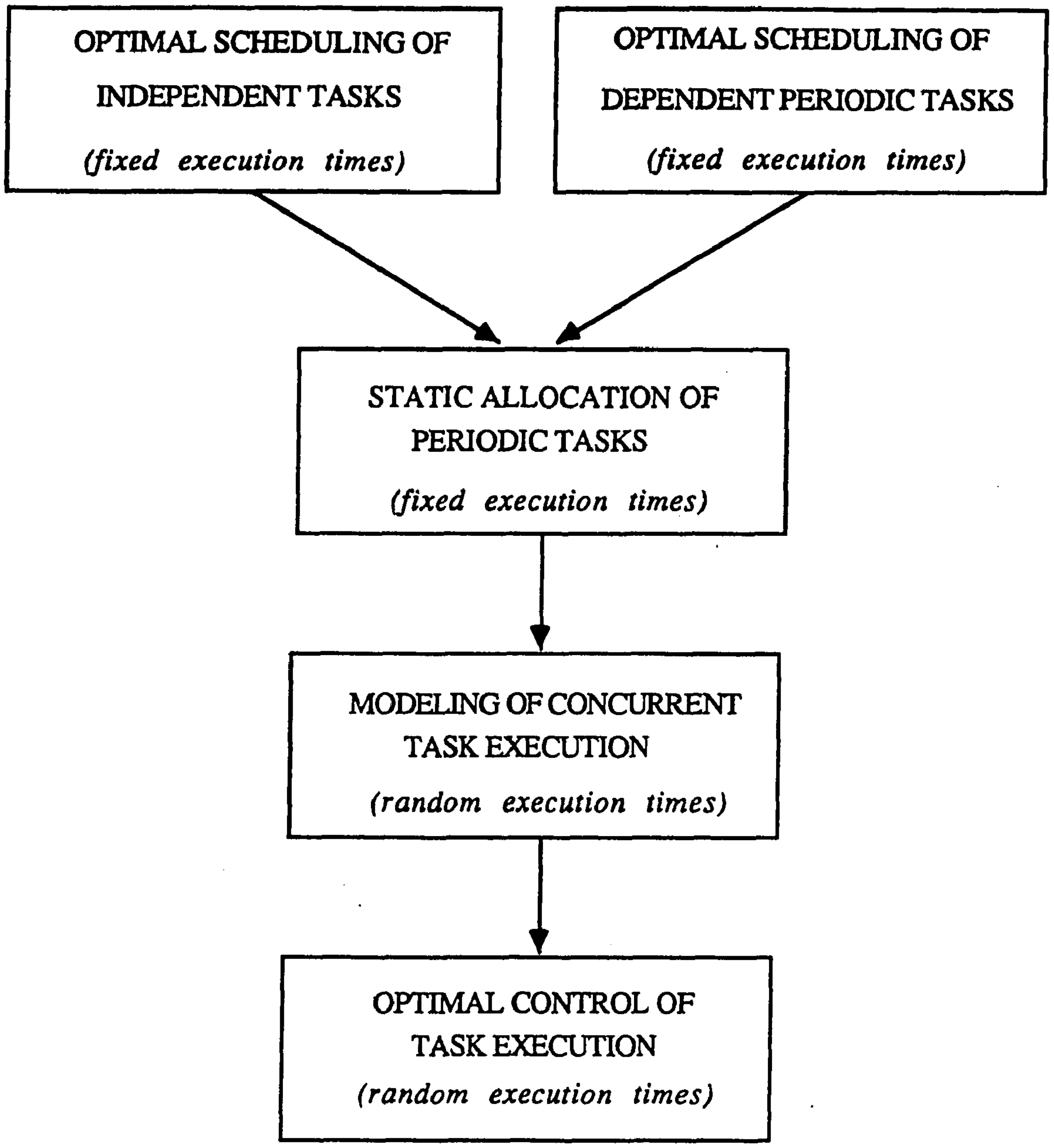

OPTIMAL SCHEDULING OF INDEPENDENT TASKS
(fixed execution times)
OPTIMAL SCHEDULING OF DEPENDENT PERIODIC TASKS
(fixed execution times)
STATIC ALLOCATION OF PERIODIC TASKS
(fixed execution times)
MODELING OF CONCURRENT TASK EXECUTION
(random execution times)
OPTIMAL CONTROL OF TASK EXECUTION
(random execution times)

Figure 1.1 Structure of Problems Addressed

Deriving an optimal schedule for a set of communicating (and thus, dependent) periodic tasks is the second problem, which is treated in the context of a multi-project scheduling problem (MPSP). The MPSP is a generalization of the conventional job-shop scheduling problem. In addition to its use for scheduling communicating tasks, an optimal solution to the MPSP is also useful, for example, in a project-based organization where precedence constraints exist among the *activities* of different projects as well as within a project. An optimal scheduling algorithm is needed for the project leaders so as to cooperate well to complete all the projects in a desired fashion.

Based on the solutions to the first two problems, an optimal allocation of periodic tasks is derived in the third problem. As mentioned earlier, the performance of our task allocation is that of task assignment determined by the subsequent optimal scheduling of the assigned tasks. This is in sharp contrast to conventional approaches, which deal with either assignment or scheduling of tasks, but not both.

Since the actual execution time of each task may not be fixed as was assumed when the tasks were first allocated, a Continuous-Time Markov Chain (CTMC) model is developed in the fourth problem for the concurrent execution of the tasks assigned. In addition to considering precedence constraints, the CTMC modeling has a finer granularity in describing the execution stages of tasks than most other modeling work. The CTMC model has high potential use for resolving various design and analysis issues of distributed real-time systems, such as task execution time estimation, message handling, time-out, etc.

Finally, one particular use of the CTMC model — deriving an optimal solution to the problem of combined periodic task and message scheduling for each PN — constitutes the last problem dealt with in the thesis. Without the CTMC model, this problem would be intractable because of its complicated problem structure.

Notice that, the proposed solutions to all but the first problem may be limited to small or medium sized problems only. For problems of large size, the computation involved may become prohibitively demanding because of the combinatorial nature of the problems addressed. Notice also that each of the research problems has been studied extensively by others for non-real-time applications. However, as we shall see in the following survey, the assumptions and criterion functions used were quite different from the ones proposed in this thesis, making these results inapplicable to our needs.

1.3. Related Work

Task scheduling has been a popular research problem in the fields of Operations Research and Computer Sciences. For scheduling tasks with deadlines, various algorithms have been proposed for different scheduling objectives, and different assumptions on the set of tasks to be scheduled and the set of processors to execute them. Task scheduling in real-time systems can be either *off-line* or *on-line*. An off-line algorithm needs a complete prior knowledge of the tasks' characteristics so that the schedules can be derived beforehand. An on-line algorithm, on the other hand, determines schedules for tasks on the fly at their random arrivals [CMM67]. Thus, off-line algorithms are usually used for scheduling periodic tasks, while on-line algorithms are needed for scheduling aperiodic ones.

As was reviewed in [CSR87], scheduling algorithms in real-time applications are concerned with one and only one objective: meeting task deadlines. An off-line algorithm is *optimal* if it generates a schedule in which each task can be completed before its deadline provided at least one such schedules exists. An on-line algorithm is optimal if it performs as good as its off-line counterpart. For example, in the case of independent tasks to be executed by a single processor, the earliest-due-date (EDD) scheduling algorithm — which always chooses the ready task with the earliest deadline to run — is both an optimal off-line [Bak74, Hor74] and on-line [Der74, Mok83] algorithm when preemption is allowed.

For systems with multiple processors, Horn [Hor74] derived an optimal off-line algorithm to schedule independent tasks with arbitrary release times and deadlines. This result was generalized by Martel [Mar82] for systems with processors of different processing speeds. Unfortunately, on-line algorithms for such systems do not exist except for some special cases [MoD78].

Scheduling tasks with precedence constraints is generally NP-hard [GaJ79] except on a single processor. Baker *et. al* [BLL83] derived an optimal off-line scheduling algorithm for a single processor so as to minimize the maximum task completion cost, where the cost associated with each task can be any monotone non-decreasing function of its completion time. In what follows, a brief survey of the work related to the five research problems are described in the order shown in Figure 1.1.

In the off-line scheduling of independent periodic tasks on a single processor, Liu and Layland [LiL73] derived and analyzed two optimal algorithms: *static* and *dynamic* algorithms. A static algorithm always schedules the ready invocation of one task before that of another if the first task is given priority over the second task. A dynamic algorithm, however, may assign different priorities to different invocations of a task. They showed that the *rate monotonic scheduling* (RMS) algorithm — which assigns a higher priority to a task with shorter invocation period — is an optimal static algorithm, whereas the EDD algorithm is optimal in the dynamic case.

The single-processor on-line scheduling of aperiodic tasks in the presence of periodic ones belongs to the case of the general single-processor on-line scheduling. Thus, the EDD algorithm is also applicable here. Recently, because of the simplicity of the static scheduling of periodic tasks, another approach to scheduling aperiodic tasks has been proposed. By assuming a minimal interarrival time for all aperiodic tasks, this approach treats and schedules the aperiodic tasks just like a new periodic task with a period of the minimal interarrival time [SLR86, LSS87, SSL89].

It is much more difficult to schedule dependent periodic tasks which have been assigned among the PNs of a distributed system than to schedule them on a single processor [BEN82, LLR81]. For example, the general job-shop scheduling problem, a special case of the stated problem, is already NP-hard for the problems even as simple as $J_2 \mid m_j \geq 3 \mid C_{\max}$ (a two-machine job-shop in which the number of operations in any job is greater than or equal to 3 and the scheduling objective is to minimize the maximum job completion time among all jobs), or $J_3 \mid m_j \geq 2 \mid C_{\max}$ [GoS78, LRB77]. Giffer and Thompson [GiT60] are the first to propose a systematic approach to generating the set of all *active schedules* based on which (and variations thereof) most implicit enumerative algorithms have been developed. A set A of active schedules is said to *dominate* another set $S \supseteq A$ of all schedules in the sense that inclusion of an optimal schedule (w.r.t. any *regular measure* [Bak74]) in S implies that in A. In other words, to find optimal schedules w.r.t. any regular measure, it suffices to consider only the set of active schedules, thus reducing the size of the state space to be searched. An extensive survey of job-shop scheduling with branch-and-bound (B&B) methods can be found in [LLR77], where job-shop scheduling was modeled by settling pairs of disjunctive arcs and a tighter bound of cost was also developed by including many other bounds as special cases. Possible extensions of the problems and variations of the solution techniques are described in [BEN82].

As mentioned earlier, the performance of task allocation for real-time tasks must be determined by the subsequent scheduling of the assigned tasks. For a set of independent periodic tasks, Dhall and Liu [DhL78] and their colleagues developed various assignment algorithms based on the RMS algorithm. If precedence constraints exist among the tasks to be allocated, then a general approach to the problem of assigning non-periodic tasks must be taken. However, most prominent methods for task assignment in distributed systems are concerned with minimizing the sum of task processing costs on all assigned processors and interprocessor communications (IPC) costs. As was reviewed in [CHL80], these methods are based on graph theoretic [Sto77, StB78], integer programming [MLT82], or heuristic [ShT85, Vir84] solutions.

Few results have been reported on the task assignment with precedence constraints, because most of such problems are NP-hard [Cof76, GaJ79, LLR81, LeK78]. This fact calls for the development of enumerative optimization methods or approximate algorithms using heuristics [KoS76]. For example, Chu and Lan [ChL87] chose to minimize the maximum processor workload for the assignment of tasks in a distributed real-time system. Workload was defined as the sum of IPC and accumulated execution time on each processor. A wait-time-ratio between two assignments was defined in terms of task queueing delays. Precedence relations were used, in conjunction with the wait-time-ratios, to arrive at two heuristic rules for task assignment.

Modeling concurrent tasks and their execution for real-time applications can be divided into two parts: the *basic objects* to be considered and the *structure* among them. The basic object belongs either to the *module level* or to the *task level*; the structure can be classified as either the *block diagram* or the *Markov Chain*. Most work related to the modeling of concurrent tasks has basic objects belonging to the task level and structures to the block diagram. However, as a more accurate estimate of program execution time or a better control of program execution becomes necessary for real-time applications, the modeling with module level objects and Markov Chain structure evolves.

The task level/block diagram modeling is exemplified by a scheme developed at MIT [War78] where the task system is modeled by a directed acyclic graph in which each node represents a task and each directed arc represents the precedence relationship between the two nodes adjacent to the arc. The task level/Markov Chain model captures the execution states of the task system when the task execution time is assumed to be independently exponentially distributed [ThB83].

There exists a relatively limited literature on the low level, such as module level, modeling of real-time concurrent tasks. This is probably because distributed real-time computing is a new discipline and the importance of the lower level analysis has not been obvious until recently. Chu and Leung [ChL84] and Leung [Leu85] developed a real-time task model to estimate the task response time based on a module level/block diagram modeling in which four types of subgraph: And-Fork to And-Join, Or-Fork to Or-Join, loop and sequential thread are used to construct the block diagram. Huang [Hua85] addressed the issue of software partitions on better resource utilization through a similar model. Woodbury [Woo86] developed a model that incorporates a more complete set of elements including exception handling of real-time tasks to compute the probability distribution function of the execution time of real-time tasks.

While scheduling tasks with fixed execution times has long been studied, its stochastic counterpart (scheduling tasks with random execution times) is relatively new [Web82]. Stochastic scheduling algorithms are normally used to optimize the expected value of certain performance quantity. There are very few research results reported on scheduling tasks with random execution times and precedence constraints in a distributed system.

1.4. Outline of the Dissertation

In Chapter 2, we present the system hazard w.r.t. which optimal static and dynamic scheduling algorithms for independent periodic tasks on a single processor are then derived. Using the optimal algorithms, two best bounds of processor utilization are derived. If, in addition to the periodic tasks, aperiodic tasks arrive randomly at the processor, no optimal on-line scheduling algorithms are shown to exist. Thus, simple mechanisms are described, which can be used to determine whether or not an arriving aperiodic task can be completed with less than the pre-specified system hazard.

Chapter 3 deals with the problem of scheduling communicating periodic tasks in a distributed system. This problem is formulated and solved in the context of Operations Research, thus calling it a multi-project scheduling problem. A multi-project is first expressed in a PERT/CPM form called the *multi-project graph* (MPG). Using the MPG, the dominance relationship between simultaneously schedulable operations are identified to reduce the set of active schedules to be searched with a B&B algorithm. Lower-bound cost estimates are derived to guide the search for an optimal schedule. Finally, a demonstrative example and some computational experiences are presented.

Using the results of optimal scheduling algorithms obtained above, an optimal task allocation can be determined. Since allocation of independent periodic tasks has been studied in considerable detail, only dependent periodic tasks are considered in Chapter 4 for their optimal allocation. While deriving the optimal allocation of periodic tasks, the problem of scheduling aperiodic tasks is also considered through determining the power for processing both types of tasks. To reduce the computation required for an optimal assignment, a polynomial-time algorithm is used to estimate the lower-bound cost for a non-terminal node in the B&B search tree.

After periodic tasks have been optimally assigned to PNs, the modeling of the concurrent execution of the tasks assigned is studied in Chapter 5. First, tasks in each PN are decomposed into *activities*. The activities and the precedence constraints among them are then modeled by a Generalized Stochastic Petri Net (GSPN). Finally, a sequence of homogeneous Continuous-Time Markov Chains (CTMCs) is built from the GSPN to model the concurrent task execution in the distributed real-time system.

In Chapter 6, an important application of the CTMC model is explored. Specifically, we want to solve the problem of optimally scheduling tasks and messages in distributed real-time systems. The scheduling problem is first transformed into a Semi-Markov Decision Processes (SMDP) and the Dynamic Programming (DP) technique is then used to solve the SMDP. We derived an optimal schedule for the centralized case and a sub-optimal schedule for the decentralized case. Moreover, we consider the scheduling problem where each PN periodically broadcasts its local information to other PNs so that a better scheduling decision can be made by each PN.

Finally, the thesis is concluded in Chapter 7 with important results and a suggestion for future research.

CHAPTER 2

A NEW PERFORMANCE MEASURE AND THE SCHEDULING OF INDEPENDENT TASKS

2.1. Introduction

The workload in a real-time system is composed of periodic and aperiodic tasks. Periodic tasks are the "base load" and invoked at fixed time intervals while aperiodic tasks are the "transient load", arriving randomly in response to environmental stimuli. In hard real-time systems such as missile navigation or robot control, execution of both periodic and aperiodic tasks must be not only logically correct but also completed in time. Specifically, there exists an associated deadline for each task before which the task must be completed.

Using different assumptions on the set of tasks to be scheduled and the set of processors to execute them, various scheduling algorithms have been proposed. (See [CSR87] for an extensive survey.) It is important to note that all of these scheduling algorithms are concerned with one and only one objective: meeting task deadlines. A scheduling algorithm is said to be *optimal* if it generates a schedule in which every task can be completed before its deadline, provided such a schedule exists. However, scheduling tasks with this objective alone has the following drawbacks:

- Prior knowledge of the task system based on which conventional scheduling algorithms were derived is not always available or accurate. For instance, the exact execution time of a task is difficult to obtain because of the uncertain behavior of loops and conditional branches in the task.

- It is impossible to evaluate the goodness of a schedule in terms of how early a task can be completed before its deadline. This information is important, especially in scheduling periodic tasks in the presence of randomly arriving aperiodic tasks which must also be completed before their deadlines.

To remedy the above drawbacks, we propose a new performance measure, called the *system hazard,* as the objective function for scheduling real-time tasks. Specifically, the system hazard, denoted by Θ, is the maximum normalized task flowtime, where the flowtime (response time or turn-around time) of a task T_j is defined as the time period between the release (arrival) (r_j) and the completion (c_j) of T_j. That is, $\Theta \equiv \max_j (c_j - r_j) / (d_j - r_j)$, where d_j is the deadline by which T_j must be completed. A schedule is said to be *optimal with respect to* (w.r.t.) Θ if it achieves the smallest possible value of Θ, denoted by Θ^*. Several insights can be drawn from Θ and Θ^* as follows. First, under the assumption that all task execution times are fixed (as with most existing scheduling algorithms), $\Theta^* \leq 1$ if and only if there exists at least one schedule under which all tasks can be completed before their deadlines. Thus, if $\Theta^* \leq 1$, an optimal schedule w.r.t. Θ is also optimal in terms of the ability to meet deadlines. Second, if task execution times are <u>random</u> rather than fixed, then Θ^* is a good measure for the system's inability of meeting task deadlines. Third, Θ^* can also be used to evaluate the goodness of task assignments in distributed real-time systems [PeS89]. An assignment with lower Θ^* is superior to the one with higher Θ^* because the former results in a lower probability of each assigned task missing its deadline.

In this paper, we shall study the optimal preemptive resume scheduling algorithms and their associated processor utilizations for both periodic and aperiodic tasks by minimizing Θ. As was done in [LiL73], this derivation is based on the assumption that A1) tasks are to be scheduled on a single processor, and A2) periodic and aperiodic tasks are independent of each other. A2 means that no precedence constraints exist between any two tasks except for those among periodic tasks implied by their invocation times.

For periodic tasks, both static and dynamic scheduling algorithms are considered. According to the definitions used in [LiL73], a scheduling algorithm is said to be *static* (*dynamic*) if all invocations of a task are assigned the same priority (different priorities). Thus in a static scheduling algorithm, if task T_i is given priority over task T_j, then an invocation of T_i has priority over all invocations of T_j. In a dynamic scheduling algorithm, on the other hand, two invocations of the same task may be assigned different priorities. We shall prove that the *rate-monotonic scheduling* (RMS) algorithm — which was proven to be optimal in meeting deadlines [LiL73] — is also an optimal static algorithm w.r.t. Θ for periodic tasks. For the dynamic case however, the *earliest due date* (EDD) scheduling algorithm — which is optimal w.r.t. many other performance measures — is shown to be not optimal w.r.t. Θ. That is, an optimal schedule derived by minimizing Θ not only meets all the deadlines that can be met by the EDD algorithm, but also offers additional benefits. For aperiodic tasks, we shall show that optimal on-line scheduling algorithms are non-existent except for some special cases.

The rest of the chapter is organized as follows. In Section 2.2, we consider the optimal static scheduling algorithms for periodic tasks as well as achievable processor utilization bounds. The dynamic version of the subject in Section 2.2 is dealt with in Section 2.3. On-line scheduling algorithms for aperiodic tasks are treated in Section 2.4, where, rather than deriving processor utilization bounds, simple mechanisms are proposed to check whether or not a randomly arriving aperiodic task can be completed with not greater than the pre-specified system hazard.

2.2. Optimal Static Scheduling of Periodic Tasks

Let $T = \{T_i \mid i = 1, 2, \cdots, m\}$ be the set of m periodic tasks to be scheduled on a processor, where each task T_i repeats itself with period p_i during the entire mission. Let $I = [0, L)$ be a *planning cycle*, where L is the least common multiple (LCM) of all p_i's. For simplicity, all tasks are assumed to be invoked simultaneously at the beginning of a planning cycle. The ν-th invocation of T_i, denoted by $T_{i\nu}$, is triggered at time $(\nu-1)p_i$ and has to be completed before its next invocation time νp_i. We want to derive a scheduling algorithm that minimizes $\Theta = \max_{\substack{1 \le \nu \le L/p_i \\ T_i \in T}} (c_{i\nu} - r_{i\nu})/p_i$, where $c_{i\nu}$ and $r_{i\nu}$ are the completion time and invocation time of $T_{i\nu}$, respectively.

2.2.1. Optimal Static Scheduling Algorithm

Liu and Layland [LiL73] showed that the rate-monotonic scheduling (RMS) algorithm — which simply assigns priorities based on task invocation periods — is optimal in the sense that it generates a *feasible* schedule provided such a schedule exists.[1] One may suggest that the RMS algorithm may also be optimal w.r.t. Θ provided at least a feasible schedule exists. In what follows, we shall prove this conjecture by first considering the case of $m = 2$ and then generalizing it.

It is necessary to state a useful result that follows directly from [LiL73].

Lemma 2.1: The maximum flowtime of a task occurs when the task is invoked simultaneously with all other higher-priority tasks.

Lemma 2.1 is obvious because the completion of a task will be delayed most if all other higher-priority tasks are invoked at the same time the task is invoked.

Lemma 2.2: For any two periodic tasks T_1 and T_2 such that $p_1 \le p_2$, an algorithm which

[1]A schedule is said to be feasible if all invocations of every task in the schedule can be completed in time.

gives T_1 priority over T_2 yields a smaller Θ than that gives T_2 priority over T_1.

Proof: The lemma can be proved by simply comparing the respective resulting values of Θ as follows. If T_1 is given priority over T_2, then the normalized flowtime of T_1 is e_1/p_1 for all invocations of T_1 within the planning cycle I, where e_i is the execution time of T_i, $i=1,2$. Since T_1 may preempt T_2, the completion of T_2 gets delayed whenever there exists an unfinished invocation of T_1. However, because of Lemma 2.1, only the first invocation of T_2 needs to be considered when determining the maximum normalized flowtime of T_2 (and, thus, the value of Θ) with the above priority assignment.

Note that T_2 can be executed only within the time interval between the completion of the current invocation of T_1 and its next invocation. Suppose T_2 is completed during T_1's $(n+1)$-th invocation interval (Fig. 2.1(a)) — that is, $n(p_1 - e_1) \le e_2 < (n+1)(p_1 - e_1)$ — where $n \le \lfloor \frac{p_2}{p_1} \rfloor$.[2] Then $\Theta = \max\ \{e_1/p_1, ((n+1)e_1 + e_2)/p_2\}$.

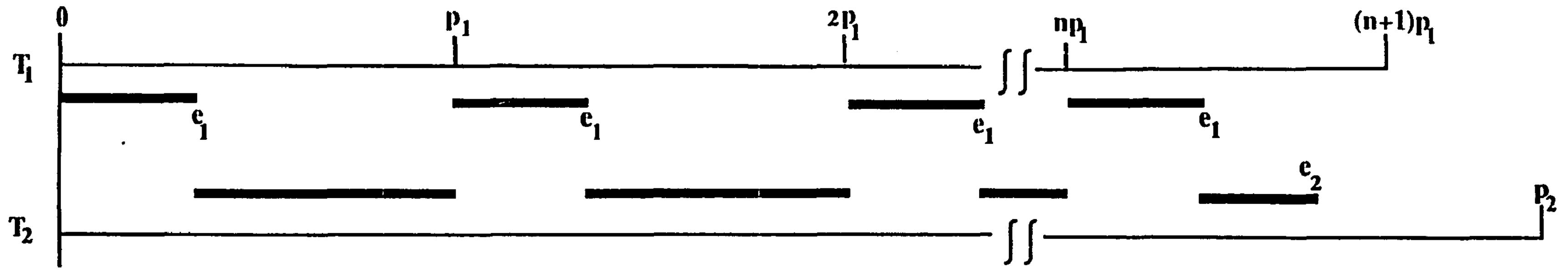
On the other hand, if T_2 is given priority over T_1 then the normalized flowtime of T_2 is e_2/p_2, and that of T_1 becomes $(e_1 + e_2)/p_1$, because T_2 preempts T_1 and $p_2 \ge p_1$ (Fig. 2.1(b)).

Since $(e_1 + e_2)/p_1 \ge e_2/p_2$ and $(e_1 + e_2)/p_1 \ge e_1/p_1$, we want to show that

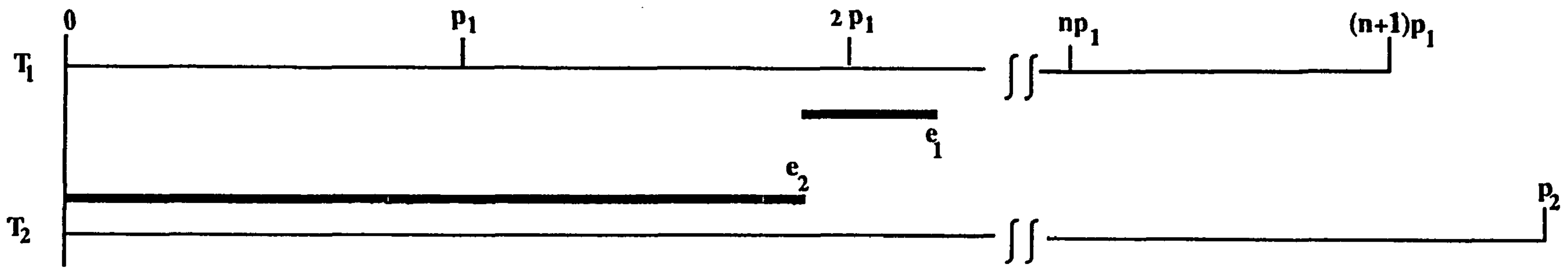
$$\frac{e_1 + e_2}{p_1} \ge \frac{(n+1)e_1 + e_2}{p_2}. \tag{2.1}$$

Eq. (2.1) is obviously true if $n = 0$. Now, consider the case of $n \ge 1$. Since $e_2 \ge n(p_1 - e_1)$, $(e_1 + e_2)/p_1 \ge n(e_1 + (p_1 - e_1))/p_1 \ge (e_1 + (p_1 - e_1))/p_1 = 1$. However, by assumption, a feasible schedule exists and the RMS algorithm always produces a feasible schedule. The RHS of Eq. (2.1) is the system hazard resulting from the application of the RMS algorithm, and

[2] $\lfloor x \rfloor$ represents the largest integer which is smaller than or equal to x, whereas $\lceil x \rceil$ the smallest integer greater than or equal to x. Thus, $\lceil x \rceil = \lfloor x \rfloor$ if x is an integer. Otherwise, $\lceil x \rceil = \lfloor x \rfloor + 1$.



2.1(a). T_1 has a higher priority over T_2



2.1(b). T_2 has a higher priority over T_1

Figure 2.1. Different Priority Assignments and Θ 's for $T=\{T_1, T_2\}$

thus, never exceeds one, making the inequality (2.1) hold. **Q.E.D.**

The above result is generalized in the following theorem.

Theorem 2.1: If a feasible schedule exists for $m \geq 2$ tasks, then the RMS algorithm is optimal w.r.t. Θ.

Proof: Suppose that T_i and T_j are any two tasks of adjacent priorities in a given schedule with $p_i < p_j$, and T_j is given priority over T_i. Then, we want to show that exchanging the priorities of these two tasks will always result in a feasible schedule with a lower Θ.

Let θ_j (θ_i) be the maximum normalized flowtime of T_j (T_i) when T_i (T_j) is given priority over T_j (T_i). We want to show $\theta_i > \theta_j$ as follows. Note that there may exist tasks with higher priorities than both T_i and T_j, which will delay the completion of T_i and T_j. Let Y denote the total delay caused by these higher priority tasks on the completion of the first[3] invocations of T_i and T_j. Following the same steps in the case of $m = 2$, we need to consider two cases: (i) $Y + e_i + e_j \leq p_j$ and (ii) $Y + e_i + e_j > p_j$. For (i), $\theta_j = (Y + e_i + e_j)/p_j$, $\theta_i = (Y + e_j + e_i)/p_i$, and thus, $\theta_i > \theta_j$ because $p_j > p_i$. For (ii), $\theta_i = (Y + e_j + e_i)/p_i \geq (Y + e_j + e_i)/p_j > 1$. However, by assumption, a feasible schedule exists and the RMS algorithm always generates a feasible schedule, thus implying that $\theta_j \leq 1 < \theta_i$. Therefore, a new schedule in which T_i is given priority over T_j will result in a lower Θ. The theorem follows since a feasible schedule based on the RMS algorithm can always be obtained from any given schedule by a sequence of pairwise exchanges as described above. **Q.E.D.**

[3]Since all task periods are assumed to begin with the start of each planning cycle, by Lemma 2.1 it is sufficient to consider the total delay on the completion of the first invocations of T_i and T_j.

2.2.2. Achievable Processor Utilization Bounds

As was done in [LiL73], the *utilization* U of a single processor system with m periodic tasks T_i's is defined as $U = \sum_{i=1}^{m} e_i/p_i$. Obviously, a higher U means a better processor utilization and $U \le 1$ must hold provided every invocation of a task can be completed before its deadline. It was shown in [LiL73] that for $\Theta = 1$ there exist two bounds of U: $U_l = m(2^{1/m} - 1)$ and $U_h = 1$. Specifically, given any m tasks with $U \le U_l$, there always exists a feasible schedule for these tasks. On the other hand, if $U > U_h$, then no feasible schedule exists for these tasks. If $U_l < U \le U_h$, then a feasible schedule may, or may not, exist depending on p_i's and e_i's. In what follows, we shall generalize these results under the condition of $\Theta \le 1$. That is, for any given value of $0 < \Theta \le 1$, we want to derive the corresponding values of U_l and U_h. (The results of [LiL73] are a special case of ours since theirs were derived for the case of $\Theta = 1$.) For any m tasks with $U \le U_l$, there always exists a feasible schedule with the maximum normalized flowtime (of all invocations of all tasks) not exceeding Θ. On the other hand, if $U > U_h$, then no such feasible schedule exists for these tasks. If $U_l < U \le U_h$, then such a feasible schedule may, or may not, exist depending on the values of p_i's and e_i's.

Notice that for any $\Theta \in (0, 1]$, U_l and U_h should be derived under the RMS algorithm — which is optimal w.r.t. Θ by Theorem 2.1 — because the RMS algorithm yields the maximal values of U_l and U_h among all static scheduling algorithms. Also, the values of U_l and U_h corresponding to any $0 < \Theta < 1$ will be shown to be smaller than those in [LiL73], which were derived for the case of $\Theta = 1$. Throughout the rest of the chapter, a schedule is said to be *feasible* for a given $\Theta \in (0, 1]$ if it results in a system hazard not exceeding Θ.

2.2.2.1. Deriving U_l

U_l is first derived for two tasks T_1 and T_2 with $p_1 \le p_2$. This result will then be extended for an arbitrary number of tasks. To derive U_l, two cases need to be considered: (C1) $\lfloor \frac{\Theta p_2}{p_1} \rfloor p_1 \le \Theta p_2 < \lfloor \frac{\Theta p_2}{p_1} \rfloor p_1 + \Theta p_1$, where $\lfloor \frac{\Theta p_2}{p_1} \rfloor \ge 1$ (Fig. 2.2(a)), and (C2) $\lfloor \frac{\Theta p_2}{p_1} \rfloor p_1 + \Theta p_1 \le \Theta p_2 < \lceil \frac{\Theta p_2}{p_1} \rceil p_1$, where $\lfloor \frac{\Theta p_2}{p_1} \rfloor \ge 0$ (Fig. 2.2(b)).

As was done in [LiL73], U_l and U_h are derived with the processor fully utilized. A processor is said to be *fully utilized* by a set T of tasks for a given Θ under a feasible schedule if increasing the execution time of any one task in T makes the schedule infeasible.

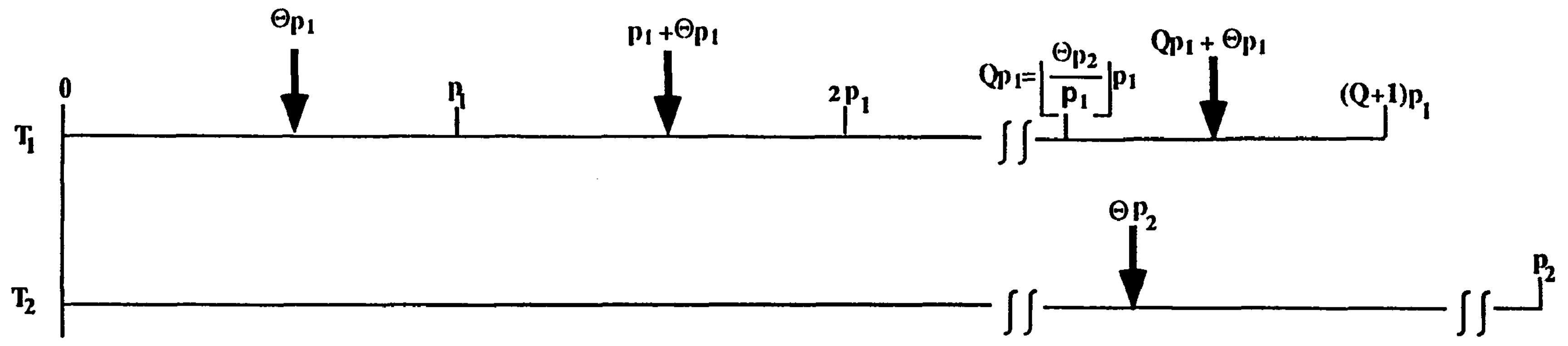
C1. In order for T_1's normalized flowtime not to exceed the given value of Θ, $e_1 \le \Theta p_1$, but e_1 may or may not exceed $\xi \equiv \Theta p_2 - \lfloor \frac{\Theta p_2}{p_1} \rfloor p_1$. When $e_1 \le \xi$, the largest e_2 that allows for at least one feasible schedule is $\Theta p_2 - \lceil \frac{\Theta p_2}{p_1} \rceil e_1$, and the corresponding utilization is

$$U = \frac{e_1}{p_1} + \frac{e_2}{p_2} = \frac{e_1}{p_1} + \Theta - \frac{e_1}{p_2} \lceil \frac{\Theta p_2}{p_1} \rceil = \Theta + e_1 \left\{ \frac{1}{p_1} - \frac{1}{p_2} \lceil \frac{\Theta p_2}{p_1} \rceil \right\}. \tag{2.2}$$

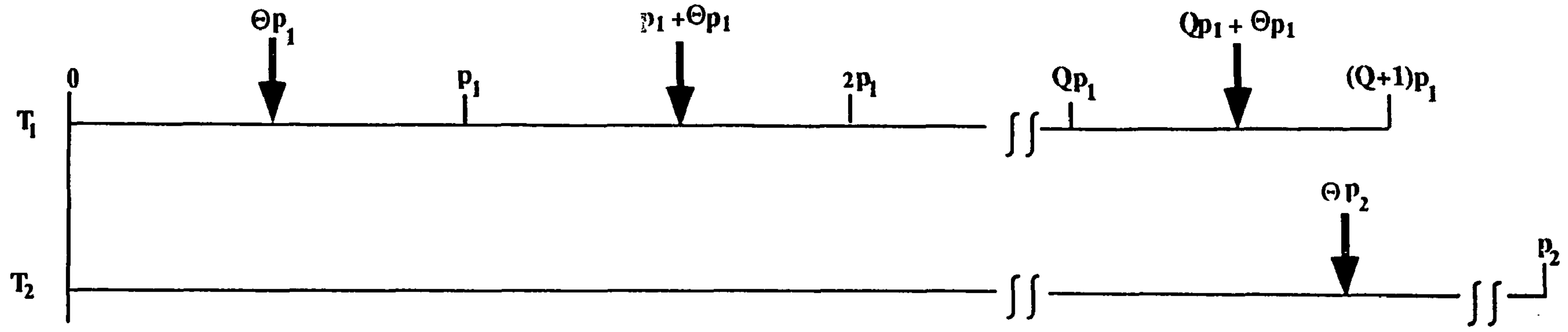
On the other hand, when $e_1 > \xi$, e_2 must be no larger than $(p_1 - e_1) \lfloor \frac{\Theta p_2}{p_1} \rfloor$, and the corresponding utilization becomes

$$U = \frac{e_1}{p_1} + \frac{p_1 - e_1}{p_2} \lfloor \frac{\Theta p_2}{p_1} \rfloor = \frac{p_1}{p_2} \lfloor \frac{\Theta p_2}{p_1} \rfloor + e_1 \left\{ \frac{1}{p_1} - \frac{1}{p_2} \lfloor \frac{\Theta p_2}{p_1} \rfloor \right\}. \tag{2.3}$$

The U in Eq. (2.3) is a non-decreasing function of e_1 because $\frac{1}{p_1} \ge \frac{1}{p_2} \lfloor \frac{\Theta p_2}{p_1} \rfloor$. Thus, the behavior of U in Eq. (2.2) as a function of e_1 determines U_l. Specifically, if the U in Eq. (2.2) is also a non-decreasing function of e_1, then U_l occurs at $e_1 = 0$. If U is a non-increasing function of e_1, then U_l occurs at $e_1 = \xi = \Theta p_2 - \lfloor \frac{\Theta p_2}{p_1} \rfloor p_1$.



2.2(a). $Qp_1 \le \Theta p_2 < Qp_1 + \Theta p_1$



2. 2(b). $Qp_1 + \Theta p_1 \le \Theta p_2 < (Q+1)\, p_1$

Figure 2.2. Different Relations Between Θp_1 and Θp_2

C2. Similarly to the case of $e_1 \le \xi$ in C1, the largest allowable e_2 is $\Theta p_2 - \lceil \frac{\Theta p_2}{p_1} \rceil e_1$, and the corresponding utilization is $U = \Theta + e_1 \left\{ \frac{1}{p_1} - \frac{1}{p_2} \lceil \frac{\Theta p_2}{p_1} \rceil \right\}$, which is a non-decreasing function of e_1 for the following reason. Since, by assumption, $\lfloor \frac{\Theta p_2}{p_1} \rfloor p_1 + \Theta p_1 \le \Theta p_2$, $\lfloor \frac{\Theta p_2}{p_1} \rfloor \le \Theta(\frac{p_2}{p_1} - 1) \le \frac{p_2}{p_1} - 1)$ implying that $\frac{1}{p_1} - \frac{1}{p_2} \lceil \frac{\Theta p_2}{p_1} \rceil \ge 0$. Thus, $U_l = \Theta$ occurs at $e_1 = 0$.

More formally, we have the following theorem.

Theorem 2.2: For a given $\Theta \le 1$ and two tasks T_1 and T_2,

$$U_l = \begin{cases} \Theta & \text{if } \Theta \le 0.5 \\ 2(\sqrt{2\Theta} - 1) + 1 - \Theta & \text{otherwise.} \end{cases}$$

Proof: We prove this theorem by deriving U_l under both C1 and C2, and choosing the smaller of the two. To find U_l under C1, it is necessary to examine the behavior of U in Eq. (2.2). First, consider the case of $\Theta \le 0.5$. Since $\frac{\Theta p_2}{p_1} \ge \lfloor \frac{\Theta p_2}{p_1} \rfloor$, $\frac{p_2}{p_1} \ge \frac{1}{\Theta} \lfloor \frac{\Theta p_2}{p_1} \rfloor \ge 2 \lfloor \frac{\Theta p_2}{p_1} \rfloor \ge 1 + \lfloor \frac{\Theta p_2}{p_1} \rfloor \ge \lceil \frac{\Theta p_2}{p_1} \rceil$. Thus, $\frac{1}{p_1} \ge \frac{1}{p_2} \lceil \frac{\Theta p_2}{p_1} \rceil$, indicating that the U of Eq. (2.2) is a non-decreasing function of $e_1 \in [0, \Theta p_1]$. Thus under C1, $U_l = \Theta$ occurs at $e_1 = 0$. The theorem follows for $\Theta \le 0.5$ because $U_l = \Theta$ also holds under C2.

Next, consider the case of $\Theta > 0.5$. It can be easily shown that there always exists at least a (p_1, p_2) pair such that $\frac{1}{p_1} < (\ge) \frac{1}{p_2} \lceil \frac{\Theta p_2}{p_1} \rceil$. In other words, the U of Eq. (2.2) could be non-increasing (non-decreasing) in e_1, depending on the values of p_1 and p_2. To derive U_l under C1 for $\Theta > 0.5$, one has to find U_l's for these two possibilities and then choose the smaller of them as U_l under C1 as follows. Since $U_l = \Theta$ if U in Eq. (2.2) is non-decreasing in e_1, we now consider the possibility of Eq. (2.2) being non-increasing in e_1. If Eq. (2.2) is non-increasing, then $U_l = U|_{e_1=\xi}$ or, from Eq. (2.2) or (2.3),

$$\begin{aligned} U_l &= \frac{p_1}{p_2}\left\lfloor\frac{\Theta p_2}{p_1}\right\rfloor + \left\{\Theta p_2 - p_1\left\lfloor\frac{\Theta p_2}{p_1}\right\rfloor\right\}\left\{\frac{1}{p_1} - \frac{1}{p_2}\left\lfloor\frac{\Theta p_2}{p_1}\right\rfloor\right\} \\ &= \frac{p_1}{p_2}\left[\left\lfloor\frac{\Theta p_2}{p_1}\right\rfloor\right]^2 + \left\{\frac{p_1}{p_2} - \Theta - 1\right\}\left\lfloor\frac{\Theta p_2}{p_1}\right\rfloor + \frac{\Theta p_2}{p_1}. \end{aligned} \tag{2.4}$$

For a given Θ, Eq. (2.4) is an expression for U_l under C1 for a given (p_1, p_2) pair. To find U_l under C1, one must find the minimum of the above U_l among all possible (p_1, p_2) pairs. To find a special (p_1, p_2) pair which minimizes U_l, let $\Theta p_2 / p_1 = Q + R$, where Q is a positive integer representing the quotient and $0 \le R < 1$ representing the remainder of $\Theta p_2 / p_1$. Then, U_l in Eq. (2.4) becomes

$$\begin{aligned} U_l &= \frac{\Theta}{Q+R}Q^2 + (\frac{\Theta}{Q+R} - \Theta - 1)Q + (Q+R) \\ &= \frac{\Theta Q^2 + \Theta Q + (R - \Theta Q)\,(Q+R)}{Q+R} \\ &= \frac{(R + \Theta - R\Theta)\,Q + R^2}{Q+R} \\ &= (R + \Theta - R\Theta) - \frac{R(\Theta - R\Theta)}{Q+R}. \end{aligned} \tag{2.5}$$

Given Θ and R, U_l in Eq. (2.5) is non-decreasing in Q because $\Theta - R\Theta \ge 0$. Therefore, the minimum of U_l must occur at $Q = 1$. Substituting $Q = 1$ into Eq. (2.5), U_l becomes $\frac{R^2 + (1-\Theta)R + \Theta}{R+1}$. To further minimize U_l w.r.t. R, we take the derivative of U_l w.r.t. R and find $R = R^*$ such that $\frac{dU_l}{dR}|_{R=R^*} = 0$. It turns out that $R^* = \sqrt{2\Theta} - 1$ and the corresponding minimum of U_l is $2(\sqrt{2\Theta} - 1) + 1 - \Theta$. To prove that $2(\sqrt{2\Theta} - 1) + 1 - \Theta$ is acceptable, we have to show that $(Q, R) = (1, R^*)$ is in the domain where Eq. (2.2) is indeed non-increasing. In other words, for any given $\Theta > 0.5$, we want to show that $\frac{1}{p_1} - \frac{1}{p_2}\left\lceil\frac{\Theta p_2}{p_1}\right\rceil \le 0$ at $(Q, R) = (1, R^*)$. Notice that

$$\frac{1}{p_1} - \frac{1}{p_2}\lceil\frac{\Theta p_2}{p_1}\rceil = \frac{1}{p_2}\left\{\frac{p_2}{p_1} - (Q+1)\right\}$$
$$= \frac{1}{p_2}\left\{\frac{1}{\Theta}(Q+R) - (Q+1)\right\}.$$

Substituting $(1, \sqrt{2\Theta} - 1)$ for (Q, R), the above becomes

$$\frac{1}{p_2}\left\{\frac{1}{\Theta}(1 + \sqrt{2\Theta} - 1) - (1+1)\right\} = \frac{1}{p_2}\left[\frac{1}{\Theta}\sqrt{2\Theta} - 2\right]$$
$$= \frac{1}{p_2}\left[\sqrt{\frac{2}{\Theta}} - 2\right] \le 0$$

provided $\Theta > 0.5$. Thus, the U_l for $\Theta > 0.5$ under C1 is $\min\{2(\sqrt{2\Theta} - 1) + 1 - \Theta,\ \Theta\}$ $= 2(\sqrt{2\Theta} - 1) + 1 - \Theta$. Choosing the smaller of this U_l and that under $C2$, the U_l in case of $\Theta > 0.5$ becomes $\min\{2(\sqrt{2\Theta} - 1) + 1 - \Theta,\ \Theta\} = 2(\sqrt{2\Theta} - 1) + 1 - \Theta$. Q.E.D.

Notice that the results presented in [LiL73] can be obtained from Theorem 2.2 by letting $\Theta = 1$. The results of Theorem 2.2 are extended below for $m \ge 2$ tasks.

Theorem 2.3: For $m \ge 2$ periodic tasks $\{T_i\}$ and a given $\Theta \le 0.5$, $U_l = \Theta$.

Proof: This theorem can also be proved by pairwise exchanges. Let $p_1 \le p_2 \le \cdots \le p_m$. To derive U_l, we again assume that the RMS algorithm is used and the processor is fully utilized.

Construct new task execution times, $e_1' = 0$, $e_i' = e_i$, $i = 2, 3, \cdots, m-1$, and $e_m' \ge e_m$, such that the processor is fully utilized. We want to show that $U' = \sum_{i=1}^{m} e_i' / p_i \le U = \sum_{i=1}^{m} e_i / p_i$. Specifically, we need to prove that $e_m' / p_m \le e_1 / p_1 + e_m / p_m$. Since $e_m' \le e_m + \lceil\frac{\Theta p_m}{p_1}\rceil\, e_1$ must hold,

$$\frac{e_m'}{p_m} \le \frac{e_m}{p_m} + \frac{e_1}{p_m}\left\lceil\frac{\Theta p_m}{p_1}\right\rceil \le \frac{e_m}{p_m} + \frac{e_1}{p_1}$$

because, as shown in Theorem 2.2, $\frac{1}{p_1} \ge \frac{1}{p_m}\left\lceil\frac{\Theta p_m}{p_1}\right\rceil$ provided $\Theta \le 0.5$. In other words, U_l must occur at $e_1 = 0$. The above arguments can be applied repeatedly between e_2 and e_m, between e_3 and e_m, and so on. Finally, we conclude that U_l must occur when $e_1 = e_2 = \cdots = e_{m-1} = 0$, $e_m = \Theta p_m$, and thus, $U_l = \Theta p_m / p_m = \Theta$. **Q.E.D.**

Before extending Theorem 2.3 to the case of $\Theta > 0.5$, it is necessary to prove the following lemma.

Lemma 2.3: For a set T of $m \ge 2$ tasks and a given $\Theta > 0.5$, if $p_{m-1} \le \Theta p_m$ and $p_m \le 2p_1$, then the minimum processor utilization occurs when (see Fig. 2.3),

$$e_i = e_i^* = \begin{cases} p_{i+1} - p_i & i = 1, \cdots, m-2 \\ \Theta p_m - p_{m-1} & i = m-1 \\ \Theta p_m - 2\sum_{i=1}^{m-1} e_i & i = m. \end{cases} \tag{2.6}$$

Proof: Assume $p_1 < p_2 < \cdots < p_m$. (As will be clear in the following steps of proof, the lemma is also true when $p_i = p_{i+1}$, $1 \le i < m$.) Let $\{e_i\}$ be an arbitrary set of execution times for which the processor is fully utilized (with utilization U). We want to show that U must be greater than that corresponding to the set of e_i^* values in Eq. (2.6). This is done by considering each task sequentially as follows.

First, consider T_1. If $e_1 < p_2 - p_1$, then construct new task execution times $\{e_i'\}$ such that $e_1' = p_2 - p_1$, $e_i' = e_i$, $i = 2, 3, \cdots, m-1$. In order to let $\{e_i'\}$ fully utilize the processor, $e_m' = e_m - 2[(p_2 - p_1) - e_1] > 0$ (Fig. 2.3). We need to show that the processor utilization U' corresponding to $\{e_i'\}$ is smaller than U, i.e., $e_1' / p_1 + e_m' / p_m < e_1 / p_1 + e_m / p_m$. This inequality holds because $\frac{(p_2 - p_1) - e_1}{p_1} < \frac{2[(p_2 - p_1) - e_1]}{p_m}$ since, by assumption, $p_1 > p_m / 2$. On the other hand,

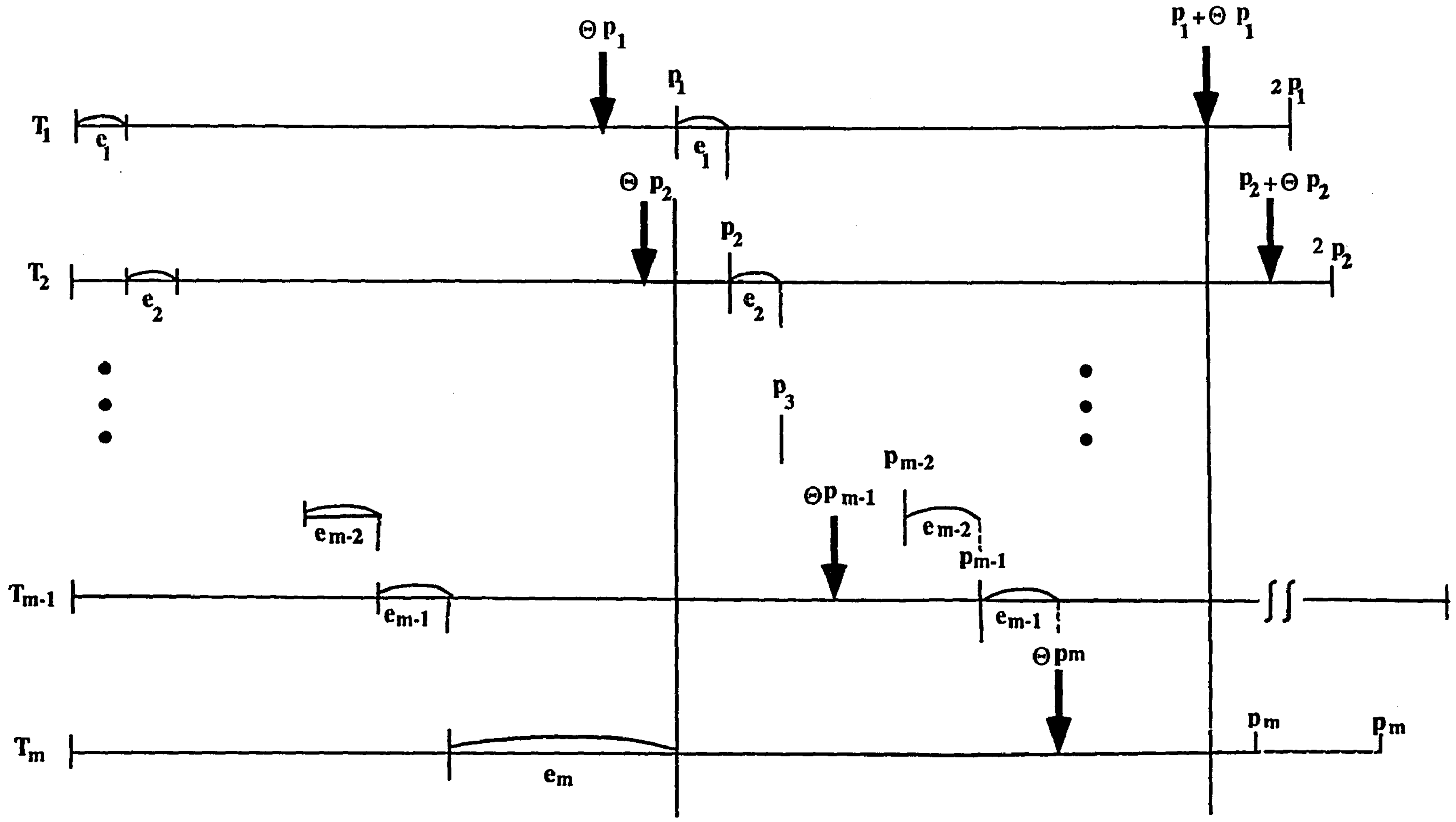


Figure 2.3. e_i 's which minimize U

if $e_1 > p_2 - p_1$, then the new execution times $\{e_i'\}$ must be constructed such that $e_1' = p_2 - p_1$, $e_2' = e_2 + [e_1 - (p_2 - p_1)]$, and $e_i' = e_i$, $i = 3, 4, \cdots, m$ to again fully utilize the processor (Fig. 2.3). Likewise, to show that the new utilization U' is less than U, we have to show that $[e_1 - (p_2 - p_1)] / p_1 > [e_1 - (p_2 - p_1)] / p_2$, which is true since $p_1 < p_2$. For the case where $e_1 = p_2 - p_1$ or after constructing $\{e_i'\}$ as above, we move on to consider T_2 based on the newly constructed $\{e_i'\}$. While considering T_2, we likewise increase (decrease) e_2' to $e_2'' = p_3 - p_2$ and decrease e_m' (increase e_3') twice the (the same) amount that e_2' has been increased (decreased) if e_2' <(>) $p_3 - p_2$. For the same reason as the case of T_1, $U'' < U'$ must hold, where U'' represents this newly generated utilization with T_2 considered. This process can be repeatedly applied to $T_3, T_4, \cdots$ and up to T_{m-2} such that the utilization is reduced each time. Finally, we consider T_{m-1} and conclude that given $e_i = p_{i+1} - p_i$, $i = 1, 2, \cdots, m-2$, $e_{m-1} = \Theta p_m - p_{m-1}$ must hold to minimize U. Since the utilization for any set of task execution times can be reduced with the above procedure until Eq. (2.6) is satisfied, the lemma follows. Q.E.D.

Using Lemma 2.3, we have the following theorem.

Theorem 2.4: For $m \geq 2$ tasks and a given $\Theta > 0.5$, if $p_{m-1} \leq \Theta p_m$ and $p_m \leq 2p_1$, then $U_l = m[(2\Theta)^{\frac{1}{m}} - 1] + 1 - \Theta$.

Proof: From Lemma 2.3, if $p_{m-1} \leq \Theta p_m$, $p_m \leq 2p_1$, and e_i's satisfy Eq. (2.6), then the resulting utilization will be minimized. We now derive U_l by searching for the minimal utilization among all possible p_i's in the domain where $p_{m-1} \leq \Theta p_m$ and $p_m \leq 2p_1$ hold. That is, we want to find the minimum of $U = \sum_{i=1}^{m} e_i / p_i$ while varying p_i's and satisfying Eq. (2.6).

To derive U_l, we use the idea in [LIL73] and define variables $g_i = (\Theta p_m - p_i) / p_i$, $i = 1, 2, \cdots, m-1$, and thus, $p_i = \Theta p_m - g_i p_i = \Theta p_m / (1 + g_i)$.

Expressing e_i's in terms of g_i's and p_i's as $e_i = p_{i+1} - p_i = g_i p_i - g_{i+1} p_{i+1}$, $i = 1, 2, \cdots, m-2$, $e_{m-1} = \Theta p_m - p_{m-1} = g_{m-1} p_{m-1}$, and $e_m = \Theta p_m - 2\sum_{i=1}^{m-1} e_i = \Theta p_m - 2(\Theta p_m - p_1) = -\Theta p_m + 2(\Theta p_m - g_1 p_1) = \Theta p_m - 2 g_1 p_1$, we get

$$\begin{aligned} U &= \sum_{i=1}^{m} \frac{e_i}{p_i} = \sum_{i=1}^{m-2} (g_i - g_{i+1} \frac{p_{i+1}}{p_i}) + g_{m-1} + \Theta - 2 g_1 \frac{p_1}{p_m} \\ &= \sum_{i=1}^{m-2} (g_i - g_{i+1} \frac{1 + g_i}{1 + g_{i+1}}) + g_{m-1} + \Theta - 2\Theta \frac{g_1}{1 + g_1}. \end{aligned} \tag{2.7}$$

Notice that if $g_1 = g_2 = \cdots = g_{m-1} = 0$ (i.e., $\Theta p_m = p_i$, $i = 1, 2, \cdots, m-1$), then $U = \Theta$. To derive U_l, Eq. (2.7) is minimized w.r.t. all g_i's. This is done by setting $\frac{\partial U}{\partial g_i} = 0$, $\forall\, i$ and solving the resulting simultaneous difference equations:

$$\begin{cases} 1 - \dfrac{2\Theta}{(1 + g_1)^2} = \dfrac{g_2}{1 + g_2} \\ 1 - \dfrac{1 + g_{i-1}}{(1 + g_i)^2} = \dfrac{g_{i+1}}{1 + g_{i+1}}, \qquad i = 2, 3, \cdots, m-2, \\ 1 - \dfrac{1 + g_{m-2}}{(1 + g_{m-1})^2} = 0. \end{cases} \tag{2.8}$$

It can be shown that

$$g_i = (2\Theta)^{\frac{m-i}{m}} - 1, \qquad i = 1, 2, \cdots, m-1, \tag{2.9}$$

solve Eq. (2.8) and minimize U of Eq. (2.7). Replacing g_i's of Eq. (2.7) with those of Eq. (2.9), one can get

$$U_l = m\,[(2\Theta)^{\frac{1}{m}} - 1] + 1 - \Theta. \tag{2.10}$$

Similarly to the proof of Theorem 2.2, we still need to check if p_i's (or g_i's) in Eq. (2.9) are in the domain where $p_m < 2p_i$ holds. This can be easily done as follows. Since $g_i = (\Theta p_m - p_i) / p_i$ by definition, $\Theta p_m / p_i = (2\Theta)^{\frac{m-i}{m}}$ from Eq. (2.9). It follows that $\frac{p_m}{p_i} = \frac{1}{\Theta} (2\Theta)\, (2\Theta)^{\frac{-i}{m}} = 2(2\Theta)^{\frac{-i}{m}} \le 2$, since, by assumption, $\Theta > 0.5$. Q.E.D.

Notice that for a given $\Theta > 0.5$, the above U_l is a decreasing function of m, and for $m = 2$, Eq. (2.10) gives the same U_l as in Theorem 2.2. Also, Eq. (2.10) becomes the same result in [LiL73] when $\Theta = 1$. It is interesting to see that $U_l \rightarrow \log (2\Theta) + 1 - \Theta$ as $m \rightarrow \infty$.

Theorem 2.4 was proved under the restriction that $p_{m-1} \le \Theta p_m$ and $p_m \le 2p_1$. In Lemma 2.4 and Theorem 2.5 below, we relax this restriction step-by-step and then present the general results on U_l.

Lemma 2.4: For a set T of $m \ge 2$ tasks and a given $\Theta > 0.5$, if $p_{m-1} \le \Theta p_m \le p_1 + \Theta p_1$, then[4] $U_l = m[(2\Theta)^{\frac{1}{m}} - 1] + 1 - \Theta$.

Proof: Suppose $p_m > 2p_i$, $i = 1, 2, \cdots, n$, and $p_m \le 2p_i$, $i = n{+}1, n{+}2, \cdots, m$. Construct $\{e_i'\}$ such that

$$e_i' = \begin{cases} 0 & i = 1, 2, \cdots, n \\ p_{i+1} - p_i & i = n{+}1, n{+}2, \cdots, m{-}2 \\ \Theta p_m - p_{m-1} & i = m{-}1 \\ \Theta p_m - 2\sum_{i=1}^{m-1} e_i & i = m. \end{cases}$$

Then, from previous discussions and Lemma 2.3, the new utilization associated with $\{e_i'\}$ will

[4]Notice that the conditions $p_m \le \Theta p_m$ and $p_m \le 2p_1$ of Theorem 2.4 together imply $p_m \le \Theta p_m \le p_1 + \Theta p_1$. But the converse is not true. Thus, the domain described by conditions $p_m \le \Theta p_m$ and $p_m \le 2p_1$ is a subset of that

not be greater than the original utilization. Further, by Theorem 2.4, the minimum utilization resulting from the $m - n$ remaining tasks becomes

$$(m-n)[(2\Theta)^{\frac{1}{m-n}} - 1] + 1 - \Theta \geq m[(2\Theta)^{\frac{1}{m}} - 1] + 1 - \Theta$$

because the U_l of Eq. (2.10) is non-increasing in m. **Q.E.D.**

Theorem 2.5: For a set T of $m \geq 2$ tasks and a given $\Theta > 0.5$, $U_l = m[(2\Theta)^{\frac{1}{m}} - 1] + 1 - \Theta$.

Proof: The theorem is proved by showing that the U_l obtained in Lemma 2.4 under the restriction $p_{m-1} \leq \Theta p_m \leq p_1 + \Theta p_1$ is the same as that obtained for the general case. Note that $p_{m-1} \leq \Theta p_m \leq p_1 + \Theta p_1$ if and only if $p_i \leq \Theta p_m \leq p_i + \Theta p_i$, $i = 1, 2, \cdots, m-1$ (see Fig. 2.3). Thus, for any $\{p_i\}$ we want to show the existence of $\{p_i'\}$ such that $p_i' \leq \Theta p_m \leq p_i' + \Theta p_i'$, $\forall\, i \neq m$, and the resulting utilization is not greater than the original utilization. This is done by the following three sequential steps. Steps 1 and 2 construct new periods and execution times such that $p_i \leq \Theta p_m \leq p_i + \Theta p_i$, $\forall\, i \neq m$ while reducing the processor utilization. Because of the way these new periods are constructed, the scheduling algorithm used to derive the utilization with the new periods may not be the RMS algorithm. Step 3 remedies this subtlety and completes the proof.

For any p_j that does not satisfy $p_j \leq \Theta p_m \leq p_j + \Theta p_j$, let $\Theta p_m / p_j = Q + R$, where $Q = \lfloor \Theta p_m / p_j \rfloor$ is the quotient and $0 \leq R < 1$ the remainder. Step 1 constructs p_j' (if any) such that $p_j' \leq \Theta p_m \leq 2p_j'$. Using the p_j''s, Step 2 constructs p_j'', if any, such that $p_j'' \leq \Theta p_m \leq p_j'' + \Theta p_j''$. Note that $Q = 1$ if $p_j \leq \Theta p_m < 2p_j$.

S1: The cases of $Q = 0$ and $Q \geq 2$ are dealt with in this step. For $Q = 0$ or $\Theta p_m < p_j$, construct $\{p_i'\}$ such that $p_j' = p_j/2$ and $p_i' = p_i$, $\forall\, i \neq j$. Also construct $\{e_i'\}$ such that $e_j' = 0$ and $e_i' = e_i$, $\forall\, i \neq j$, $i \neq m$ and $e_m' = e_m + e_j$ to fully utilize the processor. We

described by $p_m \leq \Theta p_m \leq p_1 + \Theta p_1$.

show that $Q' = 1$ and the utilization is reduced by using $\{p_i'\}$ and $\{e_i'\}$ as follows. By the assumption that $\Theta p_j \le \Theta p_m < p_j$ or $2\Theta\, p_j/2 \le \Theta p_m < 2\, p_j/2$, the relation $p_j' \le 2\Theta p_j' \le \Theta p_m < 2p_j'$, and thus, $Q' = 1$ must hold because $\Theta \ge 0.5$. Moreover, because $e_j/p_j \ge e_j/p_m$, the new utilization is not greater than the original utilization. Next for the case of $Q \ge 2$, construct the new periods $\{p_i'\}$ such that $p_j' = Qp_j$ and $p_i' = p_i, \forall\, i \ne j$. Also, construct the new execution times $\{e_i'\}$ such that $e_i' = e_i, \forall\, i \ne m$ and $e_m' = e_m + (Q-1)e_j$ to fully utilize the processor. For these new periods and execution times, $Q' = 1$. Again, the original utilization is reduced by these e_i's since

$$\begin{aligned} U - U' &= (\frac{e_j}{p_j} + \frac{e_m}{p_m}) - \left[\frac{e_j}{Qp_j} + \frac{e_m + (Q-1)e_j}{p_m}\right] \\ &= \frac{e_j}{p_j}(1 - \frac{1}{Q}) - \frac{(Q-1)e_j}{p_m} \\ &= (Q-1)e_j(\frac{1}{Qp_j} - \frac{1}{p_m}) \ge 0 \end{aligned}$$

because $Qp_j \le \Theta p_m \le p_m$.

In this step, p_j''s are constructed such that $Q' = 1$ or $p_j' \le \Theta p_m < 2p_j'$ for each p_j'. However, there may still exist some p_j''s such that $p_j' + \Theta p_j' < \Theta p_m < 2p_j'$, which is undesirable for Theorem 2.4. Therefore in Step 2, new periods and execution times are constructed from these p_j''s such that the utilization is reduced further.

S2: Suppose $p_j' + \Theta p_j' < \Theta p_m$, $1 \le j \le m-1$, where p_j''s are the new periods constructed in S1. Construct $\{p_i''\}$ such that $p_i'' = p_i', \forall\, i \ne j$ and $p_j'' \le \Theta p_m \le p_j'' + \Theta p_j''$. Also, construct $\{e_i''\}$ such that $e_j'' = 0$, $e_i'' = e_i', \forall\, i \ne j, i \ne m$, and $e_m'' = e_m' + 2e_j'$ to fully utilize the processor. Then

$$U' - U'' = \frac{e_j'}{p_j'} + \frac{e_m'}{p_m'} - \frac{2e_j' + e_m'}{p_m'}$$
$$= \frac{e_j'}{p_j'} - \frac{2e_j'}{p_m'} \geq 0$$

because by assumption, $\Theta p_m = \Theta p_m' \geq \Theta p_j' + p_j'$, and thus, $p_m' \geq 2p_j'$.

U'' is obtained under the RMS algorithm based on the original set of periods $\{p_i\}$. In other words, the scheduling algorithm which we used to obtain U'' in Step 2 may not be the RMS algorithm based on $\{p_i''\}$ except that $p_i \leq p_j$ implies $p_i'' \leq p_j''$. In Step 3 below, the utilization V resulting from the RMS algorithm based on $\{p_i''\}$ is shown to be less than or equal to the utilization U resulting from the RMS algorithm based on $\{p_i\}$.

S3: Let W be the utilization associated with $\{p_i\}$ and $\{e_i\}$ under the RMS algorithm based on $\{p_i''\}$, i.e., the same priority assignment as V. Then, from the discussions of Steps 1 and 2, $V \leq W$. On the other hand, $U \geq W$ because from Theorem 2.1, the RMS algorithm is optimal. Thus, $U \geq W \geq V$.

Based on the above discussions, we conclude that for any $\{p_i\}$, there always exists a new set $\{p_i''\}$ such that $p_i'' \leq \Theta p_m \leq p_i'' + \Theta p_i''$, $\forall\, i \neq m$, and the original utilization is reduced. Since $p_{m-1} \leq \Theta p_m \leq p_1 + \Theta p_1$ if and only if $p_i \leq \Theta p_m \leq p_i + \Theta p_i$, $\forall\, i \neq m$, the theorem directly follows from Lemma 2.4. Q.E.D.

2.2.2.2. Deriving U_h

Given $\Theta \leq 1$, U_h is the limit of maximum processor utilization achievable among all task periods and their associated execution times. As was done for U_l, U_h is first derived for the case of only two tasks, and then extended for an arbitrary number of tasks.

We need to consider the same two cases C1 and C2 used for deriving U_l. Further, the properties of the utilization in Eqs. (2.2) and (2.3) need to be used in the following theorem.

Theorem 2.6: For any two tasks and a given $\Theta \le 1$, $U_h = 1 - (1 - \Theta)^2$ occurs when Θp_2 is a multiple of p_1.

Proof: The theorem is proved by considering both C1 and C2. For C1, U is expressed as either Eq. (2.2) or (2.3), depending on whether $e_1 \le \xi$ or $e_1 > \xi$. Moreover, Eq. (2.3) is monotonically non-decreasing in e_1 while Eq. (2.2) could be monotonically non-decreasing or non-increasing in e_1. Therefore, given p_1 and p_2, the maximum of U must occur at either $e_1 = 0$ or $e_1 = \Theta p_1$, the two extreme points of e_1. It can be easily derived from Eq. (2.2) that if $e_1 = 0$ then $U = \Theta$, and from Eq. (2.3) that if $e_1 = \Theta p_1$ then

$$\begin{aligned} U = \frac{e_1}{p_1} + \frac{e_2}{p_2} &= \frac{p_1}{p_2}\left\lfloor\frac{\Theta p_2}{p_1}\right\rfloor + \Theta p_1\left\{\frac{1}{p_1} - \frac{1}{p_2}\left\lfloor\frac{\Theta p_2}{p_1}\right\rfloor\right\} \\ &= \Theta + (1 - \Theta)\frac{p_1}{p_2}\left\lfloor\frac{\Theta p_2}{p_1}\right\rfloor \ge \Theta. \end{aligned} \tag{2.11}$$

That is, given p_1 and p_2, the maximum U occurs at $e_1 = \Theta p_1$. To derive U_h under C1, the largest value of U in Eq. (2.11) over all possible p_1's and p_2's is determined as follows. The U in Eq. (2.11) will be maximized when $\frac{p_1}{p_2}\left\lfloor\frac{\Theta p_2}{p_1}\right\rfloor$ is maximized, i.e., $\left\lfloor\frac{\Theta p_2}{p_1}\right\rfloor = \frac{\Theta p_2}{p_1}$, or equivalently, Θp_2 is a multiple of p_1. Under C1,

$$U_h = \Theta + (1 - \Theta)\frac{p_1}{p_2}\frac{\Theta p_2}{p_1} = 1 - (1 - \Theta)^2.$$

For C2, the achievable utilization U is again expressed as Eq. (2.2), which, unlike that of C1, is a monotonically non-decreasing function of e_1. Therefore, for given p_1 and p_2, the maximum of U occurs at $e_1 = \Theta p_1$, and

$$U = \Theta + \Theta p_1 \left\{ \frac{1}{p} p_1 - \frac{1}{p_2} \left\lceil \frac{\Theta p_2}{p_1} \right\rceil \right\}$$
$$= 2\Theta - \Theta \frac{p_1}{p_2} \left\lceil \frac{\Theta p_2}{p_1} \right\rceil. \tag{2.12}$$

To derive U_h under C2, we likewise identify the largest value of U in Eq. (2.12) over all possible p_1's and p_2's as follows. The U in Eq. (2.12) will be maximized when $\frac{p_1}{p_2}\left\lceil \frac{\Theta p_2}{p_1} \right\rceil$ is minimized, i.e., $\left\lceil \frac{\Theta p_2}{p_1} \right\rceil = \frac{\Theta p_2}{p_1}$, or equivalently, Θp_2 is a multiple of p_1. Under C2,

$$U_h = 2\Theta - \Theta \frac{p_1}{p_2} \frac{\Theta p_2}{p_1}$$
$$= 1 - (1 - \Theta)^2,$$

which is the same as that under C1. **Q.E.D.**

Before deriving U_h for an arbitrary number of tasks, consider a set $\{p_i\}$ of periods such that both Θp_j and p_j, $j = 2, 3, \cdots, m-1$, are multiples of p_i, $i = 1, 2, \cdots, j-1$, and Θp_m is a multiple of p_i, $\forall\, i \neq m$ (see Fig. 2.4). Thus, let $\Theta p_j = \alpha_{j,i} p_i$ and $\Theta p_m = \alpha_{m,i} p_i$, where $\alpha_{j,i}$'s and $\alpha_{m,i}$'s are positive integers. Then, we have the following lemma.

Lemma 2.5: For $m \geq 2$ tasks with periods given above and given $\Theta \leq 1$, the maximum U occurs at

$$e_1 = \Theta p_1,$$
$$e_j = \Theta p_j - \sum_{i=1}^{j-1} \alpha_{j,i} e_i, \qquad j \geq 2. \tag{2.13}$$

Proof: Consider tasks in the order of $T_1, T_2, \cdots, T_m$. Suppose there exists T_j such that e_j does not satisfy (13) while all e_i, $1 \leq i < j$, satisfy (13). In particular, $e_j < \Theta p_j$ if $j = 1$, and $e_j < \Theta p_j - \sum_{i=1}^{j-1} \alpha_{j,i} e_i$ if $j \geq 2$. Then, construct $\{e_i'\}$ such that (i) Eq. (2.13) is now

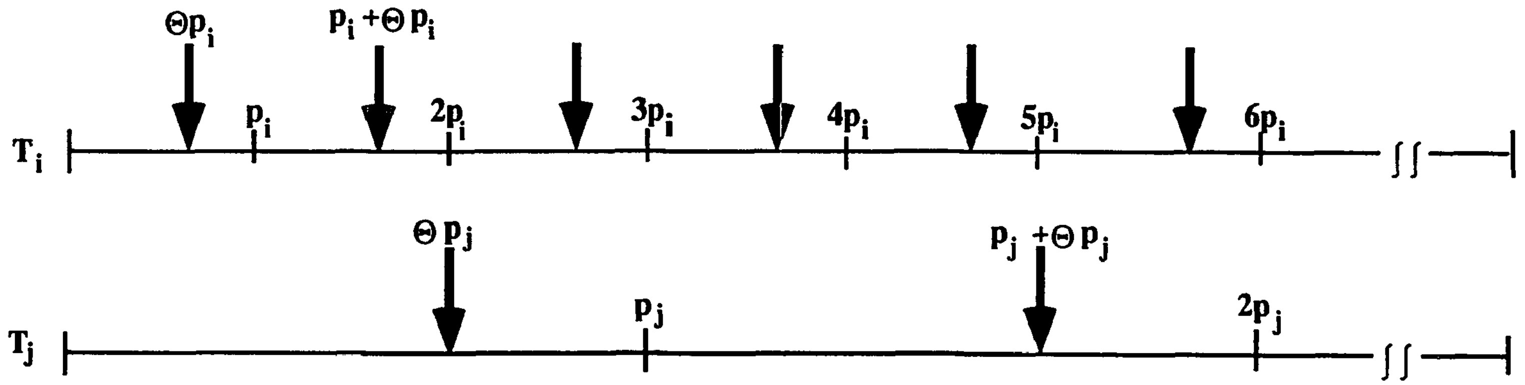


Figure 2.4. Both Θp_j and p_j are multiples of p_i

satisfied for e_i', $\forall i \leq j$, and (ii) the completion times of all invocations of other T_i's, $\forall i > j$, remain unchanged within the interval $[0, \Theta p_m)$. Specifically,

$$\begin{aligned}
e_i' &= e_i, \qquad i = 1, 2, \cdots, j-1, \\
e_j' &= e_j + \Delta_j, \\
e_{j+1}' &= e_{j+1} - \Delta_{j+1} = e_{j+1} - \alpha_{j+1\,j}\Delta_j, \\
e_i' &= e_i - \Delta_i \\
&= e_i - \left\{ \alpha_{i\,j}\Delta_j - \sum_{k=1}^{i-j-1} \alpha_{i\,j+k}\Delta_{j+k} \right\}, \qquad i = j+2, j+3, \cdots, m,
\end{aligned}$$

where $\Delta_j > 0$ is the <u>increase</u> in e_j to satisfy Eq. (2.13), and $\Delta_i \geq 0$, $i \geq j+1$, is the corresponding net <u>decrease</u>[5] in e_i to keep the completion times of all T_i's unchanged within $[0, \Theta p_m)$. A simple algebraic manipulation leads to $\Delta_i = \frac{p_i}{p_j}\Delta_j \Theta (1-\Theta)^{i-j-1}$, $\forall i \geq j+1$. It follows that

$$\begin{aligned}
U' - U &= \frac{\Delta_j}{p_j} - \sum_{i=j+1}^{m} \frac{\Delta_i}{p_i} \\
&= \frac{\Delta_j}{p_j}(1-\Theta)^{m-j} \geq 0,
\end{aligned}$$

where U and U' are the processor utilizations associated with $\{e_i\}$ and $\{e_i'\}$, respectively. If the above arguments are applied repeatly over all e_i's until Eq. (2.13) is satisfied, then one can increase the processor utilization monotonically until the maximum U is reached. **Q.E.D.**

With the above lemma, the following theorem can be easily proved.

Theorem 2.7: Let T be a set of $m \geq 2$ tasks with periods $p_1 < p_2 < \cdots < p_m$. If both Θp_j and p_j, $j = 2, 3, \cdots, m-1$, are multiples of $p_i, \forall i < j$, and Θp_m is a multiple of $p_i, \forall i < m$, then $U_h = 1 - (1-\Theta)^m$.

Proof: From the results of Lemma 2.5 and Eq. (2.13), the maximum achievable utilization U can be derived as

[5]A temporary negative e_i' will not affect the proof of the theorem.

$$U = \sum_{j=1}^{m} \frac{e_j}{p_j} = \Theta + \sum_{j=2}^{m} \Theta \left[1 - \sum_{i=1}^{j-1} \frac{e_i}{p_i} \right]$$

$$= \Theta \left[1 + (1-\Theta) + (1-\Theta)^2 + \cdots + (1-\Theta)^{m-1} \right] \qquad (2.14)$$

$$= 1 - (1-\Theta)^m . \qquad \text{Q.E.D.}$$

Let U be the utilization for a given $\Theta \le 1$. Then, it is not difficult to see that a new period p_j' of T_j can always be constructed such that both $\Theta p_j'$ and p_j' are multiples of p_i, $\forall\, i < j$, provided each of such p_i's and Θ are rational numbers. Using this observation, we have the following lemma — which is necessary to show in Theorem 2.8 that the U_h obtained in Theorem 2.7 is indeed the U_h for arbitrary m tasks.

Lemma 2.6: If p_m is modified to p_m' such that $\Theta p_m'$ is a multiple of p_i, $\forall\, i < m$, then the utilization corresponding to the new periods is greater than that to the original periods.

Proof: Consider T_m and a particular T_i, $i \neq m$, while ignoring all the other tasks. From Theorem 2.6, for any $e_i \le \Theta p_i$, the maximum utilization occurs when Θp_m is a multiple of p_i. Thus, the function

$$f_i(\Theta p_m) \equiv \frac{\Theta p_m - E_i(\Theta p_m)}{p_m} \le \Theta\left(1 - \frac{e_i}{p_i}\right), \qquad (2.15)$$

where $E_i(\Theta p_m)$ represents the total execution time of T_i within $[0, \Theta p_m)$ and the RHS of the above inequality is the part of utilization contributed by T_m as Θp_m is a multiple of p_i. Considering each (T_i, T_m) pair will lead to:

$$\sum_{i=1}^{m-1} f_i(\Theta p_m) = \sum_{i=1}^{m-1} \frac{\Theta p_m - E_i(\Theta p_m)}{p_m}$$

$$= (m-2)\Theta + \frac{\Theta p_m - \sum_{i=1}^{m-1} E_i(\Theta p_m)}{p_m}$$

$$\leq (m-2)\Theta + \Theta(1 - \sum_{i=1}^{m-1} \frac{e_i}{p_i}).$$

It follows that

$$\frac{\Theta p_m - \sum_{i=1}^{m-1} E_i(\Theta p_m)}{p_m} \leq \Theta(1 - \sum_{i=1}^{m-1} \frac{e_i}{p_i}). \tag{2.16}$$

Notice that the LHS of inequality (16) represents the part of U contributed by T_m with the original period p_m, whereas the RHS denotes that contributed by T_m as $\Theta p_m'$ is a multiple of p_i, $\forall\ i < m$. Since the part of utilization contributed by other tasks are the same, the lemma follows. **Q.E.D.**

Now, for a general $\{p_i\}$, we want to show the existence of a new set of periods which satisfies the property in Theorem 2.7 and increases processor utilization.

Theorem 2.8: For a set T of $m \geq 2$ tasks and a given $\Theta \leq 1$, $U_h = 1 - (1 - \Theta)^m$.

Proof: Consider tasks in order of $T_m, T_{m-1}, \cdots$, and T_1. Without loss of generality, Θp_m can be assumed to be a multiple of p_i, $\forall\ i < m$. (If not, from Lemma 2.6 a new p_m' can always be constructed such that $\Theta p_m'$ is a multiple of all other p_i's while increasing the processor utilization.)

Let $T_j \in T$ be the first task where both Θp_j and p_j, $2 \leq j \leq m-1$, are not multiples of p_i, $\forall\ i < j$. Then, construct a new p_j' such that both $\Theta p_j'$ and p_j' are multiples of p_i, $\forall\ i < j$. Assume that $p_j' = \sigma p_j$, where σ is a positive real number and p_j the original period of T_j. For all the other tasks,

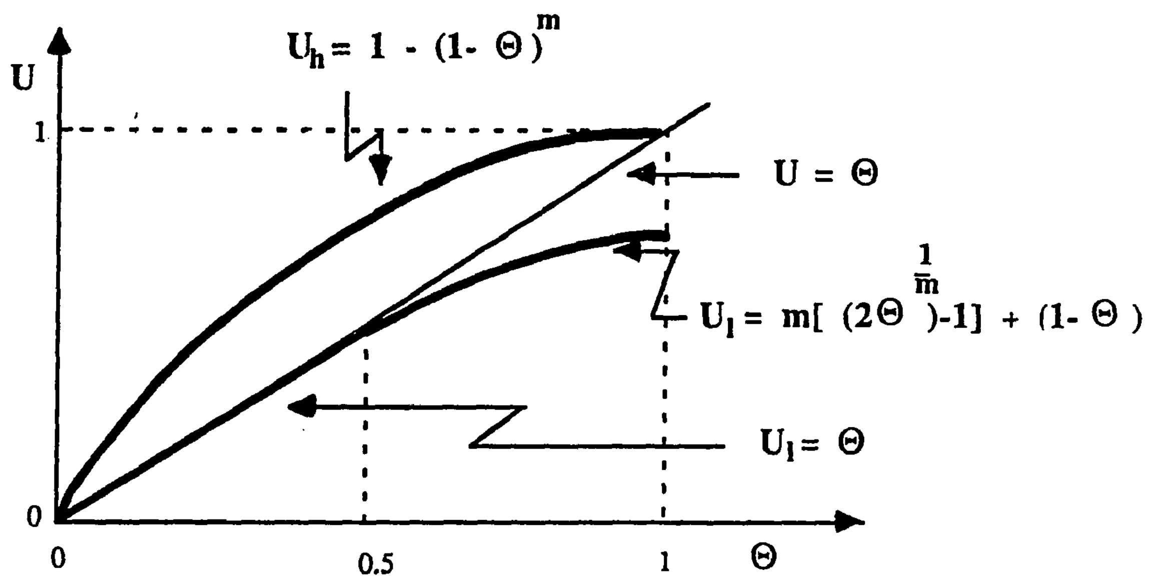
$$p_i' := \begin{cases} p_i, & i = 1, 2, \cdots, j-1, \\ \sigma p_i, & i = j+1, \cdots, m. \end{cases}$$

It is worth pointing out that after the above modification, $\Theta p_m'$, $\Theta p_i'$ and p_i', $i = j+1, \cdots, m-1$, are still multiples of p_k', $k = 1, 2, \cdots, j-1$.

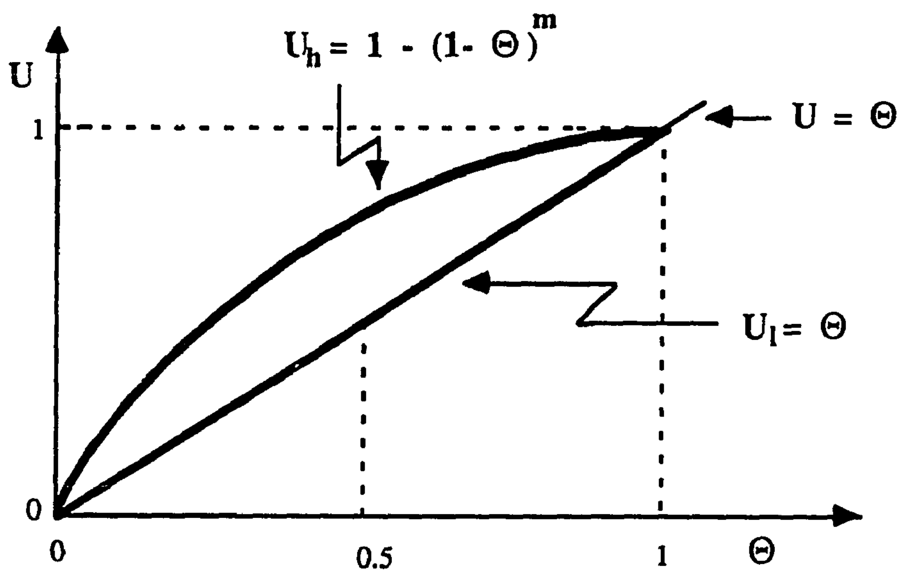
On the other hand, each e_i' is constructed as follows. First, $e_i' := e_i$, $\forall\, i < j$, to keep the part of utilization contributed by such T_i's unchanged. Second, e_j' is constructed such that e_j' and e_i', $i = 1, 2, \cdots, j-1$, fully utilize the processor. From Lemma 2.6, $U_j' = \Theta - \Theta\sum_{i=1}^{j-1} U_i' \geq U_j$, where U_j' is the new utilization contributed by T_j and U_j the old utilization by T_j. Finally, construct e_i', $i = j+1, j+2, \cdots, m$, such that e_i' and p_k', $k = 1, 2, \cdots, i-1$, fully utilize the processor. Likewise, $U_i' = \Theta - \Theta\sum_{k=1}^{i-1} U_k'$. Let $\Delta = U_j' - U_j \geq 0$, then a simple algebraic manipulation leads to $U' - U = \Delta(1-\Theta)^{m-j} \geq 0$, where $U \equiv \sum_{i=1}^{m} U_i$ and $U' \equiv \sum_{i=1}^{m} U_i'$.

The above procedure can be repeatedly applied to p_{j-1}, p_{j-2} and so on, until each Θp_j and p_j, $2 \leq j \leq m-1$, are multiples of p_i, $\forall\, i < j$, so as to increase the utilization monotonically. Thus, this theorem follows from Theorem 2.7. **Q.E.D.**

In Theorem 2.8, each of $\{p_i\}$ and Θ must be rational numbers such that both $\Theta p_j'$ and p_j' can be multiples of p_i, $i = 1, 2, \cdots, j-1$. Otherwise, p_j' must be constructed to make $\Theta p_j'$ and p_j' as close to multiples of p_i's as possible. This is the reason why U_h is termed "the limit of the maximum" of U. U_l and U_h are shown in Fig. 2.5(a).



2.5(a). Static Case



2.5(b). Dynamic Case

Figure 2.5. U_l and U_h as Functions of Θ

2.3. Optimal Dynamic Scheduling of Periodic Tasks

For the static scheduling algorithms discussed in the last section, priorities assigned to all invocations of a task are the same. For the dynamic algorithms to be addressed in this section however, different priorities are assigned to different invocations of the same task. An optimal dynamic algorithm can naturally outperform its static counterpart because it requires less stringent constraints in scheduling tasks.

2.3.1. Optimal Dynamic Scheduling Algorithm

To develop an optimal dynamic scheduling algorithm w.r.t. Θ, it is necessary to introduce an optimal single-machine scheduling algorithm developed in [BLL83]. Since this algorithm will be used in the following chapters, we shall call it *Algorithm A* in the rest of this thesis. A set of jobs with arbitrary release times and precedence constraints is to be scheduled on a single machine so as to minimize the maximum job completion cost, where the cost associated with each job can be any monotone non-decreasing function of its completion time. As can be seen from the following steps, the computational complexity of the algorithm is $O(N^2)$, where N is the number of jobs to be scheduled.

SA1. Modify job release times, where possible, to meet the precedence constraints among the jobs, and then arrange the jobs in non-decreasing order of their modified release times to create a set of disjoint blocks of jobs. For example, suppose jobs X, Y and Z are released at $t = 0, 2, 15$, respectively, X precedes Y which precedes Z, and 5 units of time are required to complete each of X and Y. Then, the job Y's release time is modified to $t = 5$, Z's release time remains unchanged at $t = 15$, and two blocks of jobs {X,Y} and {Z} will be created.

SA2. Consider a block B with block completion time $t(B)$. Let B' be the set of jobs in B which do not precede any other jobs in B. Select a job l from B' such that $f_l(t(B))$ is the minimum, where $f_l(t)$ is the nondecreasing cost function of job l if it is completed at t. This implies that l be the last job to be completed in B.

SA3. Create subblocks of jobs in the set $B - \{l\}$ by arranging the jobs in nondecreasing order of modified release times as in SA1. The time interval(s) allotted to l is (are) then the difference between the interval of B and the interval(s) allotted to these subblocks.

SA4. For each subblock, repeat SA2 and SA3 until time slot(s) is (are) allotted to every job.

By setting the cost function $f_{i\nu}(t)$ of $T_{i\nu}$, the ν-th invocation of T_i, as $f_{i\nu}(t) := (t - r_{i\nu})/p_i$, the above algorithm can be simplified to schedule independent periodic tasks while minimizing the system hazard as follows. In step SA1, the modification of job release times is unnecessary since the periodic tasks are assumed to be independent. Because $T_{i\nu}$ must be completed before $T_{i(\nu+1)}$, only one invocation from each of the m tasks needs to be considered in Step SA2 in order to determine the last invocation to be completed in each block. Therefore, the computational complexity can be reduced to $O(mn)$, where n is the total number of invocations for the m tasks in T within a planning cycle.

For example, consider two tasks, T_1 and T_2, with $p_1 = 10$, $e_1 = 3$, $p_2 = 30$ and $e_2 = 8$. That is, a total of four invocations, T_{11}, T_{12}, T_{13} and T_{21}, need to be scheduled within the planning cycle $I = [0, 30)$. As the above algorithm is applied to this example, two blocks $B_1 = \{T_{11}, T_{21}, T_{12}\}$ in the interval [0, 14], and $B_2 = \{T_{13}\}$ in [20, 23] are created from Step SA1. From SA2, B_2 is trivially allotted to T_{13}. For B_1 however, since T_{11} must be completed before T_{12}, only T_{12} and T_{21} need to be considered as to which of them should be completed last in B_1. From SA2, it turns out that $f_{12}(14) = (14 - 10) / 10 = 0.4 < 14/30 = f_{21}(14)$. Therefore, T_{12} must be completed last in B_1. Deleting T_{12} from B_1 and arranging T_{11} and T_{21} as described in SA3, a subblock $B_{11} = \{T_{11}, T_{21}\}$ in [0, 11] is created, meaning that the slot [11, 14] is allotted to T_{12}. Repeating the above steps on B_{11}, it follows that T_{11} must be completed before T_{21}, and the resulting system hazard $\Theta = 0.4$, which is the normalized flowtime of T_{12}. Notice that if the EDD algorithm [Bak74] — which simply schedules the "ready" invocations in the non-decreasing order of their deadlines — is used in the example, then the resulting Θ becomes 14/30 (the normalized flowtime of T_{21}). Therefore, the EDD scheduling algorithm is <u>not</u> an optimal algorithm in the sense of minimizing Θ.

2.3.2. Achievable Processor Utilization Bounds

As in the static case, we are also interested in deriving the two utilization bounds, U_l and U_h. Recall that both U_l and U_h must be derived with the processor fully utilized under the above optimal dynamic scheduling algorithm. However, it is not always easy to derive U_l and U_h based on this optimal dynamic scheduling algorithm. The following theorem remedies this difficulty by showing that U_l and U_h can also be derived under the EDD scheduling algorithm after properly modifying the deadlines of the task invocations to be scheduled.

Theorem 2.9: Given $\Theta \leq 1$, U_l and U_h derived under the optimal dynamic scheduling algorithm are the same as those derived under the EDD scheduling algorithm with $(v-1)p_i + \Theta p_i$ as T_{iv}'s deadline, $\forall\, i$ and v.

Proof: Given Θ, let U_l' and U_h' denote the two bounds obtainable under the above EDD algorithm. Then, we want to show that $U_l = U_l'$ and $U_h = U_h'$. Let $\{p_i\}$ and $\{e_i\}$ be the sets of periods and execution times where $U = U_l$ is obtained under the optimal dynamic scheduling algorithm. That is, given $\Theta \leq 1$, each T_{iv} can be completed by the time $(v-1)p_i + \Theta p_i$, $\forall\, i$ and v. However, by using $(v-1)p_i + \Theta p_i$ as the deadline of T_{iv}, the EDD algorithm must also be able to generate a schedule under which T_{iv} can be completed before $(v-1)p_i + \Theta p_i$, meaning that $U_l' \geq U_l$. By switching the roles of U_l' and U_l in the above argument, we can show that $U_l' \leq U_l$. Thus, $U_l = U_l'$. Similarly, $U_h = U_h'$. Q.E.D.

By using Theorem 2.9, U_l and U_h are derived in the next two subsections.

2.3.2.1. Deriving U_l

The following lemma follows directly from [LiL73].

Lemma 2.7: Let T be a set of m tasks with periods $\{p_i\}$ and execution times $\{e_i\}$. Then, using vp_i as T_{iv}'s deadline, the EDD scheduling algorithm is feasible w.r.t. the deadline if and only if the processor utilization $U \equiv \sum_{i=1}^{m} \frac{e_i}{p_i} \leq 1$.

Lemma 2.8: Given $\Theta \leq 1$, the EDD scheduling algorithm with $(v-1)p_i + \Theta p_i$ as T_{iv}'s deadline is feasible w.r.t. Θ if $U = \sum_{i=1}^{m} \frac{e_i}{p_i} \leq \Theta$.

Proof: For any set T of m tasks with $U = \sum_{i=1}^{m} e_i/p_i \leq \Theta$, we want to prove that there always exists at least a schedule in which each T_{iv} can be completed by its deadline, $(v-1)p_i + \Theta p_i$. Consider a modified task set T' of T where T' is the same as T except that the execution time e_i' in T' is e_i/Θ instead of the original e_i in T. Since $U' = U/\Theta \leq 1$, where U' is the processor utilization for T', there always exists a schedule S' in which each T_{iv} of T' can be completed before vp_i. However, since $e_i = \Theta e_i'$, there must exist a schedule in which each T_{iv} of T can be completed by its deadline $(v-1)p_i + \Theta p_i$. Since the EDD scheduling algorithm is optimal in meeting deadlines, the theorem follows. **Q.E.D.**

U_l can now be derived as in the following theorem.

Theorem 2.10: For a set T of $m \geq 1$ tasks and a given $\Theta \leq 1$, $U_l = \Theta$.

Proof: The theorem can be proved from the following three facts. First, from Lemma 2.8, a feasible schedule for the given Θ can always be derived from the EDD algorithm as long as the processor utilization does not exceed Θ. Second, consider a particular set of execution times such that $e_i = 0, \forall\, i \neq m$; for this particular instance of $\{e_i\}$, no feasible schedule with a system hazard not exceeding Θ may exist if $e_m > \Theta p_m$. That is, given $\Theta \le 1$, there exists an instance of T where no feasible schedule with a system hazard not exceeding Θ may exist should the processor utilization exceed Θ. Based on the above two facts, $\Theta = U_l'$, where U_l' is the U_l under the EDD scheduling algorithm with $(v-1)p_i + \Theta p_i$ as T_{iv}'s deadline. Finally, from Theorem 2.9, U_l — which must be obtained under the optimal dynamic scheduling algorithm — is the same as U_l'. **Q.E.D.**

Notice that for $\Theta \le 0.5$, the same U_l is obtained under either the optimal dynamic or optimal static scheduling algorithm. If $0.5 < \Theta \le 1$ however, the U_l associated with the optimal dynamic algorithm is greater than that associated with the optimal static algorithm.

2.3.2.2. Deriving U_h

Consider a task $T_i \in T$. Since T_{iv} arrives at time $(v-1)p_i$ and must be completed by its deadline $(v-1)p_i + \Theta p_i$, T_i can only be executed during the set of intervals $E_i \equiv \bigcup_{v=1}^{L/p_i} E_{iv}$, where $E_{iv} \equiv [(v-1)p_i, (v-1)p_i + \Theta p_i)$. $E \equiv \bigcup_{i=1}^{m} E_i$ is the *executable time zone* (ETZ) of T, which represents the set of time intervals within $[0, L)$ during which tasks of T can be executed. Let $|E|$ represent the total length of E. Then the *executable time ratio* (*ETR*) of T is defined as $|E|/L$. Several insights can be drawn from the notion of *ETR* as follows. First, *ETR* ≤ 1. Second, if $U >$ *ETR*, then no feasible schedule exists for the given Θ. Finally, it was also shown in Theorem 2.8 that for any $\Theta \le 1$, the maximum *ETR* occurs when both Θp_j and p_j, $j = 2, 3, \cdots, m-1$, are multiples of p_i, $\forall\, i < j$, and Θp_m is a multiple of p_i, $\forall\, i < m$. Or, the supremum of *ETR* among all instances of T with m tasks is $1 - (1 - \Theta)^m$.

U_h can now be obtained as in the following theorem.

Theorem 2.11: For a set T of $m \ge 1$ tasks and a given $\Theta \le 1$, $U_h = 1 - (1 - \Theta)^m$.

Proof: From the definition of U_h, we need to show that (i) if $U > 1-(1-\Theta)^m$, then T is never schedulable w.r.t. Θ, and (ii) there exists at least a feasible instance of T if $U < 1-(1-\Theta)^m$. The claim in (i) is obvious because $1-(1-\Theta)^m$ is the supremum of *ETR* among all instances of T. Thus, no feasible instance of T may exist if $U > 1-(1-\Theta)^m$. On the other hand, the claim in (ii) has already been shown to be true in Theorem 2.8. In particular, given $U < 1-(1-\Theta)^m$, the feasible instance of T is the same as that for the static scheduling algorithm as derived in Theorem 2.8. **Q.E.D.**

Notice that U_h for the optimal dynamic scheduling algorithm happens to be the same as that for its static counterpart. U_l and U_h are shown in Fig. 2.5(b).

2.4. On-Line Scheduling Algorithms

So far, we have derived optimal scheduling algorithms under the assumptions that the arrival times, execution times and deadlines of all tasks (invocations) are known in advance. Both the optimal static algorithm derived in Section 2.2 and the optimal dynamic algorithm in Section 2.3 are thus *off-line* algorithms. When the above information on each task is not known until it actually arrives, an *on-line* algorithm must be found.

An on-line scheduling algorithm, if any, is said to be *optimal* if it always generates a schedule which is as good as that generated by its off-line dynamic counterpart. Notice that the existence of an optimal on-line algorithm depends not only on the scheduling objective but also on the characteristics of the randomly arriving tasks. For example, in case of a single processor, it has been shown in [Der74, Mok83] that the on-line EDD scheduling algorithm is always optimal in meeting task deadlines.

In this section, we consider a system in which, in addition to periodic tasks, aperiodic tasks arrive randomly. While the information of all periodic tasks is assumed known *a priori* to the scheduler, the information of each aperiodic task is unknown until its actual arrival. We shall prove that, except for some extremely simple cases, no optimal on-line scheduling algorithms w.r.t. Θ in such a system exist. Thus, instead of deriving the two processor utilization bounds U_l and U_h, we shall describe simple on-line mechanisms which are useful to determine whether or not an arriving aperiodic task can be completed with a system hazard not exceeding Θ without disturbing the execution of periodic tasks. These mechanisms can be used to implement load-sharing strategies [WaM85, ShC88] where tasks are transferred from over-loaded processing nodes (PNs) to under-loaded PNs. A PN may use the mechanisms proposed here to decide on whether to execute an arriving task locally or transfer it to another PN for execution.

2.4.1. Optimal On-Line Scheduling Algorithms

The following theorem shows that an optimal on-line scheduling algorithm w.r.t. the system hazard does not always exist.

Theorem 2.12: No optimal on-line scheduling algorithms w.r.t. Θ exist in a system where aperiodic tasks arrive randomly with general execution times and deadlines.

Proof: First, considering a system in which there exists no periodic tasks, we shall prove the theorem by contradiction. Suppose there is an optimal on-line scheduler and consider the following scenario. At time t, there exist two active tasks T_1 and T_2. T_1 arrived at the time $t-20$ with deadline $t+20$ and remaining execution time 3, while T_2 arrived at t with deadline $t+10$ and execution time 3. Starting from t, the optimal scheduler will schedule these two tasks on a single processor. Assume no tasks arrive in $[t, t+3)$ and consider the following two cases depending on which of T_1 and T_2 is executed in $[t, t+3)$.

Case 1. T_1 is executed in $[t, t+3)$. In this case, T_1 is completed at $t+3$ and the remaining execution time of T_2 is 3. Now, consider the scenario where T_3 arrives at $t+3$ with execution time 4 and deadline $t+8$, and T_3 is the only task that will ever arrive at, or after, $t+3$. Then, $\Theta = \max\{0.575, 1.0, 0.8\} = 1.0$, which is achieved by executing T_3 in $[t+3, t+7)$ and T_2 in $[t+7, t+10)$. However, if we have had at t the knowledge of task arrivals at, or after, $t+3$, the minimum system hazard would become $\Theta = \max\{0.75, 0.3, 0.8\} = 0.8$, which is achieved by executing these three tasks in order of T_2, T_3 and T_1.

Case 2. T_2 is executed in $[t, t+3)$. In this case, T_2 is completed at $t+3$ and the remaining execution time of T_1 is 3. Now, consider the scenario where no task will ever arrive at, or after, $t+3$. Then, the minimum system hazard is $\Theta = \max\{0.65, 0.3\} = 0.65$. However, if at t we have known that no task will ever arrive at, or after, $t+3$, the minimum system hazard would become $\Theta = \max\{0.575, 0.6\} = 0.6$, which is achieved by executing these two tasks in the order of T_1 and T_2.

From the above discussion, we conclude that the assumed optimal on-line scheduler cannot always generate a schedule as good as that generated by an optimal off-line scheduler. The theorem follows since a system with only aperiodic tasks is a special case of the general system in which both periodic and aperiodic tasks exist. **Q.E.D.**

Even though an optimal on-line scheduling algorithm does not always exist, there are special cases where such an algorithm exists. For example, in a system where each (periodic or aperiodic) task has the same deadline since its arrival, the on-line EDD scheduler described above is optimal.

2.4.2. On-Line Determination of the Schedulability for an Arriving Aperiodic Task

Scheduling aperiodic tasks in the presence of periodic ones is important in real-time applications because of the unpredictable and time-critical nature of aperiodic tasks. A solution approach which is most commonly seen in the open literature [SLR86, LSS87, SSL89] suggests the following. First, some minimal interarrival time is assumed to exist for all aperiodic tasks. Second, treat the aperiodic task stream as a periodic task with a period of the minimal interarrival time. Third, use the RMS algorithm or variations thereof to determine whether or not the resulting schedule is feasible with $\Theta = 1$. Despite its simplicity, this approach has the following drawbacks:

D1. Since there may exist different types of aperiodic tasks, it may not be realistic to assume a minimal interarrival time between an aperiodic task of one type and that of another.

D2. As pointed out and analyzed in [GDB89], a system with very low processor utilization might result from the conservative nature of this approach.

In what follows, simple on-line mechanisms which are more realistic and could result in better processor utilization are described. They are presented only under the following three different scheduling algorithms for the tasks involved: A1) static scheduling of periodic tasks with the lowest priority given to aperiodic tasks, A2) static scheduling of periodic tasks with the highest priority given to aperiodic tasks, and A3) dynamic scheduling of all tasks. For all three algorithms, consider the current time instant t when an aperiodic task with execution time e and deadline $t+d$ arrives. Also, assume that all tasks residing in the processor (the periodic tasks and the aperiodic ones already accepted) before t are guaranteed to be completed in time with a system hazard not exceeding Θ.

For A1, the RMS algorithm is used for periodic tasks, and aperiodic tasks are assigned the lowest priority (the FCFS rule is used among aperiodic tasks). Let $R(t) \leq t$ denote the cumulated processor idle time in $[0, t]$ if there are only periodic tasks in the processor. Also, let $CET(t)$ be the total cumulated remaining execution time of all aperiodic tasks accepted by time t. Then, the processor may accept the arriving aperiodic task only if $R(t + \Theta d) - R(t) \geq e + CET(t)$.

A2 is similar to A1 except that the highest priority is assigned to the aperiodic tasks. The arriving aperiodic task would be accepted only if (i) it can be completed by $t + \Theta d$ and (ii) the resulting normalized flowtime is not greater than Θ for each unfinished periodic task invocation within the planning cycle. Specifically, the planning cycle *involved* is the current planning cycle if at time t there exists at least an unfinished invocation within the cycle. Then, the processor may accept the arriving task only if $\Theta d \geq CET(t) + e$ and $\max\limits_{T_{iv}}(c_{iv} - r_{iv}) / p_i \leq \Theta$, where c_{iv}, r_{iv} and p_i are the completion time, arrival times and period, respectively, of T_{iv} for all unfinished T_{iv}'s within the planning cycle.

A3 uses the on-line EDD algorithm for both periodic and aperiodic tasks. Upon arrival of an aperiodic task at time t, A3 uses the on-line EDD algorithm to check if all tasks can be completed with a system hazard not exceeding the pre-specified Θ. If they can, the arriving task is accepted. In addition to the unfinished aperiodic tasks already accepted, the tasks involved in this check includes the arriving aperiodic task and all unfinished periodic tasks.

Notice that whether to accept an arriving task w.r.t. Θ is equivalent to treating $t + \Theta d$ as the deadline of the arriving task. Depending on the specific need of the system, a different deadline such as $t + d$ may also be used in the above algorithms for an arriving task.

CHAPTER 3

MULTI-PROJECT SCHEDULING USING AN ENUMERATIVE METHOD

3.1. Introduction

In this chapter, the optimal scheduling of communicating periodic tasks which have been assigned among the PNs in a distributed system is studied in the context of Operations Research. Therefore, instead of scheduling communicating tasks on processors, cooperative projects (or jobs) are scheduled on machines.

Consider a set, $J = \{J_i \mid i = 1, 2, \cdots, |J|;\ |J| \geq 2\}$, of cooperative projects to be executed by a set, $M = \{M_k \mid k = 1, 2, \cdots, |M|;\ |M| \geq 2\}$, of machines, where $|A|$ is the cardinality of the set A. A set of cooperative projects will henceforth be called a *multi-project*. Each project J_i with release time r_i is composed of an arbitrary number of operations. Similarly to a job-shop [Bak74], each operation in the multi-project is to be executed by a pre-specified machine. Precedence constraints in a multi-project system may exist between the operations *within* a single project and *between* those of different projects. A *time delay* may also exist between the completion of an operation and its enabling of another operation on a different machine. The execution time of an operation by its corresponding machine and the period of a time delay are assumed to be given.

A PERT/CPM graph with Activity On Arc (AOA) ([Tah76]), called the *multi-project graph* (MPG), is used to describe the precedence constraints between operations and time delays, and the release and completion of each project. Unlike a general PERT/CPM graph —

which usually has only a single pair of starting and ending nodes — the MPG has multiple pairs of starting and ending nodes, each representing the release and the completion events of a project. The time a starting (ending) node is therefore "realized" is the time when the corresponding project is released (completed). Note that the completion of a project J_i means the completion of all operations (which may belong to other projects) and time delays which precede J_i's ending node in the MPG. Thus, depending on how the MPG is constructed to reflect the completion of each J_i, the set of these preceding operations may not contain all J_i's constituent operations. This means that, as in any job-shop, even the last operation(s) necessary to complete J_i may not be executed by the same machine that executes most of J_i's operations. Therefore, when solving the multi-project scheduling problem, what is more important is which machine executes an operation and which ending node the operation precedes in the MPG, rather than which project the operation belongs to. Following the convention in any PERT/CPM graph with AOA, each operation or time delay of the MPG will henceforth be called an *activity*.

Fig. 3.1 shows an example MPG consisting of four projects J_1, J_2, J_3 and J_4, which are to be executed by two machines, M_1 and M_2. Each arc in the MPG represents an activity and the weight assigned to an arc is the execution time of the corresponding operation or the delay. All operations on the LHS of the shaded area in Fig. 3.1 are to be executed by M_1 while those on the RHS by M_2. In this MPG, there are four starting nodes n_1, n_2, n_3 and n_{11} (labeled as S_1, S_2, S_3, and S_4, respectively), and four ending nodes n_{17}, n_{18}, n_9 and n_{19}, which represent the release and completion events of J_1, J_2, J_3 and J_4, respectively. Notice that for the multi-project scheduling we make no distinction on which project an activity belongs to. A project is considered completed only after all activities preceding its ending node are completed. Except for J_4, which is released at time $t = 20$, J_1, J_2 and J_3 are all released at $t = 0$, meaning that n_1, n_2, n_3 and n_{11} are "realized" at $t = 0, 0, 0$, and 20,

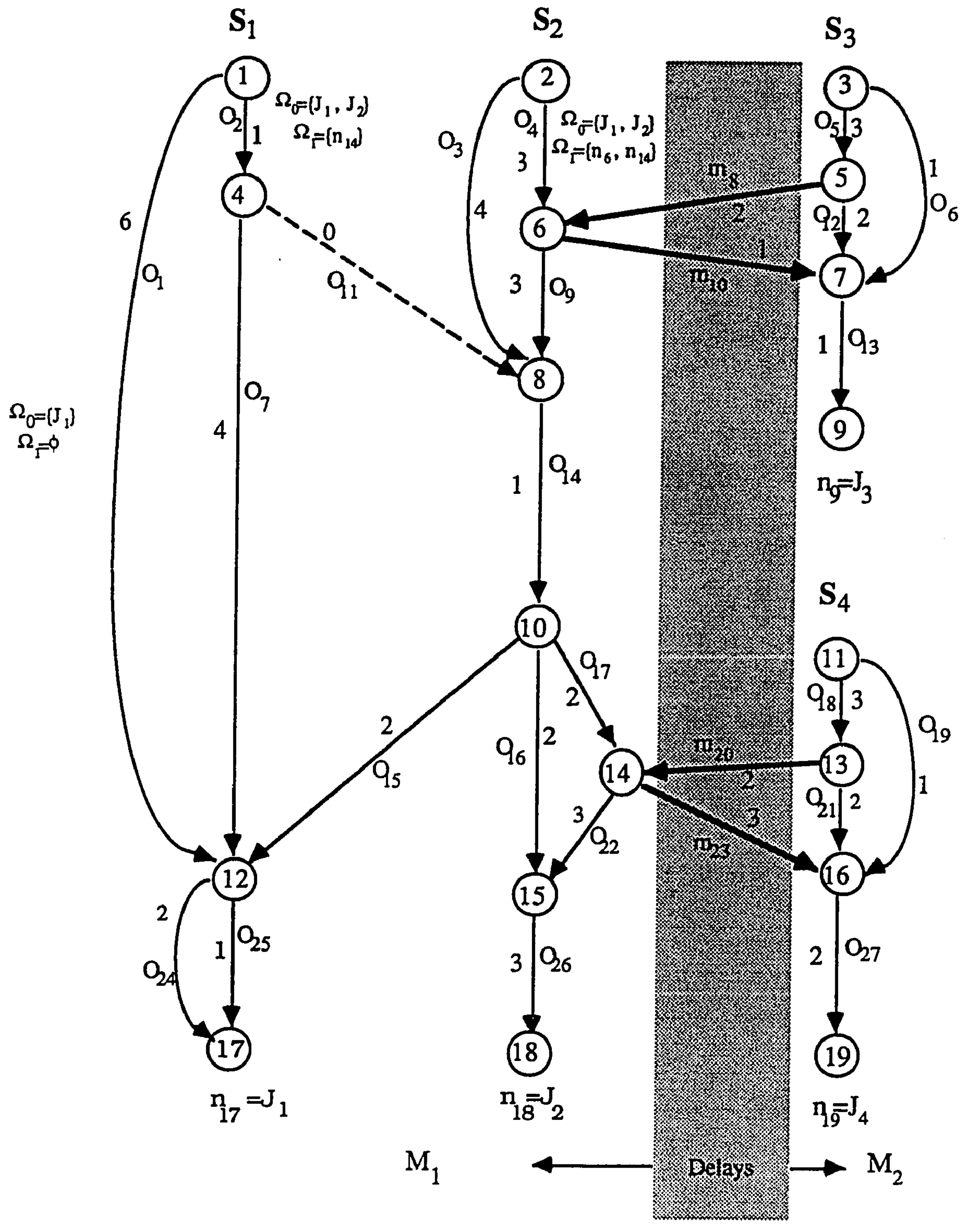


Figure 3.1. An Example MPG

respectively. For convenience, we write $n_p = J_i$ if n_p is the ending node representing J_i's completion. Also, the completions of J_1, J_2, J_3 and J_4 represent the completions of all activities preceding n_{17}, n_{18}, n_9 and n_{19}, respectively.

Given a multi-project, let c_i^ζ be the time when J_i is completed under a scheduling algorithm ζ. Then, the system hazard of the multi-project is computed as

$$\Theta^\zeta \triangleq \max_{J_i \in J} \bar{c}_i^\zeta,$$

where $\bar{c}_i^\zeta = (c_i^\zeta - r_i) / w_i$ is the normalized flowtime, r_i the release time, and $w_i > 0$ the normalization factor of J_i. We want to find an optimal multi-project preemptive (resume) scheduling (Time delays are not schedulable objects.) algorithm ζ^* such that $\Theta^{\zeta^*} = \min_{\zeta} \Theta^\zeta$. That is, ζ^* minimizes the maximum normalized project flowtime over all projects. Notice that Θ^ζ is a regular[1] performance measure. Also, the general job-shop scheduling problem is a special case ([BEN82]) of the multi-project scheduling problem (MPSP) described above.

For example, the two schedules generated by ζ_1 and ζ_2 for the multi-project in Fig. 3.1 are given in Fig. 3.2, where $w_1 = 30$, $w_2 = 40$, and $w_3 = w_4 = 20$. ζ_1 schedules operations on each machine simply according to ascending order of their indices, whereas ζ_2 considers the normalization factor for each project as well as the precedence constraints affecting operations on other machines. As shown in Fig. 3.2, we get

$$\bar{c}_1^{\zeta_1} = \frac{34}{30},\quad \bar{c}_2^{\zeta_1} = \frac{37}{40},\quad \bar{c}_3^{\zeta_1} = \frac{16}{20},\quad \bar{c}_4^{\zeta_1} = \frac{33-20}{20} = \frac{13}{20},\quad \text{and}$$
$$\bar{c}_1^{\zeta_2} = \frac{29}{30},\quad \bar{c}_2^{\zeta_2} = \frac{37}{40},\quad \bar{c}_3^{\zeta_2} = \frac{7}{20},\quad \bar{c}_4^{\zeta_2} = \frac{30-20}{20} = \frac{10}{20}.$$

Thus,

[1]A performance measure Z is said to be *regular* if (a) the scheduling objective is to minimize Z, and (b) Z increases only if one or more project completion times in the schedule increases ([Bak74]).

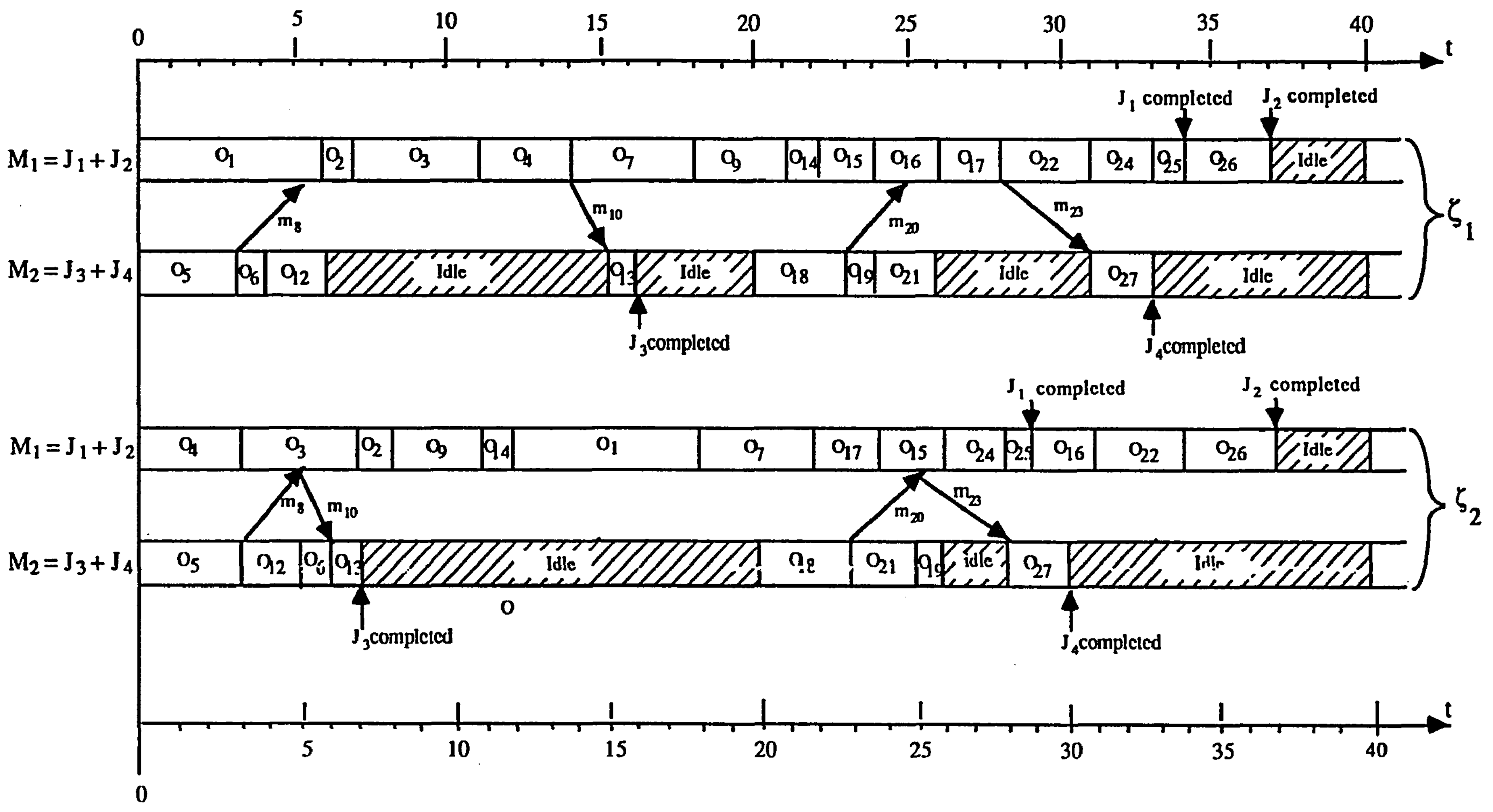


Figure 3.2. Two Schedules of the MPG in Figure 3.1.

$$\Theta^{\zeta_1} = \max\left\{\frac{34}{30}, \frac{37}{40}, \frac{16}{20}, \frac{13}{20}\right\} = \frac{34}{30} > \frac{29}{30} = \max\left\{\frac{29}{30}, \frac{37}{40}, \frac{7}{20}, \frac{10}{20}\right\} = \Theta^{\zeta_2},$$

showing the ζ_2's superiority over ζ_1.

The MPSP described above is a direct representation of the scheduling problem for periodic tasks assigned among the PNs of the distributed system. For example, a project of the former corresponds to a task invocation of the latter. Also, a time delay of the MPSP is used to represent the communication delay between two tasks assigned to different PNs. (A complete account of the correspondence of terms between the two contexts is detailed in Chapter 4.)

Even without considering the time delays, the MPSP is hard, irrespective of whether or not preemptions are allowed. For example, the general job-shop scheduling problem, a special case of the MPSP, is already NP-hard for the problems even as simple as $J_2 \mid m_j \geq 3 \mid C_{\max}$[2] or $J_3 \mid m_j \geq 2 \mid C_{\max}$ ([GoS78, LRB77]). Consequently, except in the case of a single machine ([BLL83]), no polynomial time algorithms are likely to exist for the MPSP; some form of heuristic and/or enumeration is the only recourse to the problem.

The MPSP considered here falls in the realm of *resource-constrained project scheduling* problems ([BLR83]), and thus, its solution approaches must be related to that of the latter as well as to job-shop scheduling problems. Like any other NP-hard problem, approaches to these problems are concerned with how to improve the efficiency of the search for an optimal or suboptimal solution by: 1) using dominance properties (DPs) to reduce the size of the state space to be searched, and 2) deriving a lower-bound cost as tight as possible at each stage to guide the search more efficiently. Giffler and Thompson ([GiT60]) are the first to propose a systematic approach to generating the set of all *active schedules* based on which (and

[2]a two-machine job-shop in which the number of operations in any job is greater than or equal to 3 and the scheduling objective is to minimize the maximum job completion time among all jobs ([LLR81]).

variations thereof) most implicit enumerative algorithms have been developed. A set A of active schedules is said to *dominate* another set $S \supseteq A$ of all schedules in the sense that inclusion of an optimal schedule (w.r.t. any regular measure) in S implies that in A. In other words, to find optimal schedules w.r.t. any regular measure, it suffices to consider only the set of active schedules, thus reducing the size of the state space to be searched. For example, to solve a job-shop scheduling problem, Brooks and White ([BrW65]) used two bounds, called the *job bound* and the *machine bound*, to guide the search for an optimal schedule among the set of all active schedules. Schrage ([Sch70, Sch72]) developed a similar approach to generating the set of all active schedules for preemptive and non-preemptive cases, and proposed DPs and lower-bound costs to find optimal schedules for a network scheduling problem. An extensive survey of job-shop scheduling with branch-and-bound (B&B) methods can be found in [LLR77], where job-shop scheduling was modeled by settling pairs of disjunctive arcs and a tighter bound of cost was also developed by including many other bounds as special cases. Possible extensions of the problems and variations of the solution techniques are described in [BEN82]. Unfortunately, none of these approaches are directly applicable to the MPSP due mainly to their structural differences. Further, while most known scheduling objectives are to minimize the makespan, the MPSP deals with a unique objective function.

We shall develop in this chapter a B&B algorithm to find an optimal schedule for the MPSP. We will do this by deriving and then using the dominance relationship between simultaneously schedulable operations in the MPSP. Also, lower-bound costs are derived to effectively guide the search for an optimal schedule.

A set of DPs w.r.t. all regular measures is identified in Section 3.2. Section 3.3 discusses more DPs w.r.t. the system hazard. Using all the DPs derived in Sections 3.2 and 3.3, we show in Section 3.4 how the B&B algorithm is guided to find an optimal schedule for the

MPSP. A demonstrative example and some computational experiences are presented in Section 3.5.

3.2. Dominance Properties for All Regular Measures

The main idea behind the DPs w.r.t. all regular measures is to eliminate unnecessary preemptions and reduce the machine idle times caused by precedence constraints between dependent operations.

An unfinished operation O_b in the MPG is said to be *schedulable* if (1) the project containing O_b has been released, and (2) all preceding operations and time delays, if any, of O_b have already been completed. The DPs to be identified are based on the following observations:

OB1. Preemptions which do not improve performance must be disallowed to reduce the number of possible branches in the B&B algorithm.

OB2. A machine should not be left idle if there are operations schedulable on the machine.

OB3. A schedule must always be replaced by another schedule if the latter can reduce the completion time of a project without increasing the completion time of any other project.

These observations are obvious since preemptions are allowed and regular measures used.

Let O_b and O_c be two operations schedulable on M_k, ζ_1 and ζ_2 be two preemptive scheduling algorithms, and Z^{ζ_1} and Z^{ζ_2} be the performances of ζ_1 and ζ_2, respectively. OB1 leads to the following theorem.

Theorem 3.1: Given any regular measure, the decision for M_k on which of O_b and O_c should be executed first does not change with time.

Proof : Let O_b and O_c be simultaneously schedulable on M_k at $t_1 = 0$, and suppose O_b is executed first at $t_1 = 0$. Consider the partial schedule in Fig. 3.3(a) generated by a scheduling algorithm ζ_1, where the shaded areas represent preemptions by operations other than O_b and O_c. Algorithm ζ_1 now chooses O_c over O_b to execute at $t_2 > t_1$. We want to show that there is always another scheduling algorithm which is superior to ζ_1 w.r.t. any regular measure. Construct a new schedule with another algorithm ζ_2 which simply changes the execution order of O_b and O_c as shown in Fig. 3.3(b). It is shown that using ζ_2 will never result in a larger project completion time than using ζ_1, because there are always more schedulable operations available to ζ_2 than ζ_1. Since $Z^{\zeta_2} \leq Z^{\zeta_1}$ by OB3, and the partial schedule in Fig. 3.3(a) covers all possible cases, the theorem follows. **Q.E.D.**

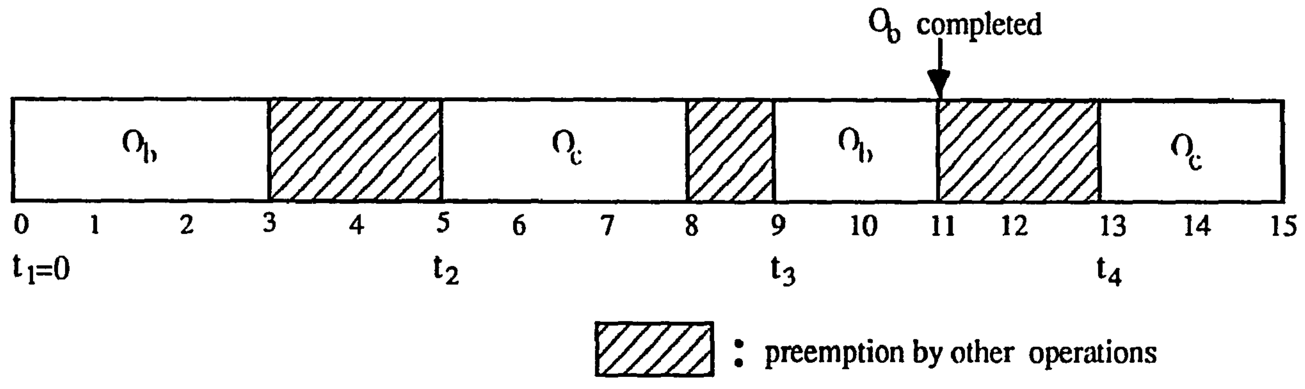
For the purpose of disallowing any unnecessary preemption, Theorem 3.1 simply states that should M_k decide to execute O_b before O_c or let O_b preempt O_c at t_0, then it is unnecessary to let O_c preempt O_b at any $t > t_0$. In what follows, the DPs are identified based on OB2 and OB3.

While the term "precede" holds its usual meaning between two nodes or two arcs in the MPG, we also want to use it between a node and an arc as follows.

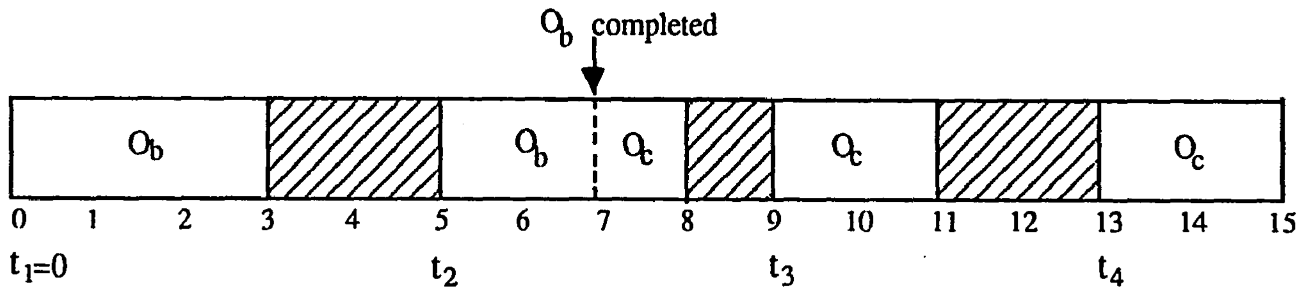
Definition 3.1: An activity (operation or time delay) A_x is said to *immediately precede* a node n_p if n_p is at the "head" of A_x in the MPG, written as $head(A_x) = n_p$, and A_x is said to *precede* n_p if either A_x immediately precedes n_p, or $head(A_x)$ precedes n_p in the MPG.

Therefore, n_p is said to be *realized* if a project whose starting event is represented by n_p has been released, and all activities that immediately precede n_p have been completed.

Since the precedence relation in the MPG is *transitive* ([TrM75]). there are usually many nodes preceded by an operation, say O_x, although only one of them may be immediately



3.3(a). The Original Partial Schedule by ζ_1



3.3(b). An Alternative Partial Schedule by ζ_2

Figure 3.3 Two Schedules Generated by ζ_1 and ζ_2

preceded by O_x. Besides, some of the nodes preceded by O_x may not even be "located" in the same machine, say M_k, that executes O_x. Let $\Omega_0(O_x)$ be the set of nodes which are preceded by an operation O_x, are located in M_k, and represent some project completions. Also, let $\Omega_1(O_x)$ be the set of machine "boundary" nodes which are preceded by O_x, are located in M_k, and are placed at the "tails" of some time delays. For the example MPG in Fig. 3.1, $\Omega_0(O_2) = \{n_{17}, n_{18}\} = \{J_1, J_2\}$ and $\Omega_1(O_2) = \{n_{14}\}$ since n_{14} is a boundary node of M_1, preceded by O_2 and located at the tail of time delay m_{23}. Likewise, $\Omega_0(O_1) = \{n_{17}\} = \{J_1\}$ and $\Omega_1(O_1) = \varnothing$. Moreover, $\Omega_0(O_4) = \{J_1, J_2\}$, $\Omega_1(O_4) = \{n_6, n_{14}\}$, and $\Omega_0(O_5) = \{J_3, J_4\}$, $\Omega_1(O_5) = \{n_5\}$, $\Omega_0(O_6) = \{J_3\}$, $\Omega_1(O_6) = \varnothing$, $\Omega_0(O_{18}) = \{J_4\}$ and $\Omega_1(O_{18}) = \{n_{13}\}$. Notice that $\Omega_0(O_2)$ does not contain J_4 since n_{19} is not located in M_1. Likewise, none of J_1 and J_2 are contained in $\Omega_0(O_5)$. Intuitively, $\Omega_0(O_x)$ and $\Omega_1(O_x)$ indicate which and how projects will benefit from the completion of O_x. Specifically, $\Omega_0(O_x)$ represents the set of projects on a machine, say M_k, whose completions must be preceded by that of O_x. On the other hand, projects on machines other than M_k can only benefit from the completion of O_x through the realization of nodes in $\Omega_1(O_x)$.

Based on the above observation and given any two operations O_b and O_c simultaneously schedulable on M_k, we write $O_c \Rightarrow^Z (=^Z)\ O_b$ if $\Omega_j(O_c) \supseteq (=)\ \Omega_j(O_b)$, $j = 1, 2$. Notice that if O_b precedes O_c then $O_b \Rightarrow^Z O_c$, but the converse is not necessarily true. The relation $\Rightarrow^Z$ is transitive and $=^Z$ is an equivalence relation. Besides, $O_b =^Z O_c$ iff $O_b \Rightarrow^Z O_c$ and $O_c \Rightarrow^Z O_b$.

Definition 3.2: Let ζ_1 be an algorithm which schedules O_b before O_c on a machine M_k at time t_0. It is said to be ***advantageous*** w.r.t. Z for M_k to execute O_c before O_b at t_0 if, for any such ζ_1, there always exists another algorithm ζ_2 which contains the ζ_1's partial schedule prior to t_0, schedules O_c before O_b at t_0, and satisfies $Z^{\zeta_2} \leq Z^{\zeta_1}$.

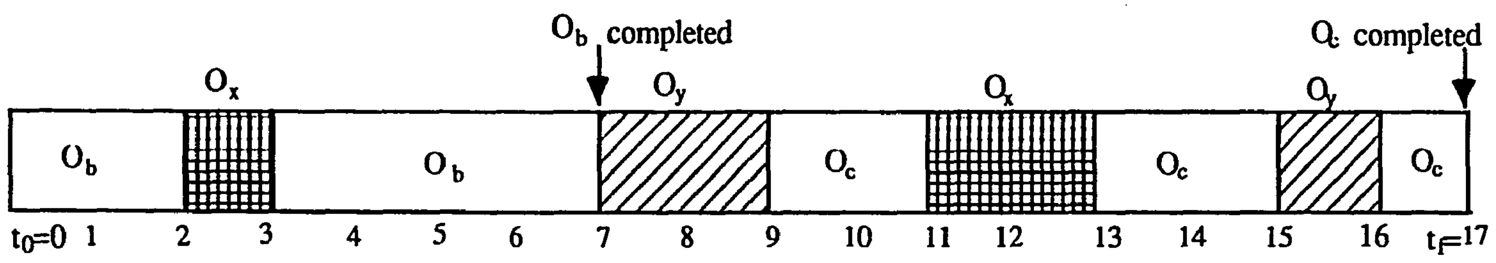
Based on OB2 and OB3, the following theorem associates the relation $=>^Z$ between O_b and O_c with the M_k's decision on which of them should be executed first.

Theorem 3.2: Given any regular measure Z, $O_c =>^Z O_b$ implies that it is always advantageous for M_k to execute O_c before O_b, where O_b and O_c are operations simultaneously schedulable on M_k.

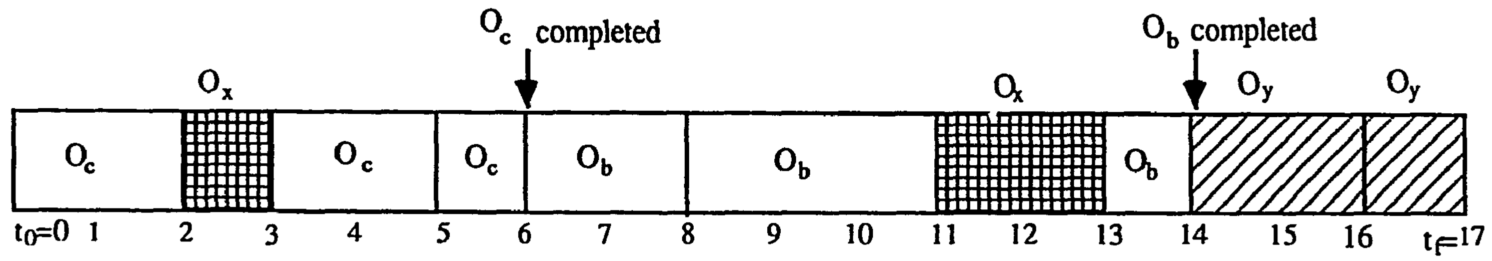
Proof : Suppose ζ_1 is any scheduling algorithm under which M_k executes O_b before O_c at time t_0. Consider the partial schedule μ_1 within the time interval $[t_0, t_f]$ generated by ζ_1, where t_0 is the time both O_b and O_c are schedulable, and t_f the time O_c is completed (Fig. 3.4(a)). In addition to O_b and O_c, there are only two other types of operations that could possibly be executed by M_k within $[t_0, t_f]$: (1) those operations which have no precedence relation with O_b and O_c, and (2) those operations which are preceded by O_b but has no precedence relation with O_c. Operations of type (1) can preempt O_b and/or O_c at any time within $[t_0, t_f]$, whereas those of type (2) can only be executed after O_b's completion, but may preempt O_c at any time within $[t_0, t_f]$. Let O_x and O_y denote, respectively, the representative operations of these two types.

Construct a new partial schedule μ_2 for M_k within $[t_0, t_f]$ as follows. While keeping the schedule for all O_x's unchanged, rearrange all the other operations to be executed in order of O_c, O_b and O_y's (Fig. 3.4(b)). μ_2 is feasible since all precedence constraints are still met within $[t_0, t_f]$. Given $\Omega_0(O_c) \supseteq \Omega_0(O_b)$ and $\Omega_1(O_c) \supseteq \Omega_1(O_b)$, we want to show that there exists another scheduling algorithm ζ_2 which contains μ_2 as well as ζ_1's partial schedule prior to t_0, and satisfies the inequality $Z^{\zeta_2} \leq Z^{\zeta_1}$.

Let $\Delta_0 = \Omega_0(O_c) - \Omega_0(O_b)$, $\delta_1 = \Omega_1(O_c) - \Omega_1(O_b)$, and Δ_1 be the set of projects, the completion event of each of which is preceded by at least a node in δ_1. Also, let $\Delta = \Delta_0 \cup \Delta_1$ and $\bar{\Delta} = J - \Delta$, where J is the set of all projects. That is, $\bar{\Delta}$ represents the set of projects



3.4(a). An Original Partial Schedule μ_1 Generated by ζ_1



3.4(b). The Modified Partial Schedule μ_2 Generated by ζ_2

Figure 3.4. Two Partial Schedules Generated by ζ_1 and ζ_2

preceded by both, or neither, of O_b and O_c, while Δ contains, among others, the set of projects preceded by O_c only. We now separately compare the possible completion times of projects in Δ and those in $\bar{\Delta}$ when either μ_1 or μ_2 is the partial schedule for M_k within $[t_0, t_f]$. Because of the way μ_2 is constructed, O_c has a smaller completion time under μ_2, but O_b and O_y's will have larger completion times as compared to the case of using μ_1. Since O_b precedes O_y, $O_b \Rightarrow^Z O_y$ and, thus, $O_c \Rightarrow^Z O_y$. That is, any project whose completion event preceded by O_y and/or O_b is also preceded by O_c, meaning that the preceded project cannot be completed without completing O_y's, O_b, and O_c. This implies that there exist a schedule ζ_2 containing both ζ_1's prior partial schedule and μ_2, under which the completion times of projects in Δ could be reduced further, without increasing those in $\bar{\Delta}$. Since these arguments are valid for any time t_0, the theorem follows by OB3. **Q.E.D.**

For the example MPG in Fig. 3.1, since $\Omega_0(O_4) = \Omega_0(O_2) \supseteq \Omega_0(O_1)$ and $\Omega_1(O_4) \supseteq \Omega_1(O_2) \supseteq \Omega_1(O_1)$, $O_4 \Rightarrow^Z O_2 \Rightarrow^Z O_1$. Therefore, it is advantageous for M_1 to execute these three operations in order of O_4, O_2 and O_1. It is worth pointing out several insights of Theorem 3.2. First, the partial schedules μ_1 and μ_2 in Theorem 3.2 have been constructed while implicitly honoring Theorem 3.1. Second, $\Omega_2(O_c) \supseteq \Omega_2(O_b)$ does not necessarily imply that it is advantageous for M_k to execute O_c before O_b, where $\Omega_2(O_x)$ is the set of *all* projects preceded by O_x. Third, if it is advantageous at t_0 for M_k to execute O_b before O_c and vice versa, then it makes no difference as to which of O_b and O_c is executed first at t_0. In other words, to search for an optimal schedule, it suffices at t_0 to consider only the case where either of O_b and O_c is executed first.

The DP in Theorem 3.2 is based only on the property expressed in the form of $\Rightarrow^Z$, which does not always exist between simultaneously schedulable operations. It is very difficult to derive any finer-grain DPs other than the above for *all* regular measures. However, if the problem is restricted to a specific regular measure, some finer DPs w.r.t. that measure can be derived. In the next section, we shall derive such DPs w.r.t. Θ, which will then be used to simplify the search further in our B&B algorithm.

3.3. Dominance Properties w.r.t. System Hazard

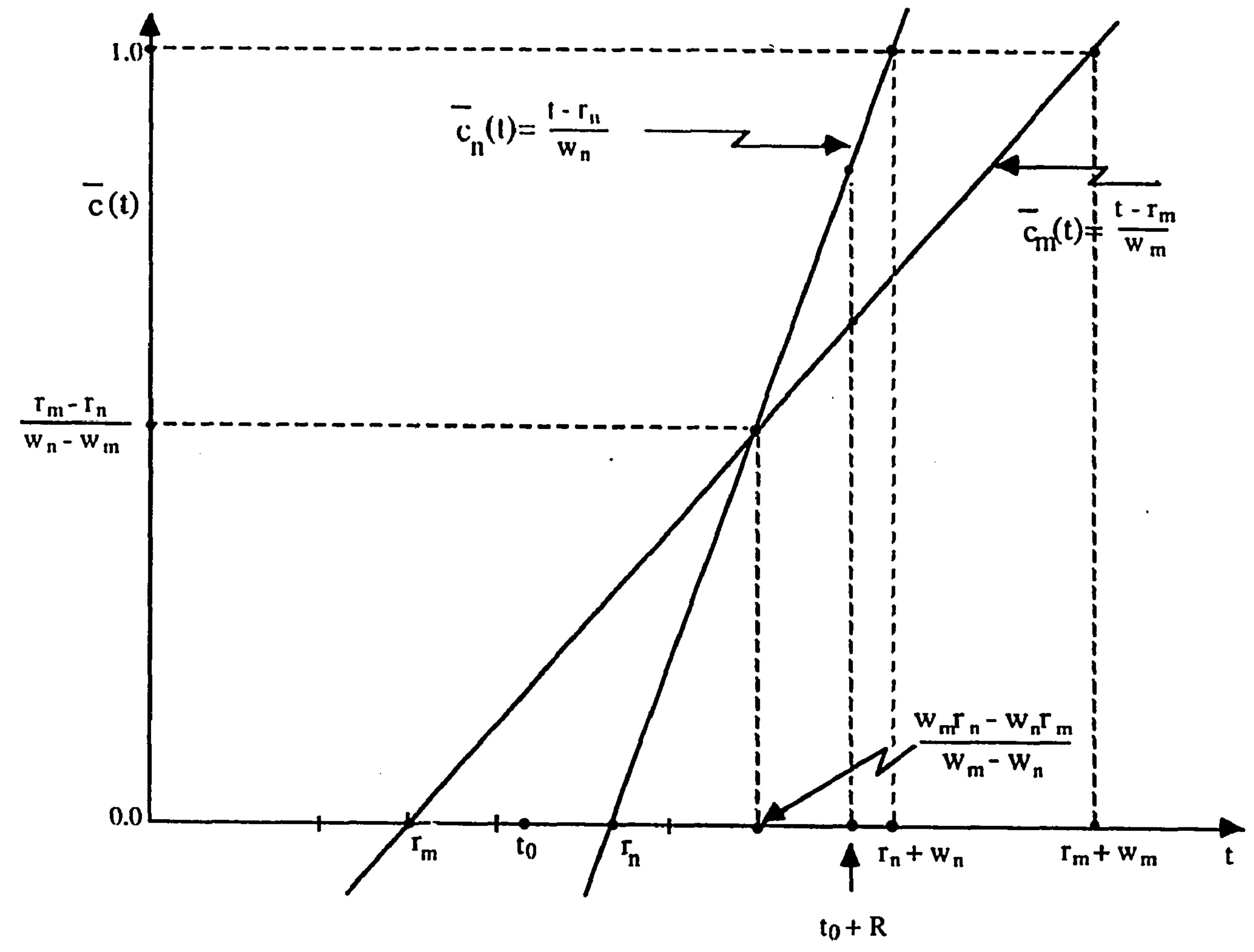
The approach to determining which of O_b and O_c to execute first w.r.t. Θ is equivalent to determining the relative "urgency" of each project in $\Omega_0(O_b) \cup \Omega_0(O_c)$. The Algorithm A described in Chapter 2, which schedules dependent jobs on a single machine, is not directly applicable here, since two operations could be assigned to different machines and must meet the precedence constraints between them. However, the following useful lemma can be obtained from Step SA2 of Algorithm A (see Section 2.3.1).

Lemma 3.1: Let O_b and O_c are two operations simultaneously schedulable on M_k at time t_0 such that $\Omega_1(O_c) \supseteq \Omega_1(O_b)$, $\Omega_0(O_b) = \{J_m\}$, and $\Omega_0(O_c) = \{J_n\}$, for some projects $J_m, J_n \in J$. Then, it is advantageous w.r.t. Θ to execute O_c before O_b at t_0 if the following inequality (see Fig. 3.5) holds:

$$\frac{t - r_n}{w_n} \geq \frac{t - r_m}{w_m}, \quad \forall \ t \geq t_0 + R, \tag{3.1}$$

where R is the sum of the remaining execution times of all operations on M_k which precede at least one of J_m and J_n.

Proof: Since $\Omega_1(O_c) \supseteq \Omega_1(O_b)$, the lemma follows if we can show that executing O_c before O_b is advantageous for J_m and J_n. Consider again the partial schedule μ_1 in Fig. 3.4(a) generated by ζ_1, and the modified partial schedule μ_2 in Fig. 3.4(b) generated by ζ_2, where ζ_2, in this case, differs from ζ_1 only in μ_1 and μ_2. In both μ_1 and μ_2, recall that O_x's represent those operations which can preempt O_b and O_c at any time within $[t_0, t_f]$, whereas O_y's are those operations which can be executed only after the O_b's completion, but can preempt O_c at



$\bar{c}(t)$
1.0
0.0
$\frac{r_m - r_n}{w_n - w_m}$
$\bar{c}_n(t) = \frac{t - r_n}{w_n}$
$\bar{c}_m(t) = \frac{t - r_m}{w_m}$
$\frac{w_m r_n - w_n r_m}{w_m - w_n}$
r_m
t_0
r_n
$r_n + w_n$
$r_m + w_m$
t
$t_0 + R$

Figure 3.5. An Example Cost Structure of J_m and J_n

any time within $[t_0, t_f]$. To prove that $\Theta^{\zeta_2} \leq \Theta^{\zeta_1}$, four cases must be considered depending on whether or not the completions of O_y and O_c represent those of J_m and J_n, respectively.

Case 1. Neither O_y nor O_c is a *completing operation*, i.e., whose completion represents the completion of J_m or J_n. In this case, all project completion times are the same under ζ_1 and ζ_2. Thus, $\Theta^{\zeta_2} = \Theta^{\zeta_1}$.

Case 2. The completion of J_m is represented by that of O_y, but O_c is not a completing operation. The normalized flowtime of J_m under ζ_2 is larger than that under ζ_1 while that of J_n remains unchanged. However, by Eq. (3.1), the normalized flowtime of J_n is larger than that of J_m under ζ_2 which, in turn, is larger than that of J_m under ζ_1. This implies that the maximum of these remain unchanged regardless whether ζ_2 (in place of ζ_1) is used or not. Thus, $\Theta^{\zeta_2} = \Theta^{\zeta_1}$.

Case 3. O_c is a completing operation, but O_y is not. The normalized flowtime of J_n is thus reduced while that of J_m remains unchanged under ζ_2 when compared to ζ_1, thereby making $\Theta^{\zeta_2} \leq \Theta^{\zeta_1}$.

Case 4. Both O_y and O_c are completing operations. From Eq. (3.1) and SA2 of Algorithm A, it is advantageous to execute O_c before O_b (and O_y), thus $\Theta^{\zeta_2} \leq \Theta^{\zeta_1}$. Q.E.D.

Notice that in the above proof, O_y, rather than O_b, is checked; if such an O_y is nonexistent (O_b itself is a completing operation of J_m), then we only need to replace O_y with O_b in the proof. Also, as is shown in Fig. 3.5, t_0 is not restricted to the time after both J_m and J_n are released; t_0 can be any time before either J_m or J_n is completed.

Eq. (3.1) holds iff the following inequality holds (Fig. 3.5):

$$t_0 + R \geq (w_m r_n - w_n r_m) / (w_m - w_n), \quad \text{if } w_m > w_n, \text{ or}$$
$$r_m \geq r_n, \quad \text{if } w_m = w_n, \tag{3.2}$$

where the RHS of the first inequality represents a particular time t such that $(t - r_m)/w_m = (t - r_n)/w_n$. It may be noted that Eq. (3.2) conforms to the various optimal policies known in scheduling theory. This fact leads to the following definition of the "relative superiority" (or urgency) w.r.t. Θ between two unfinished projects J_m and J_n.

Definition 3.3: Given that J_m and J_n are two unfinished projects on M_k, J_n is said to be

(1) *superior* to J_m w.r.t. Θ, written as J_n sp$^\Theta$ J_m, at t_0 if Eq. (3.2) holds, and

(2) *equal* to J_m w.r.t. Θ, written as J_n eq$^\Theta$ J_m, if $w_m = w_n$ and $r_m = r_n$.

Notice that J_n sp$^\Theta$ J_m is defined only when $w_n \leq w_m$ and that J_n eq$^\Theta$ J_m iff J_n sp$^\Theta$ J_m and J_m sp$^\Theta$ J_n $\forall\, t_0$. Notice also that while eq$^\Theta$ is an equivalence relation, sp$^\Theta$ is *not* transitive, meaning that J_n sp$^\Theta$ J_m and J_m sp$^\Theta$ J_p do not necessarily imply J_n sp$^\Theta$ J_p. Further, if J_n sp$^\Theta$ J_m at t_0, then J_n sp$^\Theta$ J_m at all $t_1 \geq t_0$. Based on the superiority relation between two projects, we have the following definitions on their preceding operations O_b and O_c.

Definition 3.4: O_c is said to

(1) *dominate* O_b w.r.t. Θ, written as $O_c \Rightarrow^\Theta O_b$, at t_0 if (a) for every $J_m \in \Omega_0(O_b)$ there exists a $J_n \in \Omega_0(O_c)$ such that J_n sp$^\Theta$ J_m at t_0 and (b) $\Omega_1(O_c) \supseteq \Omega_1(O_b)$, and

(2) be *similar* to O_b w.r.t. Θ, written as O_c S$^\Theta$ O_b, at t_0 if $O_c \Rightarrow^\Theta O_b$ and $O_b \Rightarrow^\Theta O_c$ at t_0.

Definition 3.4 specifies the relative urgency of a schedulable operation O_x in terms of those of the projects in $\Omega_0(O_x)$ and the projects preceded by $\Omega_1(O_x)$. It is worth examining several facts on $=>^{\Theta}$. First, $O_c =>^{Z} O_b$ implies $O_c =>^{\Theta} O_b$ at any time $t_0 \geq 0$, but the converse is not always true. In particular, it is even possible that both $O_c =>^{\Theta} O_b$ and $\Omega_0(O_c) \subset \Omega_0(O_b)$ hold. Second, $=>^{\Theta}$ is not transitive, and thus, neither is S^{Θ}. Finally, if $O_c =>^{\Theta} O_b$ at t_0, then $O_c =>^{\Theta} O_b$ at any time $t_1 \geq t_0$.

Based on the discussions thus far, the DPs between O_b and O_c are summarized in the following theorem.

Theorem 3.3: Let O_b and O_c be two operations schedulable on M_k. Then, w.r.t. Θ,

(1) it is advantageous to execute O_c before O_b at t_0 if $O_c =>^{\Theta} O_b$ at t_0, and

(2) it makes no difference as to which of O_b and O_c is executed first at t_0 if O_c S^{Θ} O_b at t_0.

Proof of (1): Since $O_c =>^{\Theta} O_b$ at t_0, for every project $J_m \in \Omega_0(O_b)$ there exists a corresponding $J_n \in \Omega_0(O_c)$ such that Eqs. (3.1) and (3.2) hold. From Lemma 3.1, executing O_c before O_b at t_0 can reduce the maximum normalized flowtime among such J_m's and J_n's. The completion time of every project other than such J_m's and J_n's does not increase because $\Omega_1(O_c) \supseteq \Omega_1(O_b)$.

Proof of (2): From the definition of S^{Θ} and the result of Part (1), the proof directly follows. **Q.E.D.**

From the discussion of Definition 3.4, there are cases where both $O_c =>^{\Theta} O_b$ and $O_b =>^{Z} O_c$ hold. This simply indicates that executing O_c before O_b (Theorem 3.3) is just as good as executing O_b before O_c (Theorem 3.2). Although Theorem 3.3 is useful, its implementation is not as straightforward as Theorem 3.2, because neither $=>^{\Theta}$ nor S^{Θ} is transitive. Corollaries 3.1 and 3.2 below indicate that, even though neither $=>^{\Theta}$ nor S^{Θ} is transitive, the orders of executing operations implied by these two relations are transitive.

Corollary 3.1: Let $S_k(t_0) = \{ O_j : 1 \le j \le s, s \ge 2 \}$ be a subset of operations schedulable on M_k at time t_0.

(a) If $O_s =>^{\Theta} O_{s-1}$, $O_{s-1} =>^{\Theta} O_{s-2}$, $\cdots$, and $O_2 =>^{\Theta} O_1$ at t_0, then it is advantageous for M_k to execute O_s before O_1 at t_0.

(b) If $O_1 =>^{\Theta} O_s$ in addition to the condition in (a), then it makes no difference as to which operation in $S_k(t_0)$ is executed first.

(The proof follows directly from Theorem 3.3 and Definition 3.2 and, thus, is omitted.)

Corollary 3.2: Let $S_k(t_0)$ be the same as in Corollary 3.1. If $O_s \; \mathrm{S}^{\Theta} \; O_{s-1}$, $O_{s-1} \; \mathrm{S}^{\Theta} \; O_{s-2}$, $\cdots$, and $O_2 \; \mathrm{S}^{\Theta} \; O_1$ at t_0, then it makes no difference as to which operation in $S_k(t_0)$ is executed first.

(The proof follows directly from Part (2) of Theorem 3.3. and is omitted.)

It is worth pointing out that $O_s =>^{\Theta} O_{s-1}$, $O_{s-1} =>^{\Theta} O_{s-2}$, $\cdots$, $O_2 =>^{\Theta} O_1$, and $O_1 =>^{\Theta} O_s$ do not imply that $O_s \; \mathrm{S}^{\Theta} \; O_{s-1}$, $O_{s-1} \; \mathrm{S}^{\Theta} \; O_{s-2}$, $\cdots$, and $O_2 \; \mathrm{S}^{\Theta} \; O_1$ at t_0; the converse does not hold either. For convenience, the set of schedulable operations for which execution order is immaterial is called an *immaterial set* (IM). As we shall see in the next section, knowledge of an IM of size as large as possible greatly simplifies the search for an optimal schedule. An important property associated with two IMs is given in the following corollary.

Corollary 3.3: Let $\Pi_k^1(t_0) = \{ O_1^1, O_2^1, \cdots, O_{s_1}^1 \}$ and $\Pi_k^2(t_0) = \{ O_1^2, O_2^2, \cdots, O_{s_2}^2 \}$ be two distinct IMs of sizes s_1 and s_2, respectively, on M_k at t_0. If there exist $O_b \in \Pi_k^1(t_0)$ and $O_c \in \Pi_k^2(t_0)$ such that executing O_c before O_b at t_0 is advantageous, then it is advantageous to execute O_j^2 before O_i^1 at t_0, $\forall \, i, j$.

Proof: By assumption, it is advantageous at time t_0 to execute O_j^2 before O_c, O_c before O_b, and O_b before O_i^1. Since O_j^2 is arbitrarily chosen from $\Pi_k^2(t_0)$ and O_i^1 from $\Pi_k^1(t_0)$, the corollary follows. **Q.E.D.**

Because $O_c \Rightarrow^{\Theta} (S^{\Theta})\ O_b$ at t_0 implies the same relation at any $t_1 \geq t_0$, Corollary 3.3 simply says that it is advantageous to execute all operations in $\Pi_k^2(t_0)$ before any operation in $\Pi_k^1(t_0)$. Corollary 3.3 — which deals with the "uninterruptability" of an immaterial set — can be thought of as another version of Theorem 3.1, which deals with the uninterruptability of a single operation.

It can be seen that the DPs in Theorem 3.3 are identified only under the condition $\Omega_1(O_c) \supseteq \Omega_1(O_b)$. Without this condition, it is very difficult to find any DP useful for our scheduling problem because of the inter-dependencies between scheduling decisions on different machines. In the next section, we shall show how the DPs presented so far are used in the B&B algorithm to find an optimal schedule.

3.4. Search for an Optimal Schedule with a B&B Algorithm

The proposed B&B algorithm is described in terms of its two phases: branching and bounding. Branching process expands an active parent vertex[3] to generate child vertices while the bounding process derives lower-bound cost of each child vertex to guide the search ([KoS76]).

3.4.1. Generation of a Small Set of Schedules Using DPs

As mentioned earlier, the set of active nonpreemptive schedules contains all optimal schedules w.r.t. any regular measure. Thus, to minimize any regular measure, it is sufficient to consider only this set of active schedules. For our MPSP which is preemptive, we want to

[3]The term "vertex" instead of the more commonly used term "node" is used to avoid possible confusion with the nodes of the MPG and the nodes of a history tree to be introduced.

generate an even smaller set of active schedules containing at least one (rather than all) optimal schedule w.r.t. Θ. An optimal schedule can then be obtained by applying the B&B algorithm only on this set. In what follows, we first summarize the implications of the DPs and then show how this set of active schedules is generated. (As before, $S_k(t_0)$ represents the set of all operations schedulable on M_k at time t_0.)

IP1. M_k must not be left idle at t_0 if $S_k(t_0) \neq \emptyset$.

IP2. M_k must execute $O_c \in S_k(t_0)$ before $O_b \in S_k(t_0)$ at time t_0 if (T1): O_c $=>^Z$ O_b and $O_c \neq^Z O_b$, and/or (T2): O_c $=>^\Theta$ O_b and, O_b and O_c do not belong to the same IM.

IP3. Once O_c has been chosen by M_k at t_0, no other operation in $S_k(t_0)$ is allowed to preempt O_c before its completion. A new operation that becomes schedulable at time $t_1 > t_0$ (and thus belongs to $S_k(t_1)$, not to $S_k(t_0)$) could preempt O_c; t_1 is the only time at which O_c can be preempted by this new operation.

IP1 and IP3 are based on OB2 and Theorem 3.1, respectively, while IP2 comes from Theorems 3.2 and 3.3 and needs further elaboration on its implementation because of the aforementioned properties of $=>^Z$, $=^Z$, $=>^\Theta$ and S^Θ:

P1. $=>^Z$ and $=^Z$ are transitive. Besides, if O_c $=>^Z(=^Z)$ O_b, then O_c $=>^\Theta(S^\Theta)$ O_b at any $t_0 \geq 0$; but the converse is <u>not</u> true.

P2. Neither $=>^\Theta$ nor S^Θ are transitive.

P3. If O_c $=>^\Theta(S^\Theta)$ O_b at t_0, then O_c $=>^\Theta(S^\Theta)$ O_b at any $t_1 \geq t_0$.

P1 suggests that T1 of IP2 be tested before T2, and that O_b be immediately excluded from consideration for scheduling if T1 holds. P2 and part (a) of Corollary 3.1 suggest that the test for T2 on the set of operations survived (called the *survival set*) the test for T1 be different from the test for T1. Specifically, all O_b–O_c pairs in the survival set are tested first for T2 and then any O_b asserted by T2 is excluded from consideration for scheduling. P3 implies that T2 (and T1, of course) need not be tested again until a new operation becomes schedulable on M_k. Let $R_k(t_0)$ denote the set of operations which survive tests for both T1 and T2. Then, by Corollary 3.2 and part (b) of Corollary 3.1, $R_k(t_0)$ may be further partitioned into q_k (yet to be determined) IMs, $\Pi_k^j(t_0)$, $j = 1, 2, \cdots, q_k$, in each of which changing the execution order does not affect the optimality. Thus, it is sufficient to pick one arbitrary operation from each $\Pi_k^j(t_0)$ and consider whether or not to execute it next. Because of P2, each $\Pi_k^j(t_0)$ can be constructed as follows. First, using Corollary 3.2 and starting with an arbitrary operation, a $\Pi_k^j(t_0)$ is formed by adding a new operation O_b to it whenever there exists an O_c already in $\Pi_k^j(t_0)$ such that $O_c \; S^{\Theta} \; O_b$. Second, use part (b) of Corollary 3.1 to merge a *cyclic* set of IMs into $\Pi_k^j(t_0)$. A set of IMs is said to be cyclic if elements from IMs of the set form a cycle of the dominance relation as described in Part (2) of Corollary 3.1. This merger is to further reduce the number of branches generated at the vertex since the single operation to be arbitrarily picked from the IM is now picked from a larger sized IM after the merger.

To meet the uninterruptability requirement, a tree, called the *history tree* (HT_k), for M_k is used to keep track of the execution order of operation on M_k. As shown in Fig. 3.6, each HT_k has only one vertical thread of branches, in which each node represents an operation that is partially completed and preempted by the one immediately above it, and the root is the operation being executed by M_k. For each operation on the vertical thread, a horizontal thread of branches is also constructed to record the set of operations which were not selected by M_k even after surviving both tests of T1 and T2. As we shall see later, each HT_k is constructed and updated such that an operation schedulable on M_k may appear at most at one node of HT_k. HT_k's are used as: (F1) a tool for checking whether or not Theorem 3.3 and the uninterruptability requirement of IMs have been violated in any machine's prior partial schedule, and (F2) a guide for selecting an operation for each machine without violating the

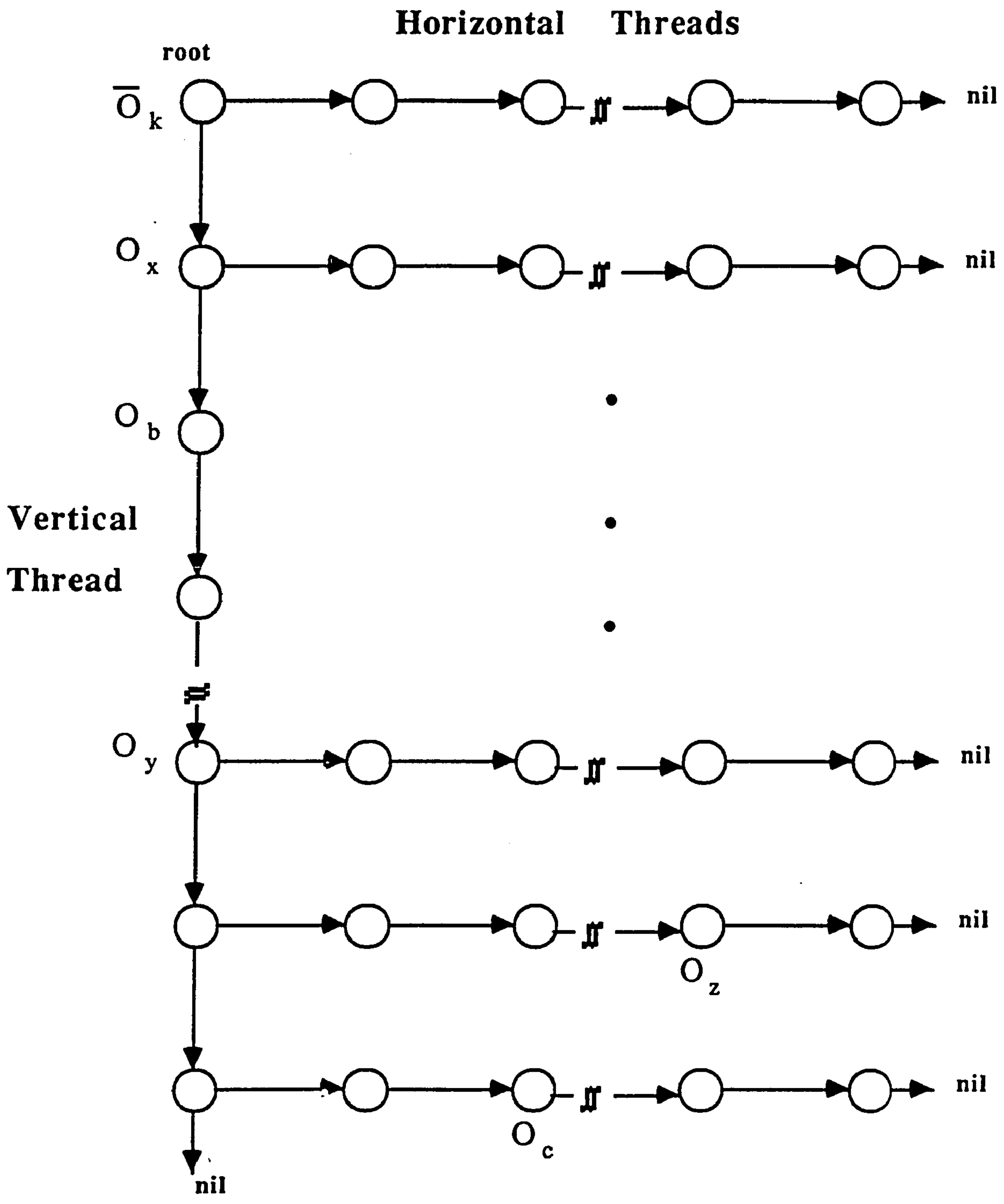


Figure 3.6. The History Tree of Mk

two conditions of F1 at least up to the time of the next scheduling decision. The violations stated in F1 are possible because more dominance relations will be established as time goes by.

In what follows, we briefly describe how HT_k's are constructed and updated, and explain how F1 and F2 are done to further simplify the search in the B&B algorithm. Let t_1 be the time when an operation or time delay is completed, or a new project is released and, thus, a new scheduling decision has to be made. Also, let $R_k(t_1) \subseteq S_k(t_1)$ be the set of all schedulable operations, each of which survived both T1 and T2 at t_1, and $\Pi_k^1(t_1), \Pi_k^2(t_1), \cdots, \Pi_k^{q_k}(t_1)$ be the q_k IMs resulting from partitioning $R_k(t_1)$. If the original HT_k is a null tree, then checking F1 is unnecessary and selecting an operation from any of $\Pi_k^j(t_1)$ will violate neither of the two conditions of F1. Let γ_k denote the operation selected by M_k. Thus, an HT_k is created by using γ_k as its root and including each operation in $R_k(t_1) - \{\gamma_k\}$ in the horizontal branch rooted by γ_k. If the original HT_k is not null, then both F1 and F2 need to be performed. For F1, there are at least two cases, V1 and V2, to be checked:

V1. There is at least a pair of operations, say, O_b and O_c with $O_c \Rightarrow^{\Theta} O_b$ at t_1, such that the node representing O_b is on the vertical thread of HT_k and has the node representing O_c on one of its branches (Fig. 3.6).

V2. There is at least a set of three operations, say, O_x, O_y and O_z, such that (1) both O_x and O_y are on the vertical thread of HT_k and O_y is rooted by O_x (i.e., O_y has ever been directly or indirectly preempted by O_x), and (2) O_z is rooted by O_y and belongs to the same IM as O_x (i.e., the IM containing O_x and O_z has ever been interrupted by that containing O_y).

If one of V1 and V2 for any machine is true, then the parent vertex y_0 being expanded is discarded because y_0 will never lead to an optimal schedule, or there exists at least another optimal schedule which does not include y_0 as its partial schedule. If the parent vertex y_0 is not discarded then F2 needs to be perfomed on each M_k such that the selected operation for M_k will not violate either of the two conditions in F1 at t_1. F2 is done by building a set of "prohibited" IMs on M_k. Specifically, a prohibited IM is the one that has ever been preempted on M_k. Thus, if an operation is chosen from a prohibited IM by M_k at t_1, then the uninterruptability requirement of this IM is violated at t_1. Further, to satisfy Theorem 3.1, if an operation is to be selected by M_k from an IM of which one operation is being executed, then M_k can only continue the operation which M_k has been executing. After applying F1 and F2, let γ_k denote the operation selected for M_k (if such a γ_k does not exist, then M_k is kept idle until the time of next scheduling decision) and let $\overline{O}_k$ represent the operation that was being executed at the time of selecting γ_k. Then, HT_k is updated according to the following rules:

Case 1. $\gamma_k \neq \overline{O}_k$ and $\overline{O}_k$ is not completed at t_1: A new node representing γ_k is created on top of $\overline{O}_k$ indicating the preemption of $\overline{O}_k$ by γ_k. For this new node, a horizontal thread is constructed to include each operation in $R_k(t_1) - \{\gamma_k\}$ if it is not already in HT_k. This is to indicate that, among all operations in $R_k(t_1)$, only γ_k is selected and the thread contains the other operations which are not selected by M_k at t_1.

Case 2. $\gamma_k \neq \overline{O}_k$ and $\overline{O}_k$ is completed at t_1: If γ_k is not the operation previously preempted by $\overline{O}_k$, then the node representing $\overline{O}_k$ is replaced by that representing γ_k, and each operation in $R_k(t_1) - \{\gamma_k\}$ must be appended to the original horizontal thread of $\overline{O}_k$ in case it is not already in HT_k. Otherwise, $\overline{O}_k$ is simply deleted from HT_k, and each operation of $R_k(t_1) - \{\gamma_k\}$ or in the horizontal thread of $\overline{O}_k$ is appended to the horizontal thread of γ_k if it is not already in HT_k.

Case 3. $\gamma_k = \bar{O}_k$ and $\bar{O}_k$ is not completed at t_1: No new node is created to preempt $\bar{O}_k$. However, operations in $R_k(t_1) - \{\gamma_k\}$ must be appended to the horizontal thread of $\bar{O}_k$.

Case 4. No γ_k is selected: Delete $\bar{O}_k$, if any, from HT_k to let HT_k become null and M_k be idle. This is because $\bar{O}_k$ must be completed at t_1 and no more schedulable operations are available to M_k.

Also, to ensure that γ_k appears only once in HT_k, the γ_k already included in one of the horizontal threads must be deleted in the above rules. Except as $\bar{O}_k$ or as the operation being preempted by $\bar{O}_k$ in Case 2, γ_k can never be any node in the vertical thread.

Based on the above discussions of IP1-3, we can now derive the algorithm which generates a small set of active schedules using DPs. It is helpful to summarize and/or introduce the notation to be used in the algorithm.

A: Set of operations without preceding activities on the MPG. An operation in A becomes schedulable whenever the project containing it is released.

B: Set of operations with preceding activities on the MPG. An operation in B becomes schedulable upon completion of all its preceding activities as well as the release of the project containing it.

Y: Set of time delays. Each time delay must have at least one preceding operation.

t_0: The current time or the time a scheduling decision has been made.

t_1: The earliest time since t_0 at least an operation or time delay is completed or a project is released. That is, t_1 is the time when a new scheduling decision has to be made.

S_k: Set of schedulable operations at t_1 on M_k including those partially completed.

L_d: The set of ready time delays, m_i's, at t_1. The remaining delay period of m_i is denoted by v_i. Since time delays are not schedulable objects, each v_i is reduced to the passage of time.

R_k: Subset of schedulable operations on M_k, each of which survives tests for both T1 and T2 of IP2 at t_1. Thus, $R_k \subseteq S_k$.

Π_k^j: The j-th IM on M_k at t_1. Each Π_k^j and the total number of IMs on M_k, denoted by q_k, may change with time.

y_0: The "parent" vertex in the state space to be searched.

y_1: The "child" vertex of y_0.

α_b: Release time of $O_b \in A$.

β_k: Completion time of an operation which is executed by M_k.

β_d: The earliest completion time among all time delays in L_d.

Since preemptions are allowed, a vertex y is represented by the following information:

PS_k: The partial schedule of M_k containing all scheduled (completed and uncompleted) operations.

HT_k: The history tree of M_k.

Using the DPs and notation introduced thus far, we present the following algorithm for generating a small set of active schedules which contains at least one optimal schedule.

```
PROCEDURE create_root_of_search-tree

For $k = 1,2, \cdots, |M|$ do
   1. Initialize $S_k := \emptyset$, $R_k := \emptyset$.
   2. Set $PS_k = HT_k := \emptyset$.
end_do.

Set $t_1 := 0$, $t_0 := 0$, and $L_d := \emptyset$ to create the root vertex $y_0 = y_1$.

end_create_root_of_search-tree.
```

```
MAIN PROGRAM generate_active_schedules

 S1. Initialize $A$, $B$ and $Y$.
 S2. create_root_of_search-tree.
```

S3. While the generated vertex y_1 does not represent a complete schedule and each HT_k survives V1 and V2 of F1 do:

3a. Create S_k's (L_d) by moving all operations (time delays) that become schedulable (ready) at t_1 from A and B (Y) to the corresponding S_k's (L_d).

3b. Perform the dominance tests T1 and T2 on each S_k to generate R_k, and partition each R_k to derive Π_k^j, $j = 1, 2, \cdots, q_k$.

3c. Perform F2 to choose Π_k^i, $i = 1, 2, \cdots, \hat{q}_k$, from that in Step 3b, where i is the new index and $\hat{q}_k \le q_k$. Let $\Gamma_k = \{O_k^1, O_k^2, \cdots, O_k^{\hat{q}_k}\}$ be the set of operations chosen by M_k.

3d. Set $y_0 := y_1$, and create a child vertex y_1 of y_0 for each element in $\Gamma_1 \times \Gamma_2 \times \cdots \times \Gamma_{|M|}$. Let γ_k denote the operation chosen by M_k in y_1. (If $\hat{q}_k = 0$ then $\gamma_k := null$, meaning that M_k is left idle in y_1.)

3e. For each created y_1 do:

3.e.1 Update HT_k, $k = 1, 2, \cdots, |M|$, according to the rules described above.

3.e.2 Set $t_0 := t_1$ and prepare to obtain a new t_1 as below.

3.e.3 Determine $\alpha^* = \min_{O_b \in A} \{\alpha_b\}$, $\beta^* = \min\ \{\beta_d, \min_{k=1,\cdots,|M|} \{\beta_k\}\ \}$, and set $t_1 := \min\ \{\ \alpha^*, \beta^*\ \}$, the earliest time a new scheduling decision has to be made.

3.e.4 Modify PS_k by scheduling γ_k during the interval $[t_0, t_1]$ (let M_k be idle if $\gamma_k = null$), $k = 1,2, \cdots, |M|$.

3.e.5 Go to Step S3.

While satisfying the DPs derived earlier, the above algorithm recursively generates a set of preemptive active schedules in a depth-first fashion for the machines. This set of active schedules contains at least an optimal schedule. Also, because of the uninterruptability in Theorem 3.1, the depth of the search tree generated is at most twice the total number of operations to be scheduled.

3.4.2. Estimation of Lower-Bound Cost

Once the rule for expanding a vertex is determined, the efficiency of search for an optimal schedule with a B&B algorithm depends solely on $\hat{\Theta}(y)$, the lower-bound cost of vertex y, and the computation needed to obtain $\hat{\Theta}(y)$. Based on Algorithm A, various $\hat{\Theta}(y)$'s can be derived depending on how the precedence constraints are relaxed and how the release time and cost function of each operation (time delay) are determined. For one extreme, we may ignore all precedence constraints between any two machines, assume that all projects have been released, apply Algorithm A to obtain the minimal maximum cost for each machine, and use the maximal mini-max cost among all machines as our lower-bound cost $\hat{\Theta}_1(y)$. For the other extreme, we may consider all precedence constraints and project release times, and use the same method to derive the lower-bound cost $\hat{\Theta}_2(y)$. Obviously, $\hat{\Theta}_2(y)$ is tighter than $\hat{\Theta}_1(y)$ but requires more computations to derive. Since all other bounds in between these two extremes can be derived similarly, we shall consider these two extreme bounds only.

Consider a vertex y at $t = t_1$, let $g_i(y)$ be the actual path cost of J_i from the root to y. $g_i(y)$ can be easily computed from the partial schedule represented by y as follows. Since some projects may have been completed before t_1, $g_i(y)$ is thus determined as the *normalized partial flowtime* of J_i at t_1:

$$g_i(y) \triangleq \begin{cases} \bar{c}_i & \text{if } J_i \text{ is completed before } t_1, \\ (t_1 - r_i) / w_i & \text{otherwise,} \end{cases} \tag{3.3}$$

where $\bar{c}_i$, r_i and w_i are the normalized flowtime, release time and normalization factor of J_i, respectively. Note that $g_i(y) < 0$ if J_i is not yet released before t_1.

To derive $\hat{\Theta}_1(y)$, precedence constraints between two machines are ignored, and all unfinished projects are assumed to have been released. These assumptions make it possible to schedule projects, rather than individual operations, with Algorithm A. Specifically, in Step SA2 of Algorithm A, set the cost function of unfinished J_i as

$$f_i(t) := (t - r_i) / w_i = g_i(y) + (t - t_1) / w_i \, , \tag{3.4}$$

and let $t_1 + R_k(B)$ be the block completion time $t(B)$, where $R_k(B)$ is the sum of all remaining execution times of the set of unfinished operations each of which precedes at least a

o

project in B. Suppose $\hat{\theta}_{1k}(y)$ is the maximum between (a) the mini-max cost of M_k obtained from Algorithm A, and (b) the maximum $g_i(y)$ obtained from Eq. (3.3) among all projects completed before t_1 on M_k. Then, $\hat{\Theta}_1(y) \triangleq \max_{M_k \in M} \hat{\theta}_{1k}(y)$ is a lower-bound cost of y, and the computational complexity is $O(\mid M \mid |\bar{J}|^2)$, where $\mid M \mid$ is the total number of machines, $|\bar{J}|$ the average number of projects on each machine.

$\hat{\Theta}_2(y)$ is derived while considering all precedence constraints and project release times. This also implies that operations, rather than projects, are the objects to be scheduled by Algorithm A. In a simpler case, the release time s_b and cost function $f_b(t)$ of an unfinished operation O_b can be determined as follows. Since O_b is not schedulable until each project that contains at least an operation preceding O_b has been released, s_b is set to the maximum release time among all such projects. Because it is the completion of a project, rather than that of an operation, that accounts for the cost, $f_b(t)$ may be set as:

$$f_b(t) := \begin{cases} (t - r_i) / w_i & \text{if } \quad head(O_b) = J_i, \\ 0 & \text{otherwise,} \end{cases} \tag{3.5}$$

where t is the completion time of J_i, and $head(O_b) = J_i$ means that $head(O_b)$ is the node representing the completion event of J_i. After applying Algorithm A, define $\hat{\theta}_{2k}(y)$ similarly to $\hat{\theta}_{1k}(y)$, $k = 1, 2, \cdots, \mid M \mid$. Then, $\hat{\Theta}_2(y) \triangleq \max_{M_k \in M} \hat{\theta}_{2k}(y)$ is a tighter lower-bound cost than $\hat{\Theta}_1(y)$ of y, and the computational complexity is $O(\mid M \mid |\bar{N}|^2)$, where $|\bar{N}|$ is the average number of operations on each machine. Note that more accurate but more computational intensive release time and cost function for each unfinished operation are possible. For example, the minimum time from a project's release to an operation's release may also be considered to determine the release time of the operation. Further, the effect of an operation's completion to the completions of projects on the other machines may be included in deriving a more accurate cost function of that operation. These issues are partially addressed in [PeS89].

As mentioned earlier, $\hat{\theta}_2(y) \geq \hat{\theta}_1(y)$, $\forall y$, because $\hat{\theta}_1(y)$ is a lower-bound of $\hat{\theta}_2(y)$. From $\hat{\theta}_1(y)$ to $\hat{\theta}_2(y)$, there is always a tradeoff as to which lower-bound to use between its accuracy and computational complexity.

3.5. An Example and Computational Experiences

In this section, a demonstrative example and some computational experiences of the proposed B&B method are presented.

3.5.1. An Example

Consider the MPSP for the MPG in Fig. 3.7, where six projects $J_1, J_2, \cdots, J_6$ are to be executed by M_1 and M_2. All operations on the LHS of the shaded area of Fig. 3.7 are to be executed by M_1 and those on the RHS by M_2. Each project is released by realizing its starting node and completed by realizing its ending node. The release times and normalization factors are $r_1 = 0$, $r_2 = 5$, $r_3 = 0$, $r_4 = 3$, $r_5 = r_6 = 10$, $w_1 = 30$, $w_2 = 20$, $w_3 = 30$, $w_4 = 25$, $w_5 = 15$ and $w_6 = 20$. Notice that, while all the other projects are released and completed on the same machine, J_3 (J_4) is released on M_1 (M_2) but completed on M_2 (M_1). Fig. 3.8 shows $\Omega_0(\bullet)$ and $\Omega_1(\bullet)$ of each operation and Fig. 3.9 gives the cost function for each project. Using $\hat{\theta}_1(y)$, we show in Fig. 3.10 all the vertices that have been generated in ascending order of their indices, and in Fig. 3.11 the corresponding optimal schedule.

Before reaching an optimal schedule, which is represented by 17 vertices from the root to vertex v_{25}, a total of only 25 vertices are generated. The first five vertices $v_1 — v_5$ are generated because no DPs exist between O_1 and O_3 as well as between O_{16} and O_3. Since O_3 is denied and O_{16} is chosen at v_5, vertices where O_3 preempts O_{16} will never be generated by expanding v_5 except at, or after, O_{16}'s completion (Theorem 3.1). When expanding v_6, where O_3 and O_2 are both schedulable, only v_7, where O_3 is chosen by M_1, is generated because $O_3 \Rightarrow^z O_2$. This branch of the tree is expanded until v_9, whose lower-bound cost

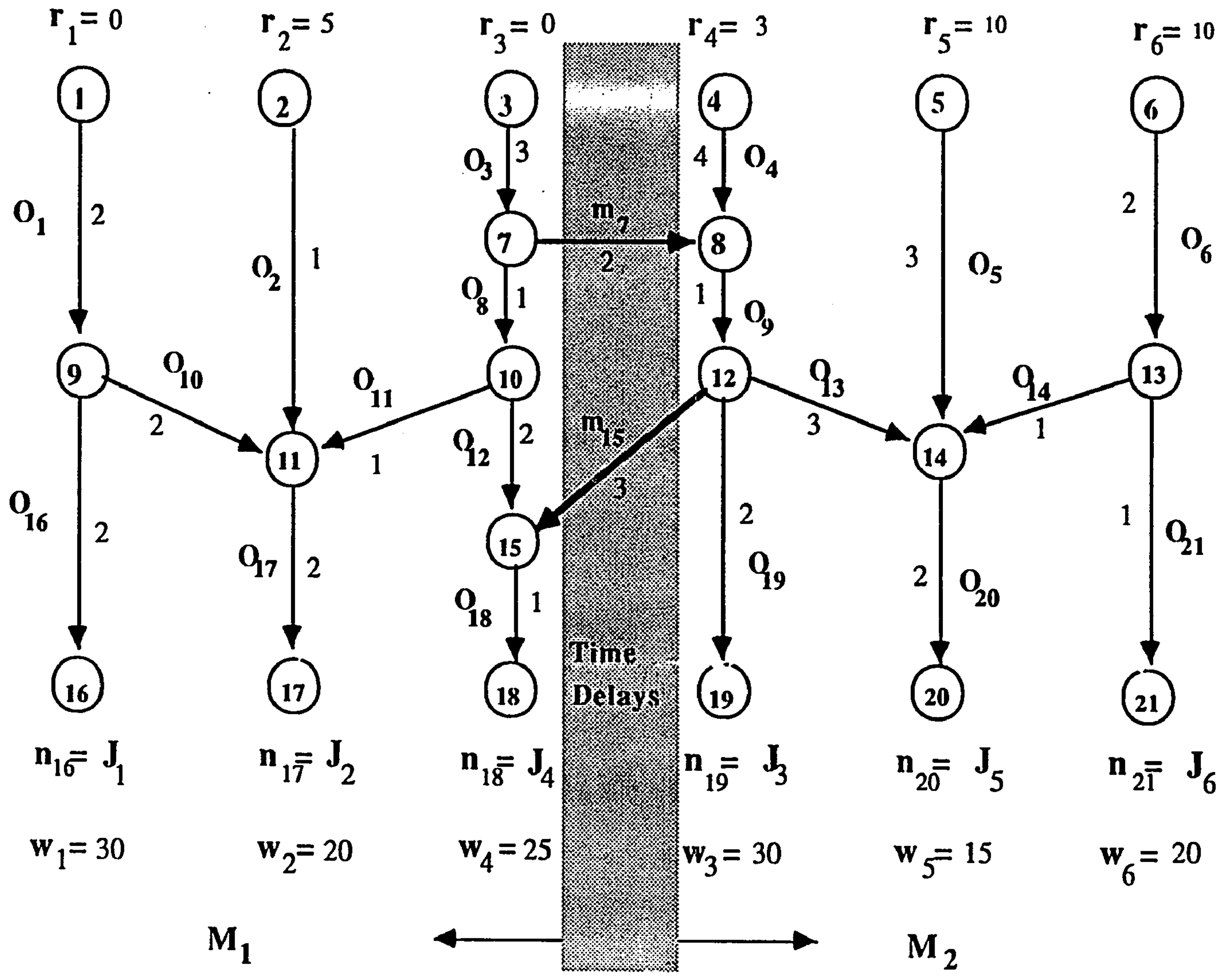


Figure 3.7. The Example MPG

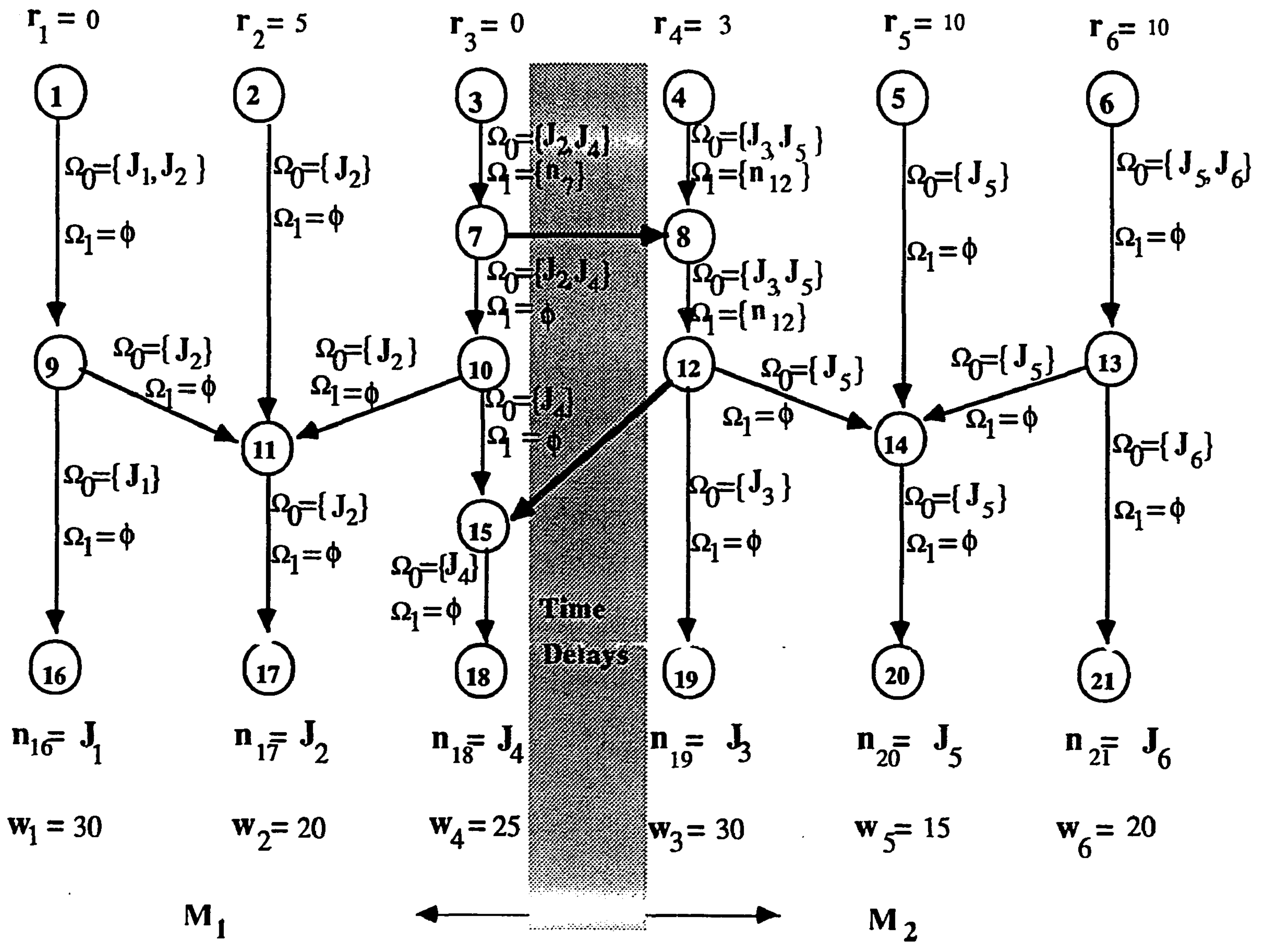
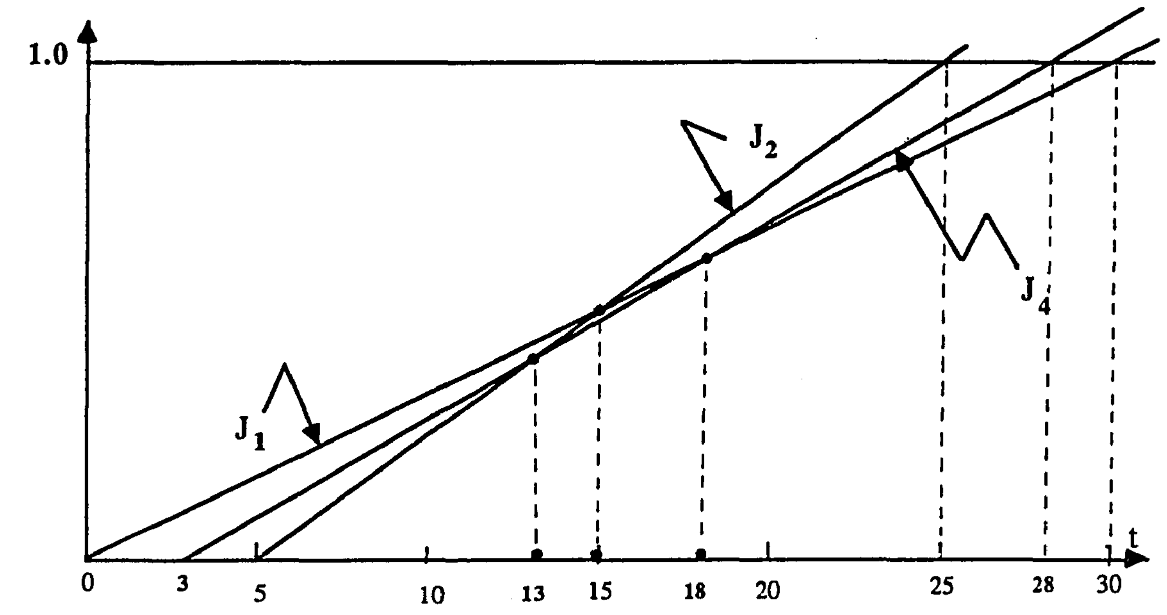
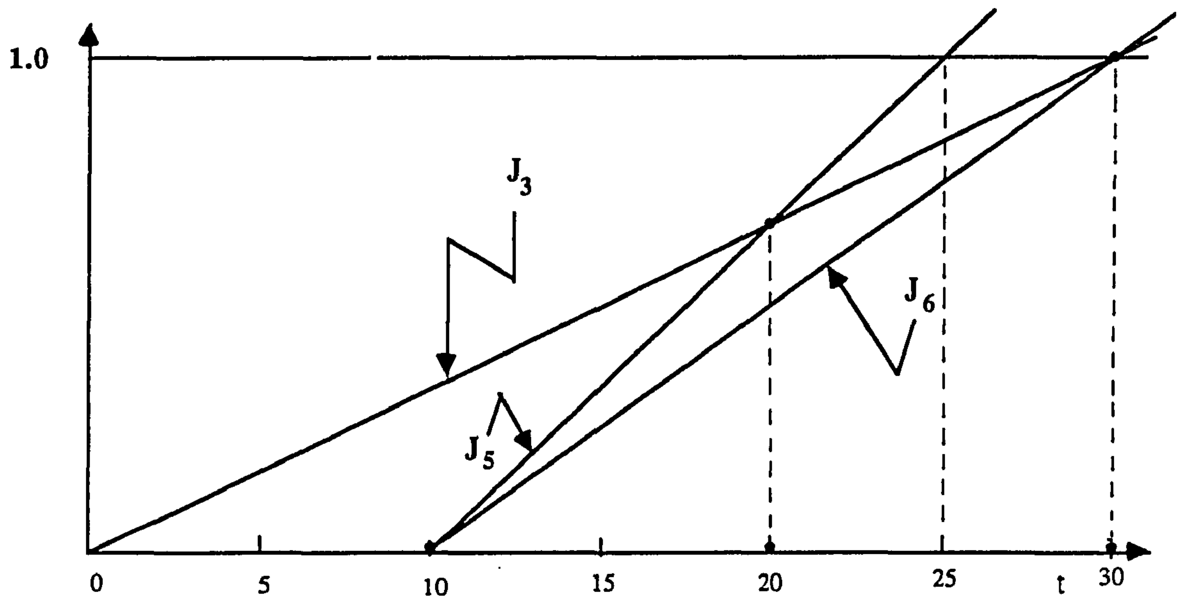


Figure 3.8. The $\Omega_0(\bullet)$ and $\Omega_1(\bullet)$ Sets of the MPG in Figure 3.7



3.9(a). projects on M_1



3.9(b). projects on M_2

Figure. 3.9. Cost Functions of Projects

becomes 0.73. Even with the same lower-bound cost 0.70, v_4, rather than v_3, is chosen for expansion because the depth-first policy is used to break a tie on lower-bound costs.

As v_{10} is expanded, both v_{11} and v_{12} have to be generated because no DPs between O_8 and O_{16} exist (since neither J_2 sp$^\Theta$ J_1 nor J_4 sp$^\Theta$ J_1). From v_{12}, only v_{13}, where O_8 is selected, is generated because $O_8 =>^Z O_2$ and $O_8 =>^Z O_{10}$ although O_2, O_8 and O_{10} are all schedulable at $t = 7$ (i.e., t_1 of v_{12} and t_0 of v_{13}). While expanding v_{13}, four (4) operations (O_2, O_{10}, O_{11} and O_{12}) on M_1 and two (2) operations (O_{13} and O_{19}) on M_2 are schedulable, making a total of 8 combinations. Since O_2, O_{10} and O_{11} belong to the same IM and dominate ($=>^\Theta$) O_{12}, an arbitrary operation, say O_2, is chosen for M_1. On the other hand, $O_{13} =>^\Theta O_{19}$, so only O_{13} is chosen by M_2, although both are schedulable. Therefore, out of these 8 combinations, only one vertex v_{14} needs to be generated without sacrificing optimality, and thus, significantly simplifies the search.

Another situation to be noted is when both O_5 and O_6 become schedulable on M_2 at $t = 10$ as v_{15} is expanded. Since $O_6 =>^Z O_5$ (O_{13}) and O_5 (O_{13}) $=>^\Theta O_{19}$ at $t = 10$, the only vertex to be generated is for O_6 to preempt O_{13} on M_2 although a total of 4 operations (O_5, O_6, O_{13} and O_{19}) are schedulable on M_2. After completing O_6, O_{13} resumes its execution. We can easily see how the DPs are followed similarly by the rest of the optimal schedule.

For completeness, we show how $\hat{\Theta}_1(y)$ and $\hat{\Theta}_2(y)$ can be derived for the above example. Consider v_{14} (see Figs. 3.10 and 3.11) for instance, where $t = 9$ when J_1 is completed while neither J_5 nor J_6 is released. The normalized partial flowtimes at $t = t_1 = 9$ of projects are: $g_1(v_{14}) = Cb1 = (7 - 0) / 30 = 7/30$, $g_2(v_{14}) = (9 - 5) / 20 = 4/20$, $g_3(v_{14}) = (9 - 0) / 30 = 9/30$, $g_4(v_{14}) = (9 - 3) / 25 = 6/25$, $g_5(v_{14}) = (9 - 10) / 15 = -1/15$, $g_6(v_{14}) = (9 - 10) / 20 = -1/20$. To derive $\hat{\Theta}_1(v_{14})$, precedence constraints between M_1 and M_2 are

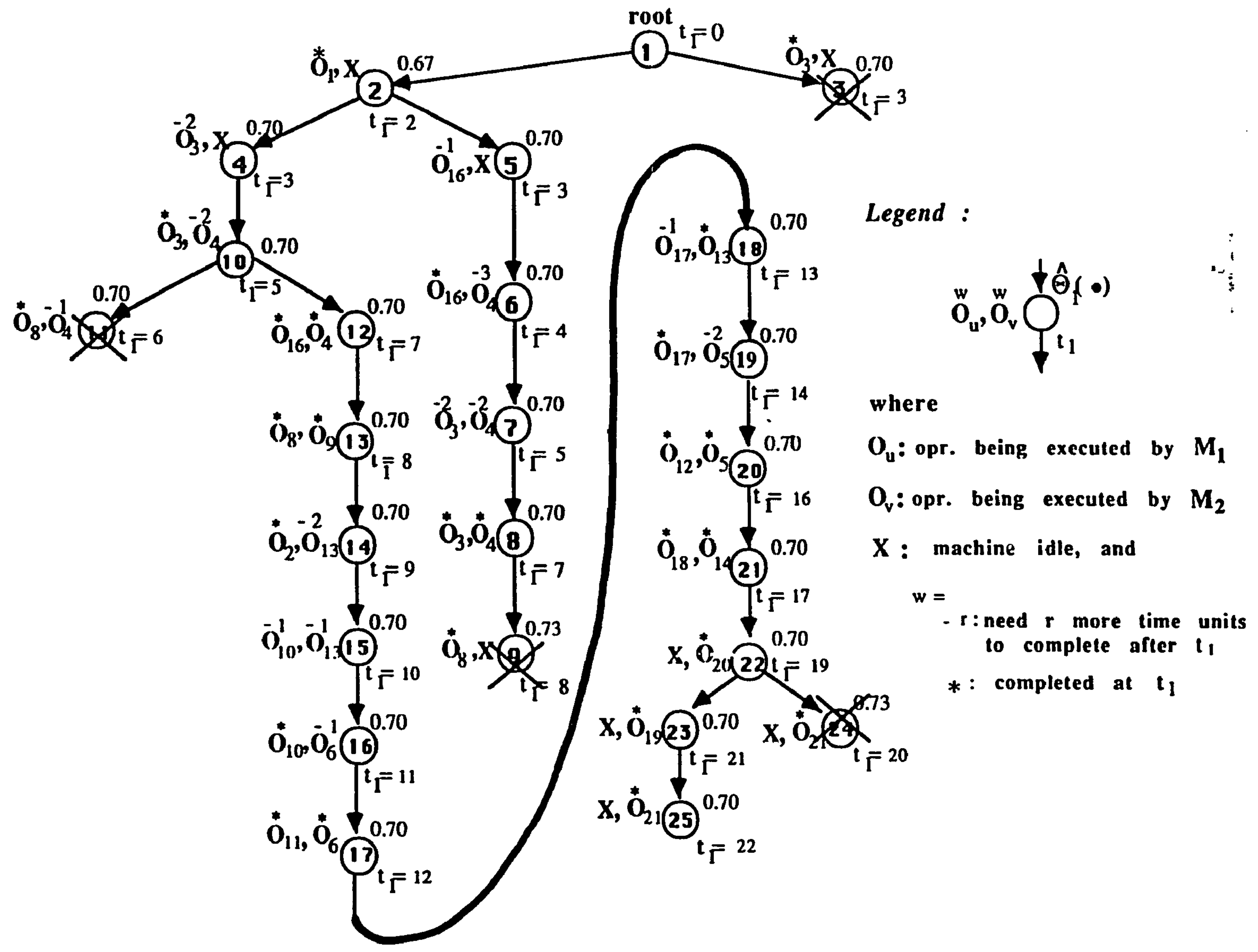
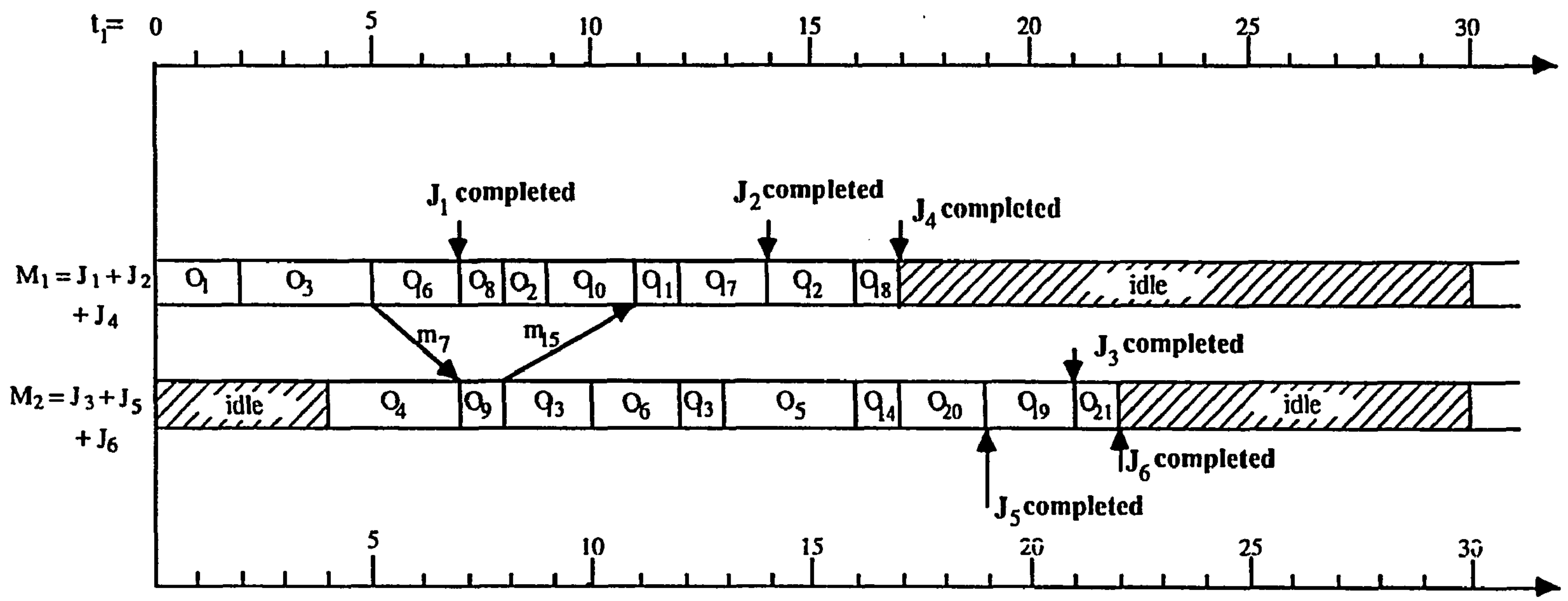

root
Legend :
where
O_u: opr. being executed by M_1
O_v: opr. being executed by M_2
X : machine idle, and
w =
- r : need r more time units to complete after t_1
* : completed at t_1

Figure 3.10. The Search Tree of the MPG of Figure 3.7.



$$\Theta = \frac{21}{30} = 0.70$$

Figure 3.11. The Optimal Schedule of Figure 3.10.

relaxed and all projects are assumed to have been released. The block completion time $t(B)$ for unfinished projects J_1 and J_2 on M_1 is determined to be $t_1 + R_1(B) = 9 + 8 = 17$, where $R_1(B)$ is the sum of all remaining execution times of the set of all unfinished operations on M_1, (i.e., $O_{10}, O_{11}, O_{12}, O_{17}$ and O_{18}), each of which precedes J_2 or J_4. Since $f_2(17) > f_4(17)$ (Fig. 3.9), by Algorithm A, J_4 should be completed last. Delete those operations which precede only J_4 and compute the block completion time for those which precede only J_2 to obtain $t_1 + 5 = 14$. This implies that the mini-max lower-bound cost for J_2 and J_4 on M_1 be max $\{f_2(14), f_4(17)\}$ = max $\{g_2(v_{14}) + 5/20, g_4(v_{14}) + 8/25\}$ = max $\{9/20, 14/25\}$ = 14/25. Thus, $\hat{\theta}_{11}(v_{14})$ = max $\{14/25, 7/30\}$ = 14/25.

Similarly, we proceed with projects J_3, J_5 and J_6 on M_2. They should be completed in the order of J_5, J_3 and J_6, and with completion times 19, 21 and 22, respectively. Thus, $\hat{\theta}_{12}(v_{14})$ = max $\{f_3(21), f_5(19), f_6(22)\}$ = max $\{21/30, 9/15, 12/20\}$ = 21/30. It follows that $\hat{\Theta}_1(v_{14})$ = max $\{\hat{\theta}_{11}(v_{14}), \hat{\theta}_{12}(v_{14})\}$ = 21/30 as shown in Fig. 3.10.

$\hat{\Theta}_2(v_{14})$ is derived while honoring all precedence constraints and all project release times. The release times of unfinished operations on M_1 at v_{14} are $s_{10} = s_{11} = s_{12} = 0$, $s_{17} = 5$ and $s_{18} = 3$, and by Eq. (3.5), the cost functions for these operations are 0 except O_{17} and O_{18}, whose cost functions are those of J_2 and J_4, respectively. By applying Algorithm A and comparing the result with $g_1(v_{14})$, we obtain $\hat{\theta}_{21}(v_{14})$ = max$\{7/30, 14/25\}$ = 14/25, which is the same as $\hat{\theta}_{11}(v_{14})$. For the unfinished operations on M_2 at v_{14}, $s_5 = s_6 = s_{14} = s_{20} = s_{21} = 10$, and $s_{13} = s_{19} = 0$. Similarly, the cost functions for O_5, O_6, O_{13} and O_{14} are 0 while those for O_{19}, O_{20} and O_{21} are those for J_3, J_5 and J_6, respectively. By applying Algorithm A, $\hat{\theta}_{22}(v_{14}) = 21/30$ is obtained, which is again equal to $\hat{\theta}_{12}(v_{14})$. Thus, $\hat{\Theta}_2(v_{14}) = \hat{\Theta}_1(v_{14}) = 21/30$. These results are expected because the unfinished operations always create the same single block in Step SA2 of Algorithm A regardless whether $\hat{\Theta}_1(y)$ or

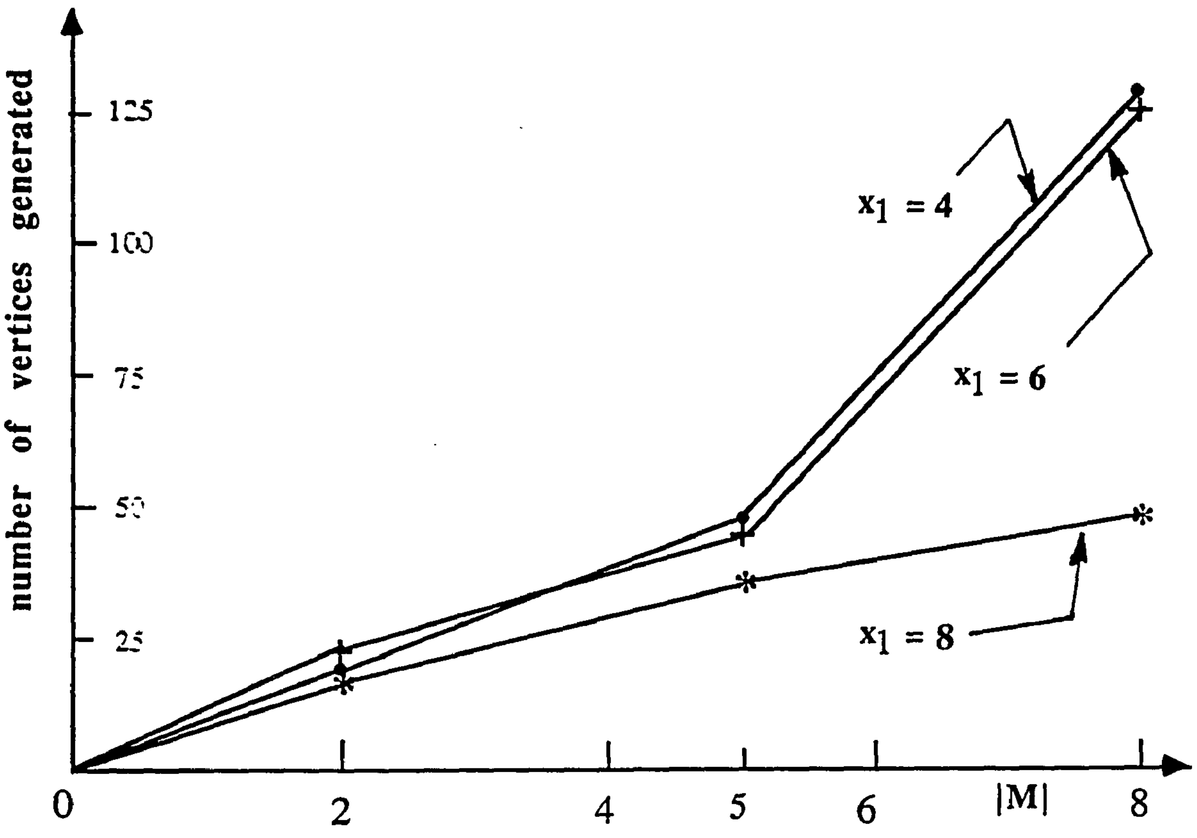
$\hat{\Theta}_2(y)$ is derived.

3.5.2. Computational Experiences

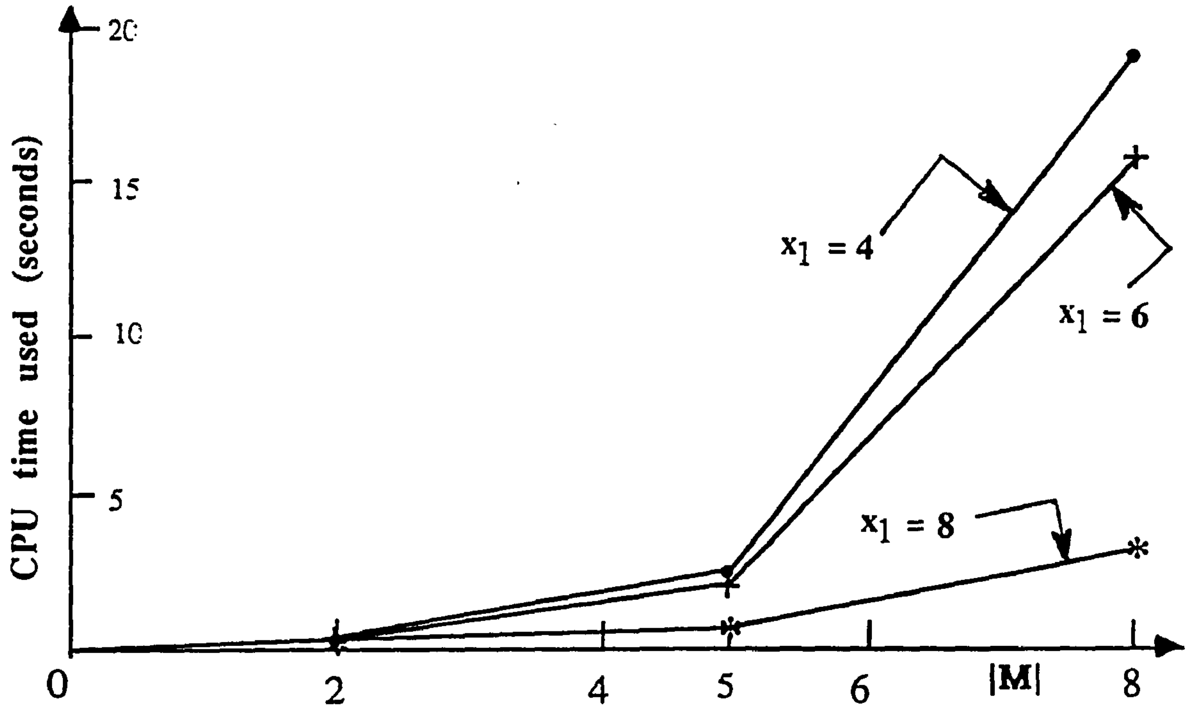
The proposed B&B algorithm, which embodies Theorems 3.1-3.3 and Corollaries 3.1-3.3, was coded in Pascal and run on a VAX-8600 computer, where 4.3 BSD UNIX is used as the operating system. In order to test a wider class of sample problems, a total of 90 multi-projects were randomly generated according to the classification: (a) the average number of operations per project is either 4, 6 or 8 and (b) the number of machines in the system is either 2, 5 or 8.

Consider the class where the operations per project is x_1 and the number of machines x_2. A total of 30, 20 and 10 locally numbered nodes are initially set up on each machine for $x_1 = 4$, 6 and 8, respectively. The operation between nodes n_i and n_j, $i < j$, on the same machine is generated if the outcome from a random experiment using a uniform distribution is greater than the threshold $p\rho^r$, where $p \le 1$ is the initial probability, $\rho = 0.5$ the *discount factor*, and $r = j - i - 1$ the *discount period.* That is, given p, the larger the difference $j - i$, the less likely is an operation generated between n_i and n_j. Therefore, by tuning p and considering the total number of projects to be created, we can generate a set of multi-projects with the desired x_1 value. After removing each ''dangling'' node, projects are created by randomly assigning their starting and ending nodes to the remaining nodes. The number of projects created for each class is pre-determined such that the average numbers of projects on each machine are 2, 1.5 and 1 for $x_1 = 4$, 6 and 8, respectively. For example, two projects on the average are created for the class where $x_1 = 8$ and $x_2 = 2$. Ten projects are created for the class where $x_1 = 4$ and $x_2 = 5$, and so on. Also, time delays are generated for each project and randomly inserted between the project and other projects on different machines to supply the required precedence constraints.

For each of these 9 classes, 10 multi-projects were tested. Table 3.1(a) summarizes the (rounded) average numbers of activities, and projects created for each class. The test results, which include the (rounded) average number of vertices generated and the CPU time (the sum of user and system times) consumed, are recorded in Table 3.1(b) and plotted in Fig. 3.12. According to our experiences, the variance of each entry in Table 3.1(b) grows rapidly as the number of machines increases. For example, for $x_1 = 8$, the variance of the number of vertices generated (CPU time used) for the entry $x_2 = 2$ is about 27 (1389) times higher than that for the entry $x_2 = 8$. Therefore, the test results are very sensitive to the way in which the random samples are generated as well as to the set of multi-projects actually tested as the number of machines increases.



3.12(a). number of vertices generated



3.12(b). CPU time used (seconds)

Figure. 3.12. Test Results

Table 3.1. Test Results

3.1(a). The Nine Classes of MPGs Tested

x_2 / x_1	$\|M\| = 2$	$\|M\| = 5$	$\|M\| = 8$
4	20,* 4**	55, 10	92, 16
6	20, 3	55, 7	90, 12
8	18, 2	47, 5	78, 8

*: number of activities
**: number of projects

3.1(b). The Test Results of the Nine Classes

x_2 / x_1	$\|M\| = 2$	$\|M\| = 5$	$\|M\| = 8$
4	19,* 0.23**	48, 2.51	130, 18.23
6	20, 0.20	47, 2.24	126, 16.25
8	18, 0.11	37, 0.86	49, 3.02

*: number of vertices generated
**: CPU time used (seconds)

CHAPTER 4
STATIC ALLOCATION OF COMMUNICATING PERIODIC TASKS

4.1. Introduction

Using the results of an optimal task scheduling derived in the last two chapters, the optimal allocation of periodic tasks can be determined. Since the allocation of independent tasks is a special case of that of dependent tasks, we deal in this chapter exclusively with the *static allocation* of communicating periodic tasks to the PNs in a distributed real-time system. The allocation is static because it remains unchanged during the entire mission lifetime as long as PNs are fault-free. By 'allocation' we mean *assignment* with the subsequent *scheduling* considered. Specifically, the performance of our task allocation is that of task assignment determined by the subsequent optimal scheduling of assigned tasks. This is in sharp contrast to conventional methods which deal with either assignment or scheduling of tasks, but not both. Note that the performance or cost of any assignment strongly depends on how the assigned tasks are assigned. For the generality of our allocation model, PNs in the system are assumed to be heterogeneous in the sense that different PNs have different processing power for different tasks. The allocation of aperiodic tasks is usually treated as a *dynamic load sharing* problem and is beyond the scope of this chapter.

Three features that distinguish our task allocation problem from others are:

F1. Tasks communicate with one another to accomplish a common system goal. Thus, precedence constraints among tasks must be considered in deriving an optimal task allocation.

F2. The tasks to be allocated are invoked periodically at fixed time intervals during the mission lifetime.

F3. Tasks are usually time-critical, meaning that each task execution is associated with a hard deadline. If the execution of a task is not completed before its deadline, catastrophic outcomes might ensue, e.g., a robot may collide with another robot or even with an operator.

F1 and F2 describe the structure of the task system, while F3 specifies the criterion function for the task allocation problem. Because of F2, only those task invocations in a *planning cycle* I need to be considered for the allocation of tasks since the behavior of task invocations within I repeats itself for the entire mission lifetime. The criterion function to be minimized for our allocation problem is the *system hazard* Θ, or the maximum normalized task flowtime. Besides, as we shall see in the next section, our task allocation model has a finer granularity in describing the precedence constraints between tasks. This captures the fact that in many cases tasks are communicating with each other during the course of their execution.

Task assignment and scheduling problems are studied extensively in both fields of Operations Research and Computer Science [Bak74, CHL80, ChL87, Fre82, WoS74]. For a set of independent periodic tasks, Dhall and Liu [DhL78] and their colleagues developed various assignment algorithms based on the *rate monotonic scheduling* (RMS) algorithm [LiL73], or *intelligent fixed priority algorithm* [Ser72]. However, if precedence constraints exist among tasks, a general approach to nonperiodic task assignment problems must be taken, and the set of tasks to be assigned includes all task invocations within a planning cycle. (Of course, all invocations of the same task must be assigned to the same PN.) Depending on the assumptions and criterion functions used, nonperiodic task assignment problems are formulated in different ways. However, most prominent methods for task assignment in distributed systems are concerned with minimizing the sum of task processing costs on all assigned processors and interprocessor communications (IPC) costs. As was reviewed in [CHL80], these methods are based on graph theoretic [Sto77, StB78], integer programming [MLT82], or heuristic [ShT85, Vir84] solutions. Real-time constraints are difficult to impose when the graph theoretic approach is used. Integer programming methods, on the other hand, allow for constraints that all of the tasks assigned to a processor must be completed within a given time. However, these constraints do not account for task queueing and precedence constraints between tasks.

Few results have been reported on the task assignment with precedence constraints, because most of such problems are NP-hard [Cof76, GaJ79, LeK78, LLR81]. This fact calls for the development of enumerative optimization methods or approximate algorithms using heuristics [KoS76]. For example, in [KaN84] an enumeration tree of task scheduling is generated and searched using a heuristic algorithm called the CP/MISF (Critical Path/Most Immediate Successors First) and an optimal/approximate algorithm called the DF/IHS (Depth-First/Implicit Heuristic Search) to obtain an approximate minimum schedule length (i.e., makespan) for a set of tasks. Chu and Lan [ChL87] chose to minimize the maximum processor workload for the assignment of tasks in a distributed real-time system. Workload was defined as the sum of IPC and accumulated execution time on each processor. A wait-time-ratio between two assignments was defined in terms of task queueing delays. Precedence relations were used, in conjunction with the wait-time-ratios, to arrive at two heuristic rules for task assignment.

To the best of our knowledge, there are no results reported in the literature on the task allocation problem dealing with all of the foregoing three features of distributed real-time systems. This is probably because the problems associated with designing and analyzing such real-time systems are very hard. For example, even for a given assignment, it is shown in [GoS78, LRB77] that most job-shop scheduling problems, which are special cases of our scheduling problem after task assignment, are already NP-hard. Thus, heuristic or enumeration algorithms must be sought.

A branch-and-bound (B&B) algorithm is proposed to solve our task allocation problem. To derive an optimal allocation, the exact cost (i.e., the system hazard) of a terminal vertex[1] (i.e., a complete assignment) must be obtained against which the lower-bound (exact) costs of other partial (complete) assignments can be compared and pruned if so asserted by the algorithm [KoS76]. The results of Chapter 2 provides such an exact cost if the tasks to be allocated are independent. For dependent tasks to be dealt with in this chapter, the results of Chapter 3 can readily be used after conversion of terms as shown in the next section. For non-terminal vertices (i.e., partial assignments) however, lower-bound (rather than exact) costs are estimated by a polynomial-time algorithm to ease the ensuing computational difficulty.

The rest of this chapter is organized as follows. Since inter-task communications are an essential part of our task allocation model, Section 4.2 is devoted to the description of the precedence constraints imposed by such communications. Also, the problem is formulated and the correspondence of terms is shown such that the solution to the MPSP of Chapter 3 can be used for the exact system hazard of a complete assignment. Section 4.3 presents a polynomial-time algorithm to evaluate a "good" lower-bound cost for a non-terminal vertex of the B&B algorithm. This lower-bound cost is used, together with the exact cost obtained, to derive an optimal allocation of periodic tasks. An example is presented in Section 4.4 to demonstrate the power and utility of our task allocation method.

[1]We use the term "vertex" again, instead of the more frequently used "node", to avoid confusion with PNs and the nodes of a task graph (to be described).

4.2. Task System Description and Problem Formulation

It is desirable to model critical real-time systems in great detail such that the true nature of the system under study is accurately described. As will be elaborated on below, the description of the task systems under study covers the inter-task communications and the heterogeneity of PNs. For example, a graph is needed to describe the precedence constraints imposed by inter-task communications. Further, the computation time needed by pure computations as well as by inter-task communications can only be accurately specified after the communicating tasks have been assigned. Therefore, the notion of *nominal computation* and *nominal delay* needs to be introduced for a general description of the task system.

4.2.1. Task System Description

Let $T = \{T_i \mid i = 1, 2, \cdots, |T|\}$ be the set of $|T| \geq 2$ periodic tasks to be allocated among the set of $|N| \geq 2$ processing nodes, $N = \{N_k \mid k = 1, 2, \cdots, |N|\}$, of the system, where $|A|$ is the cardinality of the set A. It is assumed that any two tasks residing in different PNs can communicate with each other by using the usual primitives SEND-RECEIVE-REPLY and QUERY-RESPONSE [ShE87]. If $T_i \in T$ issues a SEND to $T_j \in T$, T_i remains blocked until a REPLY from T_j is received, thus establishing a precedence relation. If T_j executes a RECEIVE before the requested message arrives, it also remains blocked. A task may QUERY another task for information, which replies by executing a RESPONSE. Unlike SEND-RECEIVE-REPLY, the task being queried does not get blocked regardless whether the QUERY has arrived or not.

Each $T_i \in T$ with period p_i consists of one or more computation modules M_{ia} for pure computation, and a single communication module associated with each communication X_{ij} from T_i to T_j using either of the two primitives described above. For example, X_{ij} includes message packetization on T_i's side and packets assembly on T_j's side. It is worth pointing out that M_{ia}'s and all communication modules on T_i's side do not all have to be completed in order to complete T_i. For example, the completion of T_i does not always require the completion of the communication module representing T_i's RESPONSE to T_j's QUERY.

Fig. 4.1 shows an example task system in a PERT/CPM form, which consists of three tasks T_1, T_2 and T_3 with periods 40, 40 and 20, respectively. Within the planning cycle $I = [0, 40)$, except for T_3 which is invoked twice, both T_1 and T_2 are invoked only once. As shown in the figure, an arc represents either a computation module M_{ia}, a communication module, or a message delivery delay. The various communication modules involved can be explained as follows. T_1 packetizes (a) a QUERY message X_{12} to T_2 for information, T_2 assembles (b) the message upon its arrival, and then T_2 packetizes (c) the RESPONSE message X_{21} which contains the queried information and sends it back to T_1. After the RESPONSE message arrives, T_1 assembles (d) the message to get the queried information. On the other hand, T_3 and T_2 exchange messages using SEND-RECEIVE-REPLY for both information and synchronization. T_3 packetizes (e) a SEND message X_{32} to T_2, and T_2 executes a RECEIVE and packetizes (g) the corresponding REPLY message X_{23}. The REPLY message, which may also contain information for T_3, is used by T_2 to unblock T_3, and thus, can only be sent out after the original SEND message from T_3 has been received. As T_2 may proceed to assemble (f) the SEND message only after receiving it, T_3 may also assemble (h) the REPLY message only after arrival of the message at T_3. Notice that the arcs between nodes 4 and 8, 14 and 17, 5 and 6 (13 and 15), and 6 and 7 (15 and 16) represent communication delays for X_{12}, X_{21}, X_{32}, and X_{23}, respectively. Also, the completion of module c — a RESPONSE from T_2 to T_1 — is required for the completion of T_1, rather than T_2, meaning that T_2 may choose to finish its own computation first before responding to T_1's query.

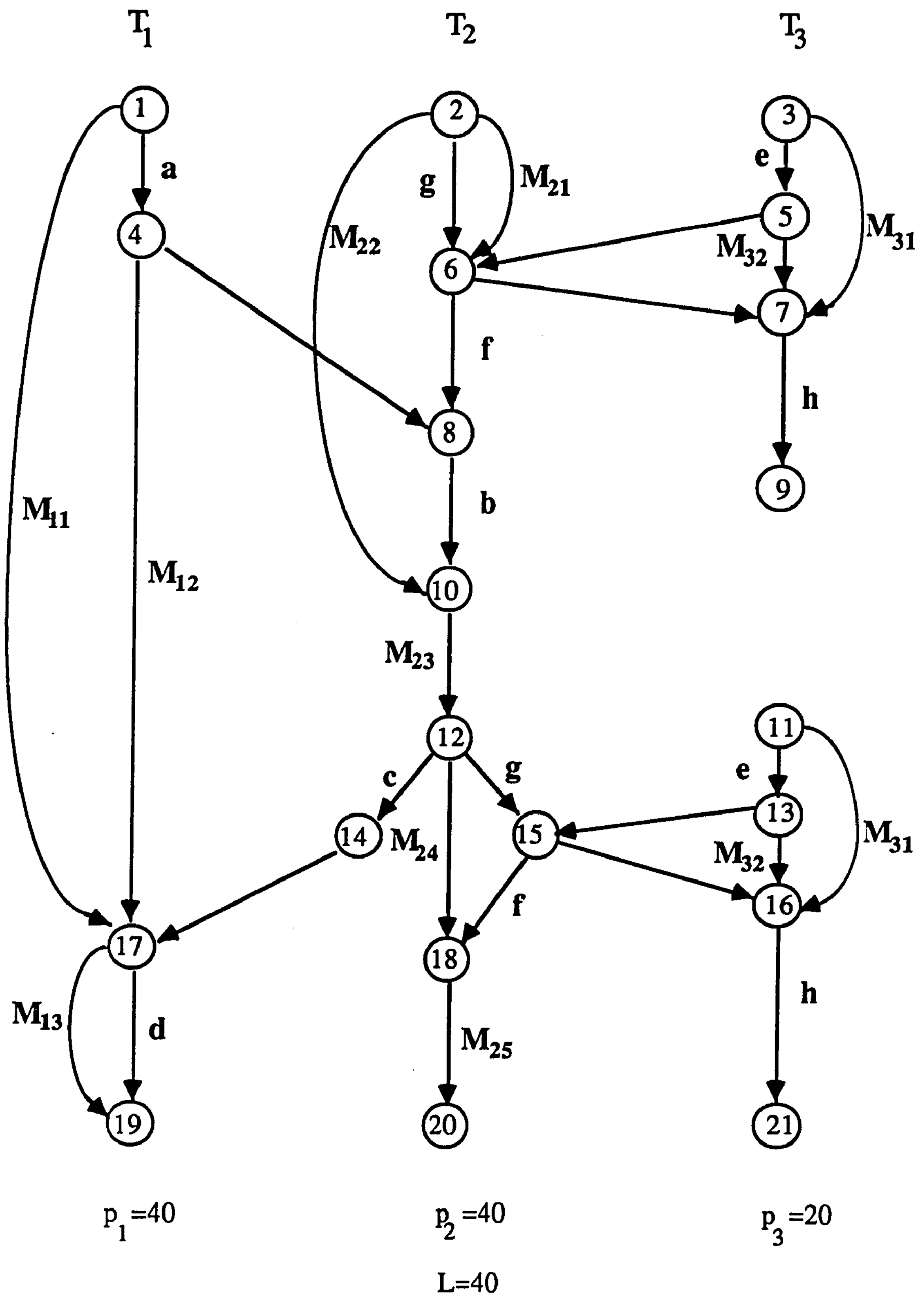


Figure 4.1. An Example Task System

Let $e_{ia} > 0$ be the nominal computation (measured in number of basic steps) of M_{ia}. Then, the execution time (measured in seconds) of M_{ia} on N_k is obtained as the ratio of e_{ia} to the processing power (measured in number of basic steps per second) of N_k for T_i.

Unlike e_{ia}'s, the nominal computations of the two communication modules associated with X_{ij} (one on each of T_i's and T_j's sides) depend on the assignment of T_i and T_j. If T_i and T_j are assigned to the same PN, then the nominal computations of the two communication modules are small since such communications can be achieved via accessing shared memory. Otherwise, larger nominal computations will be needed for packetizing the message and assembling the packets. Similarly to M_{ia}'s, one may obtain the execution time of a communication module on T_i's side as the ratio of its nominal computation to the processing power of N_k for T_i, when T_i is assigned to N_k. As we shall see, the division of T_i into computation and communication modules is essential to the B&B algorithm presented in Section 4.3.

Let $y_{ij} > 0$ be the delay per unit distance (measured in seconds) from T_i to T_j and $d_{hk} \geq 0$ be the communication distance (measured in distance units) from N_h to N_k. Then, the actual communication delay (measured in number of seconds) from T_i to T_j, which are assigned to N_h and N_k, respectively, is determined as $y_{ij}d_{hk}$. Note that y_{ij} is usually a function of the "size" of message to be transferred, and d_{hk} a function of the actual distance between N_h and N_k. Therefore, if $h = k$, we may set $d_{hk} := 0$ to ignore the actual communication delay. On the other hand, we may set $d_{hk} := \infty$ if there is no path between N_h and N_k. Following the usual practice, $d_{hk} = d_{kh}$ is assumed in our problem formulation. Besides, we assume that, for any communication between T_i and T_j, it is always possible to identify which invocations of T_i and T_j this communication is associated with.

Consider all invocations of periodic tasks in a planning cycle $I = [0, L)$, where L is the least common multiple (LCM) of $\{p_i \mid T_i \in T\}$. $T_{i\nu}$, the ν-th invocation of T_i, has to be completed by the $(\nu+1)$-th, $1 \le \nu \le L/p_i$, invocation time of T_i. To describe the nominal computation of each module and the delay of each communication within I and the precedence constraints among them, an acyclic directed task graph (TG) with Activity On Arc (AOA) [Tah76] is used, where an arc represents a module and a node is an event representing the completion of some module(s). Following common practice, the nodes in TG are numbered in such a way that the number assigned to the event at the tail of an arc is always smaller than that assigned to the event at its head. The weight of an arc represents the nominal computation or delay of the corresponding module in the TG. As a node n_p in TG can also represent the completion of a certain invocation $T_{i\nu}$, we write $n_p = T_{i\nu}$ should this happen.

For example, the TG shown in Fig. 4.2 for the task system in Fig. 4.1 is obtained by simply adding the various nominal quantities to each arc. Notice that, unlike M_{ia}'s or communication delays, a nominal computation of the form "u/v" is assigned to each communication module, where u and v represent the nominal computations when the associated communicating tasks are and are not assigned to the same PN, respectively.

4.2.2. Problem Formulation

Let B_k be the subset of tasks assigned to N_k under an algorithm δ such that each task is assigned to one and only one PN. Also, let c and r be the time instants of completion and invocation of a task invocation, respectively. Define the *normalized task flowtime* $\bar{c}$ of the task invocation as

$$\bar{c} \triangleq \frac{c-r}{p}, \tag{4.1}$$

where p is the invocation period of the task. Notice that c (and, thus, $\bar{c}$) depends on the

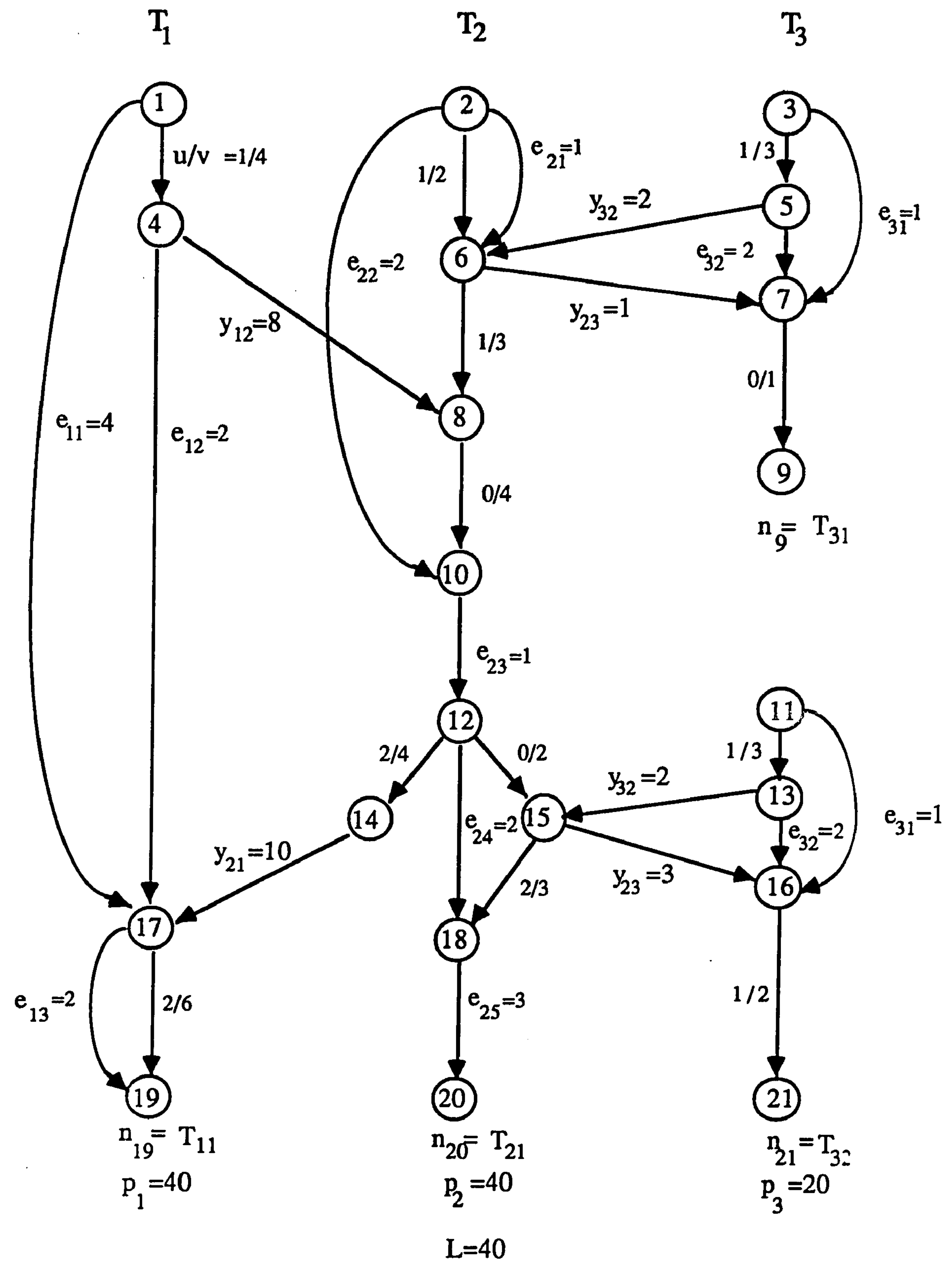


Figure 4.2. An Example TG

assignment algorithm δ. Let $\bar{c}_{iv}$ indicate the normalized flowtime of T_{iv}. Then, the *node hazard* of N_k, θ_k^δ, and the *system hazard*, Θ^δ, are defined as:

$$\theta_k^\delta \triangleq \max_{T_i \in B_k} \left[\max_{1 \le v \le L/p_i} \bar{c}_{iv} \right], \text{ and}$$
$$\Theta^\delta \triangleq \max_{N_k \in N} \theta_k^\delta. \tag{4.2}$$

In other words, Θ^δ is the maximum $\bar{c}_{iv}$ over all invocations of all tasks in T which are distributed over the PNs of the system. We want to find an optimal task assignment algorithm δ^* such that $\Theta^{\delta^*} = \min_\delta \Theta^\delta$. Since δ^* minimizes the maximum $\bar{c}_{iv}$, it allows for better system load <u>sharing</u> (rather than <u>balancing</u>) and a smaller probability of each task missing its deadline. Notice that $\bar{c}_{iv}$ (thus θ_k^δ and Θ^δ) depends not only on δ, but also on how the tasks assigned under δ are actually scheduled on each PN. An optimal preemptive (resume) scheduling algorithm ζ^* needs to be used such that the Θ^δ obtained is the smallest value of $\bar{c}_{iv}$ for each δ.

For example, suppose the TG in Fig. 4.2 is to be assigned to two identical PNs, N_1 and N_2, each with unity processing power for each task, and $d_{12} = d_{21} = 1$. Then, only four different assignment algorithms need to be considered: δ_0, δ_1, δ_2 and δ_3, where δ_0 assigns all three tasks to a single PN, while δ_i assigns T_i to a PN and the other two tasks to the other PN. By using an optimal scheduling algorithm ζ^* under each of these assignments, we obtain $\Theta^{\delta_i} = \max\{\theta_1^{\delta_i}, \theta_2^{\delta_i}\}$ as follows.

$$\Theta^{\delta_0} = \max\left\{\frac{31}{40}, 0\right\} = \frac{31}{40}, \qquad \Theta^{\delta_1} = \max\left\{\frac{39}{40}, \frac{29}{40}\right\} = \frac{39}{40},$$
$$\Theta^{\delta_2} = \max\left\{\frac{34}{40}, \frac{42}{40}\right\} = \frac{42}{40}, \qquad \Theta^{\delta_3} = \max\left\{\frac{10}{20}, \frac{32}{40}\right\} = \frac{32}{40}.$$

Therefore, $\delta^* = \delta_0$, for Θ^{δ_0} is the smallest among these four system hazards. This counter-intuitive result comes from the fact that communication modules and delays are the predominant part of this example. Thus, the algorithm which assigns all tasks to a single PN is superior to the others. The TG with execution times and actual communication delays under δ_3 is given in Fig. 4.3, and the associated optimal schedule in Fig. 4.4.

Our task allocation problem is general enough to capture the three aforementioned features of distributed real-time systems. For example, X_{ij} may be defined for any (T_i, T_j) pair and the precedence constraints imposed by X_{ij} can always be embedded into the TG. Moreover, the criterion function, Θ^{δ}, is directly related to F3.

In Table 4.1, we show the correspondence of terms such that the optimal solution to the MPSP derived in Chapter 3 can readily be used as the optimal scheduling algorithm ζ^* for a complete assignment.

In the next section, the lower-bound cost for a partial assignment is derived with a polynomial-time algorithm such that the computation complexity is reduced.

4.3. Lower-Bound Cost of a Non-Terminal Vertex

The optimal scheduling algorithm for a terminal vertex is necessary for the B&B allocation algorithm to find an optimal assignment. But, the optimal scheduling algorithm is too costly to use for all the non-terminal vertices (i.e., partial assignments) in the B&B allocation algorithm. This is because the B&B algorithm is guaranteed to find an optimal assignment as long as the cost estimate for each non-terminal (terminal) vertex is a lower-bound (true value) of the optimal cost for this vertex [KoS76]. A looser, but inexpensive, lower-bound cost may be more attractive than a tighter, expensive one. In this section, we present such a lower-bound cost which is obtainable in polynomial time.

The lower-bound cost $\hat{\Theta}(x)$ of a non-terminal vertex x on the B&B search tree is a lower-bound estimate of $\Theta^*(x)$, the minimum system hazard among the set of all complete

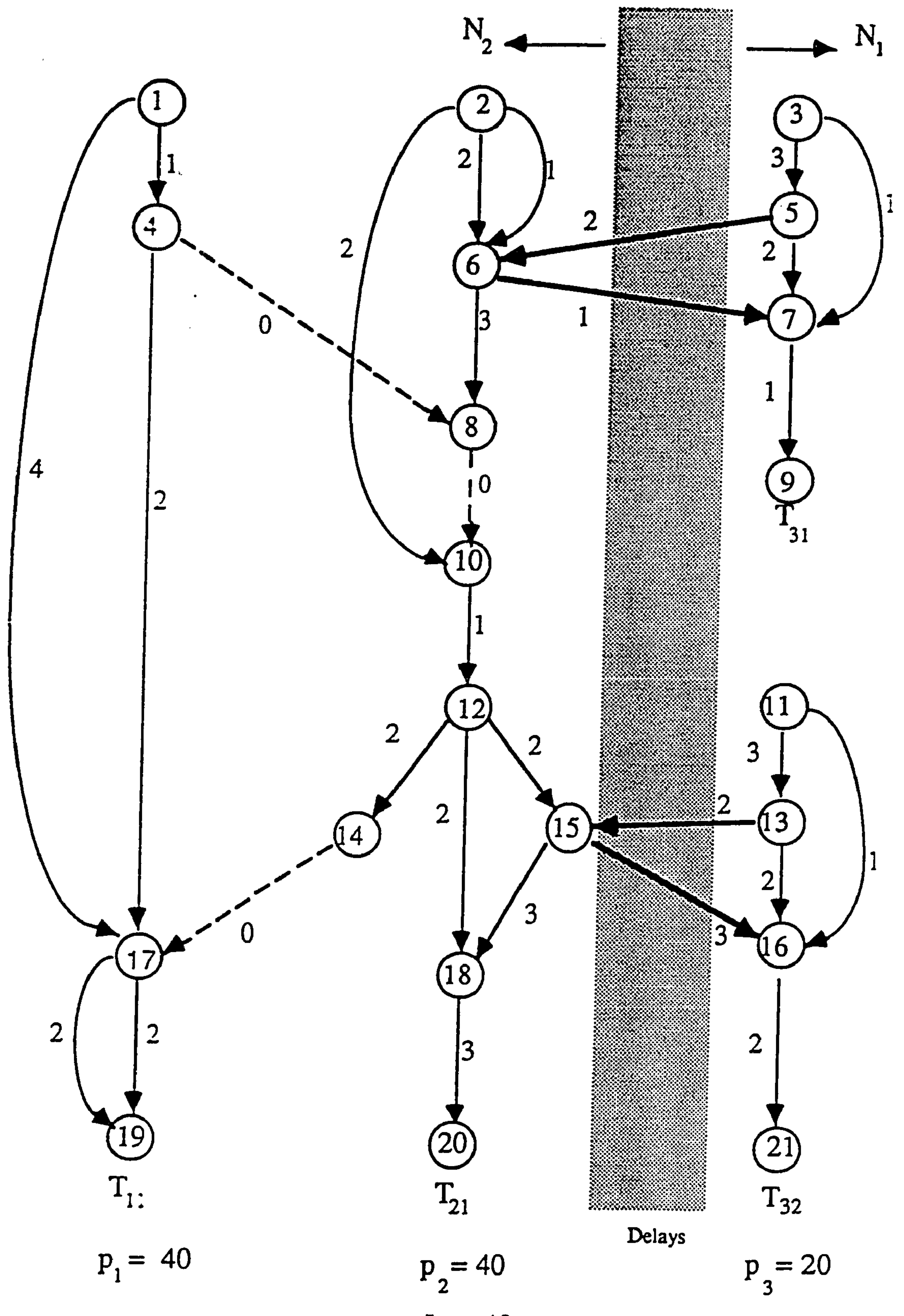
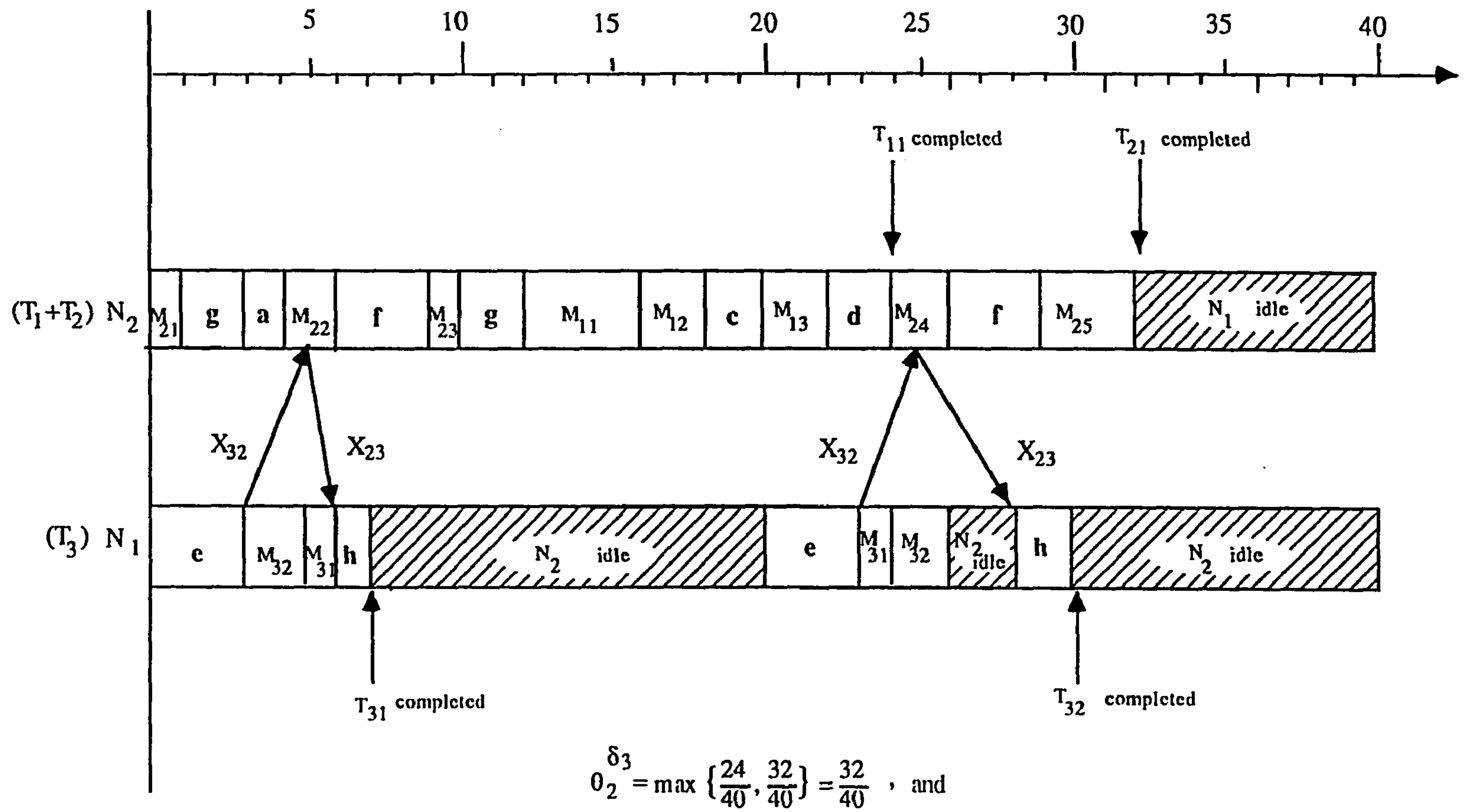


Figure 4.3. The TG for δ_3



$$\theta_2^{\delta_3} = \max\left\{\frac{24}{40}, \frac{32}{40}\right\} = \frac{32}{40}, \text{ and}$$

$$\theta_1^{\delta_3} = \frac{10}{20}$$

$$\Theta^{\delta_3} = \max\left\{\frac{10}{20}, \frac{32}{40}\right\} = \frac{32}{40}.$$

Figure 4.4. An Optimal Schedule for the Assignment by δ_3

Table 4.1. The Correspondence of Terms Between MPSP and Task Allocation

Multi-Project Scheduling	→	Complete-Assignment Scheduling
project J_i	→	task invocation T_{iv}
machine M_k	→	processing node (PN) N_k
multi-project graph (MPG)	→	task graph (TG)
operation	→	computation or communication module
time delay	→	communication delay
release time r_i of J_i	→	invocation time r_{iv} of T_{iv}
completion time c_i of J_i	→	completion time c_{iv} of T_{iv}
normalization factor w_i of J_i	→	invocation period p_i of T_i
normalized project flowtime $\overline{c}_i$ of J_i	→	nomalized task flowtime $\overline{c}_{iv}$ of T_{iv}
system hazard Θ	→	system hazard Θ.

assignments each with x as its partial assignment. The non-terminal vertex x can be represented as a subset of $\{(T_i, N_k) \mid T_i \in T, N_k \in N\}$, where each pair (T_i, N_k) represents the assignment of T_i to N_k.

For each partial assignment x, the following two steps are necessary to derive $\hat{\Theta}(x)$.

- Construct a precedence graph $TG_k(x)$ for each N_k from the original task graph TG.
- Apply an optimal scheduling algorithm ζ on a simplified version of $TG_k(x)$ to obtain the lower-bound cost $\hat{\Theta}_k(x)$. The desired $\hat{\Theta}(x)$ is then chosen as the maximum of $\hat{\Theta}_k(x)$ over all k.

These steps are detailed in the next two subsections.

4.3.1. Obtaining $TG_k(x)$

Each $TG_k(x)$ is composed of two subgraphs: $TG1(x)$ that represents those tasks already assigned within partial assignment x and is common for all k, and $TG2_k(x)$ that represents the load to be imposed on N_k by those tasks that have not yet been assigned within x. To obtain $TG_k(x)$, the following notation is used:

q_k^i: The processing power of N_k for T_i.

$e_{ia}^k \triangleq e_{ia}/q_k^i$: the execution time (measured in seconds) of M_{ia} on N_k, where e_{ia} is the nominal computation of M_{ia}.

$X_{ij,e}$: The communication module on T_e's side, $e = i, j$, associated with X_{ij}.

$\chi_{ij,e}$: the nominal computation (measured in number of basic steps) of $X_{ij,e}$. The value of $\chi_{ij,e}$ depends on whether both T_i and T_j are assigned to the same PN or not. The value of $\chi_{ij,e}$ is denoted as $\chi_{ij,e}(0)$ ($\chi_{ij,e}(1)$) if T_i and T_j are (not) assigned to the same PN.

$\chi_{ij,i}^{k} \triangleq \chi_{ij,i}/q_k^i$: the execution time (measured in number of seconds) of $X_{ij,i}$ on N_k. The value of $\chi_{ij,i}^{k}$ depends also on whether both T_i and T_j are assigned to the same PN or not. The value of $\chi_{ij,i}^{k}$ is denoted as $\chi_{ij,i}^{k}(0)$ ($\chi_{ij,i}^{k}(1)$) if T_i and T_j are (not) assigned to the same PN.

$y_{ij}^{hk} \triangleq y_{ij}d_{hk}$: the actual delay (measured in number of seconds) of communication X_{ij}, when T_i and T_j are assigned to N_h and N_k, respectively.

For a given partial assignment x, let $B_k(x)$ be the set of tasks already assigned to N_k, $B(x) = \bigcup_k B_k(x)$, and $\bar{B}(x) = T - B(x)$, the set of tasks not yet assigned within x. $TG\,1(x)$ can be obtained from TG by the following steps.

S1. For all a, compute e_{ia}^{k} if $T_i \in B_k(x)$, and let $e_{ia}^{k} = 0$ if $T_i \in \bar{B}(x)$. This represents the computation load on N_k contributed by T_i under the partial assignment x.

S2. Case 1: $T_i \in B_k(x)$ and $T_j \in B_h(x)$, $h \neq k$.

Compute $\chi_{ij,i}^{k} = \chi_{ij,i}(1) \,/\, q_k^i$ and $\chi_{ji,i}^{k} = \chi_{ji,i}(1) \,/\, q_k^i$ to represent the load on N_k imposed by the communication between T_i and T_j when T_j is assigned to N_h.

Case 2: $T_i \in B_k(x)$ and either $T_j \in B_k(x)$ or $T_j \in \bar{B}(x)$.

Obtain $\chi_{ij,i}^{k} = \chi_{ij,i}(0) \,/\, q_k^i$ and $\chi_{ji,i}^{k} = \chi_{ji,i}(0) \,/\, q_k^i$ to represent the load on N_k imposed by the communication between T_i and T_j when T_j is either assigned to N_k or not yet assigned at all.

Case 3: $T_i \in \bar{B}(x)$.

Set $\chi_{ij,i}^{k} := \chi_{ji,i}^{k} := 0$.

S3. Set $y_{ij}^{kh} := y_{ij}d_{hk}$ if $T_i \in B_k(x)$ and $T_j \in B_h(x)$, $k \neq h$. Otherwise, set $y_{ij}^{kh} := 0$ to represent the case where at least one of T_i and T_j is not yet assigned within x.

S4. Assign the above e_{ia}^{k}, $\chi_{ij,i}^{k}$, $\chi_{ji,i}^{k}$, and y_{ij}^{kh} as the weights of their corresponding arcs on the TG.

S5. Simplify and restructure the resulting task graph to obtain $TG1(x)$ by deleting and/or adding *dummy* modules to preserve the precedence and timing constraints. (A dummy module is a module with zero computation time or communication delay.)

The resulting $TG1(x)$ is the partial task system corresponding to those tasks already assigned within x while ignoring those tasks not yet assigned. For example, consider the TG shown in Fig. 4.2, where T_1, T_2 and T_3 are to be assigned to N_1 and N_2 with $d_{12} = d_{21} = 1$. Assume $q_1^1=q_1^2=q_1^3=2$ and $q_2^1=q_2^2=q_2^3=1$. Let $x=\{(T_1, N_1), (T_2, N_2)\}$, i.e., T_1 is assigned to N_1, T_2 to N_2 and $\bar{B}(x) = \{T_3\}$. The resulting $TG1(x)$ is shown in Fig. 4.5 with all related execution times and actual communication delays properly indicated, but all modules associated with T_3, an unassigned task, are ignored. While all the nodes associated with T_3 are merged and/or deleted in the restructured TG, n_{12} and n_{15} cannot be merged because the earliest time n_{15} can be realized is 20 (T_{32}'s invocation time), but n_{12} may be realized before or after t=20. For $x = \{(T_1, N_1), (T_3, N_2)\}$ it is interesting to see $TG1(x)$ in Fig. 4.6 that a dummy module between n_4 and n_{16}, and another between n_5 and n_{17} are added to preserve the precedence constraints between modules of T_1 and T_3 as T_2 is ignored.

$TG2_k(x)$ represents the total minimum load on N_k imposed by the unassigned tasks, and thus, can be expressed as $TG2_k(x) = \{\psi_{iv}^{k}(x) \mid v = 1, 2, \cdots, L/p_i, T_i \in \bar{B}(x)\}$, where $\psi_{iv}^{k}(x)$ represents the minimum load imposed on N_k by T_{iv}, the v-th invocation of $T_i \in \bar{B}(x)$. $\psi_{iv}^{k}(x)$ is derived by considering whether or not T_i will be assigned to N_k. For each $T_i \in \bar{B}(x)$, let $MIL(x)$ ($\overline{MIL}(x)$) represent the *Minimum Imposed Load* on N_k by T_{iv} when T_i is (not) to be assigned to N_k. $MIL(x)$ consists of three parts: 1) T_{iv}'s computation modules, 2) communication modules $X_{ij,i}$'s and $X_{ji,i}$'s between T_{iv} and T_j's invocations that have

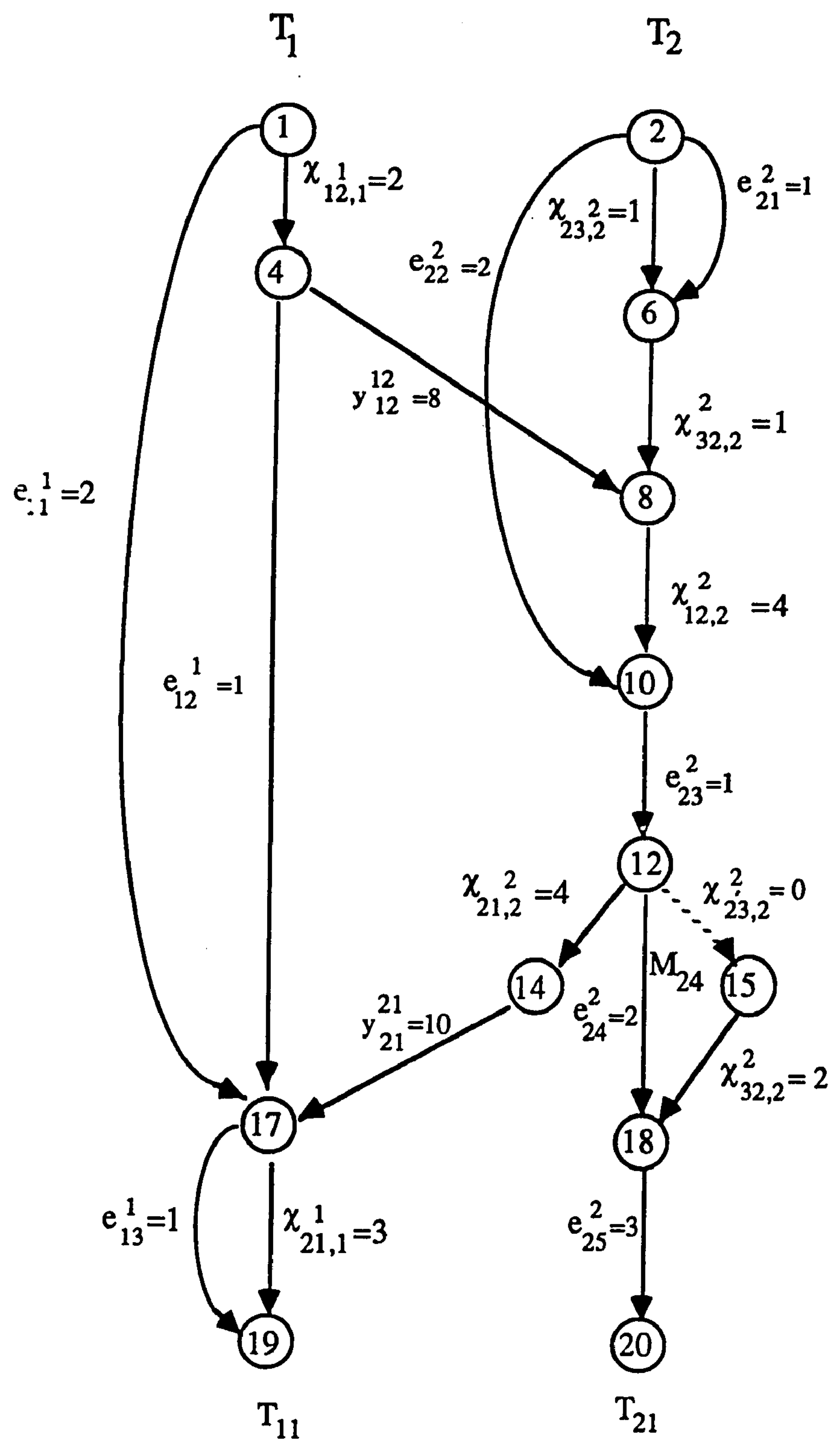


Figure 4.5. TG1(x) for $x = \{ (T_1, N_1), (T_2, N_2) \}$

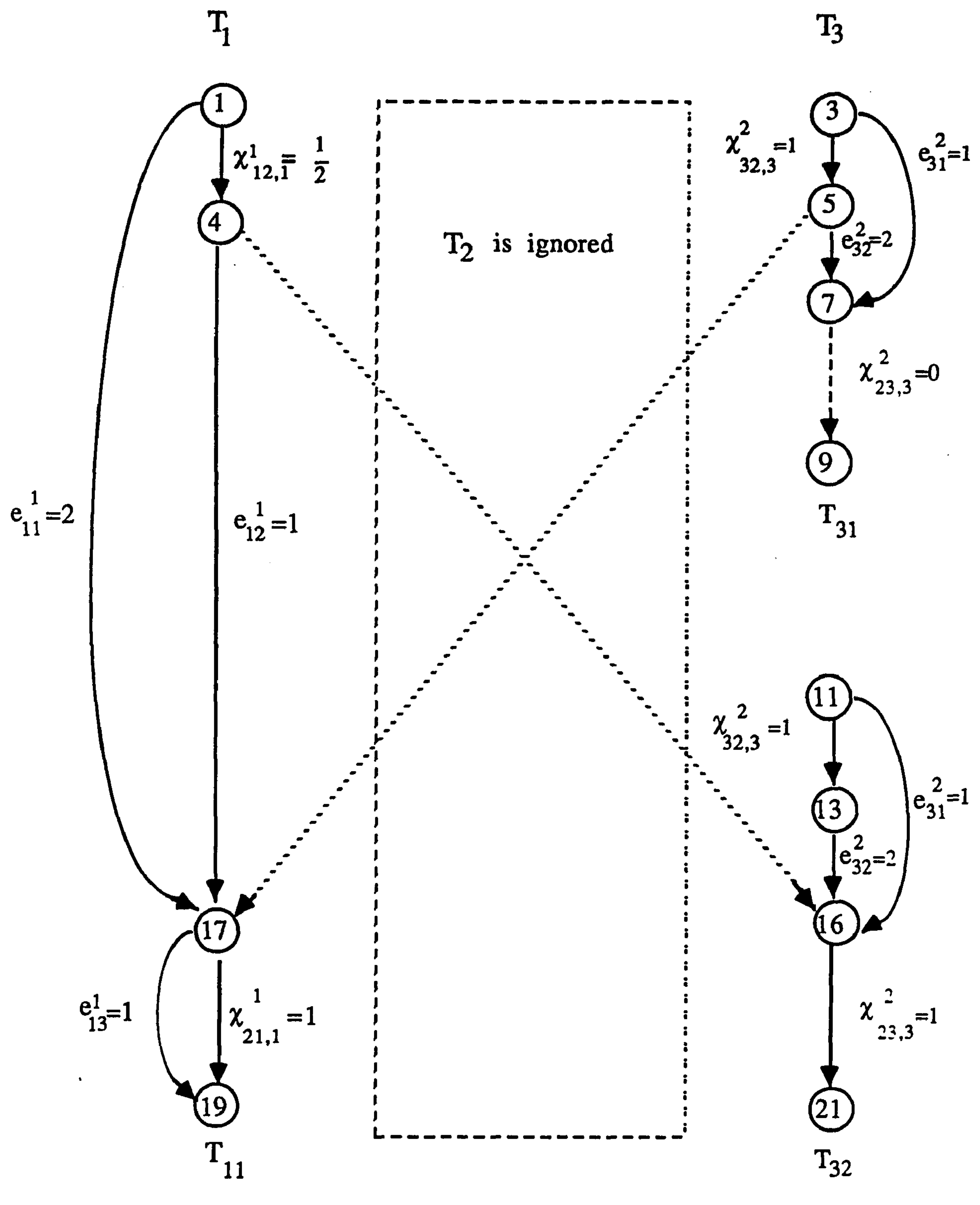


Figure 4.6. TG1(x) for $x=\{(T_1, N_1), (T_3, N_2)\}$

already been assigned to N_h, $h \neq k$, and 3) all the other communication modules to those tasks that have been either assigned to N_k, or not yet assigned. Because of the way $TG\,1(x)$ is constructed, $\overline{MIL}(x)$ is composed of $(\chi^{k}_{ji,j}(1) - \chi^{k}_{ji,j}(0))$'s and $(\chi^{k}_{ij,j}(1) - \chi^{k}_{ij,j}(0))$'s for each communication between $T_j \in B_k(x)$ and T_{iv}. In other words, $\overline{MIL}(x)$ is an extra load on N_k to facilitate the IPC between N_k and N_h, $k \neq h$, to which T_{iv} is assigned. Since each module of $MIL(x)$ ($\overline{MIL}(x)$) usually has a different release time and deadline, and is necessary for the completion of other tasks, it is very difficult to determine the actual minimum load imposed by T_{iv} without further simplifications of $MIL(x)$ and $\overline{MIL}(x)$. We have thus taken the following simplifying steps:

SP1. Since only those modules which are required for T_{iv}'s completion are of interest to us, those modules unnecessary for T_{iv}'s completion are excluded from $MIL(x)$ and $\overline{MIL}(x)$.

SP2. $MIL(x)$ and $\overline{MIL}(x)$ of SP1 are then treated as two independent single modules with the following execution times $E^{k}_{iv}(x)$ and $\overline{E}^{k}_{iv}(x)$, respectively:

$$E^{k}_{iv}(x) = \sum_{a} e^{k}_{ia} + \sum_{\substack{T_j \in B_h(x),\\ h \neq k}} \left[\chi^{k}_{ij,i}(1) + \chi^{k}_{ji,i}(1)\right] + \sum_{\substack{T_j \in B_k(x) \text{ or}\\ T_j \in \overline{B}(x)}} \left[\chi^{k}_{ij,i}(0) + \chi^{k}_{ji,i}(0)\right], \tag{4.3}$$

$$\overline{E}^{k}_{iv}(x) = \sum \left\{ \left[\chi^{k}_{ji,j}(1) - \chi^{k}_{ji,j}(0)\right] + \left[\chi^{k}_{ij,j}(1) - \chi^{k}_{ij,j}(0)\right] \right\}. \tag{4.4}$$

$$\overline{E}^{k}_{iv}(x) = \sum_{T_j \in B_k(x)} \left\{ \left[\chi^{k}_{ji,j}(1) - \chi^{k}_{ji,j}(0)\right] + \left[\chi^{k}_{ij,j}(1) - \chi^{k}_{ij,j}(0)\right] \right\}.$$

Note that the three terms of the RHS of Eq. (4.3) correspond to the three parts of $MIL(x)$ mentioned above.

It can be seen that if $MIL(x)$ ($\overline{MIL}(x)$) of SP1 together with $TG\,1(x)$ generates a lower-bound of $\Theta^{*}(x)$, then so does $MIL(x)$ ($\overline{MIL}(x)$) of SP2 together with $TG\,1(x)$, because the single-module representation of $MIL(x)$ ($\overline{MIL}(x)$) has less precedence and timing constraints than its counterpart of SP1. Even though each of $MIL(x)$ and $\overline{MIL}(x)$ of SP2 is needed for

T_{iv}'s completion, they are not necessarily released at the same time. This is because $\overline{MIL}(x)$ is part of the T_{iv}'s communication partner T_j, whose period is p_j, whereas $MIL(x)$ is part of T_{iv}, whose period is p_i. Therefore, it is still difficult to determine the minimum imposed load by simply comparing $E_{iv}^k(x)$ and $\bar{E}_{iv}^k(x)$. The following step can be used to overcome this difficulty.

SP3. The minimum load $\psi_{iv}^k(x)$ imposed on N_k by T_{iv} is constructed with a) the lumped execution time being min $\{E_{iv}^k(x), \bar{E}_{iv}^k(x)\}$, b) the release time being the minimum of the release times of $MIL(x)$ and $\overline{MIL}(x)$, and c) the cost function $= (t - r_{iv}) / p_i$.

The above $\psi_{iv}^k(x)$ is the minimum load imposed on N_k by T_{iv} in the sense that $\psi_{iv}^k(x)$'s together with $TG1(x)$ generate a lower-bound of $\Theta^*(x)$ regardless whether T_i is assigned to N_k or to a different PN. Using $\psi_{iv}^k(x)$, the minimum load $TG2_k(x)$ imposed by all the tasks in $\bar{B}(x)$ is calculated as $TG2_k(x) = \bigcup_{\substack{1 \le v \le L/p_i \\ T_i \in \bar{B}(x)}} \psi_{iv}^k(x)$. Finally, we calculate

$TG_k(x) \triangleq TG1(x) \cup TG2_k(x)$.

4.3.2. Lower-Bound Cost $\hat{\Theta}(x)$

The lower-bound node hazard of N_k, $\hat{\Theta}_k(x)$, (and thus, $\hat{\Theta}(x) \triangleq \max_{N_k \in N} \hat{\Theta}_k(x)$) can be obtained by applying an optimal scheduling algorithm on the partial task graph $TG_k(x) = TG1(x) \cup TG2_k(x)$. Since the problem of deriving such a $\hat{\Theta}_k(x)$ is NP-hard, some form of approximation is called for. Some of precedence and/or timing constraints on $TG_k(x)$ need to be relaxed in order to derive an approximate polynomial-time algorithm. Several candidate $\hat{\Theta}_k(x)$'s can be obtained depending on the extent to which these constraints are relaxed. For example, in $TG1(x)$, we may ignore timing and/or precedence constraints between PNs and derive a $\hat{\Theta}_k(x)$ as was done in deriving the two lower-bounds in Section

3.4.2. However, both of these two bounds are obtained without considering the completion of any other tasks on different PNs. In what follows, a third method for deriving a $\hat{\Theta}_k(x)$ will be described by considering both task completions on N_k as well as on other PNs. Thus, the resulting $\hat{\Theta}_k(x)$ should be the best among the three.

Since Algorithm A (Section 2.3.1) is to be used for N_k, the main theme in deriving a better $\hat{\Theta}_k(x)$ relies solely on how to embed the effects of those tasks of $TG\,1(x)$ located in PNs other than N_k into the release times and cost functions of those modules of $TG\,1(x)$ located in N_k. The release time of any module m_b of $TG\,1(x)$ (the release times and the cost functions of $TG\,2_k(x)$ were already determined in the last subsection) is relatively easy to obtain because it is the latest task invocation among those having at least a module preceding m_b. For example, M_{24} (and also the dummy $X_{23,2}$) of Fig. 4.5 has a release time of 0 since T_{11}, T_{21} and T_{31} are all invoked at time 0 (see also Fig. 4.2). However, the release time of $X_{32,2}$ is 20 because T_{32} is invoked at time 20.

To derive the cost function for each module of $TG\,1(x)$ located in N_k, the notion of *outgoing communication point* (OCP) needs to be introduced. An OCP of N_k is a node on $TG\,1(x)$ which is located at the tail of a delay module. A module m_b in N_k is called an *OCP module* if $head(m_b)$ is an OCP, i.e., a module whose completion may enable some modules in PNs other than N_k. See Fig. 4.5 for an example. Since n_4 is an OCP of N_1, $X_{12,1}$ is an OCP module. Consider an OCP and a node $n_q = v_z$ on $TG\,1(x)$, i.e., n_q is the node representing the completion of certain task invocation v_z. Assume that v_z has been assigned to a PN other than N_k, where the OCP is located. If the OCP precedes v_z (i.e., n_q), then the *critical path* (it may contain some modules in N_k) of length β_z from the OCP to v_z represents the minimum time to complete v_z after realizing the OCP. Otherwise, the completion time of v_z is independent of the realization time of the OCP.

Let $V(m_b)$ denote the set of all invocations preceded by the OCP, $head(m_b)$. Also, let v_y be a task invocation on N_k, and $f_y(t)$ be the cost function, r_y the invocation time and p_y the period of v_y. The cost function $f_b(t)$ of m_b can now be obtained using the following rules.

R1. If m_b is not a completing module and m_b is not an OCP module then $f_b(t) := 0$.

R2. If m_b is not a completing module and m_b is an OCP module then $f_b(t) := \max_{v_z \in V(m_b)} g_z(t)$, where $g_z(t) = (t + \beta_z - r_z) / p_z$, and β_z is the length of the critical path from $head(m_b)$ to v_z.

R3. If $head(m_b) = v_y$ and m_b is not an OCP module then $f_b(t) := f_y(t) = (t - r_y) / p_y$, the cost function of v_y.

R4. If $head(m_b) = v_y$ and m_b is an OCP module then $f_b(t) := \max \{ f_y(t), \max_{v_z \in V(m_b)} g_z(t) \}$, where $g_z(t)$ as is in $R2$.

Once the release time and cost function of each module on $TG_k(x)$ located in N_k are determined, Algorithm A can be applied to obtain $\hat{\Theta}_k(x)$, and thus, $\hat{\Theta}(x) = \max_{N_k \in N} \hat{\Theta}_k(x)$. Notice that the optimal schedule obtained above is only for N_k in this simplied version of $TG_k(x)$. Also, the computational complexity of deriving $\hat{\Theta}(x)$ is $O(|N|Q^2)$, where $|N|$ is the number of PNs and Q the total number of modules in the system.

4.4. An Example

Consider an example of allocating the three tasks T_1, T_2 and T_3 to two PNs, N_1 and N_2, in Fig. 4.2. Within the planning cycle [0, 40), T_1 and T_2, both with period 40, are invoked only once while T_3, with period 20, is invoked twice. Suppose $q_1^1 = 2$, $q_1^2 = 1$, $q_1^3 = 1$, $q_2^1 = 1$, $q_2^2 = 2$, $q_2^3 = 1$, and $d_{12} = d_{21} = 1$.

Fig. 4.7 shows all the vertices which are numbered in the order of their generation times. The assignment and $\hat{\Theta}(x)$ associated with each vertex x are also indicated in Fig. 4.7. Our B&B algorithm is shown to be quite efficient in this example because only two (out of a total of 8) terminal vertices are generated before an optimal assignment is found. Specifically, vertex 6, with $\hat{\Theta}(6) = 39/40$, is first eliminated as soon as vertex 8 with the exact cost $28/40 \leq \hat{\Theta}(6)$ is generated. Then, all active vertices 4, 5, and 8 are eliminated after vertex 9 is generated since the value of H of this complete assignment is 23.5 / 40 is the smallest. Thus, vertex 9, which assigns all three tasks to N_2, is an optimal assignment, and its optimal schedule derived by Algorithm A is shown in Fig. 4.8. One reason for this counter-intuitive result was already given in Section 4.2; communication modules and delays are the major part of the TG, and thus, assigning all tasks to a single PN to minimize the communication and delays is superior to the others. There is another reason why all tasks are not assigned to N_1. Since T_{32} is invoked at t=20, the length of a path from n_{15} to n_{20} (Fig. 4.2) is critical to the performance of the assignment. If all tasks are assigned to N_1 which has a smaller processing power for T_2 than N_2, then this path will be longer than the optimal solution.

To see how $\hat{\Theta}(x)$ is obtained for a non-terminal vertex x, $\hat{\Theta}(5)$ is computed as follows. Since $B_1(5)=\{T_1\}$, $B_2(5)=\{T_2\}$, $\bar{B}(5)=\{T_3\}$, and $q_1^1 = q_2^2 = 2$, $e_{1a}^1 := e_{1a}/2$, $e_{2a}^2 := e_{2a}/2$, $\forall a$, $\chi_{12,1}^1(1) := \chi_{12,1}(1) / 2$, $\chi_{21,1}^1(1) := \chi_{21,1}(1) / 2$, $\chi_{12,2}^2(1) := \chi_{12,2}(1) / 2$, $\chi_{21,2}^2(1) := \chi_{21,2}(1) / 2$, $\chi_{23,2}^2(0) := \chi_{23,2}(0) / 2$, $\chi_{32,2}^2(0) := \chi_{32,2}(0) / 2$ to accommodate the loads imposed on N_1 and N_2 by the tasks in $B_1(5)$ and $B_2(5)$, respectively. The resulting $TG\,1(5)$ is shown in Fig. 4.9, ignoring T_3 and the communication delays between T_3 and T_2. (Note the differences between this figure and Fig. 4.5.)

To derive $\hat{\Theta}_1(5)$, we need to obtain the minimum load $TG\,2_1(5)$ imposed only by T_3 on N_1 since $\bar{B}(5) = \{T_3\}$. Thus, $TG\,2_1(5) = \{\psi_{31}^1(5), \psi_{32}^1(5)\}$ since T_3 is invoked twice in any planning cycle. However, a further examination on the original TG shows that there is no

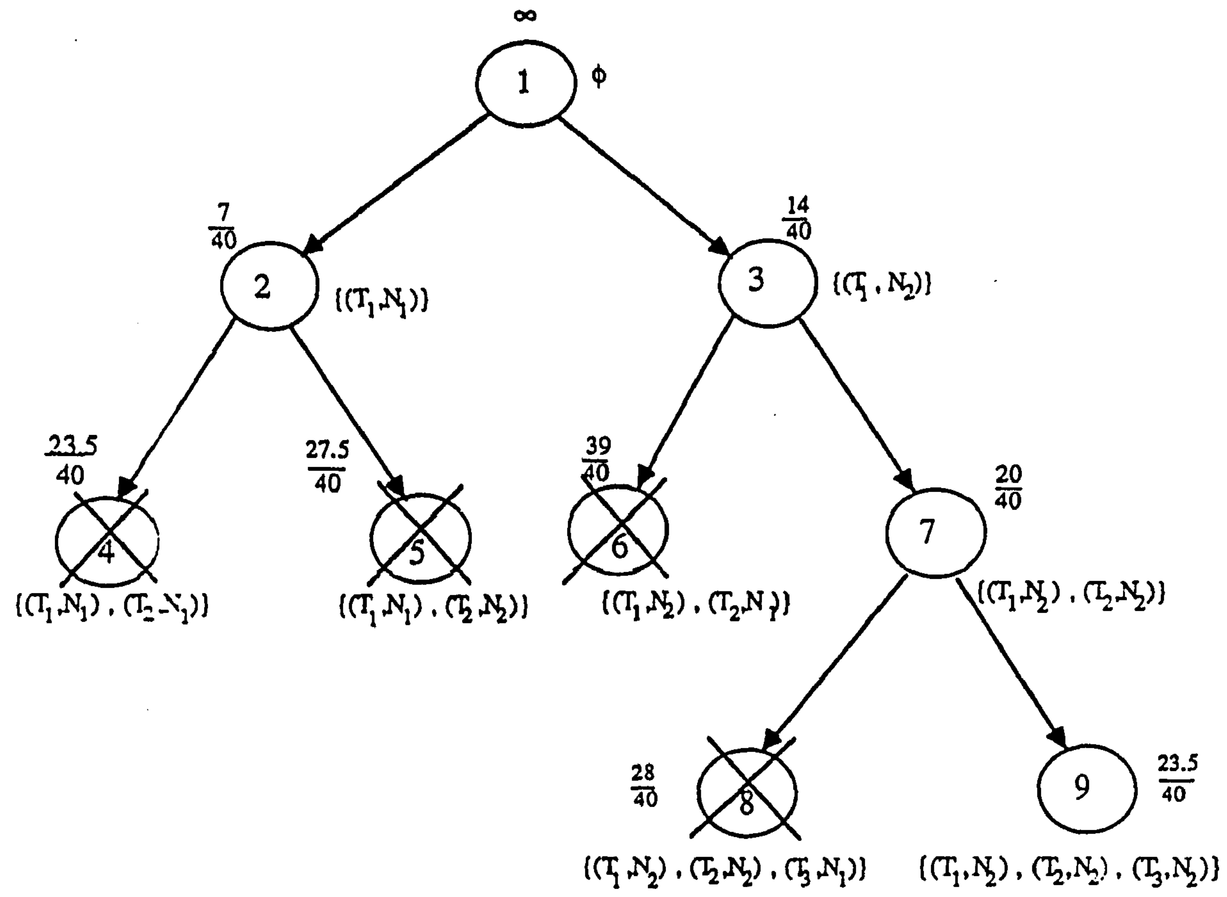


Figure 4.7. Search Tree Generated for the Numeric Example

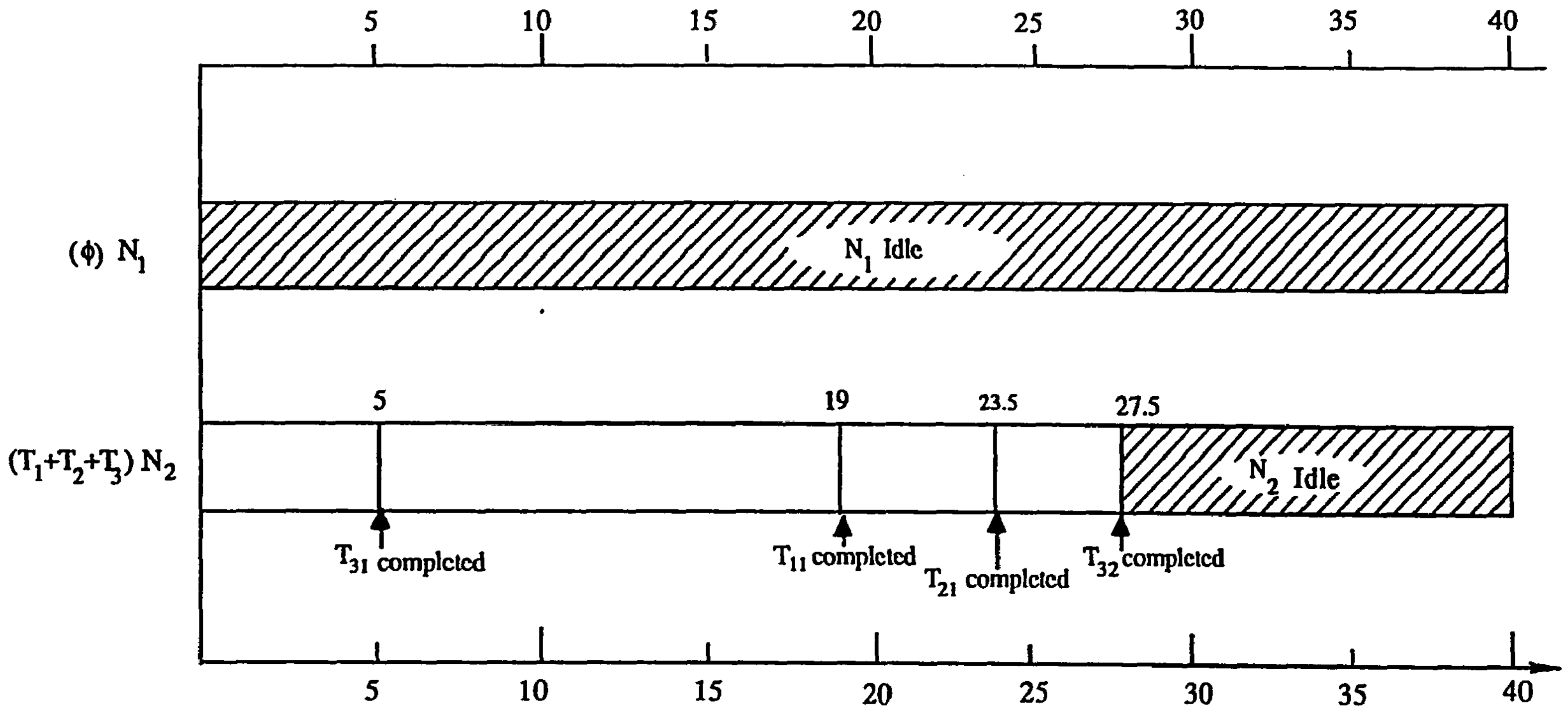


Figure 4.8. An Optimal Schedule $x = \{(T_1, N_2), (T_2, N_2), (T_3, N_2)\}$

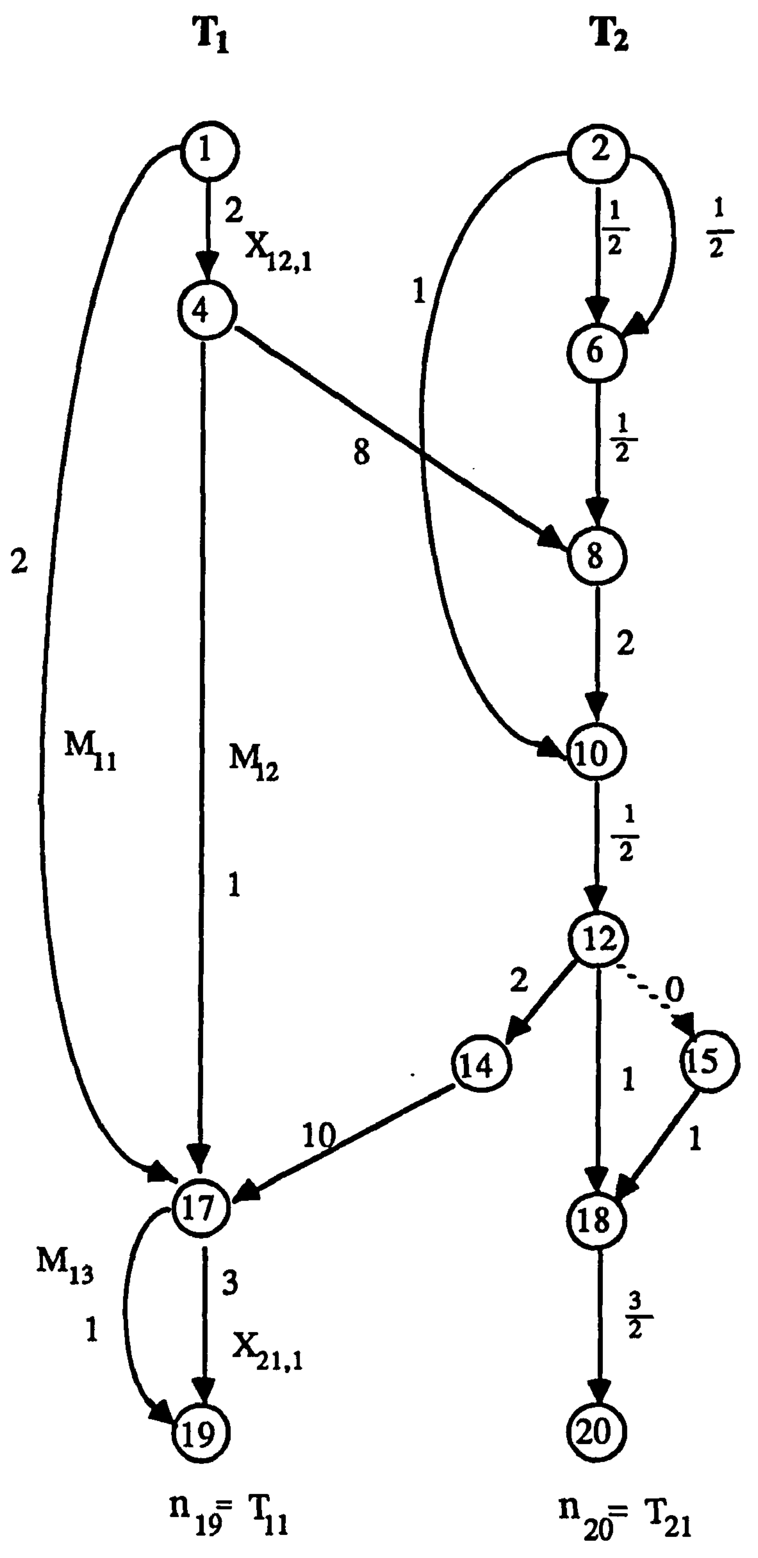


Figure 4.9. TG1(x) for x =$\{(T_1, N_1), (T_2, N_2)\}$

extra load imposed on N_1 by T_3, if T_3 is not assigned to N_1, because T_3 does not communicate with T_1. It follows that, although assigning T_3 to N_1 will impose some load on N_1, we need not consider T_3 to derive $\hat{\Theta}_1(5)$. That is, $\hat{\Theta}_1(5)$ can be obtained by scheduling only $TG_1(5) = TG1(5)$. The release time and cost function of each module in T_1 are determined as follows. Since no modules of T_1 are preceded by T_{32}, each of them has a release time of $t = 0$. To find the cost functions, rules R1-R4 of the last section are followed. Specifically, from R3 the cost functions of M_{13} and $X_{21,1}$ (Fig. 4.9) is $t/40$, which is identical to that of T_1. From R1, the cost functions of M_{11} and M_{12} are 0. Finally, the cost function of $X_{12,1}$ is determined by considering the critical paths of lengths β_1 and β_2 from n_4, which is an OCP, to n_{19} and n_{20}, respectively. That is, from R2, the cost function of $X_{12,1}$ is max $\{g_1(t), g_2(t)\} = g_1(t)$, where $g_1(t) = (t + \beta_1 - 0) / 40 = (t + 25.5) / 40$ and $g_2(t) = (t + \beta_2 - 0) / 40 = (t + 13) / 40$. As a result of applying Algorithm A, $\hat{\Theta}_1(5) = 27.5/40$.

To derive $\hat{\Theta}_2(5)$, we need to obtain the minimum load $TG2_2(5) = \{\psi^2_{31}(5), \psi^2_{32}(5)\}$ imposed by T_3 on N_2 when T_3 is and is not assigned to N_2. If T_3 is assigned to N_2, from Eq. (4.3), $E^2_{31}(5) = 4$ ($E^2_{32}(5) = 5$) since all modules of T_3 contributing to the summation are required for T_{31}'s (T_{32}'s) completion. On the other hand, if T_3 is not assigned to N_2, the extra loads imposed on N_2 (Eq. (4.4)) are $\bar{E}^2_{31}(5) = (2 - 1)/2 = 0.5$ and $\bar{E}^2_{32}(5) = (2 - 0)/2 = 1$ since the first (second) $X_{32,2}$ is not required for T_{31}'s (T_{32}'s) completion. Following SP3 in the last section, we derive the $\psi^2_{31}(5)$'s execution time as min $\{4, 0.5\} = 0.5$, release time as 0, and cost function as $(t - 0)/20$. $\psi^2_{32}(5)$ can be derived similarly to $\psi^2_{31}(5)$ except for its release time. Specifically, the execution time is derived as min $\{5, 1\} = 1$, cost function $= (t - 20) / 20$, and release time as 0 since $X_{23,2}$ is executable before T_{32} is invoked. In order to use Algorithm A for deriving $\hat{\Theta}_2(5)$, the release time and cost function of each module in N_2 on $TG1(5)$ also need to be determined. As a result of scheduling the simplified version of $TG_2(5)$, we get $\hat{\Theta}_2(5) = 20/40$. Therefore, $\hat{\Theta}(5)$ = max

$\{\hat{\Theta}_1(5), \hat{\Theta}_2(5)\} = \hat{\Theta}_1(5) = 27.5/40.$

CHAPTER 5

MODELING OF CONCURRENT EXECUTION OF TASKS

5.1. Introduction

In the previous chapters, task execution times were assumed to be fixed so that optimal scheduling and allocation of the tasks can be properly handled. However, in practical systems, task execution times may be random. In such a case it is virtually impossible to guarantee that each task be completed before its deadline under any task allocation. Therefore, modeling concurrent execution of tasks, termed *task system modeling*, is an essential step in the analysis and design of distributed real-time systems and is the main purpose of this chapter.

Conventional task modeling approaches either consider each task as a basic unit or decompose a task into smaller units, called *modules*. The former is called a *task-oriented* model [Hua85, Vir85], whereas the latter is called a *module-oriented* model [Mok83, ChL84]. Most of these models, however, have some of the following disadvantages:

- The task-oriented model is too coarse. Most tasks communicate with each other during the course of execution, and inter-task communications impose complex precedence constraints which are usually difficult to analyze [Hua85].
- The inter-task or inter-PN communication delay is either assumed to be constant or have a fixed probability distribution. It is impossible to describe the delay under, for example, different communication protocols, or different task scheduling or message handling policies adopted by communication partners.

- It is not easy to describe which task stage each PN has been executing. This information is essential when we need to dynamically prioritize the execution of some part of the task system for each PN.

In order to remedy the above shortcomings, we present a module-oriented model with a finer granularity than the conventional module-oriented models. Our modeling process consists of the following two steps:

(1) Contiguous stretches of executable code of the task system are combined into a set of basic entities called *activities*. A set of activities is formed in such a way that any inherent precedence constraint within the task system and the expected task execution times are preserved.

(2) A Generalized Stochastic Petri Net (GSPN) [MBC84] is used to model the activities and their precedence constraints. These activities are then modeled by a sequence of Continuous-Time Markov Chains (CTMC's) [Kle75, Ros83] by performing *reachability analysis* on the GSPN and assuming independently, exponentially distributed transition firing delays[1].

The *state* of each CTMC describes the task execution stage each PN (which could be a multi-processor) is in, and a state transition corresponds to the execution of an activity. As will be seen later, the use of a sequence of CTMC's is to facilitate *time-driven* task invocations. The CTMC model offers a useful base on which various problems, such as task response time estimation, optimal message handling policy, and optimal time-out policy, can be rigorously studied.

[1]The set of transitions that can be fired is time dependent, however.

The rest of this chapter is organized as follows. Section 5.2 describes the task system and states some simplifying assumptions. Section 5.3 introduces necessary concepts, definitions and notation. The problem statement and the proposed modeling process are presented in Section 5.4. Finally, the probability of missing deadlines is computed in Section 5.5 to demonstrate the use of our model.

5.2. The Task System

Only periodic tasks are considered here since they are not only the nucleus but also the unique feature of real-time tasks.

As mentioned earlier, most periodic tasks communicate to accomplish the overall control mission. The communication between two cooperating tasks usually establish precedence constraints between them. These constraints specify the completion of some parts of one task to enable some parts of the other task to be ready for execution. As we shall see later, these precedence constraints may even form a chain of *cyclic* precedence relations. To identify a set of tasks with precedence constraints, all tasks in the system can be partitioned into mutually exclusive sets of tasks called *precedence classes*. Two tasks are in the same class if and only if they communicate, directly or indirectly, with each other.

For the purpose of modeling, we need the following assumptions:

A1. At any given time, all resources in the system are dedicated to a single control mission.

A2. Tasks are pre-assigned to the PNs, and remain unchanged throughout the control mission lifetime.

A3. Each aperiodic task alone forms a single precedence class.

A1 indicates that no other control mission can begin execution before the current one is completed. This assumption excludes a rare and more complex case where more than one control mission simultaneously compete for system resources. A2 is usually the case in practice due to the time overhead associated with on-line assignment of tasks. A3 is to simplify the treatment of the task system. Specifically, if each aperiodic task alone forms a distinct class, by definition, it does not communicate with any other task and, thus, imposes no precedence constraints in the task system. Notice that the above assumptions do not exclude the case where the periodic tasks may form more than one precedence class.

To analyze normal system behavior, a planning cycle $I = [0, L)$ of the task system is identified such that the task system can be fully characterized in I, where L is the least common multiple (LCM) of $\{p_i \mid i = 1, 2, \cdots, m\}$, p_i the invocation period of T_i and m the total number of periodic tasks in the task system. The example in Fig. 5.1 shows a system with 3 tasks T_1, T_2 and T_3, with $p_1 = 6$, $p_2 = 3$ and $p_3 = 4$ time units. Within $I = [0, 12)$, T_1, T_2 and T_3 are invoked 2, 4 and 3 times, respectively.

It is also assumed that no pipelining of tasks is allowed. The current invocation of a task must be completed before its next invocation; if a task is not completed prior to its next invocation, it is simply discarded. Since the process under control may have changed, for example, its sensor values by the time of the next invocation of a task, there is no need to execute a previous invocation of the task with obsolete sensor data.

5.3. Definitions and Notation

This section introduces concepts, definitions and notation that will be used throughout this chapter. The *task flow graph* (TFG), and its *task tree*, which serve as the input to our modeling process, are first presented and then followed by a discussion of the combination and expansion processes which deal with modules.

A TFG describes a task to be executed by a PN in the distributed system. The TFG is composed of four types of subgraphs: *chain, And-Fork to And-Join, Or-Fork to Or-Join,* and *loop*.

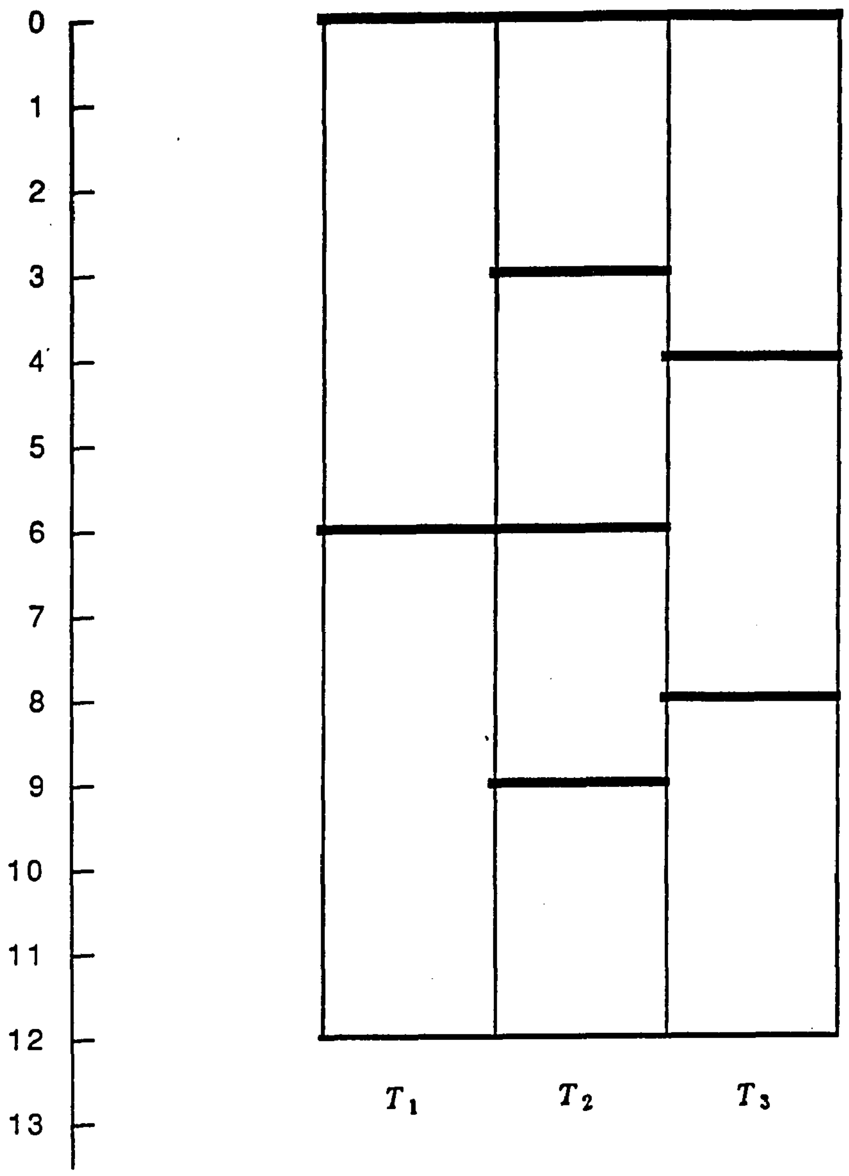


Figure 5.1. A Planning Cycle of a Task System with 3 Tasks

A chain (Fig. 5.2) is the largest possible concatenation of multiple entities called *execution objects*. A *communication point* is an execution object which represents one of the six communication primitives (to be elaborated on in Section 5.4) used in the system or an output signal sent to the processes under control. A sequential code stretch or a communication point is called a *basic* execution object, and a single execution object, which is not a chain by definition, is called a *stand-alone* execution object. A stand-alone execution object may be a basic execution object, an And-Fork to And-Join subgraph, an Or-Fork to Or-Join subgraph, or a loop.

An And-Fork to And-Join subgraph (or simply called an *And-Subgraph*) (Fig. 5.3) consists of more than one branch, all of which must be executed (possibly in parallel). A branch of the And-Subgraph may be a stand-alone execution object or a chain.

Similarly, an Or-Fork to Or-Join subgraph (or simply called an *Or-Subgraph*) (Fig. 5.4) consists of more than one branch. However, one and only one branch of the Or-Subgraph is executed, and the probability of choosing each branch is assumed to be given. Another point that differentiates an Or-Subgraph from an And-Subgraph is that a branch of the Or-Subgraph could contain no execution object at all.

A loop (Fig. 5.5) consists of a loop body with a ''looping back'' probability p. Like a branch of the And-Subgraph, the loop body may be a stand-alone execution object or a chain.

Also shown in Figs. 5.2 thru 5.5 are the upper-end and the lower-end points of a chain, the fork and join points of an And-Subgraph and an Or-Subgraph, and the collecting and branching points of a loop. Since these points serve as boundaries of their respective subgraphs, they are called *structure points*. Note that there is no structure point defined for a basic execution object, and that there may or may not be structure points for a stand-alone execution object, depending on the specific execution object it represents.

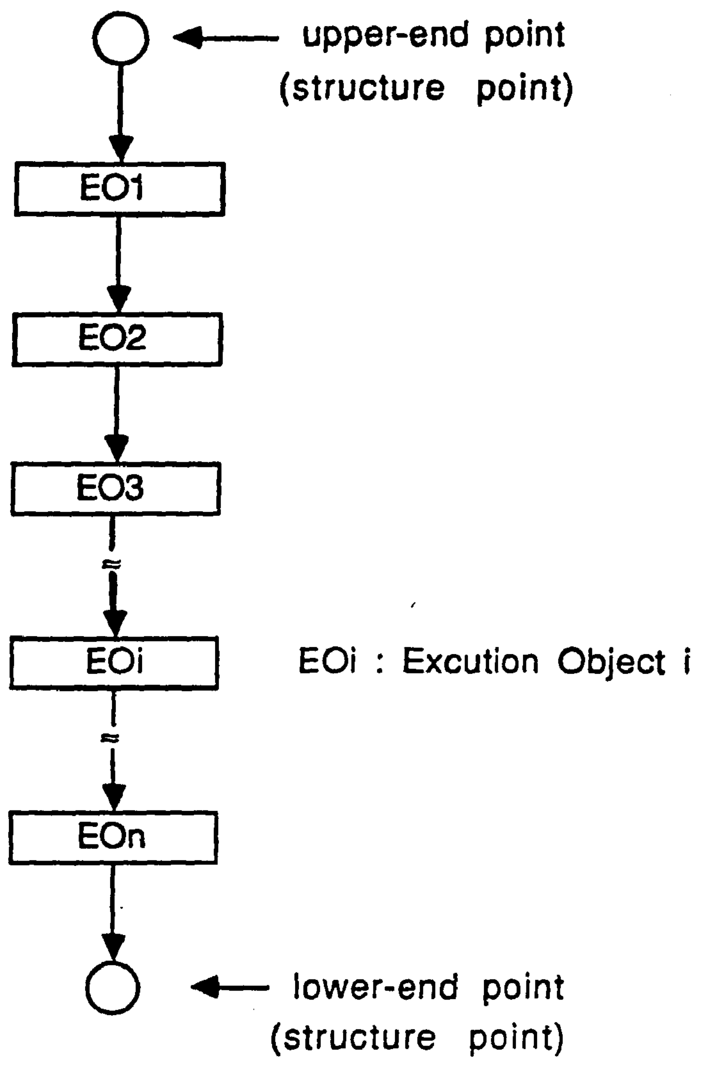
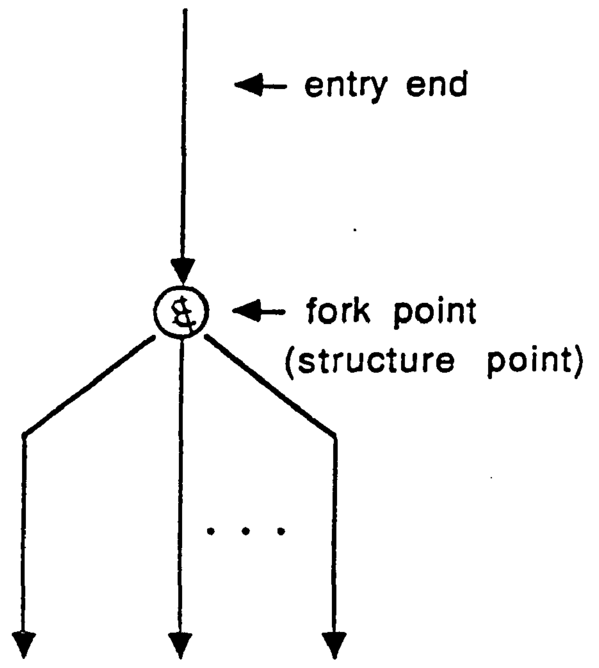
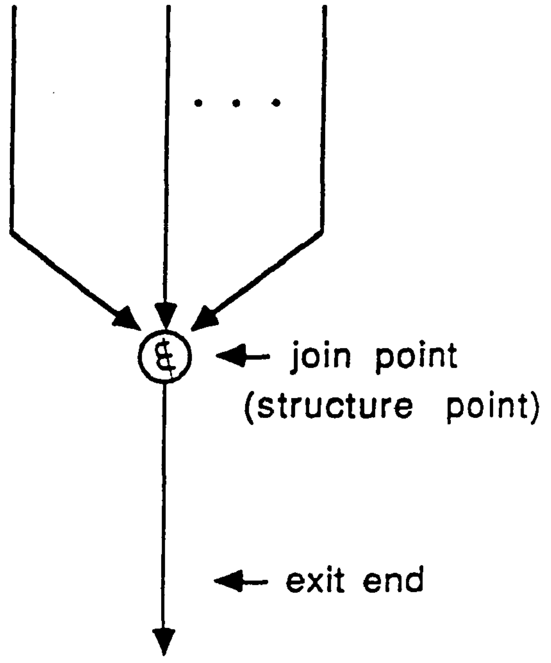


Figure 5.2. A Chain

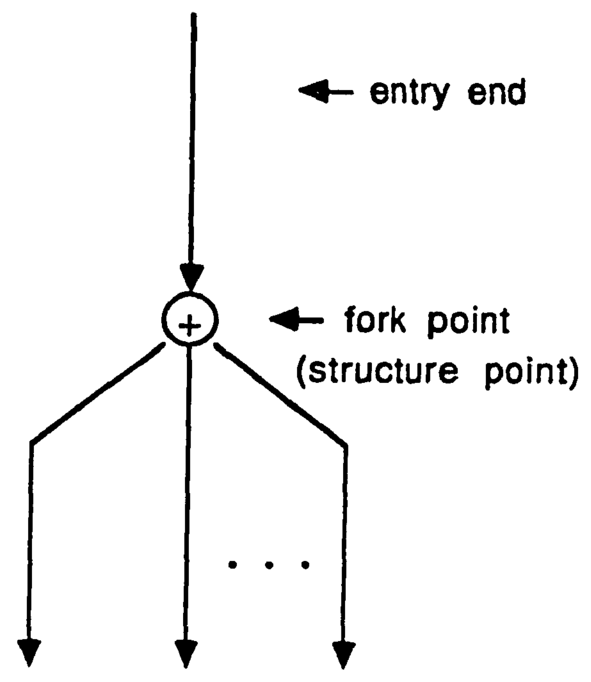


5.3(a). And-Fork

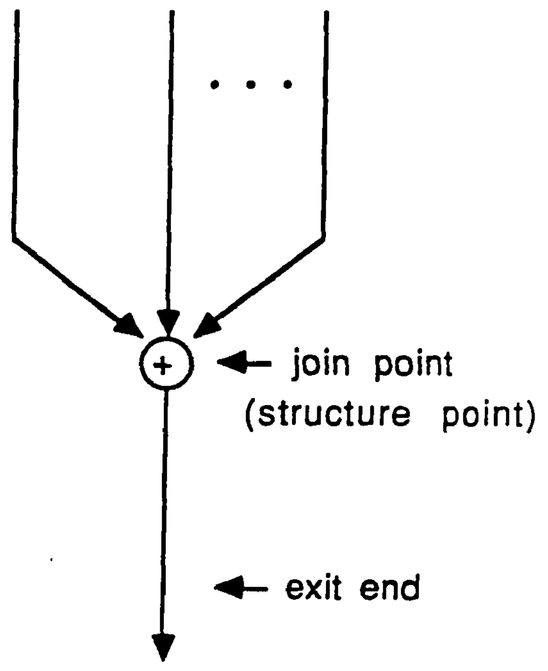


5.3(b). And-Join

Figure 5.3. An And-Subgraph



5.4(a). Or-Fork



5.4(b). Or-Join

Figure 5.4. An Or-Subgraph

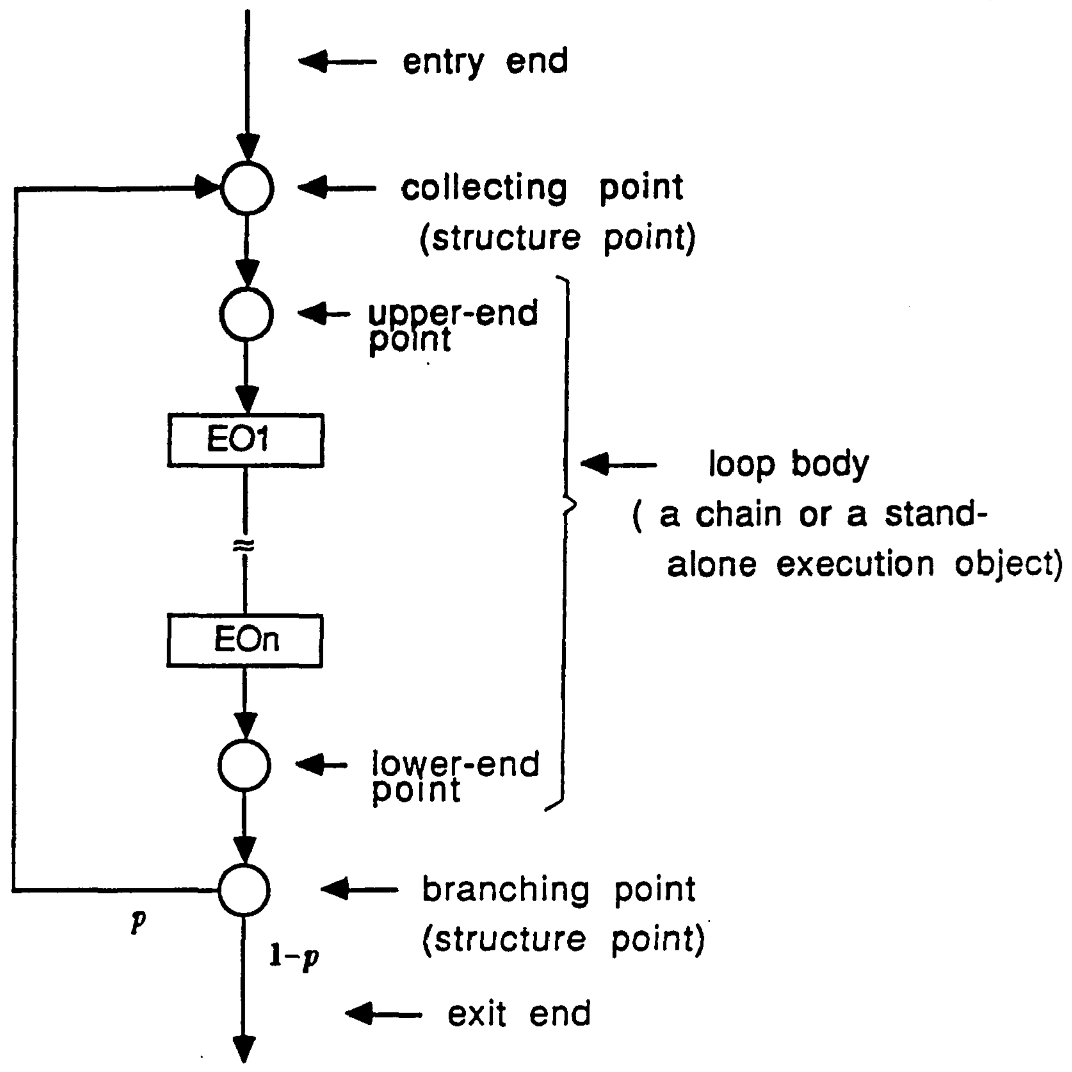

entry end
collecting point (structure point)
upper-end point
EO1
EOn
loop body (a chain or a stand-alone execution object)
lower-end point
branching point (structure point)
p
1-p
exit end

Figure 5.5. A Loop

A task tree describes how the TFG is organized by the four types of subgraphs and the basic execution objects mentioned above. The root of the task tree is the TFG itself, the leaves are the basic execution objects while the non-leaf nodes are the four types of subgraph. A *layer number* is a non-negative integer assigned to each node in the task tree to indicate the relative position of the node in the tree and, consequently, of the subgraph and the basic execution object in the TFG. A higher layer number is assigned to a node of an inner layer, while a lower number is assigned to a node of an outer layer, and the lowest layer number, 0, is assigned to the node of the outermost layer, the TFG itself.

For example, Fig. 5.6 is a TFG whose task tree is shown in Fig. 5.7. The TFG in Fig. 5.6 as a whole is a chain which consists of two execution objects: E1 and an And-Subgraph. The And-Subgraph has three branches: the first is a stand-alone (and basic) execution object E2, the second is a stand-alone execution object (a loop), while the third is a chain which contains two execution objects: E5 and an Or-Subgraph. The loop body in the second branch is also a chain consisting of two basic execution objects E3 and E4. Since E1 and the And-Subgraph are one layer "inside" the chain (or the chain is said to be one layer "outside" E1 and the And-Subgraph) representing the TFG itself, each of them is assigned the layer number 1. Likewise, each of the three branches of the And-Subgraph is one layer inside the And-Subgraph, so each of the branches is assigned layer number 2, and so on. Note that an outer layer node contains all inner layer nodes under itself. For example, the chain on the third branch of the And-Subgraph contains E5, the Or-Subgraph and, thus, E6 and E7.

In a chain, an execution object is not ready for execution until the one immediately preceding it has been completed. In an And-Subgraph, however, no precedence constraints are needed among its branches. Because of the constructs of an Or-Subgraph and loop, it is not possible to use a relation such as *partial ordering* to describe the precedence constraints between any two basic execution objects.

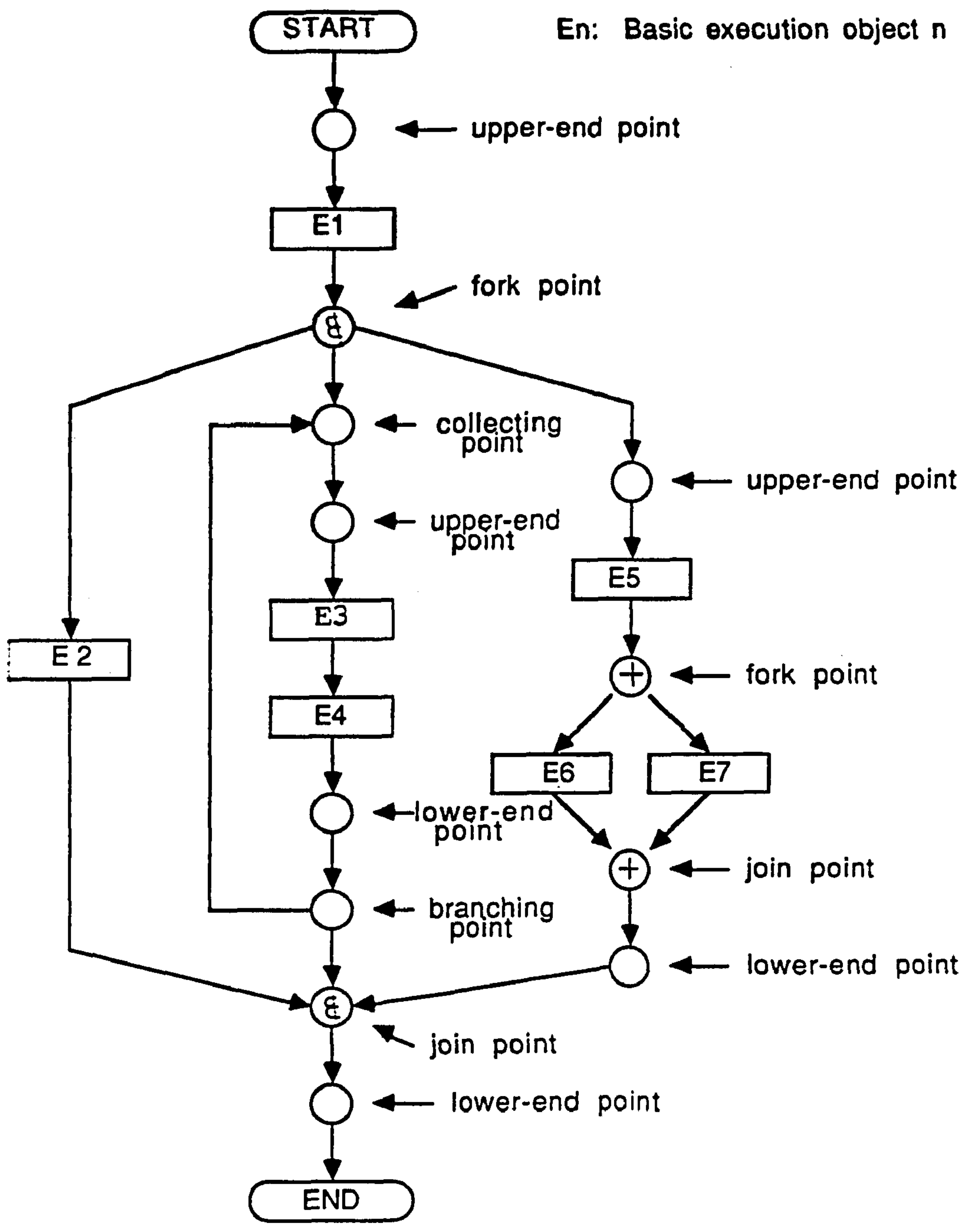


Figure 5.6. A Task Flow Graph (TFG)

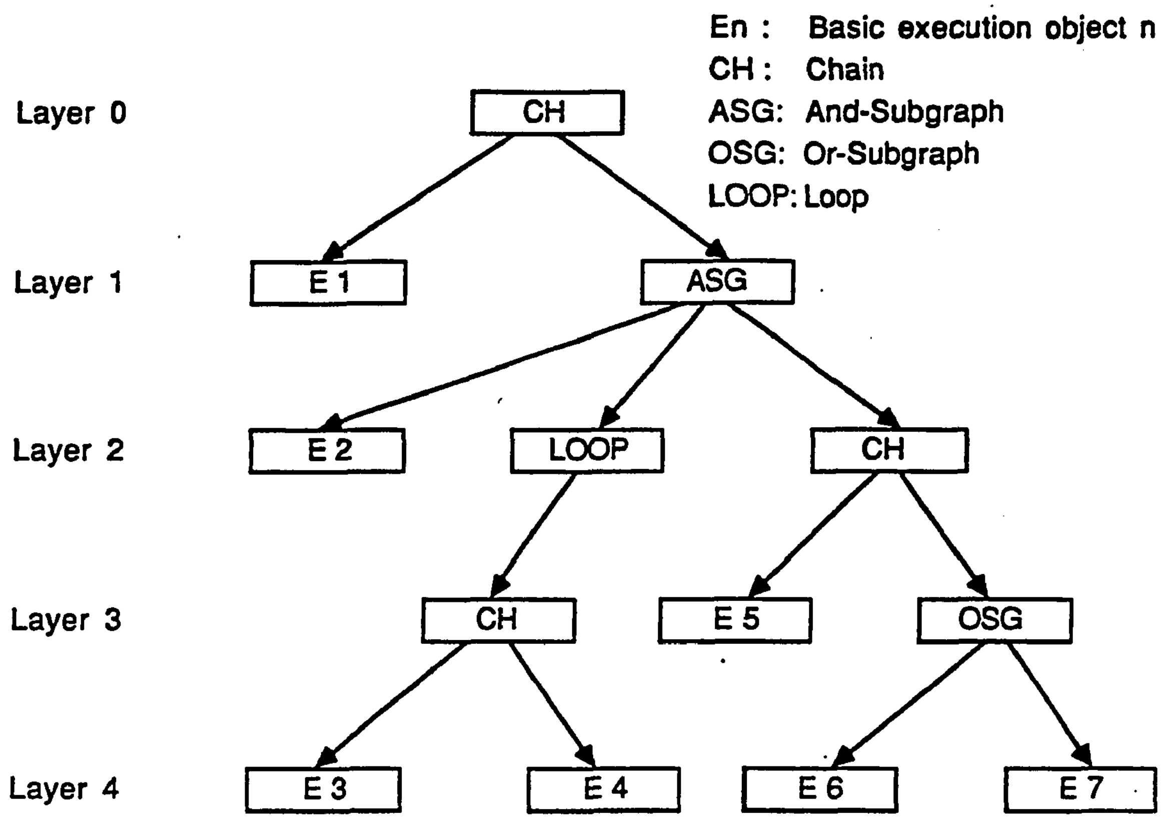


Figure 5.7. The Task Tree for the TFG in Figure 5.6.

A *module* is an entity resulted from combining a set of two or more code stretches or modules.[2] A combination is said to *retain the precedence constraints* among the original stretches if and only if the combined set belongs to one of the two types: 1) a set of contiguous modules on a chain, and 2) a set of branches of a single module of an And-Subgraph or Or-Subgraph. All combinations to be discussed here are of this type. When a set of modules is combined into a module e, e is said to be an *equivalent* module of the combined set. The largest module that can be formed without violating any precedence constraint is called an *activity*. Depending on how far the combination process can go in the TFG, the boundary of an activity may or may not be a communication point. The boundary of an activity which is not a communication point is called a *control point*. After all the activities are found, the resulting graph is called a *Communication Flow Graph* (CFG). More on this will be discussed in Section 5.4.

We have introduced the TFG, its task tree and the combination process which are necessary tools for our modeling process. In what follows, a brief discussion on a GSPN [MBC84] and related definitions necessary to our modeling process is given.

A Petri Net [Pet81] is a four-tuple, $C = (P, T, I, O)$, where $P = \{\phi_1$[3]$, \phi_2, \ldots, \phi_p\}$ is the set of places, $T = \{t_1, t_2, \cdots, t_p\}$ the set of transitions, $I : T \rightarrow P$ the input function, and $O : T \rightarrow P$ the output function. A marking $\beta : P \rightarrow I^+$ represents the number of tokens for each place $\phi_i \in P$, where I^+ is the set of non-negative integers. $M = (P, T, I, O, \beta)$ is a Petri Net with a marking β. After being enabled, a transition fires by removing one token from each of its input places and adding one token to each of its output places.

[2]This is a recursive definition. Further, we will use the term "module" to refer to both the code stretch and module in the rest of the chapter.

[3]We use ϕ_i, instead of the more frequently used p_i, to represent the i-th place of a Petri Net to avoid confusion with the period of the i-th periodic task.

A GSPN is a type of marked Petri Net, where a non-negative random variable representing the firing delay is associated with each transition. Depending on its firing delay, each transition of the GSPN can be classified into one of the two types: *immediate* and *timed.* A transition is immediate if its firing delay is zero with probability one, and timed otherwise. A place p_i in the GSPN is said to be *instantaneous* if a transition with ϕ_i as a sole input place is immediate, and *non-instantaneous* otherwise. The *state* of the GSPN is the marking of the set of all non-instantaneous places. A state is *vanishing* if it enables at least one immediate transition and *tangible* otherwise.

The procedure to find the reachability set S of states from some initial marking μ is called the *reachability analysis* of the GSPN. A non-instantaneous place ϕ_i of a GSPN with an initial marking β is *safe* if $\forall \beta' \in S$, $\beta'(\phi_i) \leq 1$. A GSPN is safe if every non-instantaneous place in the net is safe. As will become clearer, all GSPN's considered in this chapter are safe. The remaining part of this section will deal with the CTMC model describing the task system.

Let N be the set of all PNs in the system, each of which is assigned *a priori* a set of tasks to be executed, and $Z(k)$ be the number of processors of $N_k \in N$. Assuming that, in a planning cycle of the task system, tasks are invoked at times $\omega_1, \omega_2, \cdots, \omega_l$, where $\omega_1 = 0$, the beginning of the cycle, and $\omega_1 < \omega_2 < \cdots < \omega_l < \omega_{l+1} = L$, a sequence of CTMC's, $\{(S_u, \Lambda_u, \Theta_u) \mid u = 1, 2, \cdots, l\}$, is necessary to model the task system.

State space S_u is the set of states of the CTMC model reachable during time interval $I_u = [\omega_u, \omega_{u+1})$. Then, $S = \bigcup_{u=1}^{l} S_u$ is the *total* state space.

$\Lambda_u : S_u \times S_u \rightarrow T$ is the event-driven transition function between two states in S_u. Each element in T is a triplet (μ, ξ, k), where $\mu \geq 0$ is the transition rate whose inverse represents the activity execution time, ξ the prespecified branching probability, and k the PN to execute the activity. Let μ_{ij}, ξ_{ij}, and k_{ij} denote the transition rate, the branching probability, and the PN associated with states s_i and s_j, respectively. A transition with $\mu_{ij} = 0$ is trivial, meaning no transition between these two states. A set of non-trivial transitions from s_i to some other states s_j such that $\sum_j \xi_{ij} = 1$ is called a *branching set*, and each transition in the set with $\xi_{ij} < 1$ is called a *branching* transition. On the other hand, a transition with $\xi_{ij} = 1$ is called a *lone* transition. As will be seen in Section 5.4, a branching set results from either an Or-Fork or a loop.

A set of *event-driven* transitions associated with $s_i \in S_u$ and $N_k \in N$ is defined as: $OUT(s_i, k) = \{(\mu_{ij}, \xi_{ij}, k) \mid s_j \in S_u, \mu_{ij} \neq 0\}$, the set of all nontrivial transitions from s_i to be fired in N_k during I_u. Given that the task system is in s_i, the number of transitions, $|OUT(s_i, k)|$[4], may be larger than the number of processors, $Z(k)$, for some $N_k \in N$. Therefore, a decision must be made as to which activities/transitions will be chosen[5] to fire. An *activity selection policy* π specifies this choice for each $N_k \in N$ and for each $s_i \in S$. Let D_i denote the set of all decisions available at s_i. The *policy space* Π is defined as the set of all such policies, i.e., $\Pi = \{\pi\} = D_1 \times \cdots \times D_i \times \cdots \times D_{|S|}$, where $|S|$ is the cardinality of S.

Given the policy π, the set of transitions chosen for concurrent firing at s_i is called an *active transition set*, ATS_i^π. Note that ATS_i^π is a subset of $OUT(s_i, *) = \bigcup_{N_k \in N} OUT(s_i, k)$.

$\Theta_u : S_u \rightarrow S_{u+1}$ is the *time-driven* transition function. All transitions specified by Θ_u occur at time ω_{u+1} and take no time to complete the transitions. Specifically, Θ_u specifies which state in S_{u+1} to start with at time ω_{u+1} for each state in S_u upon some task invocations.

[4]Each branching set as a whole is counted as a single transition.

[5]If $Z(k) \geq |OUT(s_i, k)|$, then all elements in $OUT(s_i, k)$ are chosen.

5.4. The Modeling Process

Given the Task Flow Graphs (TFG's) and their task trees for each PN, our objective is to model the task system with a sequence of CTMC's such that:

C1. the resulting CTMC model has as small a state space as possible,

C2. the precedence constraint between any two basic execution objects in the original TFG is retained, and

C3. the expected execution times for each task invocation and for the task system as a whole are preserved in the resulting CTMC model.

C1 is concerned with optimality, while C2 and C3 are constraints. Since the size of the total state space S is generally an exponential function of the number of activities, modules should be combined, whenever possible, to build the smallest set of activities while satisfying C2 and C3. This combination may cause the following two problems: P1) loss of potential parallelism, and P2) dependency among resulting activities. P1 is due to the combination of modules in an And-Subgraph. This is acceptable since analysis based on such a combination will err on a conservative side in meeting hard deadlines. P2 is caused by the combination between modules inside and outside an Or-Subgraph. A combination of this kind merges an outside module with a module on each branch of the Or-Subgraph, and, thus, introduces dependency between the resulting modules on any two branches. This calls for the use of a stochastic process with mutually dependent transition delays. However, since such a combination could drastically reduce the size of state space, we choose to approximate the model by ignoring this effect.

The overall procedure to build the proposed CTMC model of the task system is divided into three sequential stages: building the smallest activity set of a TFG, constructing a GSPN of concurrent task execution, and deriving the CTMC model from the GSPN.

5.4.1. The Smallest Activity Set

Given the TFG's and their task trees for each PN, the set of activities of each TFG is determined by identifying their boundaries. We propose a procedure consisting of N iterations of *combination* and *expansion* phases, where N is the largest layer number in the task tree of interest. The combination phase merges modules within subgraphs of the same layer while the expansion phase penetrates the boundaries of the subgraphs being worked on and performs some preprocessing for the next combination phase.

Starting from the second innermost (i.e., layer $N-1$), and working through the outer layers, each iteration works on all subgraphs in the layer until the outermost layer is expanded. The layer which is currently being worked on is called the *active layer.*

5.4.1.1. The Combination Phase

To meet C2, only three types of combinations are allowed: *vertical, horizontal,* and *total* combinations. A vertical combination merges vertically adjacent modules of a chain into a new equivalent module. A horizontal combination operates on two or more branches in an And-Subgraph or an Or-Subgraph. A total combination merges the whole subgraph into a single equivalent module as was done in [ChL84].

A chain can have vertical and total combinations, an And-Subgraph or an Or-Subgraph can have horizontal and total combinations, while a loop can have total combination only. These combinations are performed only in the subgraphs in the active layer of the TFG.

To satisfy C3, the expected execution time of an equivalent module is computed as follows. The expected execution time of the equivalent module after a vertical combination in a chain or the horizontal combination in an And-Subgraph[6] is the sum of those of all the

[6]As mentioned before, potential parallelism is lost if, for example, there are more than two processors available to execute in parallel any two branches to be combined.

component modules combined into it. For the horizontal combination in an Or-Subgraph, the branching probability, p_e, and the expected execution time, $E[T_e]$, of the equivalent module e are determined by the following two equations:

$$p_e = \sum_{i \in G} p_i \tag{5.1}$$

$$E[T_e] = \frac{\sum_{i \in G} E[T_i]p_i}{p_e}, \quad p_e \neq 0, \tag{5.2}$$

where G is the non-empty set of modules combined into the equivalent module e, p_i and $E[T_i]$ are respectively the branching probability and the expected execution time of module[7] i in G. Note that both Eqs. (5.1) and (5.2) also hold for the total combination of an Or-Subgraph. For a loop, however, the expected execution time $E[T_e]$ after the total combination is:

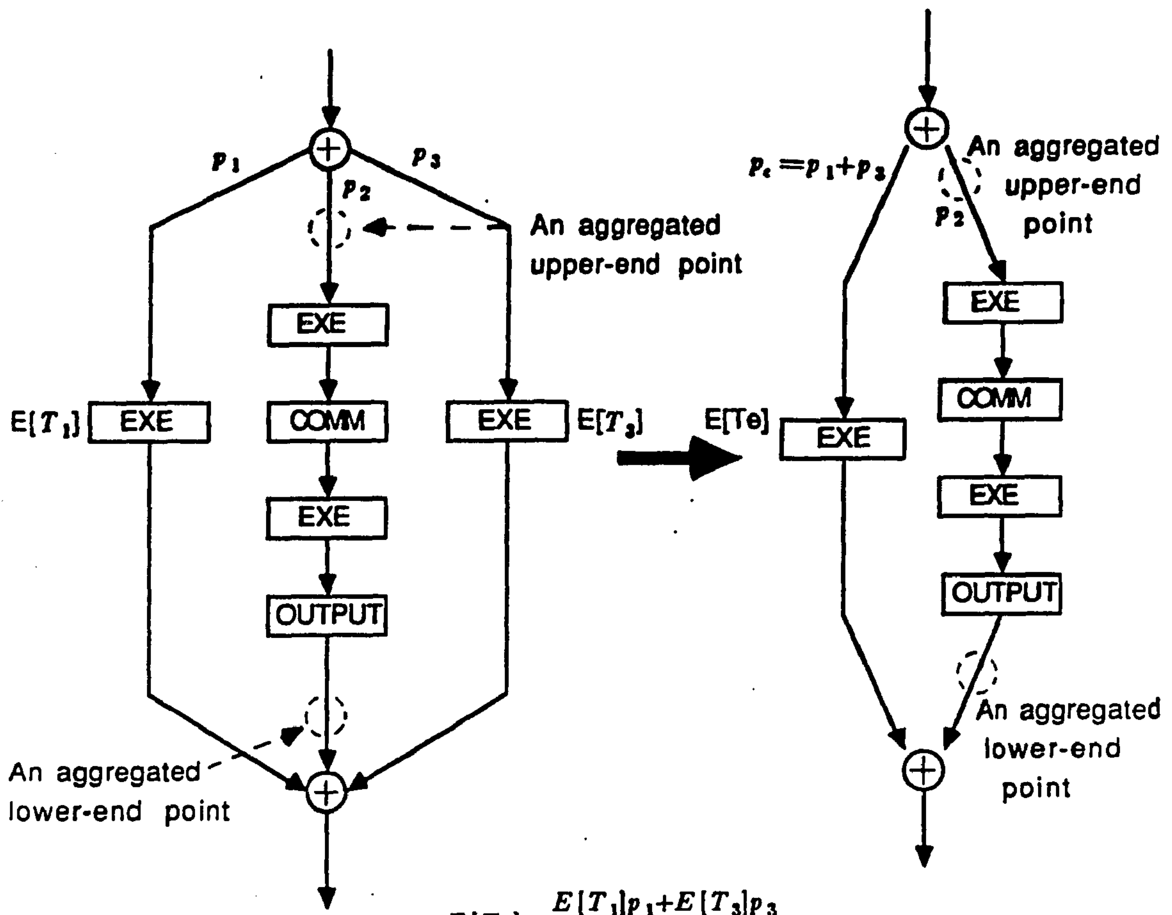
$$E[T_e] = \frac{E[T_l]}{1-p}, \quad p \neq 1, \tag{5.3}$$

where $E[T_l]$ is the expected execution time of the single equivalent module in the loop body.

To facilitate the next phase, if a total combination is performed in any of the four types of subgraph, the subgraph is replaced with its equivalent module, and its two structure points are removed.

For example, Fig. 5.8 illustrates the horizontal combination of an Or-Subgraph, and Fig. 5.9 shows the total combination of a loop. In the figures, $\boxed{\text{COMM}}$ represents a communication primitive, $\boxed{\text{OUTPUT}}$ is the point where an output signal is sent, and $\boxed{\text{EXE}}$ represents a module. Also indicated in Fig. 5.8 are structure points which have been aggregated in the previous expansion phase.

[7]The context is clear enough to avoid confusions on T_i between representing the execution time of module i and representing the i-th periodic task.


p_1
p_2
p_3
An aggregated upper-end point
EXE
$E[T_1]$
EXE
COMM
EXE
$E[T_3]$
EXE
OUTPUT
An aggregated lower-end point
$p_e = p_1 + p_3$
An aggregated upper-end point
p_2
E[Te]
EXE
EXE
COMM
EXE
OUTPUT
An aggregated lower-end point

$$E[T_e] = \frac{E[T_1]p_1 + E[T_3]p_3}{p_e}$$

Figure 5.8. The Combination Process for the Or-Subgraph

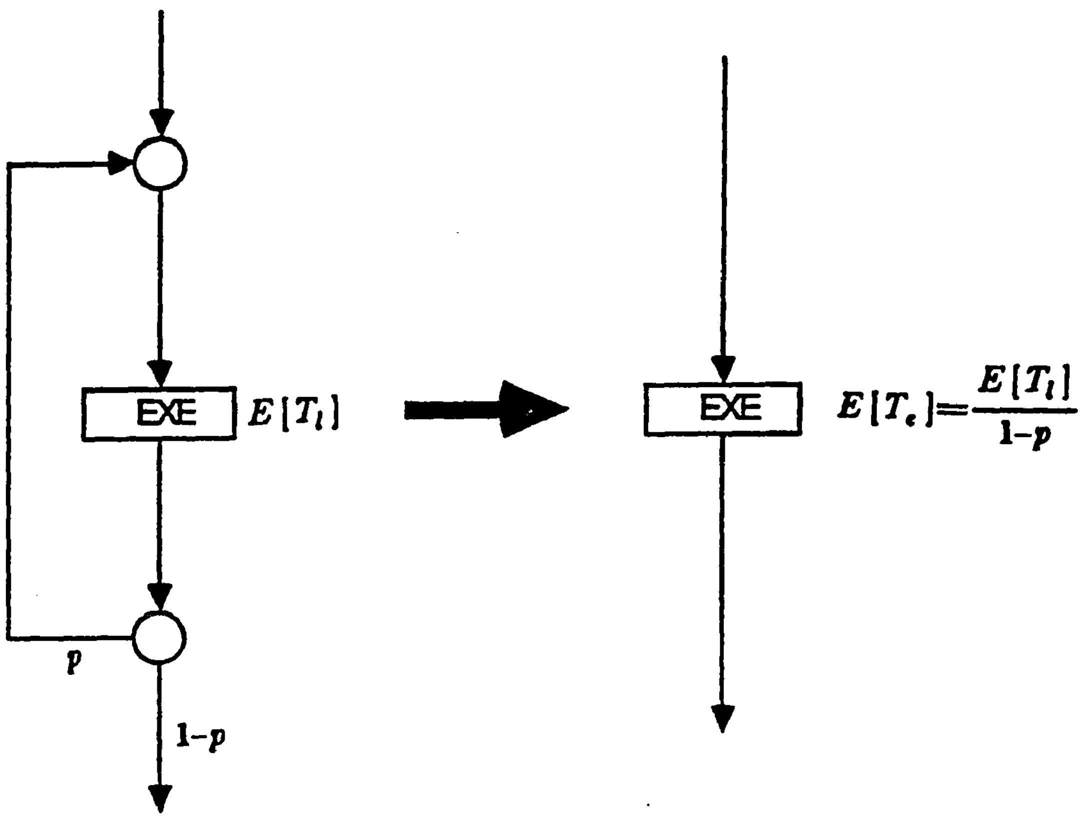


Figure 5.9. The Total Combination of a Loop

5.4.1.2. The Expansion Phase

The expansion phase is to remove the structure points of the subgraphs of the active layer such that the next combination phase can be applied across the original boundaries. This is done by applying two operations: *module migration* and *structure points aggregation*. Modules outside an Or-Subgraph and adjacent to its join point are moved inside the subgraph only if the next combination phase can reduce further the total number of resulting activities. This is the case when there is a module on each branch of the Or-Subgraph ready to be combined with the migrated module. Two adjacent structure points are aggregated into one if the aggregation should decrease the number of instantaneous places in the resulting GSPN model.

While performing the above two operations, control points need to be assigned and communication points to be identified to "stop" further combination on modules located between these points in order not to violate the precedence constraints. Therefore, each of these "stopping points" will serve as the boundary of the two activities, if any, on each side of the point, and no other point in the TFG will serve the same purpose.

The rules for module migration are summarized as follows.

R1. Vertically combine all contiguous modules in the active layer into their equivalent modules as in the combination phase described above.

R2. If a module is outside only one Or-Subgraph whose joint point is adjacent to the module, move the module into the subgraph if there is a module adjacent to the join point on each branch of the Or-Subgraph[8] (Fig. 5.10).

R3. If a module is outside only one Or-Subgraph whose fork point is adjacent to the module, or the module is between two Or-Subgraphs, assign control point(s) to the structure

[8]This is the case that introduces dependency relationship among combined modules on any two branches of the Or-Subgraph.

point(s) associated with the subgraph(s) without moving the module (Fig. 5.11).

R4. For any module that is adjacent to an And-Subgraph or a loop, assign a control point to the adjacent structure point of the subgraph without moving the module (Fig. 5.12).

R5. For any subgraph that is adjacent to a communication point, assign a control point to the adjacent structure point of the subgraph without moving the module.

After module migration is performed, structure points should be aggregated as required to eliminate redundant structure points and prepare for the next combination phase. Apparently, different modeling bases have different modeling efficiency, making the conditions for aggregation different. For example, a structure point regarded as redundant by a more efficient modeling base, such as Stochastic Activity Networks [MoM84], may not be regarded redundant by a less efficient base such as a GSPN,[9] whose modeling efficiency is limited by its execution rules. Since the GSPN cannot efficiently model a logic structure which is more complex than the fork or joint point of an And-Subgraph or Or-Subgraph, a structure point is regarded redundant only if aggregation on this point should result in a logic structure which is logically less complex than AND Fork, AND Join, OR Fork, and OR Join. (Each of these is called an "AND/OR logic").

The notation P1 $\rightarrow$ P2 is used to denote a case of two adjacent structure points, where P1 is immediately followed by P2. Depending on the relative positions of P1 and P2 in the TFG, three different classes can be identified:

- P2 is in the active layer, which is one layer inside the layer of P1.
- P1 is in the active layer, which is one layer inside the layer of P2.

[9]Irrespective of the various modeling bases used, the resulting CTMC models are the same. However, using a more efficient modeling base makes the corresponding reachability analysis more difficult.

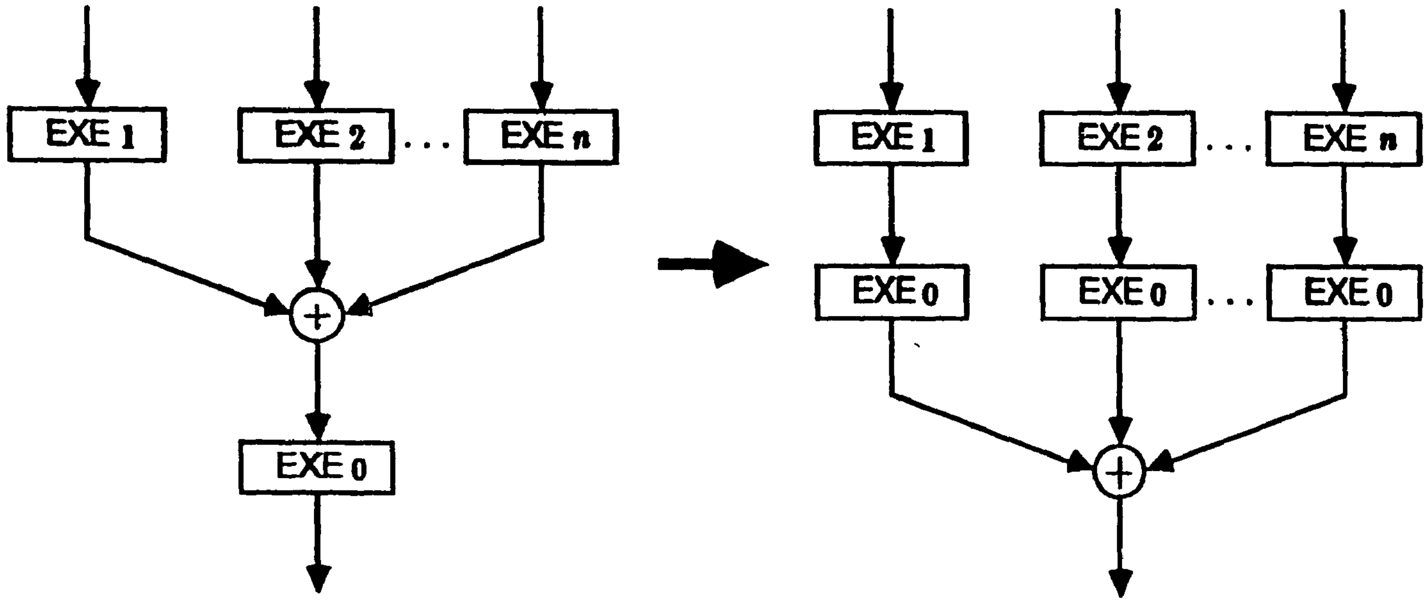


Figure 5.10. The Movement of a Module into an Or-Subgraph

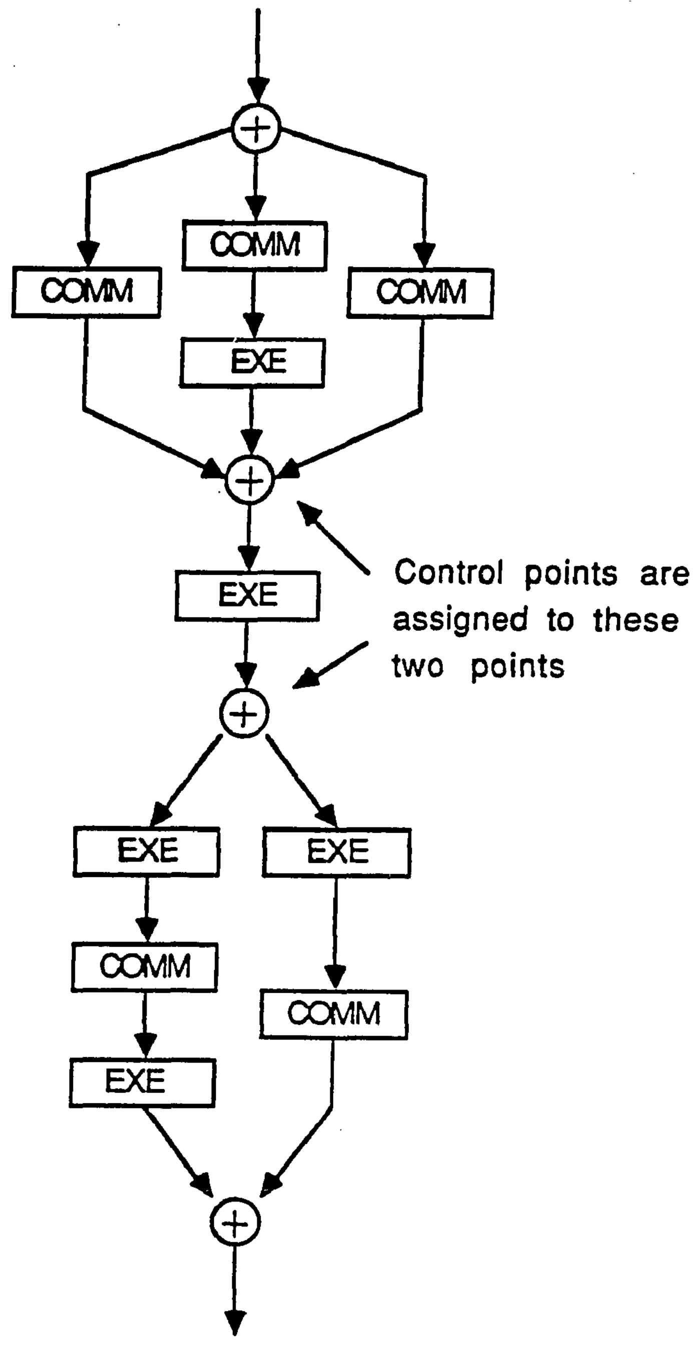


Figure 5.11. A Module Between Two Or-Subgraphs

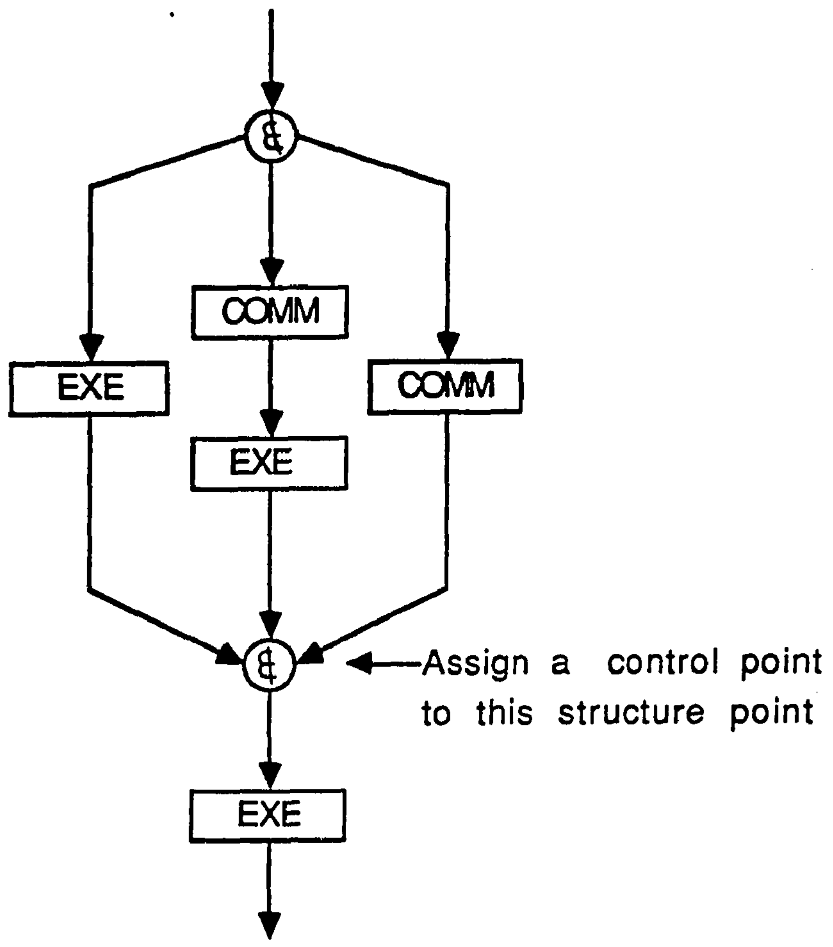


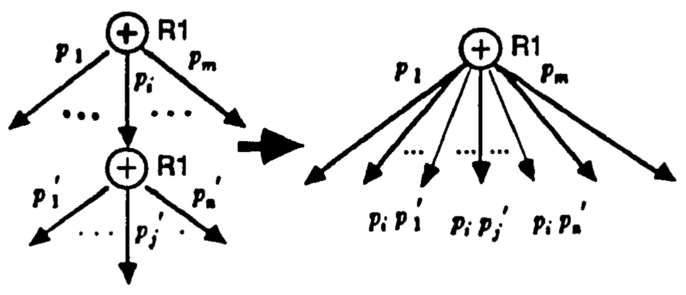
Figure 5.12. A Module Adjacent to an And-Subgraph

- Both P1 and P2 are in the active layer.

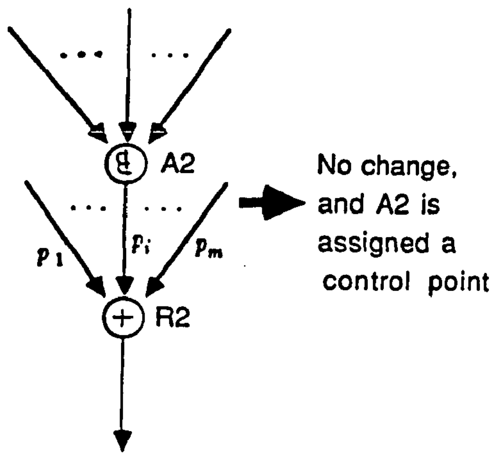
One example of structure point aggregation of each of the above classes is shown in Fig. 5.13 where A1, A2, R1 and R2 represent the fork and join points of the And-Subgraph and the Or-Subgraph, respectively. When the inner Or-Subgraph is expanded (Fig. 5.13(a)), the inner R1 is redundant and, thus, aggregated into the outer R1 to form an AND/OR logic. In order to include new branching probabilities after aggregation, each branching probability in the inner Or-Subgraph should be adjusted by multiplying each branching probability in the inner subgraph by their probability in the outer subgraph. In this case, since further combination may still be possible, no control point is assigned. In Fig. 5.13(b), the inner And-Join cannot be aggregated into the outer Or-Join to form an AND/OR logic, implying that no aggregation is possible and, thus, a control point is assigned to the inner And-Join to represent a boundary between certain activities. Fig. 5.13(c) shows an Or-Subgraph immediately followed by an And-Subgraph in the same layer. Since no AND/OR logic can be formed by aggregating these two structure points, they remain unchanged, except that each of them is assigned a control point.

Since only four types of subgraph are considered, it is not difficult to (i) enumerate all different cases in which two structure points are adjacent to each other, and (ii) set up the aggregation rules for each case. All 39 cases and their aggregation rules are given in the Appendix A.

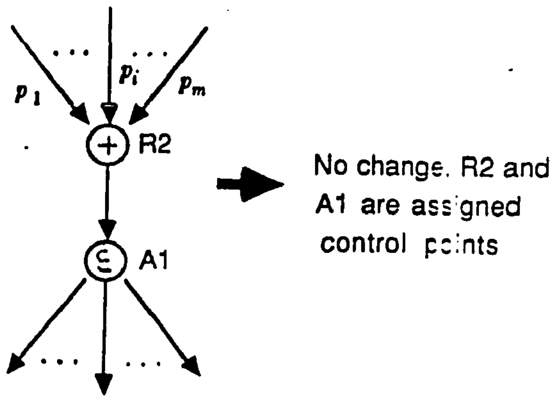
Upon completion of the current iteration of both phases, the active layer is *raised outward* by one so that the next iteration can be applied on the new active layer. This procedure is repeated until the outermost layer is finally expanded. Fig. 5.14 is the summary of the process described thus far. Clearly, since the precedence constraints and the expected execution times are preserved under each operation, C2 and C3 are both satisfied, and the



5.13(a). Case (12)



5.13(b). Case (23)



5.13(c). Case (34)

Figure 5.13. Three Examples of Aggregation Rules

```
PROCEDURE combination_phase( layer_num)
BEGIN /*** combination phase for the current active layer ***/
FOR each subgraph whose layer number = layer_num DO
    BEGIN / ***combination phase for one subgraph ***/
        Perform vertical, horizontal, and total combinations on modules
        in the subgraph as described in Section 5.4.1.1.
    END /*** combination phase for one subgraph ***/
END /*** combination phase for the current active layer ***/


PROCEDURE expansion_phase( layer_num)
BEGIN /*** expansion phase for the current active layer ***/
    Identify all communication points whose layer number = layer_num.
    FOR each subgraph whose layer number = layer_num DO
        BEGIN /*** code stretch movement for one subgraph ***/
            Move modules whose layer number = layer_num into the
            subgraph according to the rules R1, R2, R3, R4 and R5 as
            described in Section 5.4.1.2.
        END /*** code stretch movement for one subgraph ***/

    FOR each structure point whose layer number = layer_num, and has
        no module adjacent to it DO
        BEGIN /*** structure point aggregation ***/
            Aggregate the structure point, and assign control point
            according the rules listed in Appendix A.
        END /*** structure point aggregation ***/
END /*** expansion phase for current active layer ***/


/****************** Main Program **********************/
BEGIN /*** Decomposition Process ***/
    FOR active_layer = N-1 to 0 DO
        BEGIN /* 1 iteration of combination and expansion phases */
            combination_phase( active_layer );
            expansion_phase( active_layer );
        END /*** 1 iteration ***/
END /*** decomposition process ***/
```

Figure 5.14. Summary of the Process to Build the Smallest Activity Set

resulting set of modules is the minimal set of activities. Because the decomposition process is *communication-oriented*, the resulting graph is called a *Communication Flow Graph* (CFG).

As an example, the resulting CFG for the TFG in Fig. 5.15 is shown in Fig. 5.16. Notice that the CFG does not maintain the structure of the original TFG. The CFG is just a collection of activities, communication and control points properly organized by AND/OR, sequential and loop logic structures and, more importantly, no longer contains the four types of standard subgraph.

5.4.2. GSPN Representation

To fully describe the task system with a GSPN, the following three aspects of the task system must be properly modeled and incorporated:

F1. the precedence constraints among activities within a single task, which are imposed by the corresponding CFG,

F2. the precedence constraints among activities of two communicating tasks, which are imposed by the semantics of the communication primitives used, and

F3. the time-driven, rather than event-driven, task invocations.

F1 and F2 determine the structure, whereas F3 affects the initial markings, of the resulting GSPN. F1 requires to model sequential, AND/OR, and looping logics on the activities in a CFG. This modeling process is straightforward and, thus, omitted here (interested readers may consult [Pet81, Mol81]). F2 and F3 are addressed in Sections 5.4.2.1 and 5.4.2.2, respectively.

5.4.2.1. The GSPN Models of Communication Primitives

The communication primitives to be modeled include SEND-RECEIVE-REPLY, QUERY-RESPONSE, and WAITFOR. Although these primitives were proposed in [ShE87]

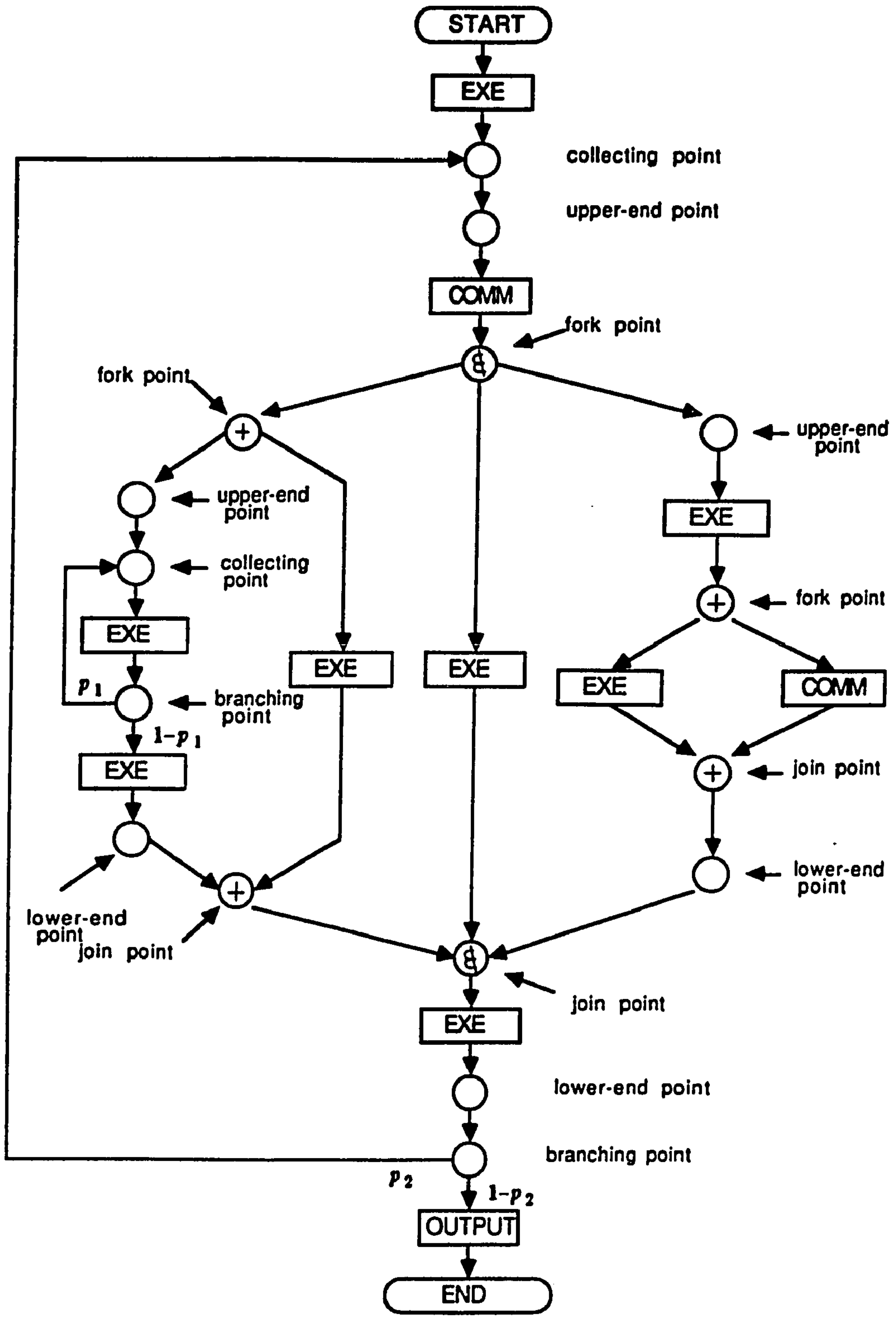


Figure 5.15. The TFG of a PN

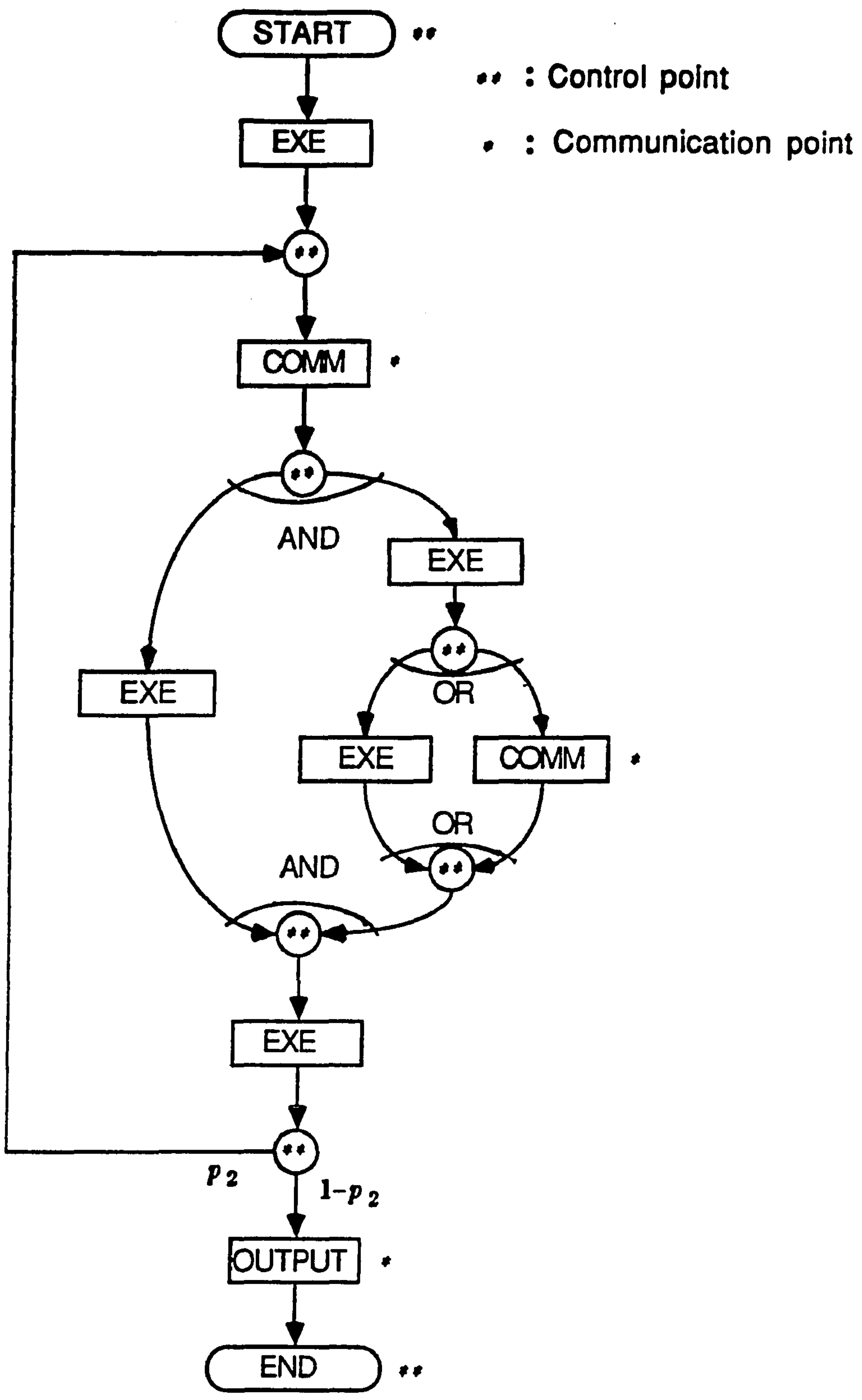


Figure 5.16. The CFG of the TFG in Figure 5.15.

for inter-task communications in an integrated multi-robot system (IMRS), they are typical to real-time control systems. If task A issues a SEND to task B, A remains blocked until a REPLY message from B is received. If B executes a RECEIVE before the message arrives, it also remains blocked. QUERY and RESPONSE are used to allow one task to interrupt another for information, e.g., to avoid collision between two robots that share the same workspace. If A issues a QUERY to B, A remains blocked until the RESPONSE message from B arrives. Upon arrival of the QUERY message from A, B can decide to either accept the QUERY and respond to A immediately or queue the QUERY for a later RESPONSE. WAITFOR is the primitive to allow more than two tasks to synchronize among themselves. Note that a processor can switch to other tasks while the current task is being blocked.

A. SEND-RECEIVE-REPLY

Assuming task A issues a SEND to task B in a different PN, the GSPN model for this communication is given in Fig. 5.17, where the integer numbers in circles are the place numbers.

When A issues a SEND, a token will be placed in ϕ_1, and a timed transition T_1, which represents the transmission delay of the message, is fired. A token will be created at ϕ_2 at the other end of the message transmission after t_1 units of time, the time required for T_1. A token will also be placed in ϕ_3 if B executes a RECEIVE. This token together with that in ϕ_2 will enable the timed transition T_2, which represents the necessary processing by B after the message is received. Upon completion of T_2, two tokens are generated. One is placed in ϕ_4 to unblock B, the other in ϕ_5 to issue a REPLY to unblock A. Again, the delay for the REPLY message is represented by the timed transition T_3. If A and B are executed on the same PN, then the message passing delays T_1 and T_3 are zero, implying that ϕ_1 and ϕ_6 will coincide with ϕ_2 and ϕ_5, respectively.

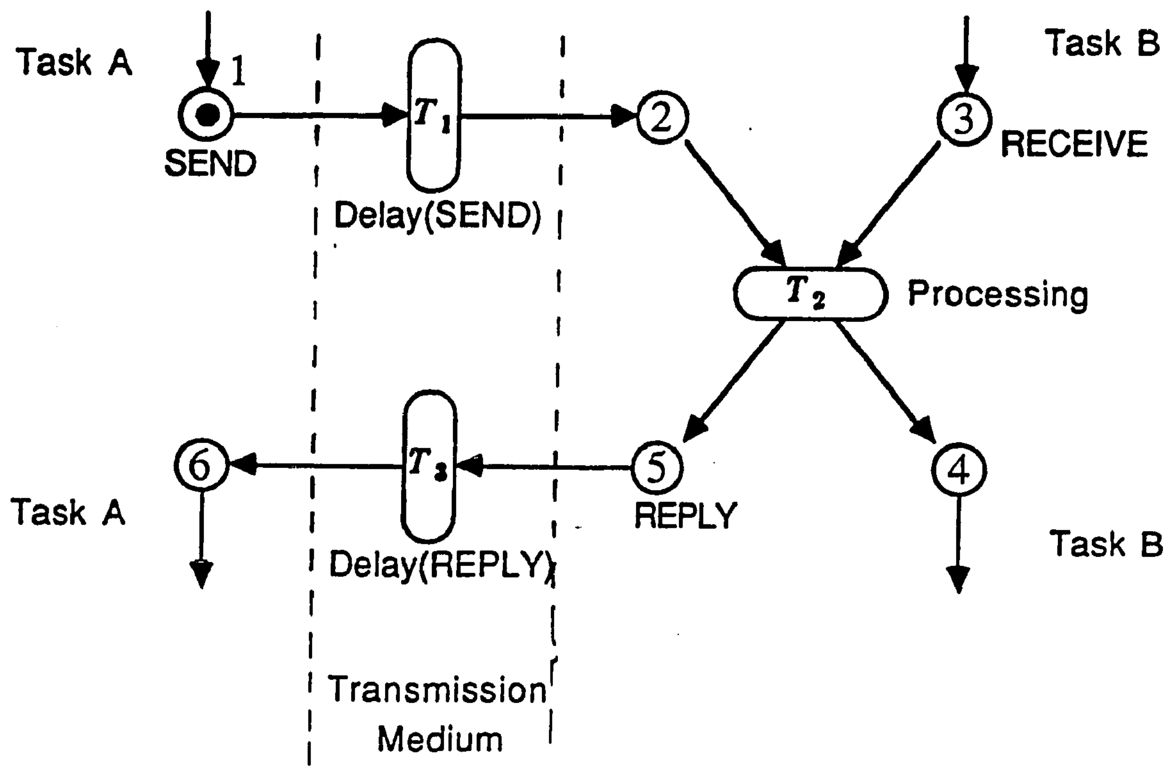


Figure 5.17. GSPN Model for SEND-RECEIVE-REPLY

B. QUERY-RESPONSE

QUERY is used to allow a task to interrupt another task for information. The queried task will decide to either accept or queue the QUERY. When the task accepts the QUERY, it stops its current thread of control, starts executing the RESPONSE routine, returns to where it left off after the RESPONSE routine is completed, and issues a REPLY to unblock the querying task. The GSPN model for QUERY is shown in Fig. 5.18. In essence, the task issuing a QUERY creates an activity, i.e., the RESPONSE routine, to be executed by the task accepting the QUERY. Upon arrival of the QUERY message, the corresponding RESPONSE routine will be ready to be scheduled for execution by the accepting task. Note that such a GSPN allows a blocked task to respond to or queue a query, and to schedule RESPONSE's in case of multiple queries.

C. WAITFOR

WAITFOR allows more than two tasks to synchronize with one another. It can be implemented as follows. When a task T_i in a PN executes a WAITFOR, it sends a message, say $waitfor_i$, to each of the tasks named in its WAITFOR list. T_i remains blocked until it receives the $waitfor$ messages from all the tasks in its waitfor list. After T_i is unblocked, a named function F_i included in the WAITFOR will be executed. When F_i is completed, execution of the task continues from the point immediately after the WAITFOR. Fig. 5.19 is the GSPN model of WAITFOR for three tasks residing at different PNs participating in a three way synchronization.

After precedence constraints F1 and F2 are properly modeled, a system-wide GSPN can be obtained by "pasting together" all GSPN's corresponding to each individual task invocation in a planning cycle. The system-wide GSPN is unmarked until tasks are invoked. Section 5.4.2.2. describes how the unmarked GSPN is marked at each task invocation to

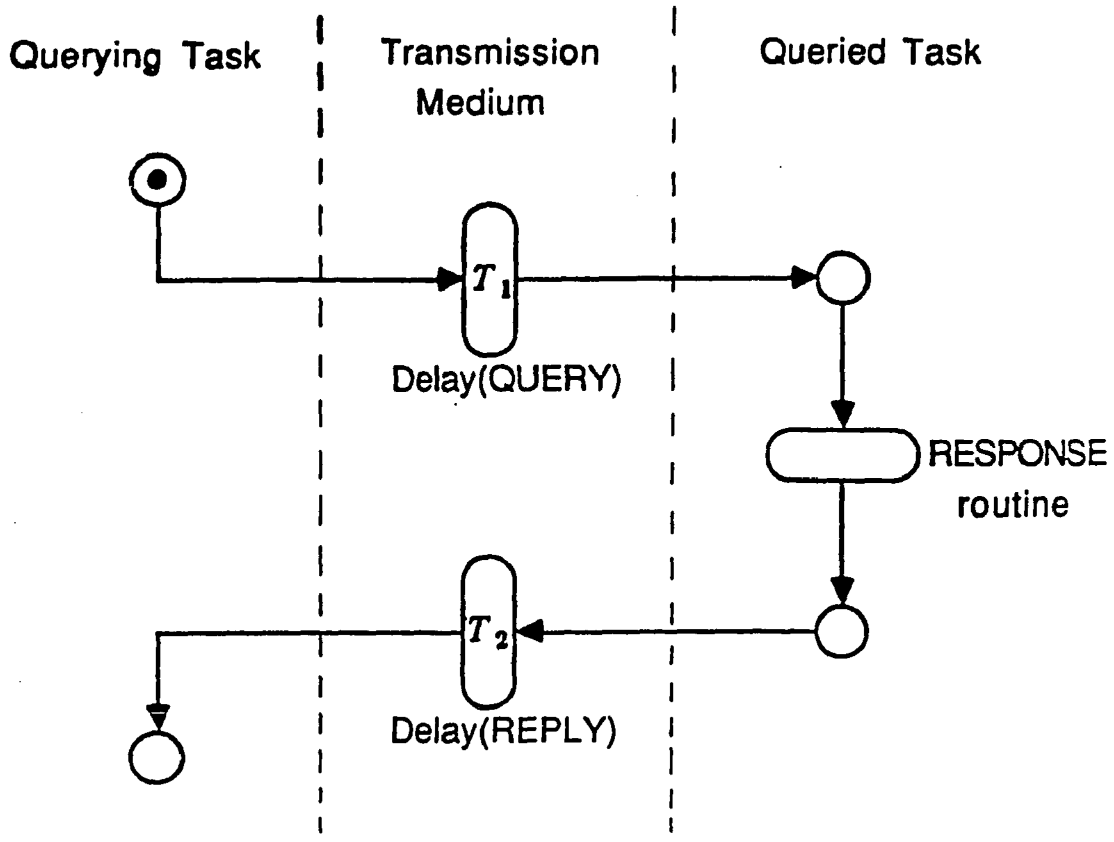


Figure 5.18. GSPN Model for QUERY-RESPONSE

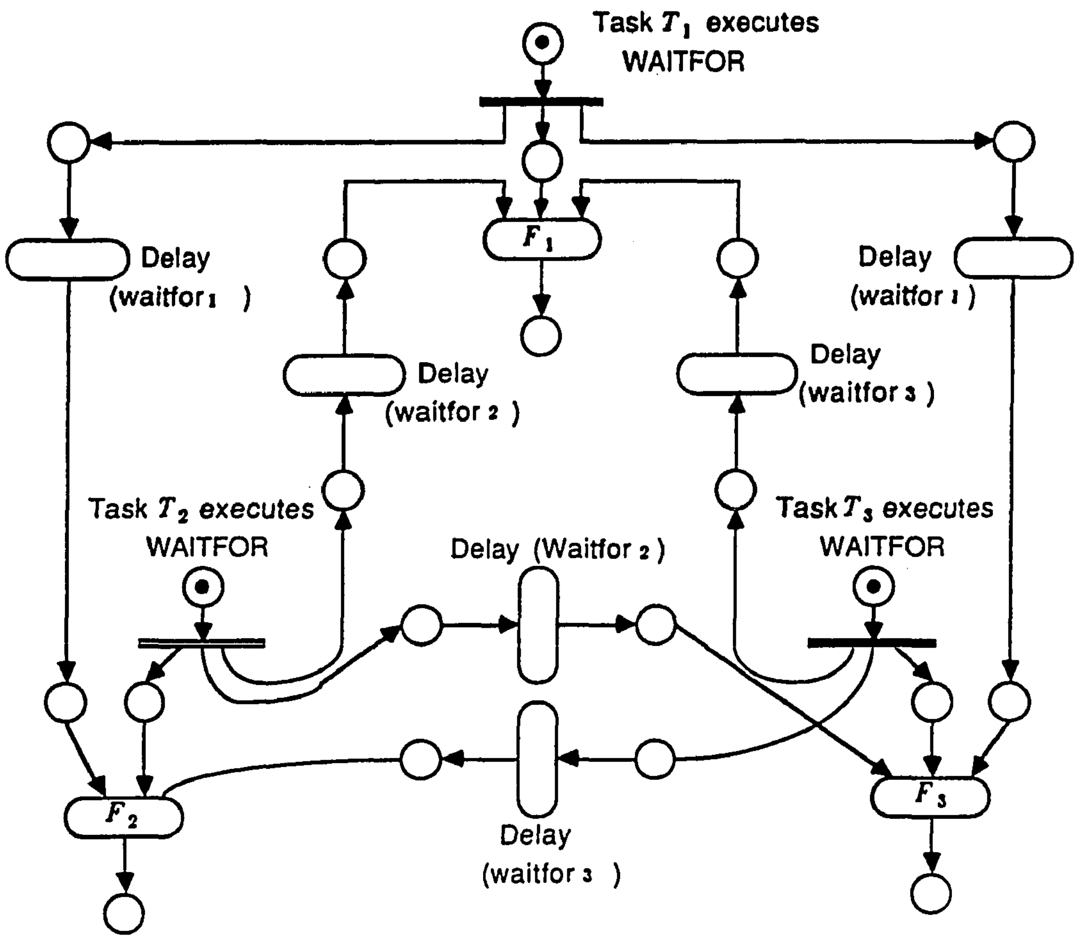


Figure 5.19. Three Tasks Executing WAITFOR's

correctly model the behavior of the task system.

5.4.2.2. Modeling the Time-Driven Task Invocations

Since Petri net type modeling bases are good only for event-driven parallel transition firings, time-driven task invocations add more complexities to our modeling process. This is because time-driven task invocations dictate the state evolution of the (marked) GSPN at invocation times. Therefore, instead of a single CTMC, a sequence of CTMC's has to be used to correctly model the task system.

Assume, as before, that each periodic task begins its first invocation at time 0, and that the set of invocation times in a planning cycle $[0, L)$ is $\{\omega_1, \omega_2, \cdots, \omega_l\}$, where $0 = \omega_1 < \omega_2 < \cdots < \omega_l < L$. The following steps show how the unmarked system-wide GSPN built above is marked such that a sequence of CTMC's, $\{S_u, \Lambda_u, \Theta_u) \mid u = 1, 2, \cdots, l\}$, can be built for the task system.

1) At the beginning of a planning cycle, the unmarked system-wide GSPN is marked by creating a token in each place corresponding to the START points of the individual GSPN's of all periodic tasks to serve as the initial marking of the planning cycle.

2) The marking at time $t \in [\omega_1, \omega_2)$ is determined by the event-driven transition firings in the system-wide GSPN with its initial marking at time ω_1.

3) The marking at time ω_j, $2 \leq j \leq l$, is determined by three parts: (i) a token is created in each place corresponding to the START points of the individual GSPN's of the tasks invoked, (ii) to disallow task pipelining, all tokens associated with previous task invocations are removed from the individual GSPN's before the same tasks are invoked again, and (iii) the number of tokens in each place is determined by the event-driven transition firings of the marked system-wide GSPN with its initial marking at time ω_{j-1}. This marking serves as the initial marking of the GSPN beginning at time ω_j.

4) At time $t \in [\omega_j, \omega_{j+1})$, $2 \leq j \leq l$, where $\omega_{l+1} = L$, the marking is determined by the event-driven transition firings in the system-wide GSPN with its initial marking at time ω_j.

For example, the unmarked GSPN of a task system that consists of three periodic tasks T_1, T_2, and T_3 residing in three different PNs N_1, N_2 and N_3 is shown in Fig. 5.20, where T_1 queries T_2 for information and T_3 communicates with T_2 by sending messages. T_1, T_2 and T_3 with periods 5, 10 and 5, respectively, are first invoked at time 0. In Fig. 5.20, immediate transitions are denoted by "bars", timed transitions by "ovals", and integer numbers in circles are the place numbers. At the beginning of the planning cycle, a token is generated in ϕ_1, ϕ_9 and ϕ_{18}; at time $t \in [0, 5)$, the state evolution is driven by transition firings. At $t = 5$ when T_1 and T_3 are invoked again, the marking of the GSPN is determined by: (i) a token is generated in ϕ_5 and ϕ_{23} to indicate the second invocations of T_1 and T_3, (ii) all tokens in ϕ_1 - ϕ_4 and ϕ_{18} - ϕ_{22} are removed to disallow task pipelining, and (iii) the tokens in ϕ_9 - ϕ_{17} are determined simply by event-driven transition firings since $t = 0$. Likewise, at $t \in [5, 10)$, the state evolution is determined by event-driven transition firings. At $t = 10$, the same is repeated again for the next planning cycle. Notice that the random switches at ϕ_{13} and ϕ_{21} should be the same to assure fault-free message passing since no time-out provisions are made in receiving messages. The same also holds for the two random switches at ϕ_{15} and ϕ_{26}.

Following the execution rules of GSPN introduced in Section 5.2, it is easy to prove that if the task system is fault-free, then the system-wide GSPN marked above is safe.

5.4.3. Construction of the CTMC Model

Given the system-wide GSPN marked as above, the state-transition-rate (STR) diagram of the u*-th* CTMC is built as follows.

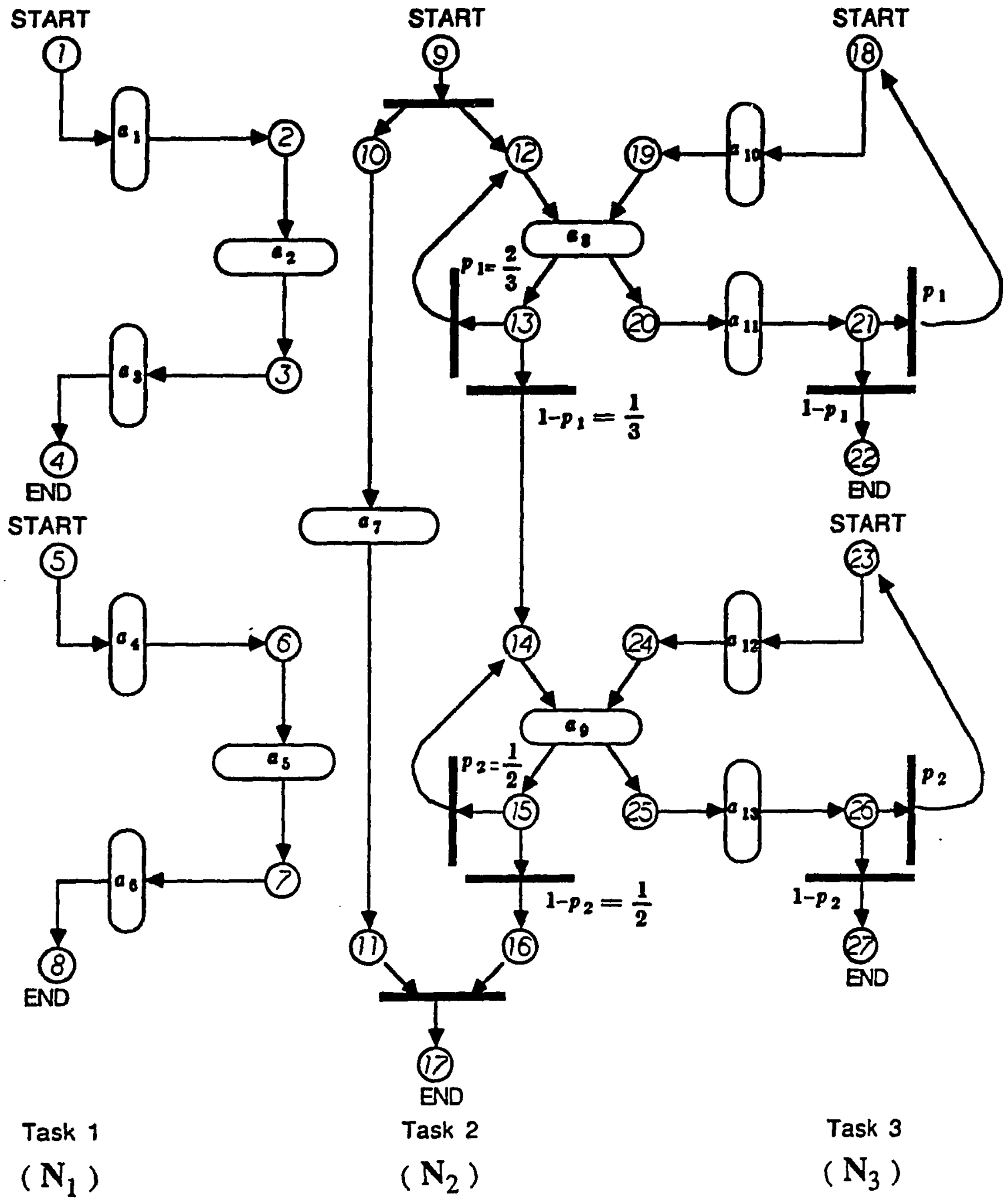


Figure 5.20. System-Wide GSPN for an Example Task System

1) For the ease of analysis, replace each random switch with its GSPN equivalent [MoM84] as shown in Fig. 5.21.

2) Find S_u by performing reachability analysis on the GSPN initially marked at ω_u. This set serves as the state space of the STR diagram in $[\omega_u, \omega_{u+1})$. Note that, except for those introduced above, there are no vanishing states in S_u.

3) Assume that each timed transition delay between states is independently and exponentially distributed with a transition rate equal to the reciprocal of the expected execution time of the corresponding activity.

For example, the STR diagrams of the task system in Fig. 5.20 are given in Fig. 5.22 with $\mu_1 = \mu_3 = \mu_4 = \mu_6 = 2$, $\mu_2 = \mu_5 = 4$, $\mu_7 = \mu_{10} = \mu_{11} = \mu_{12} = \mu_{13} = 1$, $\mu_8 = 6$, $\mu_9 = 4$, and branching probabilities $p_1 = 2/3$, and $p_2 = 1/2$, where μ_i is the transition rate for activity a_i.

As shown in Figs. 5.22(a) and (b), all states are represented by a set of integers representing the set of (non-instantaneous) places each with one token, and the number on each arc indicates the transition rate.

Examination of these diagrams leads to the following insights:

- Each state in the *total* state space $S = \bigcup_{u=1}^{l} S_u$ globally describes which stage of the tasks each PN has been executing. For example, (2, 10, 12, 19) in Fig. 5.22(a) represents a state where T_1 is blocked waiting for a response from T_2, T_2 has neither finished a_7 nor responded to the query from T_1, and the message from T_3 has arrived at N_2 waiting to be handled.

- The state spaces for any two different invocation intervals are disjoint. Further, S_1 contains a unique *starting* state s_0, and S_l contains a unique *ending* state s_f of the planning cycle. For example, state (1, 10, 12, 18) (Fig. 5.22(a)) is the starting state,

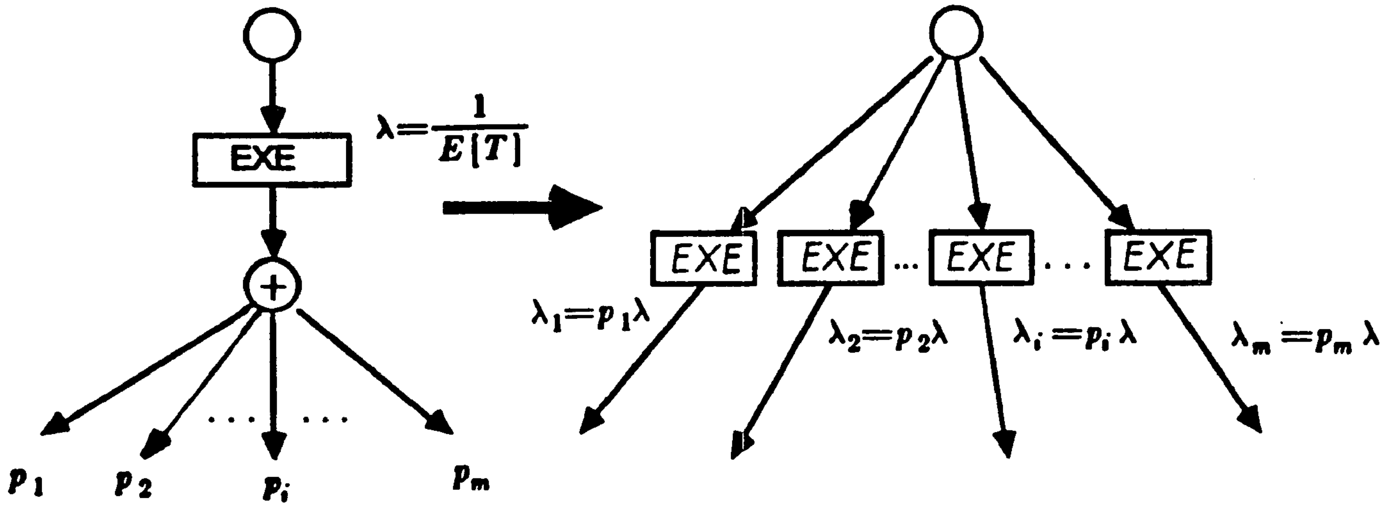
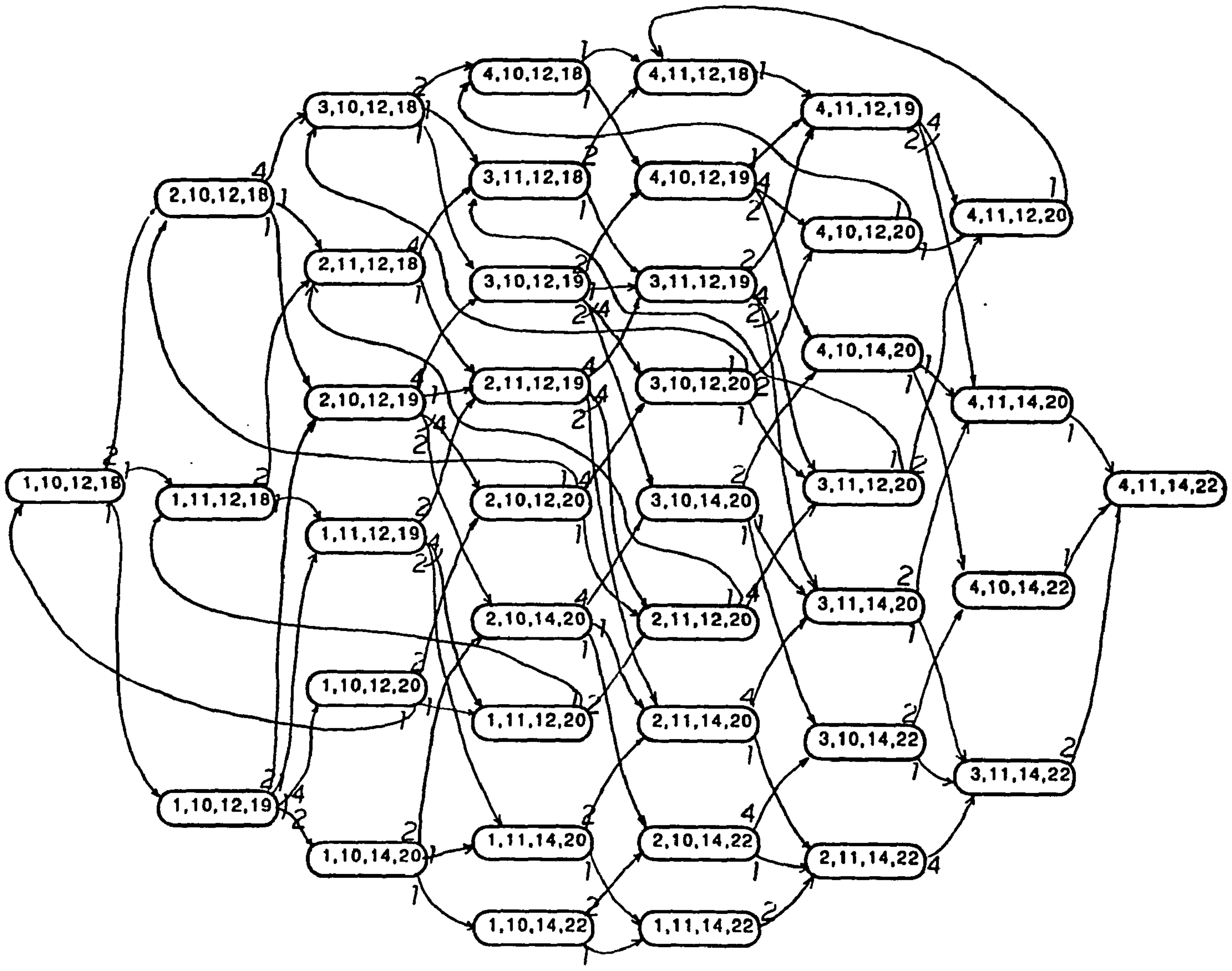
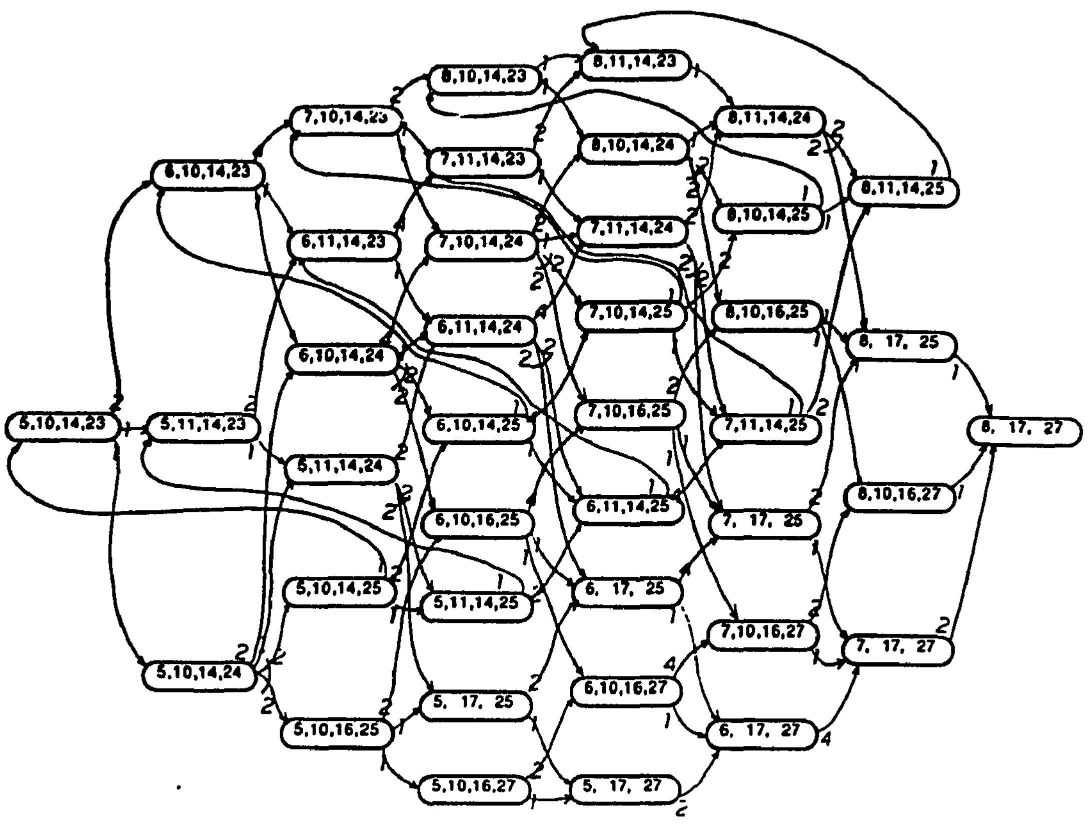
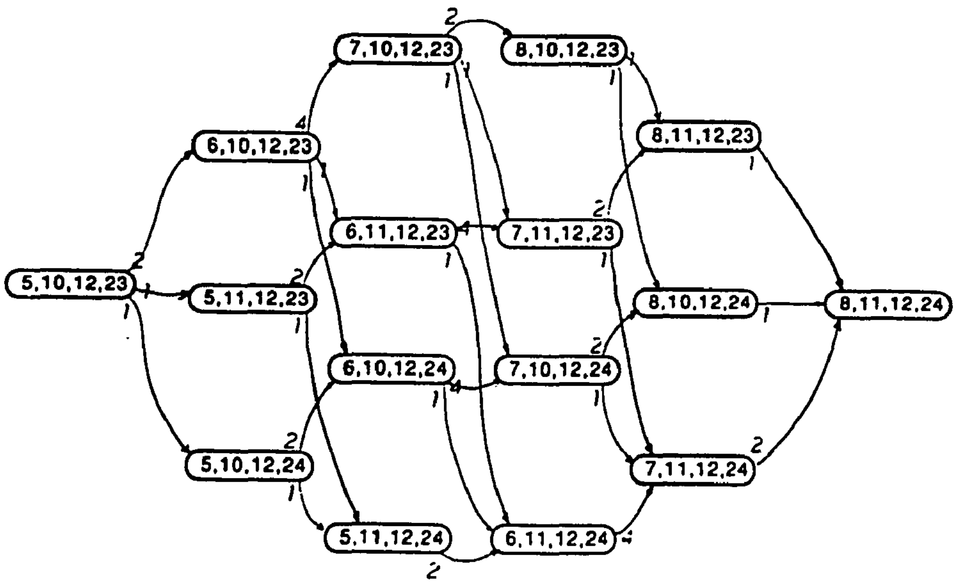


Figure 5.21. The GSPN Equivalent of a Random Switch


1,10,12,18
2,10,12,18
1,11,12,18
1,10,12,19
3,10,12,18
2,11,12,18
2,10,12,19
1,11,12,19
1,10,12,20
1,10,14,20
4,10,12,18
3,11,12,18
3,10,12,19
2,11,12,19
2,10,12,20
2,10,14,20
1,11,12,20
1,11,14,20
1,10,14,22
4,11,12,18
4,10,12,19
3,11,12,19
3,10,12,20
3,10,14,20
2,11,12,20
2,11,14,20
2,10,14,22
1,11,14,22
4,11,12,19
4,10,12,20
4,10,14,20
3,11,12,20
3,11,14,20
3,10,14,22
2,11,14,22
4,11,12,20
4,11,14,20
4,10,14,22
3,11,14,22
4,11,14,22

5.22(a). t ∈ [0, 5)





5.22(b). t ∈ [5, 10)

Figure 5.22. The CTMC Model for the Task System in Figure 5.20.

while state (8, 17, 27) (Fig. 5.22(b)) is the ending state.

- If a CFG contains loops, then all STR diagrams will be cyclic as shown in Figs. 5.22(a) and (b), and acyclic otherwise.
- Depending on the marking at ω_j, the STR diagram in $[\omega_j, \omega_{j+1})$ could be disconnected (Fig. 5.22(b)).
- Since the precedence constraints among all activities are properly conveyed in the system-wide GSPN, the resulting STR diagrams implicitly show all possible execution sequences of activities of the task system in a planning cycle.
- Depending on the task system, some states could be *time critical.* For example, if there is a hard deadline for N_1 to complete the first invocation of T_1, then at least one state of the form (4, *, *, *) in Fig. 5.22(a) has to be reached before that deadline.
- For a specific state, the number of activities that can be concurrently executed by a PN may be larger than that of the processors available to that PN. For example, the system is in state (6, 10, 14, 24), three activities (a_5, a_7, and a_9) can be concurrently executed by N_2. If N_2 has only two processors, then a decision must be made as to which two of the three activities in N_2 are to be chosen for parallel execution.

5.5. An Application Example

In this section, we demonstrate the use of the proposed CTMC model $\{(S_u, \Lambda_u, \Theta_u) \mid u = 1, 2, \cdots, l\}$ by computing the probability of missing a hard deadline given the activity selection policy and the local state of a PN. Clearly, estimating task execution time is a special case of this problem.

The concurrent task execution can be thought as the "movement" of tokens in the GSPN. A place in the GSPN is said to be *realized* if it has a token implying the completion of those activities that "precede" the place. Therefore, a hard deadline in the task system is associated with the realization of a place. For example, a hard deadline for N_1 in Fig. 5.20 to complete the first invocation of T_1 is the maximum time allowed for ϕ_4 to be realized. A place is said to be ***time critical*** if it has a hard deadline. A state s_i which contains a set, say R_i, of realized time critical places is called a ***goal state with the realized set*** R_i, and written as $\psi(s_i) = R_i$. Denote the set of all time critical places by Ψ. Clearly, $\psi(s_i) \subset \Psi$, $\forall\, s_i \in S$. For the previous example, if $\Psi = \{4, 8, 17\}$, then $\psi((4, 11, 14, 21)) = \{4\}$, and $\psi((6, 17, 25)) = \psi((7, 17, 27)) = \{17\}$. If $\Psi = \{11, 22, 27\}$, then $\psi((6, 11, 14, 25)) = \{11\}$, $\psi((8, 17, 27)) = \{11, 27\}$,[10] and $\psi((8, 10, 16, 25)) = \varnothing$. The set G_h of goal states each of whose realized sets contains the time critical place ϕ_h is called the ***goal set with respect to place*** ϕ_h and written as $\Upsilon(\phi_h) = G_h$. Obviously, $\Upsilon(\phi_h) \subset S$, $\forall\, \phi_h \in \Psi$. For the same task system in Fig. 5.20, $\Upsilon(\phi_{17})$ is the subset of all states of the form (*, 17, *).

The ***first passage time*** $Y_{i,j}$ from s_i to s_j is a r.v. representing the time needed for the task system to reach s_j for the first time from s_i. Denote the CDF of $Y_{i,j}$ by $Q_{i,j}(t) = Pr\{Y_{i,j} \le t\}$. A set of initial states, E, with known element probabilities could sometimes be given instead of a single initial state s_i. Similarly, a set of final states F may be known in place of a single final state s_j. In such a case, the first passage time is defined as the time needed for the task system to reach any $s_j \in F$ for the first time given that the task system is initially in some state $s_i \in E$ at time t_c. Denote the CDF of the first passage time of this type by $Q_{E,F}(t_c, t')$, where t' represents the time measured from t_c.

The local state space S_u^k associated with N_k within $I_u = [\omega_u, \omega_{u+1})$ is constructed as follows: identify the set of places, P_k, belonging to N_k, and then select the markings of P_k from S_u. Note that for any $v \ne u$, $S_v^k \cap S_u^k = \varnothing$ because $S_v \cap S_u = \varnothing$. Taking the task system in Fig. 5.20 again as an example, ϕ_1, ϕ_4, ϕ_5, and ϕ_8 belong to N_1, S_1^1 = {(1), (4),

[10]The realized set of (8, 17, 27) is that of (8, 11, 16, 27).

$(nil)\}$ and $S_2^1 = \{(5), (8), (nil)\}$, where (1), (4), (5), and (8) stand for the local states that each of ϕ_1, ϕ_4, ϕ_5, and ϕ_8 contains a token, and (nil) for the local state where no place contains a token. Clearly, $S_u \subset S_u^1 \times S_u^2 \times \cdots \times S_u^k \times \cdots \times S_u^n$, $\forall\, u = 1, 2, \cdots, l$, where $n = |N|$.

Given the activity selection policy π^* used by the task system and the local state x^k at time $t_c \geq 0$, we want to compute the probability σ_e of missing the hard deadline $d_e \geq t_c$ for a time critical place ϕ_e in the current planning cycle. More formally, we want to calculate

$$\sigma_e = Pr\{W_e > d_e \mid \pi = \pi^*, X^k(t_c) = x^k\}, \tag{5.4}$$

where W_e is a r.v. representing the time ϕ_e is realized for the first time, π a variable representing the activity selection policy used by the task system, and $X^m(t)$ a r.v. for the local state of N_k at time t.

Given the GSPN and CTMC models of the task system, the above problem can be solved by using the following CTMC properties.

S1. Construct the CTMC model that includes the activity scheduling policy π^*. This is done by deleting all arcs not in the active transition set $ATS_i^{\pi^*}$ from the set $OUT(s_i, *)$ for each $s_i \in S$ to queue the corresponding activities for later execution.

S2. Solve the forward Chapman-Kolmogorov (C-K) equations for the sequence of CTMC's from S1 to obtain the prior probability $P_i(t_c)$ and apply Bayes' Theorem [DeG70] to get the posterior probability $P_i'(t_c)$ that the system is in s_i at time t_c given $X^k(t_c) = x^k$. The initial probability of state s_b of the CTMC for interval $I_u = [\omega_u, \omega_{u+1})$ is determined from the final probability of state s_a of the CTMC for $I_{u-1} = [\omega_{u-1}, \omega_u)$, $u \neq 1$, by the following equation:

$$P_b(\omega_u) = \begin{cases} 0 & \text{if } B = \varnothing \\ \sum\limits_{s_a \in B} P_a(\omega_u) & \text{otherwise,} \end{cases} \tag{5.5}$$

where $B = \{s_a | \Theta_{u-1}(s_a) = s_b\}$. By Bayes' Theorem, the posterior probability is $P_i'(t_c) = Pr\{X(t_c) = s_i | X^k(t_c) = x^k\} =$

$$\begin{cases} \dfrac{P_i(t_c)}{\sum\limits_{s_j \in L_{x^k}} P_j(t_c)} & \text{if } s_i \in L_{x^k} \\ 0 & \text{otherwise,} \end{cases} \tag{5.6}$$

where L_{x^k} is the set of states with $X^k(t_c) = x^k$.

S3. Identify the goal set $Y(\phi_e) = G_e$. Since each $s_j \in G_e$ is a state with ϕ_e realized, the event that ϕ_e is realized is equivalent to the event that at least one state in G_e is reached.

S4. Using the posterior probabilities $P_i'(t_c)$ computed in S2 as the initial probabilities at time t_c, compute the CDF of the first passage time $Q_{L_{x^k}, G_e}(t_c, t')$ to any s_j in Gk given that $X^k(t_c) = x^k$, where t' is the time period measured from t_c. $Q_{L_{x^k}, G_e}(t_c, t')$ can be found by making each state in G_e an absorption state in the CTMC model, and then solving the C-K equations again for these new CTMC's to compute $\sum\limits_{s_j \in G_e} Pr\{X(t') = s_j | X^k(t_c) = x^k\}$.

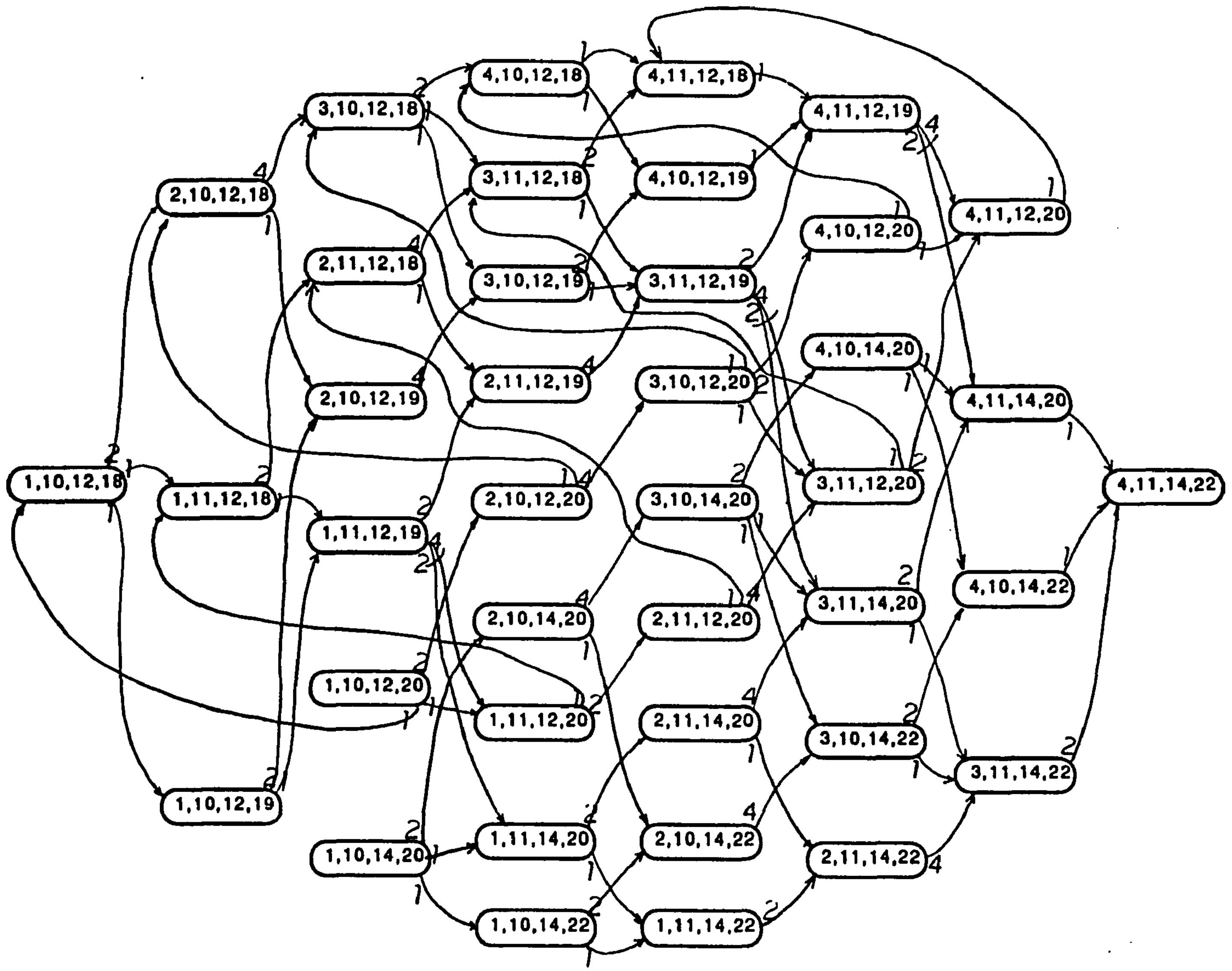
S5. Considering the current time t_c and $Q_{L_{x^k}, G_e}(t_c, t')$ computed above, σ_e is determined by:

$$\begin{aligned} \sigma_e &= Pr\{W_e > d_e | \pi = \pi^*, X^k(t_c) = x^k\} \\ &= 1 - Q_{L_{x^k}, G_e}(t_c, d_e - t_c). \end{aligned} \tag{5.7}$$

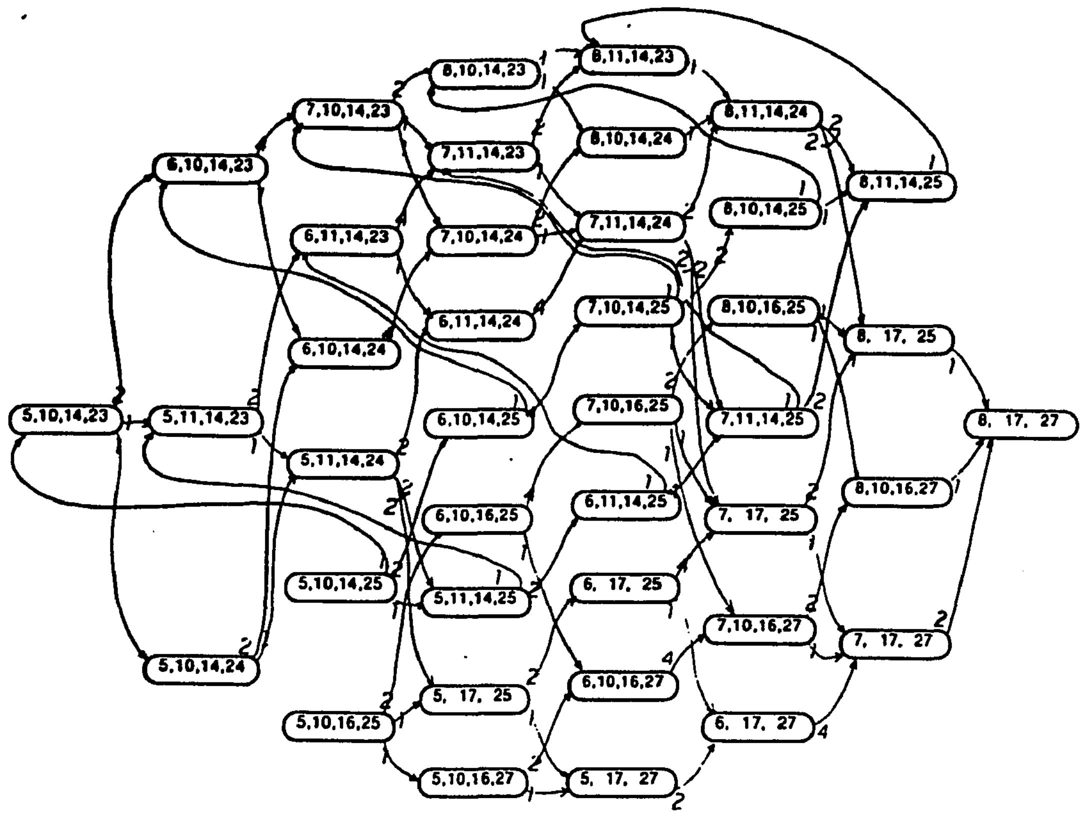
As an example, each N_1, N_2, or N_3 in Fig. 5.20 is assumed to have only one processor dedicating to normal computations. From Fig. 5.20 or 5.22, it is easy to see that N_1 is the only PN with an insufficient number of processors since, among the set $\{a_2, a_7, a_8\}$ or

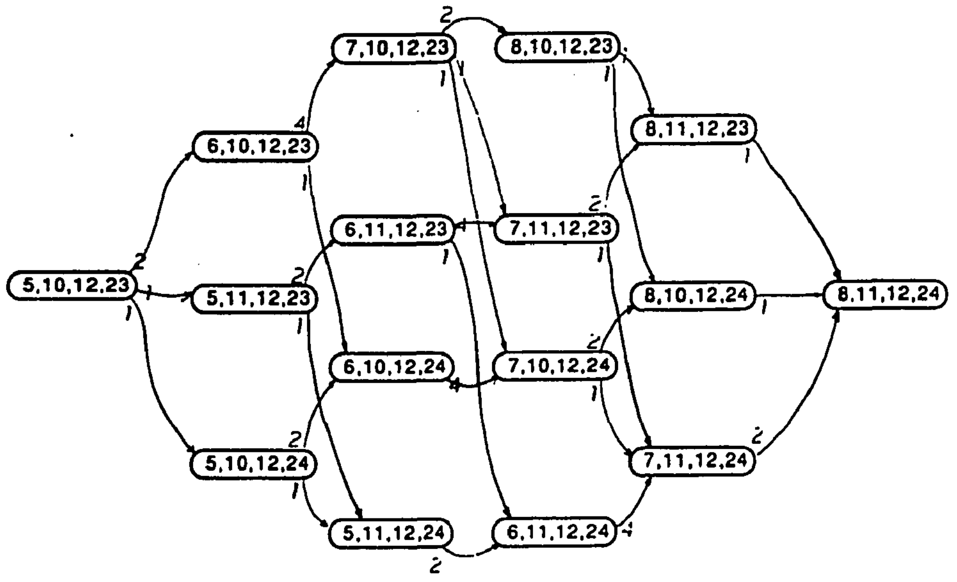
$\{a_5, a_7, a_9\}$, two or more transitions can be fired simultaneously at some state. Thus, the activity selection policy is determined by how N_2 picks its transition to fire at each state. In this example, we assume that N_2 picks the transition according to the order a_2, a_7, a_8 or a_5, a_7, a_9 to fire in each state. Fig. 5.23 is the CTMC's corresponding to this policy.

Suppose the only information available to derive σ_e is that ϕ_1 is realized in [0, 5) and ϕ_5 is realized in [5, 10). Suppose also that ϕ_4, ϕ_8, ϕ_{17}, ϕ_{22} and ϕ_{27} are all time critical with hard deadlines $d_4 = d_{22} = 5$, $d_8 = d_{17} = d_{27} = 10$. Following the above solution steps, we obtain Fig. 5.24 showing the probabilities of missing these hard deadlines as functions of t_c, the time the information is observed. It is not surprising that σ_4 and σ_8 are identical, since N_2 always chooses a_2 or a_5 first, whenever possible. This implies that the ordered activities $\{a_1, a_2, a_3\}$ or $\{a_4, a_5, a_6\}$ will be executed without interruption. For $t_c \in [0, 5)$ and $t_c \in [5, 10)$, σ_{17} is nearly stable at the value 0.575 because of the dominating fact that, after the second invocation of T_3, T_2 may be waiting hopelessly at ϕ_{12} for the message which will never arrive from the first invocation of T_3 that has already been discarded. σ_{22} is a monotonically decreasing, although not significant, function of t_c. The high probability of missing τ_{22} is due to the high branching back probability p_1 as well as the tightness of the deadline. σ_{27} fluctuates around 0.628, a value larger than σ_{22} by approximately 0.1, because, in addition to similar reasons for σ_{22}, the message from the current invocation of T_3 may wait hopelessly to be processed by T_2 which, unfortunately as discussed above, is also waiting hopelessly for the message from the already-discarded invocation of T_3. From this example, we can see the importance of the time-out mechanism on message communications in a distributed real-time system.


1,10,12,18
2,10,12,18
1,11,12,18
1,10,12,19
3,10,12,18
2,11,12,18
2,10,12,19
1,11,12,19
1,10,12,20
1,10,14,20
4,10,12,18
3,11,12,18
3,10,12,19
2,11,12,19
2,10,12,20
2,10,14,20
1,11,12,20
1,11,14,20
1,10,14,22
4,11,12,18
4,10,12,19
3,11,12,19
3,10,12,20
3,10,14,20
2,11,12,20
2,11,14,20
2,10,14,22
1,11,14,22
4,11,12,19
4,10,12,20
4,10,14,20
3,11,12,20
3,11,14,20
3,10,14,22
2,11,14,22
4,11,12,20
4,11,14,20
4,10,14,22
3,11,14,22
4,11,14,22

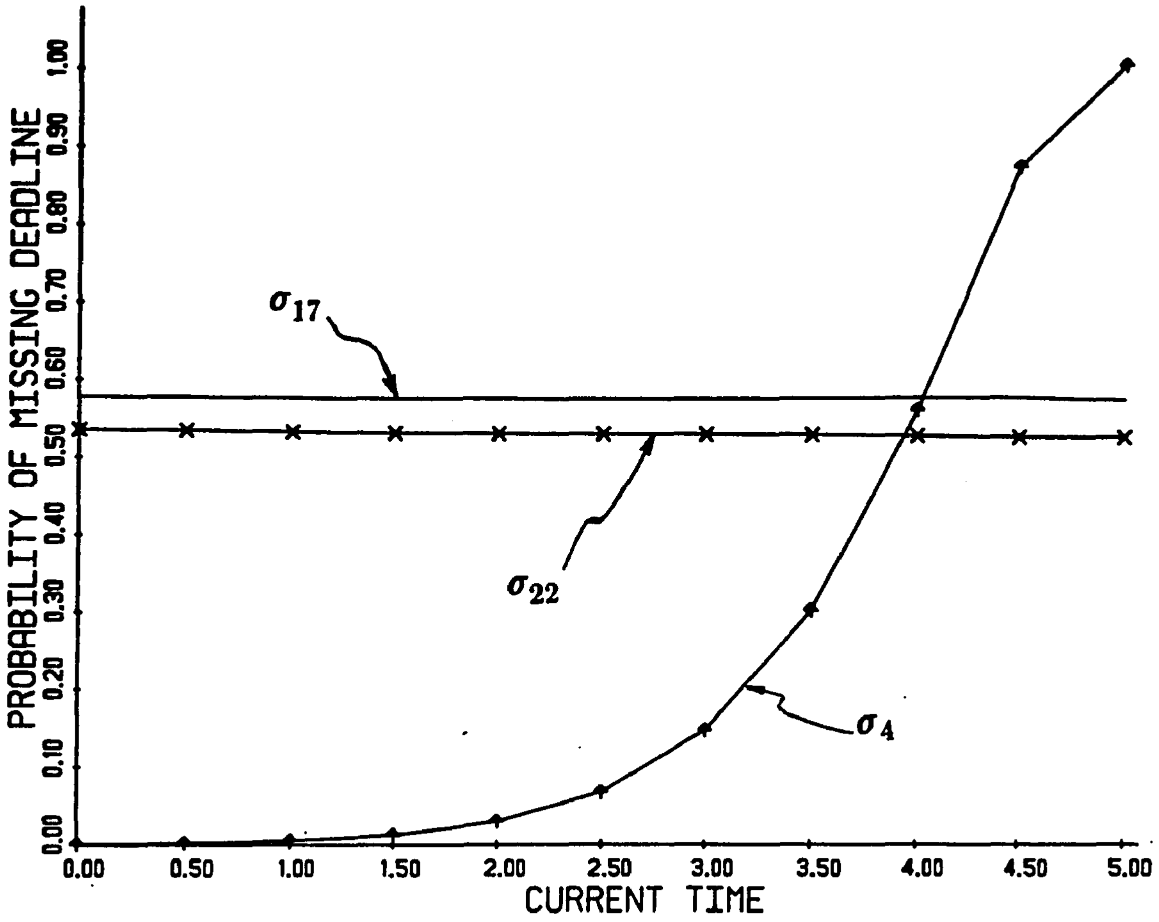
5.23(a). t ∈ [0, 5)





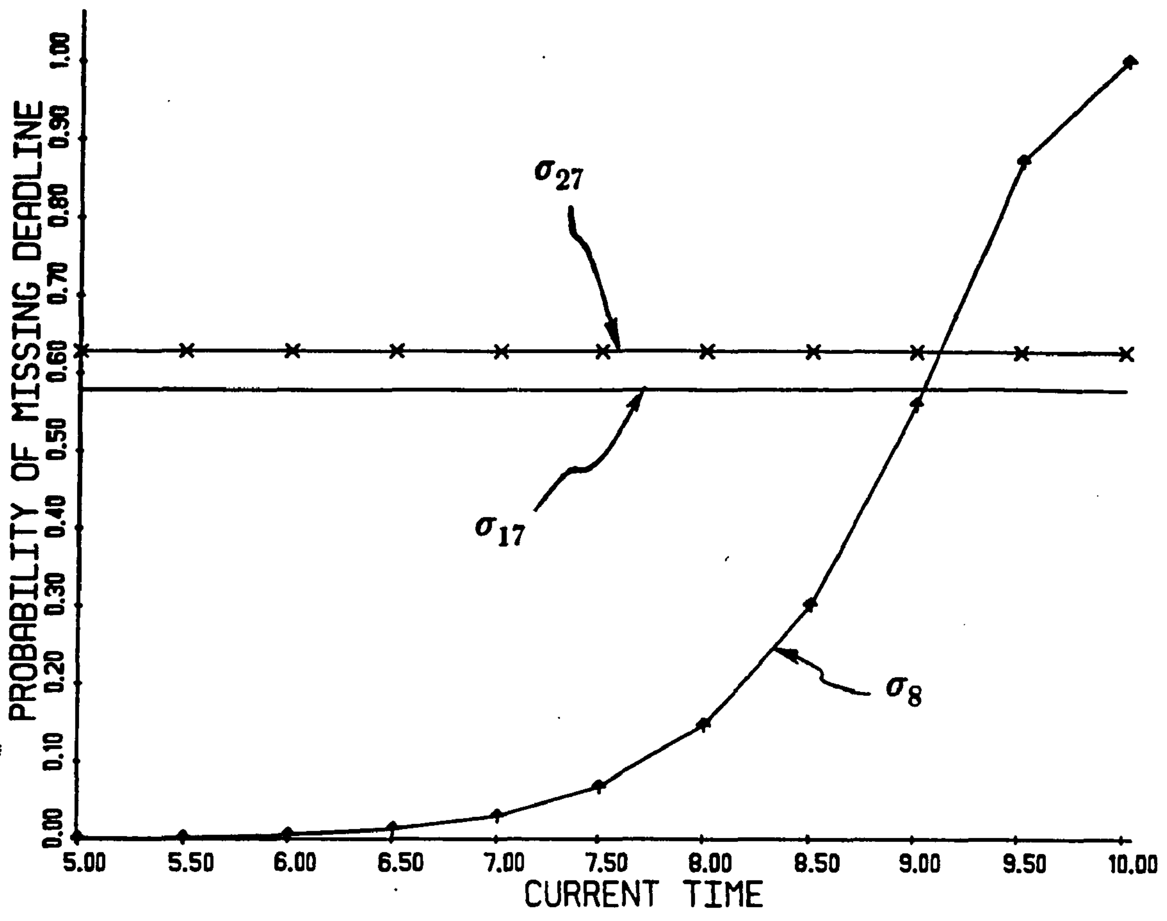
5.23(b). t $\in$ [5, 10)

Figure 5.23. The CTMC's Tailored for the δ^* Selected



PROBABILITY OF MISSING DEADLINE
0.00
0.10
0.20
0.30
0.40
0.50
0.60
0.70
0.80
0.90
1.00
0.00
0.50
1.00
1.50
2.00
2.50
3.00
3.50
4.00
4.50
5.00
CURRENT TIME
σ_{17}
σ_{22}
σ_4

5.24(a). t $\in$ [0, 5)



5.24(b). t $\in$ [5, 10)

Figure 5.24. The Probability of Missing Deadlines

CHAPTER 6
OPTIMAL SCHEDULING OF PERIODIC TASKS AND MESSAGES

6.1. Introduction

Since inter-task communications impose precedence constraints among tasks, the scheduling of communication messages is an indispensable part of task scheduling. The main objective of this chapter is to formulate and solve the integrated problem of scheduling periodic tasks and messages in a distributed real-time system using the CTMC model developed in the last Chapter. The scheduling objective is to minimize the long-term expected number of periodic tasks missing their deadlines.

The integrated scheduling problem is described as follows. At any time t, each PN has to make the following decisions on the execution of its periodic tasks:

D1. While waiting for a specific message to arrive, the PN either continues to wait for the message or abandons the waiting and executes, instead, a certain *default activity.*

D2. Which of ready activities in the PN should be executed next if the number of free processors at the PN is less than that of ready activities?

D3. Which of the queued messages must be processed first?

D2 and D3 are self-explanatory, while D1 needs further elaboration as follows. Since a given goal is accomplished through the cooperative execution of periodic tasks, they communicate with one another to synchronize or exchange information. The communicating partners are usually blocked [ShE87] until the communication is completed, meaning that no activities requiring the information are allowed to continue. This blocking communication scheme could result in a situation where one or more of the communicating partners are delayed indefinitely waiting for a message which may never arrive. This could be due to, for instance, the failure of the sending PN or a communication link. To assure the timely completion of each task, there must be a provision for a communicating partner to carry on its execution even in the absence of the requested information. Of course, to compensate for the missing information, some form of default activity should be invoked. This default activity introduces an extra computation load to the corresponding PN. It is D1 that deals with the decision on whether to wait for an outstanding message or to invoke a default activity. D1-D3 are actually three dependent parts of a single problem since neither of them can be solved without considering the others. Therefore, we shall henceforth call them collectively the *task/message scheduling problem* (TMSP).

While scheduling tasks with fixed execution times has long been studied, scheduling tasks with random execution times is a relatively new and more difficult problem [Web82]. To our best knowledge, the TMSP with <u>dependent</u> periodic tasks has not been addressed in the literature. Perhaps, the most relevant work is the scheduling of a fixed set of dependent tasks with stochastic execution times (e.g., [Dem81, PiS81]). However, none of these addressed scheduling problems which are as complicated as the TMSP. We will first transform the TMSP into a semi-Markov decision process (SMDP) [Ros70] and then use the Dynamic Programming (DP) [Den82] technique to solve the SMDP. The SMDP is constructed based on the extended CTMC model of periodic tasks with the notion of default activity included.

Both *centralized* and *decentralized* solutions to the TMSP are derived. In the centralized case, a central monitor with perfect observations of the execution stages of all PNs is assumed to exist, which determines a globally optimal scheduling decision for each PN. In the decentralized case, however, each PN makes its own scheduling decision based on private and, perhaps, out-dated information received from other PNs.

The rest of the chapter is organized as follows. In Section 6.2, the original CTMC model of periodic tasks is extended to include a default activity if the system gives up on waiting for an outstanding message for whatever reasons, e.g., time-out. The centralized TMSP is formulated and solved in Section 6.3. In Section 6.4, we solve the decentralized TMSP for which each PN periodically broadcasts its local state to other PNs. Finally, we also determine an optimal frequency of state broadcast.

6.2. The Extended CTMC Model of Periodic Tasks

Since message scheduling is not separable from the TMSP, the original CTMC model built in Chapter 5 needs to be extended to include the notion of default activity.

A default activity is extra work that a PN has to perform in order to assure the quality of computation in case the system gives up on waiting for an outstanding message for some reason. For each message that is supposed to be received, the execution time of the corresponding default activity is assumed to be exponentially distributed with a fixed rate. According to the semantics of blocking communication and the purpose of a default activity, it is assumed that no activities in the program located after the receipt point of a message r can be executed before either receiving r or completing the corresponding default activity R_r. Notice that a communication primitive, such as SEND-RECEIVE-REPLY, may account for two message receipts (one on sender's side and the other on receiver's side), and the default activities for these two receipts need not be the same.

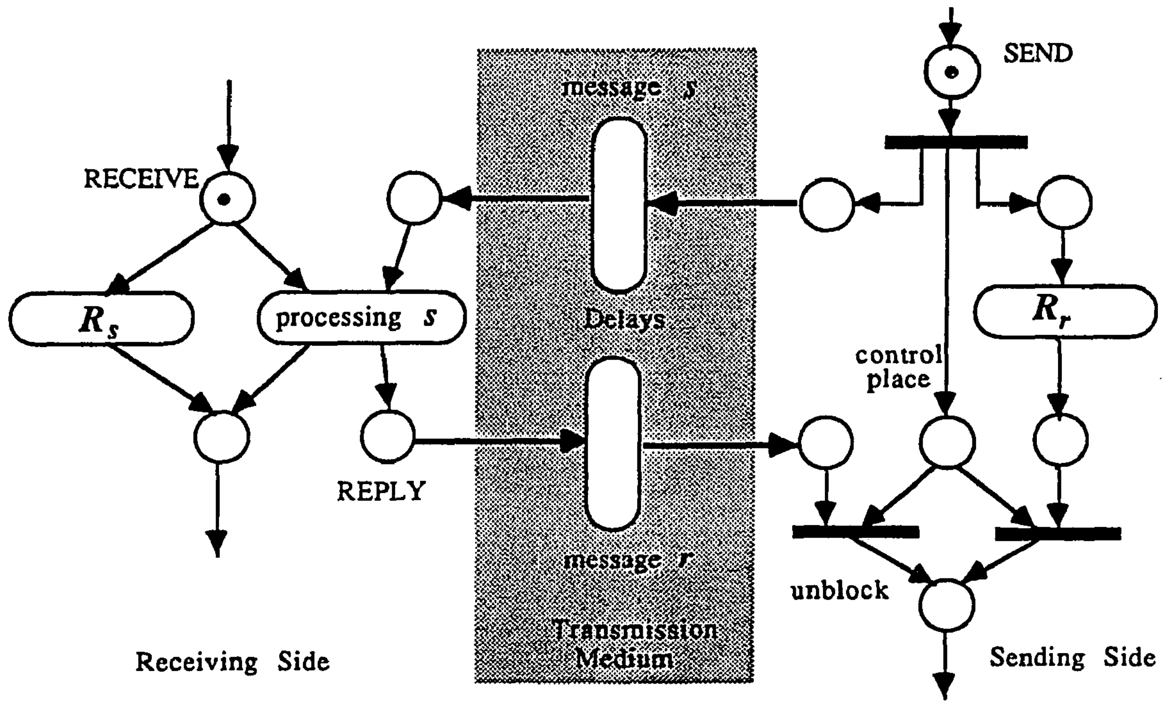
Default activities are implemented as follows. While waiting for an outstanding message r, the PN may or may not choose to execute the default activity R_r. The PN's next action will depend on the following two cases:

CS1. R_r is chosen and completed before r arrives. The PN proceeds as if the precedence constraints imposed by r were met, and does not send a REPLY to unblock the communicating partner from which the r was sent.

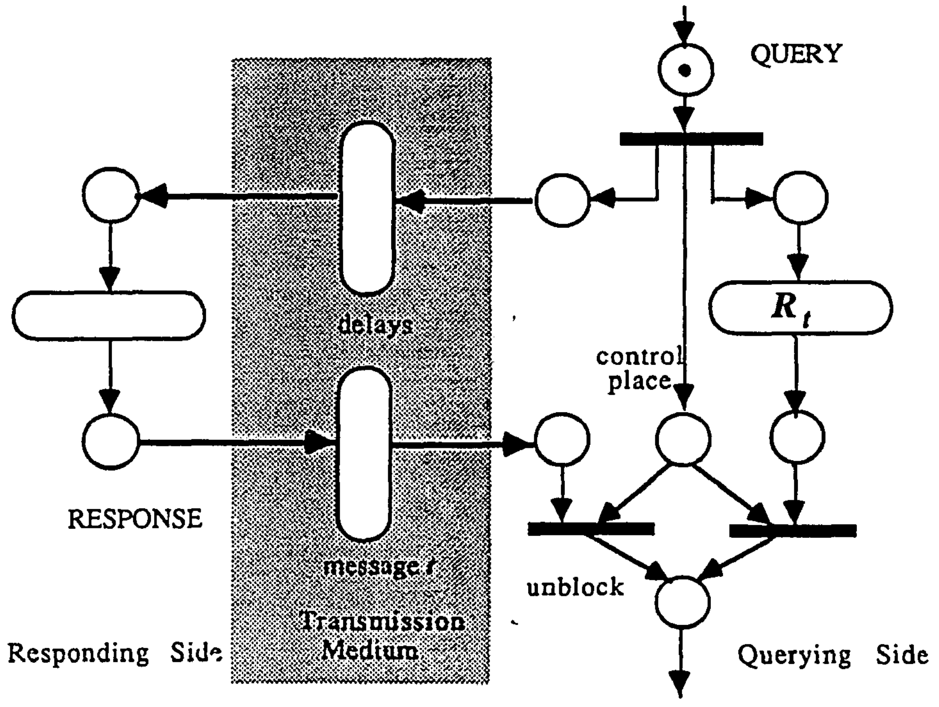
CS2. r arrives before completing (choosing) R_r. The PN stops executing (choosing) R_r, processes r, and replies to unblock its communicating partner.

The GSPN representation for this extended model can be built, as shown in Fig. 6.1, by adding a timed transition representing R_r to the original GSPN. Notice that in Fig. 6.1(a), where the extended GSPN model for SEND-RECEIVE-REPLY or WAITFOR is shown, if the receiving PN gives up on waiting for an outstanding message s and completes a default activity R_s instead, then the sending PN will eventually be forced not to wait for the associated reply message r (which will never arrive anyway), and, instead, execute R_r. Consequently, a control place is added to assure that only one immediate transition will be fired after either receiving the expected message or completing the corresponding default activity R_r. On the other hand, from CS2, if message s arrives before completing R_s, then the receiving PN switches immediately to processing s provided a free processor is available. The extended CTMC model can then be constructed by performing reachability analysis on the extended GSPN. For convenience, the extended CTMC model will simply be called the CTMC model for the rest of the chapter as long as it does not cause any confusion/ambiguity.

For example, consider the task system shown in Fig. 6.2 (which will also be used throughout the chapter). It shows an extended GSPN where three periodic tasks T_1, T_2 and T_3 are pre-assigned to N_1, N_2 and N_3, respectively. T_1, T_2 and T_3 with periods $p_1 = 5$, $p_2 = 10$ and $p_3 = 5$ are invoked twice, once and twice, respectively, within the planning cycle $I = [0, 10)$. T_1 queries T_2 for information while T_3 communicates with T_2 for synchronization. Suppose default activities, labeled as a_{14} and a_{15} in Fig. 6.2, are provided only for T_1 to estimate the values of two parameters related to T_2's response, whereas no



6.1(a). SEND-RECEIVE-REPLY or WAITFOR



6.1(b). QUERY-RESPONSE

Figure 6.1. Extended GSPNs

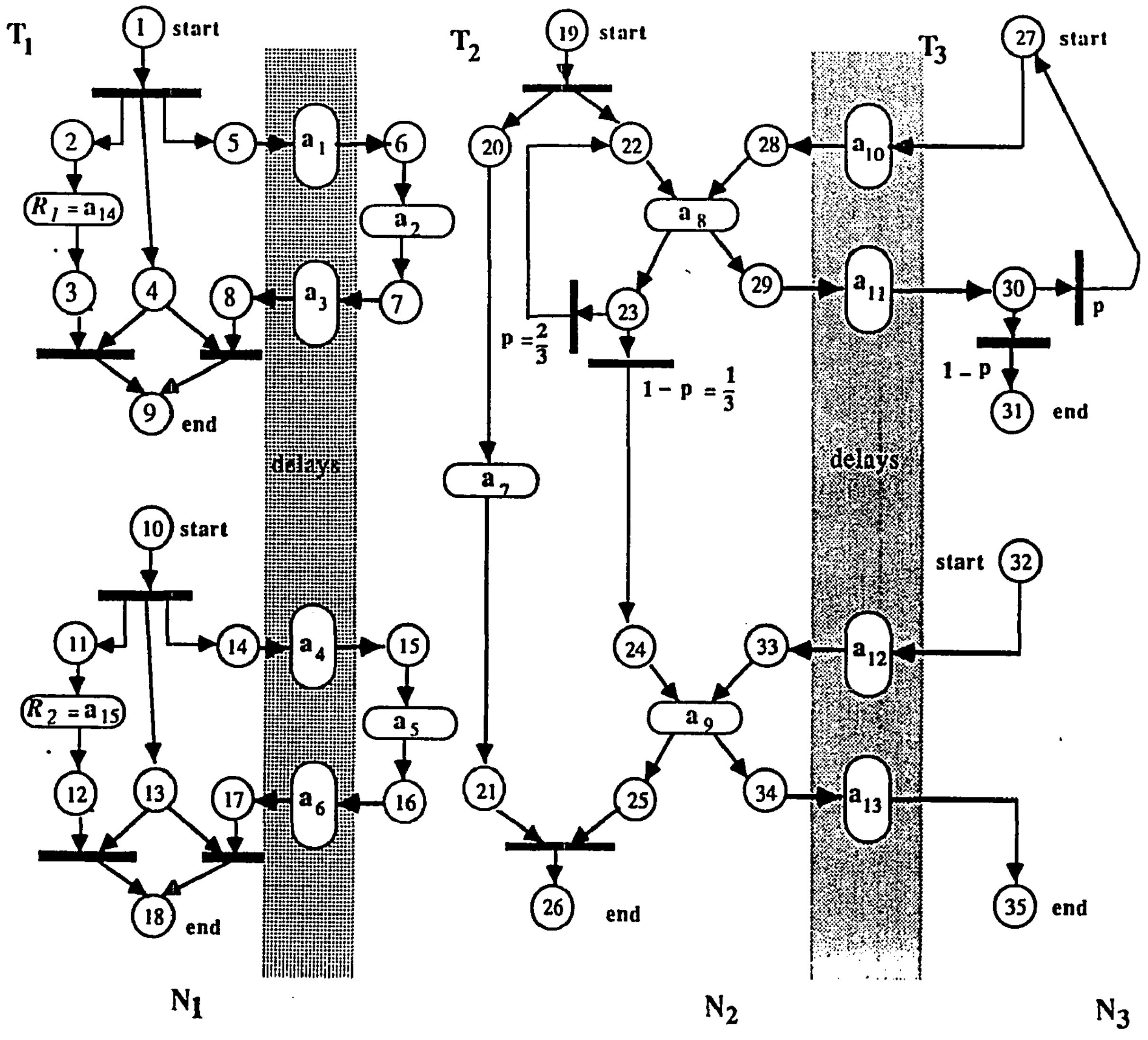


Figure 6.2. The Extended GSPN of the Example

default activity is provided for synchronization between T_2 and T_3. Following the common practice, immediate transitions are denoted by bars and timed transitions by ovals, and integer numbers in the circles are place numbers. All but a_8 and a_{11}, each of which creates a branching set, are lone transitions. Further, to assure successful synchronization between T_2 and T_3, the random switches at places ϕ_{23} and ϕ_{30} are assumed to be identical.

At the beginning of I, a token is generated in each of ϕ_1 (thus, ϕ_2, ϕ_4 and ϕ_5), ϕ_{19} (ϕ_{20} and ϕ_{22}) and ϕ_{27}; at time $t \in (0, 5)$, states evolve based on event-driven transition firings. At $t = 5$ when T_1 and T_3 are invoked again, the marking of the extended GSPN is determined by: 1) a token is generated in ϕ_{10} and ϕ_{32}, 2) all tokens in ϕ_1 — ϕ_9 and ϕ_{27} — ϕ_{31} are removed to abandon unfinished invocations of T_1 and T_3,[1] and 3) the tokens in ϕ_{19} — ϕ_{26} are determined simply by event-driven transition firings since $t = 0$. At $t \in (5, 10)$, the state evolution is again determined by event-driven transition firings until $t = 10$ when the system repeats itself for the next planning cycle. Let $\mu_1 = \mu_3 = \mu_4 = \mu_6 = \mu_{15} = 2$, $\mu_2 = \mu_5 = 4$, $\mu_7 = \mu_{10} = \mu_{11} = \mu_{12} = \mu_{13} = \mu_{14} = 1$, $\mu_8 = 6$, $\mu_9 = 4$, and branching probabilities $p = 2/3$, where μ_i is the execution rate of a_i. After performing reachability analysis on the extended GSPN, a total of 90 (108) non-vanishing states are generated within $I_1 = [0, 5)$ ($I_2 = [5, 10)$) as shown in the Appendix B (Appendix C).

6.3. Optimal Centralized Periodic TMSP

The TMSP is formulated as a semi-Markov decision process (SMDP) by discretizing a planning cycle into a number of small intervals or epochs. We first justify why it is sufficient to base the criterion function on one planning cycle, rather than the whole mission lifetime. Then, the three characterizing factors of the SMDP, *decision set*, *one-step transition probability*, and *one-step cost* [KuV86], are identified/calculated in the context of the

[1]An invocation of a periodic task is simply discarded if it is not completed before the next invocation, since its execution results will become obsolete.

underlying problem so that functional equations can be built and the dynamic programming (DP) technique applied for an optimal solution.

Consider a renewal reward process $\{N(t),\ t \geq 0\}$ [Ros83] with inter-arrival times V_n, $n = 1, 2, \cdots$, and suppose further that a reward R_n is earned at the time of the n-th renewal. Let $C(t) \overset{\Delta}{=} \sum_{n=1}^{N(t)} R_n$ be the total reward earned by time t. For every $n \geq 1$ if $R_n = R$ is independently, identically distributed (iid) and $E[V_n] = E[V]$ is finite, then we get the long-term expected reward $\lim_{t \to \infty} C(t) / t = E[R] / E[V]$ with probability one. This implies that the policy that optimizes the long-term expected reward also optimizes the mean reward in any renewal cycle. Thus, the optimal solution to the TMSP is the one that minimizes the expected number of periodic tasks missing deadlines within a planning cycle $I = [0, L)$.

Unfortunately, there is no closed form solution to this type of optimization problems. A well-known approach is to approximate the optimal solution by discretizing the planning cycle into a number of equally-spaced epochs and then applying the DP technique to find an optimal solution at each of these time epochs and states [Dij84, Mil68].

Following the common practice, we will divide the planning cycle I into V equally-spaced epochs, or intervals of length h each, such that each task invocation time coincides with one of these epochs. Since the solution algorithm of DP is well-known, we need to formulate the problem to be solved in terms of DP's three characterizing factors: decision sets, one-step transition probability and one-step cost. In what follows, these three factors are established.

6.3.1. Decision Set

The decision set D_i for state $s_i \in S$ — the total state space of the CTMC model — is the set of all scheduling options available when the system is in s_i. D_i is determined by combining over all N_k's the set of all scheduling options available to N_k when the system is in s_i. For example, if there is only one processor in each PN, then the set of scheduling options available to N_k is the set of all activities ready for execution on N_k. A policy π specifies which activity to choose for each PN for each given state $s_i \in S$ at epoch v, $0 \le v \le V-1$. The policy space Π is defined as the set of all such π's. For convenience, the set of all activities chosen under π in state s_i by all PNs at the v-th epoch is denoted by $ATS_i^{\pi}(v)$.

6.3.2. One-Step Transition Probability

The epochs resulting from the discretization of a planning cycle serve as the stages of the corresponding dynamic programming network [Den82], i.e., transitions occur only between states of adjacent epochs. Let $P_{ij}^{\pi}(v)$ denote the one-step transition probability from state $s_i \in S$ at epoch v within $I_u = [\omega_u, \omega_{u+1})$ to $s_j \in S$ at epoch $v+1$ under a certain policy π. Also, let L_i be the total transition rate of all transitions in $ATS_i^{\pi}(v)$, i.e., $L_i = \sum_{j \in A_i} \xi_{ij}\mu_{ij}$, where $A_i = \{s_j \mid \Lambda_u(s_i, s_j) \in ATS_i^{\pi}(v)\}$, ξ_{ij} is the branching probability, and μ_{ij} the transition rate from s_i to s_j. Depending on whether or not a task will be invoked at the next epoch, $P_{ij}^{\pi}(v)$ is determined as follows.

TP1: $(v+1)h < \omega_{u+1}$.

$$P_{ij}^{\pi}(v) = \begin{cases} \xi_{ij}\mu_{ij}h & j \in A_i \text{ and } j \ne i \\ 1 - L_i h + \xi_{ii}\mu_{ii}h & j = i \\ 0 & j \notin A_i \text{ and } j \ne i \end{cases} \tag{6.1}$$

because the execution time of each activity is assumed to be exponentially distributed.

TP2: $(v+1)h = \omega_{u+1}$. The transition probability in this case is similar to that of TP1 except that the time-driven transition function Θ_u is fired. Specifically,

$$P_{ix}^{\pi}(v) = \sum_{j \in B_x} P_{ij}^{\pi}(v), \qquad s_x \in S_{u+1}, \tag{6.2}$$

where $B_x = \{s_j \mid \Theta_u(s_j) = s_x\}$, and $P_{ij}^{\pi}(v)$ is the transition probability obtained from Eq. (6.1).

6.3.3. One-Step Cost

Before deriving the one-step cost corresponding to the mean number of tasks missing deadlines, it is necessary to recall the concepts of the *goal state* and its *realized set* introduced in Chapter 5. In the CTMC model, concurrent task execution can be viewed as the movement of tokens in the corresponding GSPN. A place in the GSPN is said to be *realized* if it has a token, implying the completion of all the activities that precede the place. In other words, a deadline in the task system is essentially associated with the realization of a place. A place is time-critical if a deadline is associated with it. A state s_i containing a set R_i of realized time-critical places is then called a goal state with realized set R_i. A time-critical place is assumed to occur only at the conclusion of the task containing the place. This assumption allows the deadline of a task invocation to be set to its next invocation time. Let Ψ_u and $\Phi_i = \Psi_u - R_i$, $1 \le u \le l$, denote, respectively, the set of all current invocations that should be completed before the next task invocation time ω_{u+1}, and those that have not been completed while in s_i before ω_{u+1}. Since whether or not a task misses its deadline is not known until its next invocation, $\forall\, \pi \in \Pi$ and $\forall\, s_i \in S_u$, it is natural to set the cost function as:

$$c^{\pi}(v, s_i) = \begin{cases} 0 & \dfrac{\omega_u}{h} \le v < \dfrac{\omega_{u+1}}{h} - 1 \\[2ex] \sum\limits_j P_{ij}^{\pi}(v)\, |\Phi_j| & v = \dfrac{\omega_{u+1}}{h} - 1, \end{cases} \tag{6.3}$$

where $P_{ij}^{\pi}(v)$ is the one-step transition probability without considering the time-driven transition function Θ_u and $|\Phi_j|$ represents the number of time-critical places that should be, but will not be, realized at ω_{u+1}.

Under the assumption that a central monitor exists and has a perfect observation of all s_i's, one can construct the following functional equations and then apply backward recursion to derive an optimal solution [KuV86]:

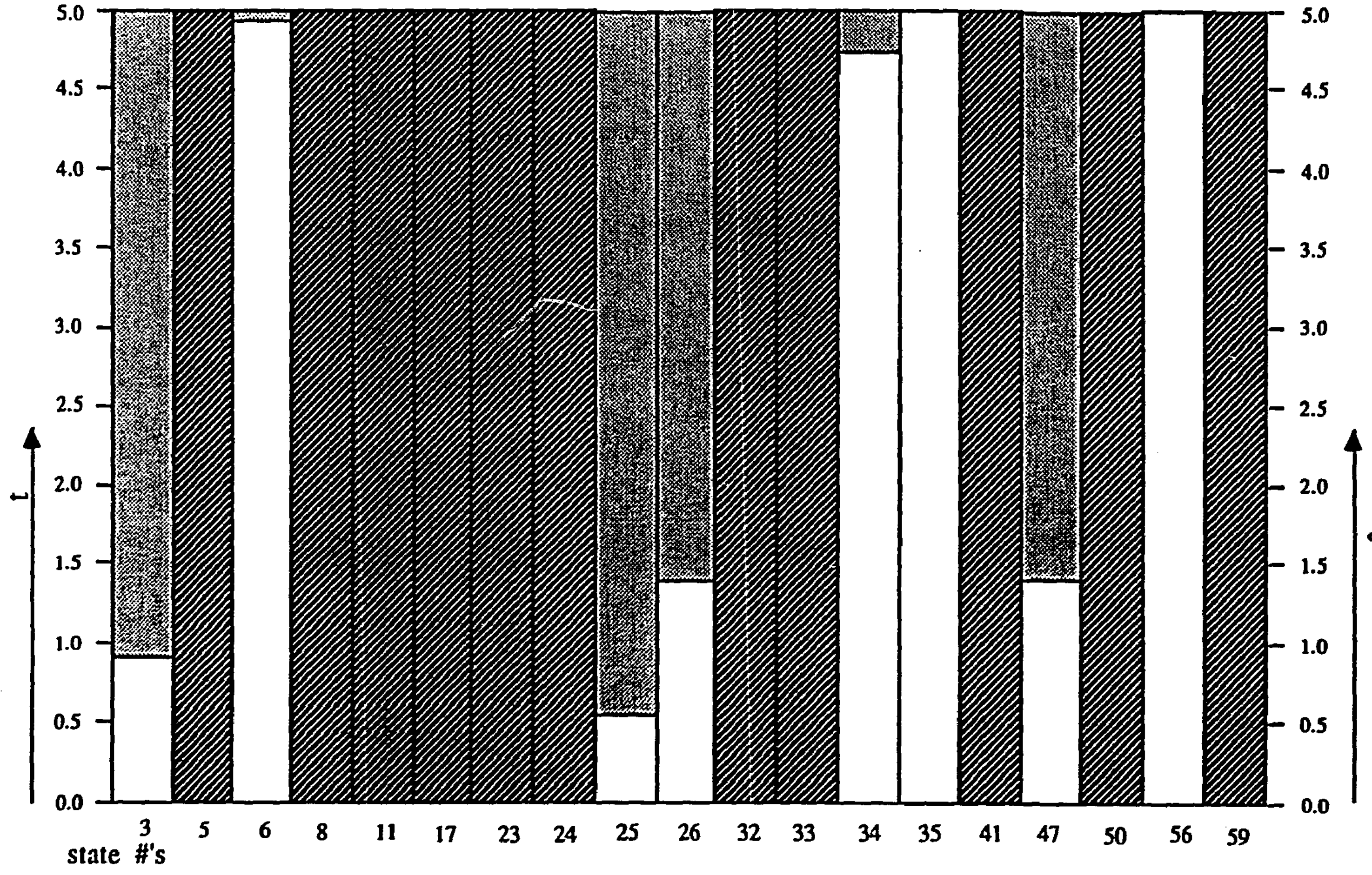
$$U_V(s_i) = 0\,, \tag{6.4}$$

$$U_v(s_i) = \min_{\pi \in \Pi} \left\{ c^{\pi}(v, s_i) + \sum_j P_{ij}^{\pi}(v)\, U_{v+1}(s_j) \right\}, \qquad 0 \le v < V. \tag{6.5}$$

Note that the minimum expected number J^* of tasks missing deadlines — which is achieved under an optimal policy π^* — is equal to $J^* = U_0(s_o)$, where s_o is the unique starting state of the CTMC model of the task system.

Consider again the task system of Fig. 6.2 as an example. Since alternative decisions are available only to N_2, it is necessary to derive the optimal policy for N_2 only. Recall that the deadlines for the first and second invocations of T_1 and T_3 are 5 and 10, respectively, and that for T_2 is 10. To obtain a good approximate solution, the planning cycle $I = [0, 10)$ is discretized into $V = 1000$ intervals of length $h = 0.01$ each. The first epoch (v=0) is the beginning of I while the last epoch, i.e., $v = V = 1000$, is the beginning of the next planning cycle. Our objective is to find the optimal policy π^* that specifies the activity to be chosen by N_2 at each epoch v, $0 \le v \le 999$, such that the expected number of task invocations missing their deadlines within I is minimized.

The optimal policy obtained by solving Eq. (6.5) backward recursively for each epoch v is shown in Figs. 6.3(a) and (b) which correspond to $I_1 = [0, 5)$ and $I_2 = [5, 10)$, respectively. The optimal decisions at different epochs are shown only for those states in which N_2 has



Legend: ▨ : a_2 □ : a_7 ▨ : a_8

Figure 6.3(a). Optimal Decisions of N_2 Within $I_1 = [\ 0,\ 5)$

multiple options. For example, as the system is in state $s_3 = (2, 4, 6, 20, 22, 27) \in S_1$ (see Appendix B.) at time $t \in I_1$, the optimal policy for N_2 (Fig. 6.3(a)) is to execute a_7 if $t \leq 0.89$, and execute a_2 (i.e., respond to T_1's query) if $t \geq 0.90$. Suppose the system is in s_3 at $t < 0.89$, when N_1 executes $R_1 = a_{14}$, N_2 executes a_7 and N_3 sends a message to N_2. Assume further that a_{14} is finished at the next epoch $t+h$ bringing the system to state s_6. Then, as shown in Fig. 6.3(a), the optimal policy for N_2 is still to execute a_7. However, if the message from N_3 arrives at N_2 before completing a_7 or a_{14} (i.e., bringing the system to s_{11}), then N_2 should reply to N_3 (i.e., execute a_8) immediately, instead of executing a_7. The optimal policy in any other state and at any other time can be obtained similarly from Fig. 6.3. Notice that a_8 (a_9) is always executed before a_2 (a_5) or a_7 within I_1 (I_2). This is because the completion of a_8 (a_9) is essential for those of T_2 and both (the second) invocations of T_3. Thus, a_8 is more "urgent" than a_7 and a_2, since the completion of the latter enables only one task to complete. Also, in s_4, s_9, s_{12} and s_{18} within I_2, a_5 is always executed before a_7 (Fig. 6.3(b)). This is obvious, since at these states, the token is in ϕ_{22}, rather than ϕ_{24}, meaning that T_2 is stuck at waiting hopelessly for a message that will never arrive. Hence, it is meaningless for N_2 to execute a_7 since T_2 will not be completed in time anyway.

The optimal cost $J^* = 1.3045$, which is rather high. Two reasons explain this subtlety: (1) the absence of default activities for N_2 in waiting for outstanding messages, and (2) the tightness of deadlines. To see how the tightness of deadlines affects performance, we solved the same problem with each deadline extended and with $V = 10{,}000$. Specifically, the deadlines for the first invocations of T_1 and T_3 are extended to 50, those for T_2 and the second invocations of T_1 and T_3 to 100 while maintaining $h = 0.01$. These changes resulted in $J^* = 0.00056$, a significant reduction. This reduction can be explained by the fact that with a much higher probability, the token originally in ϕ_{22} can now move to ϕ_{24} before $t = 50$ and, thus, each invocation can be completed before its deadline. It is worth pointing out that the

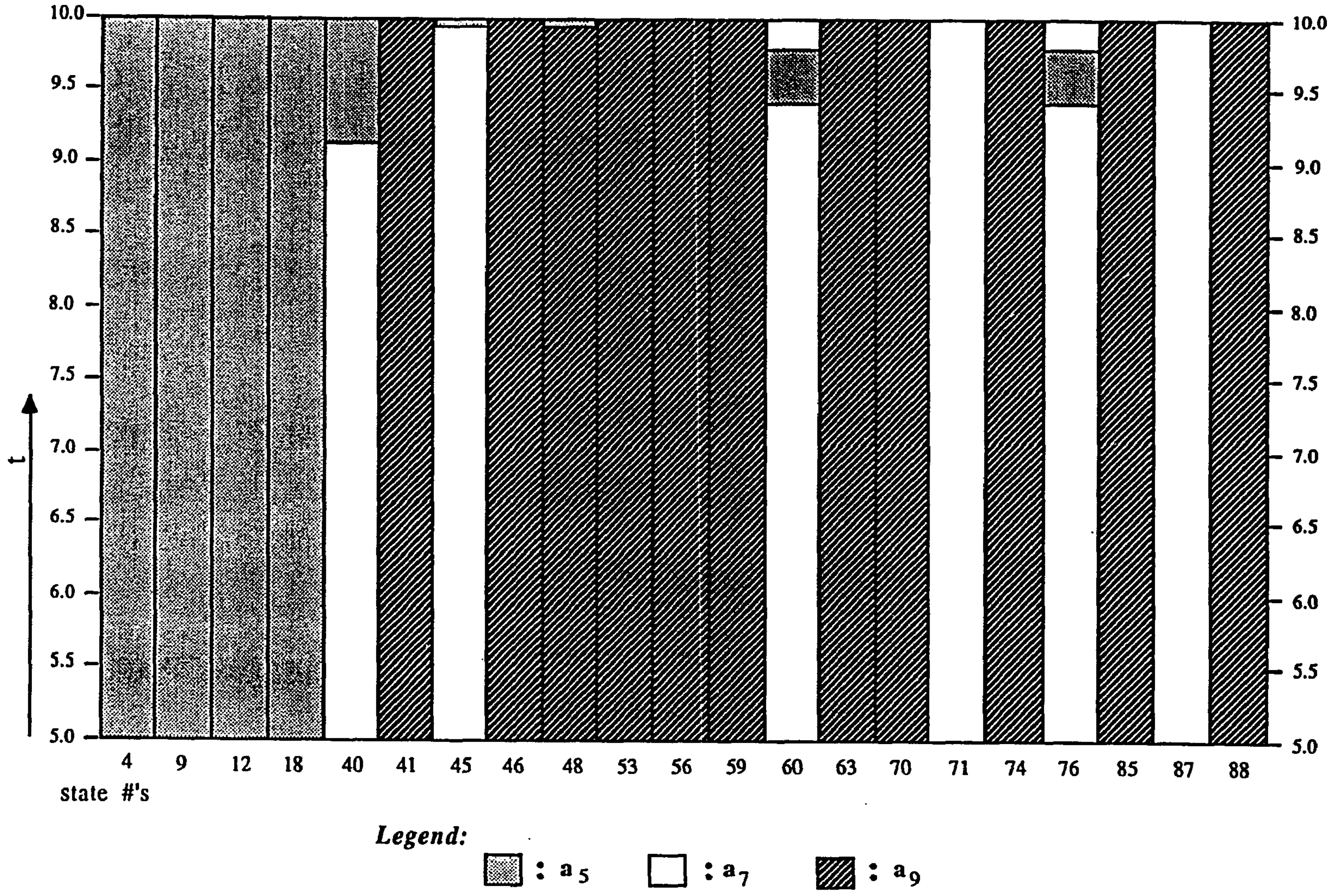


Figure 6.3(b). Optimal Decisions of N_2 Within I_2 = [5, 10)

optimal policies of these two cases for N_2 are not necessarily the same nor similar.

6.4. Sub-Optimal Decentralized TMSP

Since there does not usually exist a central monitor with complete knowledge of every PN's current task execution status, it is important to derive a decentralized policy under which each PN makes its own part of scheduling decision using its locally available information such that the overall system performance is still optimized (to a lesser extent than the centralized case). A PN's local information consists of its current state and possibly out-dated information on the other PNs. Specifically, each PN synchronously and periodically broadcasts its local state to all the other PNs, based on which scheduling decisions are made. Simultaneous broadcasts can be accomplished by using a system-wide common time base established by either a software method [LaM85] or hardware method [ShR87]. (Note that there are many other reasons why a common time base is needed as outlined in [Lam84]). Such broadcasts can be accomplished, for instance, by using a method similar to the one proposed in [GrS88], where fault-tolerant distributed communication is achieved through maintaining synchronous replicated data. The algorithms used to maintain and recover synchronous replicated data include atomic broadcast, handshake, membership, and clock and join synchronizations. Except for the handshake algorithm, all the other algorithms are based on the notion of *diffusion*, or the passing of information from neighbor to neighbor in a point-to-point communication network. Further, to prevent the broadcast messages from saturating the network, we assume that each PN broadcasts state information only after the previous broadcast state has been received by all other PNs.

Theoretically, the more frequently does each PN receive the other PNs' state information, the better scheduling decisions it will make. However, frequent state broadcasts induce a higher overhead to normal inter-task communications, thus degrading the overall system performance. Our objective is then to find an optimal frequency of state broadcasts and the corresponding optimal decentralized policy that minimizes the expected number of tasks missing deadlines.

Given a state broadcast frequency, system performance depends heavily on how each PN uses the broadcast information and makes a best possible scheduling decision. This problem is known as a dynamic team[2] decision problem with delayed shared information structure [HoC72], and is frequently encountered in large scale systems, such as transportation, data communication and power systems. Unfortunately, dynamic team decision problems are known to be extremely difficult to solve [Wit71, Wit73] except in a particular class of problems with so called *one-step delay sharing* (1SDS) information pattern [ADM87, HsM82, VaW78, YoS75]. Therefore, a sub-optimal solution is derived using the DP technique.

In the following subsections, we first derive the delays both in state broadcasts and in normal inter-task communications. Then, the ''probability state'' of the DP network, the set of action rules, one-step transition probability, one-step cost and the functional equations are described.

6.4.1. State Broadcasts and Their Effects on Normal Communication

The communication subsystem is approximated by two single-server queues: an M/M/1 queue for inter-task communications and a D/D/1 queue for state broadcasts.

Communication delays depend essentially on what portion of the communication subsystem's power is used to serve each of these two queues. Let λ_x and b_x be respectively the arrival rate of inter-task messages and the average service need measured in number of bits for each inter-task message. For the D/D/1 queue, let λ_y and b_y be the number of synchronous state broadcasts per unit time and the service need for each broadcast (to be

[2]A team is said to be *dynamic* if the action of one member affects the information of another member. Otherwise, the team is said to be *static* [MaR71].

elaborated further). Then, the portion of the communication subsystem's power for serving inter-task messages is $\xi_x = \dfrac{\lambda_x b_x}{\lambda_x b_x + \lambda_y b_y}$ and that for state broadcasts is $\xi_y = 1 - \xi_x = \dfrac{\lambda_y b_y}{\lambda_x b_x + \lambda_y b_y}$. Recall that in the CTMC model without state broadcasts, the delay of an inter-task message i was represented by an exponentially distributed random variable with rate μ_i. The average system (sojourn) time $\bar{S}$ of messages in the communication subsystem is then approximated by $\bar{S} = \dfrac{1}{n} \sum_{i \in A} \dfrac{1}{\mu_i}$, where A is the set of all inter-task messages within a planning cycle and $n = |A|$. It follows from [Kle75] that the service rate μ_x of the M/M/1 queue becomes $\mu_x = \lambda_x + 1/\bar{S}$. Thus, given ξ_x, the "adjusted" service rate μ_x' of inter-task messages is determined as $\mu_x' = \xi_x \mu_x = \xi_x (\lambda_x + 1/\bar{S})$. This means that the average sojourn time $\bar{S}'$ of inter-task messages becomes

$$\begin{aligned} \bar{S}' &= \frac{1}{\mu_x' - \lambda_x} = \frac{1}{\xi_x \mu_x - \lambda_x} \\ &= \frac{1}{\xi_x (\lambda_x + 1/\bar{S}) - \lambda_x} = \frac{\bar{S}}{\xi_x - \xi_y \lambda_x \bar{S}} . \end{aligned} \tag{6.6a}$$

Notice that $0 < \xi_x - \xi_y \lambda_x \bar{S} \le 1$; the total traffic density will otherwise saturate the communication subsystem. Given $\bar{S}'$, the adjusted transition rate μ_i' as a result of introducing state broadcasts becomes:

$$\mu_i' = \mu_i \frac{\bar{S}}{\bar{S}'} = \mu_i \left[\xi_x - \xi_y \lambda_x \bar{S} \right]. \tag{6.6b}$$

Since the communication subsystem's power is actually allocated indiscriminately among all the messages, $\lambda_x W_x = \lambda_y W_y$ must hold provided the communication subsystem is not saturated, where W_x (W_y) is the average service time of each inter-task message (state broadcast) at the M/M/1 (D/D/1) queue. Therefore, the delay (sojourn time) of each state broadcast, W_y, can be expressed as:

$$\begin{aligned} W_y &= \frac{\lambda_x}{\lambda_y} W_x = \frac{\lambda_x}{\lambda_y} \frac{1}{\mu_x'} \\ &= \frac{\lambda_x}{\lambda_y} \frac{1}{\xi_x(\lambda_x + 1/\bar{S})} \\ &= \frac{\lambda_x}{\lambda_y} \frac{\bar{S}}{\xi_x + \xi_x \lambda_x \bar{S}} . \end{aligned} \tag{6.7}$$

In the above derivation, while each inter-task message is treated as a single arrival at the M/M/1 queue, all the states broadcast simultaneously by all PNs at a given time are combined into a single arrival at the D/D/1 queue. The length or service need of this "combined message" is the sum of those of $m(m-1)$ individual messages, and no individual message is considered received by a PN until the combined message at the D/D/1 queue is completely served. Note that all of the $m(m-1)$ broadcast messages will not arrive at their respective PNs at the same time even though they are broadcast simultaneously, and that a message is not considered usable by the receiving PN until all the other PNs receive their respective messages. The inter-broadcast interval $1/\lambda_y$ must always be larger than W_y, since broadcast messages alone will otherwise require more power than the communication subsystem's capacity [Kle75]. A PN broadcasts its state only after its previously broadcast state has been received by all other PNs. This broadcast scheme is essential for the applicability of the DP technique.

In what follows, we assume that each PN broadcasts its state B times within a planning cycle $I = [0, L)$ with the inter-broadcast interval $g = L/B \geq W_y$. That is, the first broadcast is made at $t = 0$,[3] the last at $t = (B-1)g$, and all messages broadcast at t will be received at $t + W_y \leq t + g$. Further, the time axis is discretized into epochs in such a way that states are always broadcast and received at some epoch boundaries. In what follows, the decentralized

[3]The first broadcast is actually not necessary since all PNs already know the unique starting state S_0 of the system. The first broadcast is introduced simply to meet the requirements of the arrival pattern of the D/D/1 queue.

TMSP is formulated in such a way that the DP algorithm can be used to find a sub-optimal solution. Like its centralized counterpart, the decentralized TMSP is formulated by the characterizing factors of the underlying DP algorithm.

6.4.2. Probability States of the DP Network

For each epoch v in $I_u = [\omega_u, \omega_{u+1})$ define a "probability state" as the vector $\rho \overset{\Delta}{=} [p_1, p_2, \cdots, p_{|S_u|}]$, where p_i is the marginal probability that the system is in s_i at time vh. Obviously, there are infinitely many probability states. However, since 1) both $|S_u|$ and the number of available policies are finite and 2) W_y time unit old (global) state information is available (via state broadcasts) to all PNs once every g time units, only a finite subset of probability states will be generated.

6.4.3. Set of Action Rules

The set of all action rules that N_k can possibly take within I_u is the Cartesian product of the sets of all scheduling options available to N_k when N_k is in each of its local states within I_u. Thus, the entire set Γ_u of action rules for all m PNs within I_u becomes the Cartesian product of the sets of action rules for each N_k.

When some components of ρ in I_u are zero, only a subset of the action rules in Γ_u are admissible, i.e., only those with non-zero marginal probabilities are admissible. To determine this set of admissible action rules for a certain ρ, we first identify the set of all local states of N_k within I_u, each of which has a non-zero marginal probability at ρ. (This can be achieved by choosing each of those local states of N_k such that there exists at least a state s_i containing the local state with a non-zero marginal probability in ρ.) After constructing the set of all action rules for N_k from this set of local states, the set of all admissible action rules $\hat{\Gamma}_u(\rho)$ is then built as the Cartesian product of the sets of such action rules over all PNs.

6.4.4. Probability State and Its One-Step Transition Probability

For the centralized TMSP, the one-step transition probability was one of the three components needed in its DP formulation. For the decentralized case however, except at those epoches where state broadcasts are received, it is the probability state, rather than the probability of jumping into a state, that is essential to the DP formulation. This is because the system always jumps from one such probability state to another (of the next epoch). As will be seen below, a unique "destination" probability state can be identified with the one-step transition probability matrix whose elements are obtained from Eqs. (6.1) and (6.2).

Suppose the system is in probability state ρ at epoch $\nu \in I_u$. The probability states generated at epoch $\nu+1$ and their one-step transition probabilities are determined depending on whether or not epoch $\nu+1$ represents the time of receiving state broadcasts.

A: $\nu+1$ is not a message receipt epoch:

Each admissible action rule $\hat{\gamma} \in \hat{\Gamma}_u(\rho)$ identifies a unique decision for each PN for each state $s_i \in S_u$ if $p_i \neq 0$. Given that the system is in s_i at epoch ν and $(\nu+1)h < \omega_{u+1}$, one can use Eq. (6.1) to determine the marginal probability $P_{ij}^{\hat{\gamma}}(\nu)$ that the system will move to s_j at epoch $\nu+1$ if $\hat{\gamma}$ is adopted. Given the marginal probability p_i of s_i at epoch ν, one can compute the marginal probability p_j' of s_j at epoch $\nu+1$ as: $p_j' = \sum_{i=1}^{|S_u|} p_i \, P_{ij}^{\hat{\gamma}}(\nu)$. Therefore, the unique probability state $\rho' \triangleq (p_1', p_2', \cdots, p_{|S_u|}')$ at epoch $\nu+1$ given $\rho \triangleq (p_1, p_2, \cdots, p_{|S_u|})$ at epoch ν is determined by:

$$\rho' = \rho \, \mathbf{P}^{\hat{\gamma}}(\nu) \, , \tag{6.8}$$

where $\mathbf{P}^{\hat{\gamma}}(\nu)$ is the one-step transition matrix whose elements are determined by Eq. (6.1). When $(\nu+1)h = \omega_{u+1}$, the one-step transition matrix is determined similarly except with elements $P_{\tilde{u}x}^{\hat{\gamma}}(\nu)$'s, $s_x \in S_{u+1}$, determined by Eq. (6.2). For notational convenience, we write $\rho' = G^{\hat{\gamma}}(\rho))$ to denote that ρ' is the unique probability state from ρ given $\hat{\gamma}$.

From Eq. (6.8), the probability states ρ''s form the $|\hat{\Gamma}_u(\rho)|$ branches of ρ, which, in turn, is one branch of another probability state at epoch $\nu-1$, and so on. Given ρ' at epoch $\nu+1$ in between states were broadcast and received (Fig. 6.4), this relationship continues to hold through a particular probability state, written as $X(\rho')$, at the epoch when states were broadcast, and finally rooted at a unique probability state at an epoch where the last state broadcasts were received. Specifically, $X(\rho')$ represents the prior probability of each (global) state at the epoch of broadcast, and the particular state realized at that epoch will be globally known after W_y time units. $X(\rho')$ plays an important role in determining the one-step transition probability in the following case.

B: $\nu+1$ is a message receipt epoch:

Since W_y time unit old global state information is available to each PN at epoch $\nu+1$, the probability states at epoch $\nu+1$ are not generated by Eq. (6.8). Rather, from the idea of one-step delayed sharing (1SDS) information pattern, and by extending the delay to $s = W_y/h > 1$ steps[4], a total of $|S_u|\ |\Gamma_u|^{s}$[5] probability states are possible at each of such epoches. Each of these probability states corresponds to: (i) a specific state deduced from the state broadcasts received and (ii) a specific sequence of scheduling rules adopted from the state broadcast epoch to epoch $\nu+1$. In other words, each of these probability states can be identified (Fig. 6.4) as an $(s+1)$-tuple $(s_i, \hat{\gamma}'(0), \hat{\gamma}'(1), \cdots, \hat{\gamma}'(s-1))$, where s_i is the actual system state at W_y time units ago, and $\hat{\gamma}'(e)$, $e = 0, 1, \cdots, s-1$, represents the action rule adopted at the e-th epoch since the last state broadcast. Similarly, one may equivalently represent each

[4]The information pattern resulting from this extension is not the same as the common *multi-steps delayed sharing* information pattern, where the current states are broadcast, and multi-steps delayed messages are received at each epoch.

[5]Notice that if we set $s = 0$, then the number of states generated is consistent with the centralized case.

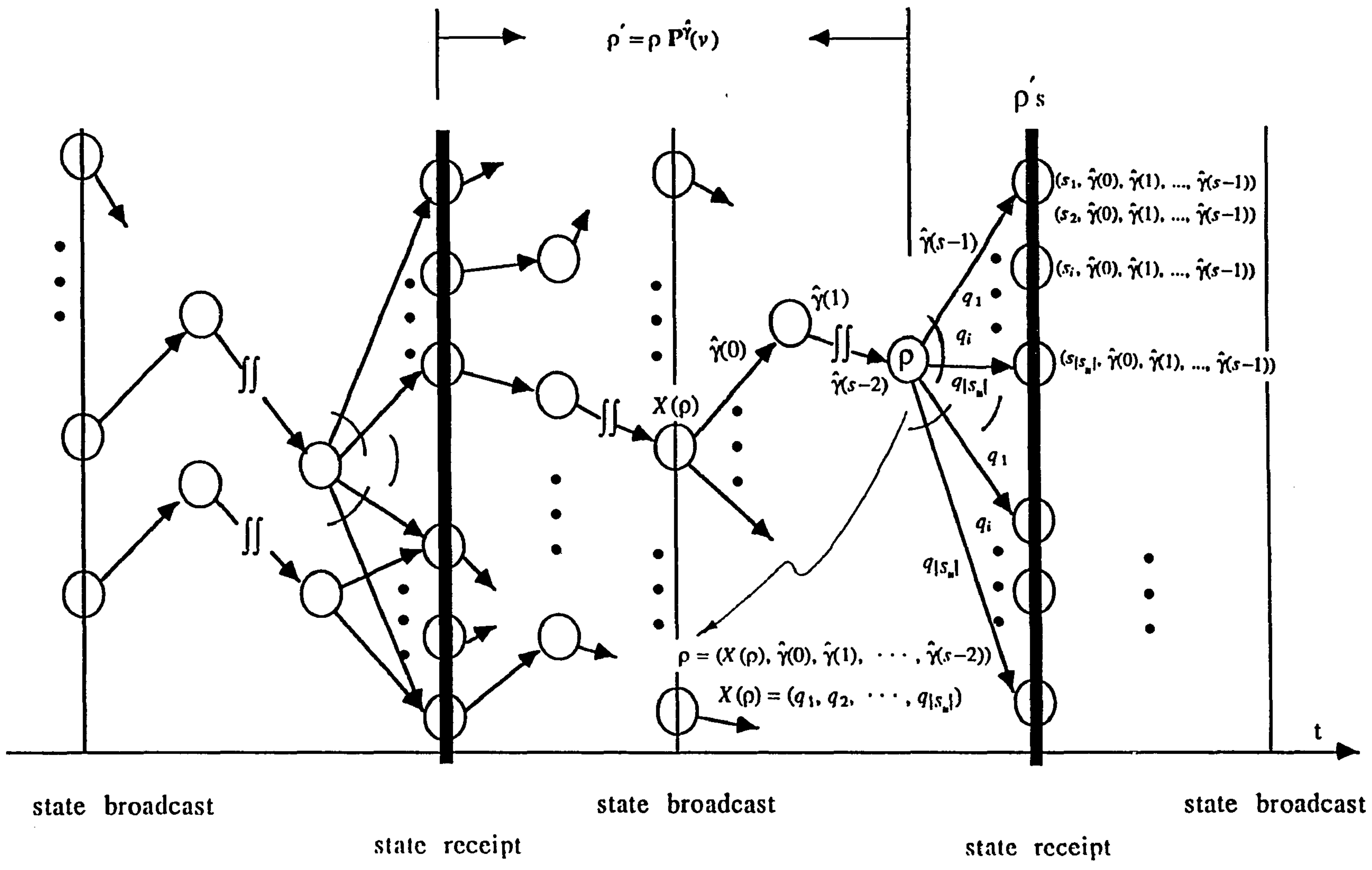

$\rho' = \rho\, \mathbf{P}^{\hat{\gamma}}(\nu)$
$\rho' s$
$(s_1, \hat{\gamma}(0), \hat{\gamma}(1), \ldots, \hat{\gamma}(s-1))$
$(s_2, \hat{\gamma}(0), \hat{\gamma}(1), \ldots, \hat{\gamma}(s-1))$
$(s_i, \hat{\gamma}(0), \hat{\gamma}(1), \ldots, \hat{\gamma}(s-1))$
$(s_{|s_n|}, \hat{\gamma}(0), \hat{\gamma}(1), \ldots, \hat{\gamma}(s-1))$
$\hat{\gamma}(s-1)$
$\hat{\gamma}(1)$
$\hat{\gamma}(0)$
$\hat{\gamma}(s-2)$
$X(\rho)$
ρ
q_1
q_i
$q_{|s_n|}$
q_1
q_i
$q_{|s_n|}$
$\rho = (X(\rho), \hat{\gamma}(0), \hat{\gamma}(1), \cdots, \hat{\gamma}(s-2))$
$X(\rho) = (q_1, q_2, \cdots, q_{|s_n|})$
t
state broadcast
state receipt
state broadcast
state receipt
state broadcast

Figure 6.4. Probability State and One-Step Transition Probability

probability state ρ at epoch v with an s-tuple $(X(\rho), \hat{\gamma}(0), \hat{\gamma}(1), \cdots, \hat{\gamma}(s-2))$, where $X(\rho)$ is the root of ρ at the epoch the states were broadcast, and $\hat{\gamma}(e)$ the action rule adopted at the e-th epoch since the last state broadcast.

Given ρ and ρ' at epoches v and $v+1$, respectively, let $\hat{\gamma}(s-1)$ denote the decision rule adopted at epoch v. Then, the one-step transition probability from ρ to ρ' can be determined as:

$$Q_{\rho\rho'}^{\hat{\gamma}(s-1)} = \begin{cases} q_i & \text{if } \hat{\gamma}(e) = \hat{\gamma}'(e),\ e = 0, 1, \cdots, s-1, \\ 0 & \text{otherwise,} \end{cases} \tag{6.9}$$

where q_i is an element of $X(\rho)$ representing the marginal probability of the system being in s_i. Since the marginal probability of s_i is q_i within $X(\rho)$, when $s = 0$ and before applying any action resulting from $\hat{\gamma}(0)$, the one-step transition probability from $X(\rho)$ to a particular $\rho' = (p_1', p_2', \cdots, p_{|S_u|}')$ must be q_i, where $p_i' = 1$ and $p_j' = 0$ if $j \neq i$. When $s > 0$, the same conclusion holds as long as the sequence of action rules adopted are the same for ρ and ρ'.

6.4.5. One-Step Cost

The one-step cost $d^{\gamma}(v, \rho)$ corresponding to the expected number of tasks missing deadlines at epoch $v \in I_u$ and probability state ρ given action rule γ can be easily determined using the one-step cost $c^{\pi}(v, s_i)$ obtained from Eq. (6.3) for the centralized TMSP. Specifically, we have

$$d^{\gamma}(v, \rho) = \sum_{i=1}^{|S_u|} p_i c^{\pi}(v, s_i), \tag{6.10}$$

where π is the centralized action rule corresponding to the decentralized action rule γ if the system is known to be in s_i.

6.4.6. Functional Equations and the Sub-Optimal Policy

Unlike the centralized case for which only one pass is needed, three passes are required to derive the sub-optimal decentralized policy. The first pass is to generate all the probability states of the DP network. The second pass solves the functional equations backward recursively to find the "optimal" action rule at each probability state ρ and each epoch ν. Since, except at the message receipt epoches, ρ is not observable by any PN, the third pass is used to identify the "optimal" policy at each epoch. In what follows, the last two passes are described; the first pass has already been described by Eqs. (6.8) and (6.9).

A: Functional Equations

Similarly to the centralized case, the functional equations $U_\nu(\rho)$ is the cost-to-go representing the number of task missing deadlines given the system is in ρ at epoch ν. $U_\nu(\rho)$ can be determined easily by using backward recursion as follows.

$$U_N(\rho) = 0, \tag{6.11}$$

$$U_\nu(\rho) = \min_{\hat{\gamma} \in \hat{\Gamma}_*(\rho)} (d^\gamma(\nu, \rho) + Q_{\rho\rho'}^{\hat{\gamma}(\varsigma-1)} U_{\nu+1}(\rho')) , \tag{6.12}$$

where $Q_{\rho\rho'}^{\hat{\gamma}(\varsigma-1)}$ is the one-step transition probability from ρ to ρ' as determined by Eqs. (6.8) and (6.9). Notice that the minimum expected number J^* of tasks missing deadline, which is achieved under the sub-optimal decentralized policy γ^*, is equal to $U_0(\rho_0)$, where ρ_0 is the probability state representation of the unique starting state s_o of the task system.

B: Sub-Optimal Policy

The sub-optimal policy is found by identifying the path with the least total cost within the DP network. Since ρ is unobservable between two adjacent message receipt epoches, the sub-optimal policy at each epoch is identified using forward recursion as follows. Let ρ_0 be the initial probability state at epoch 0 (i.e., s_o) and $\bar{\gamma}_\nu(\rho)$ the sub-optimal action rule for ρ at epoch ν. Three intervals need to be considered: 1) from epoch 0 to the first message receipt epoch, 2) from the i-th to the (i+1)-th message receipt epoch, $1 \le i \le B-1$, and 3) from the B-th message receipt epoch to the end of the planning cycle. Consider first the interval between epoch 0 and the first message receipt epoch ν_1. Define the probability state ρ_ν^* and action rule γ_ν^* at epoch ν as

$$\rho_0^* = \rho_0 , \tag{6.13}$$

$$\rho_{\nu+1}^* = G^{\bar{\gamma}_\nu(\rho_\nu^*)}(\rho_\nu^*), \qquad 0 \le \nu < \nu_1 , \tag{6.14}$$

and

$$\gamma_\nu^* = \bar{\gamma}_\nu(\rho_\nu^*), \qquad 0 \le \nu < \nu_1 , \tag{6.15}$$

where $G^\gamma(\rho)$ denote the unique probability state from ρ given γ. Next, consider the interval from the i-th to the (i+1)-th message receipt epoch, $1 \le i \le B-1$. Let ν_i (ν_{i+1}) denote the i-th ((i+1)-th) message receipt epoch. From the results of the second pass and using the information contained in the received states, a unique probability state ρ_{ν_i} at epoch ν_i is determined. Similarly to Eqs. (6.13)-(6.15), we may derive

$$\rho_{\nu_i}^* = \rho_{\nu_i} , \tag{6.16}$$

$$\rho_{\nu+1}^* = G^{\bar{\gamma}_\nu(\rho_\nu^*)}(\rho_\nu^*), \qquad \nu_i \le \nu < \nu_{i+1} , \tag{6.17}$$

and

$$\gamma_\nu^* = \bar{\gamma}_\nu(\rho_\nu^*), \qquad \nu_i \le \nu < \nu_{i+1}. \tag{6.18}$$

At epoch ν_{i+1}, when state broadcasts are received again and used, a unique probability state $\rho_{\nu_{i+1}}$ can be identified by using the results of the second pass as well as the information contained in the received states. This process repeats itself for all such intervals between epoch ν_i and ν_{i+1} until the final message receipt epoch ν_B. Within the interval from epoches

v_B to V, the last epoch of the planning cycle, one can derive:

$$\rho^*_{v_B} = \rho_{v_B} , \tag{6.19}$$

$$\rho^*_{v+1} = G^{\bar{\gamma}_v(\rho^*_v)}(\rho^*_v), \qquad v_B \le v < V , \tag{6.20}$$

and

$$\gamma^*_v = \bar{\gamma}_v(\rho^*_v), \qquad v_B \le v < V. \tag{6.21}$$

Obviously, $\gamma^* \triangleq [\gamma^*_0, \gamma^*_1, \cdots, \gamma^*_{V-1}]$ is a sub-optimal policy for the decentralized TMSP with periodic state broadcasts.

In summary, three passes are needed to solve the decentralized TMSP:

P1. Generate the DP network (forward recursively) as described in Sections 6.4.2-6.4.4.

P2. Solve the functional equations, Eqs. (6.11) and (6.12), backward recursively using the one-step cost derived in Eq. (6.10) for the DP network generated in P1.

P3. Identify the sub-optimal decentralized scheduling rules forward recursively from the solutions obtained in P2 using Eqs. (6.13)-(6.21).

Consider again the task system in Fig. 6.2 as an example. Within $I = [0, 10)$, suppose inter-task messages a_1, a_3, a_4, a_6, a_{12} and a_{13} each occur only once, while a_{10} and a_{11} each occur twice, resulting in a total of 10 messages. Recall that $\mu_1 = \mu_3 = \mu_4 = \mu_6 = 2$ and $\mu_{10} = \mu_{11} = \mu_{12} = \mu_{13} = 1$. The average sojourn time $\bar{S}$ of these inter-task messages within the communication subsystem is $\bar{S} = \frac{1}{10}\sum_i \frac{1}{\mu_i} = 4/5$. Since $\lambda_x = 10 / L = 1$, the service rate of the M/M/1 queue $\mu_x = \lambda_x + 1/\bar{S} = 9/4$. Let the arrival rate of state broadcasts $\lambda_y = 2$ and the service need $b_y = 0.2\, b_x$, where b_x is the service need for each inter-task message. Then, the portion of power used to serve inter-task messages is $\xi_x = \frac{b_x}{b_x + 0.4 b_x} = 5/7$ and the adjusted service rate $\mu_x' = \xi_x \mu_x = (5/7)(9/4) = 45/28$. From Eq. (6.6a),

$\bar{S}' = \dfrac{1}{\mu_x' - \lambda_x} = 28/17 > 4/5 = \bar{S}$, and $\bar{S}/\bar{S}' = 17/35$. From Eq. (6.7), the delay in broadcasting states becomes $W_y = \dfrac{\lambda_x}{\lambda_y} W_x = (1/2)(28/45) = 14/45$ since $W_x = 1/\mu_x' = 28/45$. Notice that the inter-broadcast interval ($= 0.5$) is larger than W_y satisfying the requirement that states are broadcast only after the previously broadcast states have been received. Also, to avoid saturating the communication subsystem, the maximum of λ_y with $b_y = 0.2\ b_x$ occurs only when $\bar{S}' = \infty$, i.e., $\mu_x' = \xi_x \mu_x = \lambda_x$ or $\xi_x = \xi_y \lambda_x \bar{S}$ (Eq. (6.6)). This occurs at $\lambda_y = 25/4$ when $\xi_x = 4/9$, where $W_x = W_y = \infty$.

To ease the computation difficulty of the DP algorithm mentioned above, the following approximations are made: a) the planning cycle is discretized into 200 intervals, b) W_y is discretized into only two stages, each of which contains several intervals, and c) the marginal probability of each state is discretized into 10 different intervals. Applying this approximation to the decentralized TMSP with $b_y = 0.2\ b_x$ and $\lambda_y = 0.4, 0.8, 1.0, 2.0, 3.0$ and 4, we obtained $J^* = 2.309, 1.577, 1.445, 1.446, 1.489$ and 1.543, respectively, showing that $\lambda_y = 1$ is the best among the five broadcast frequencies. To show the fact that the optimal frequency depends on b_y, the same algorithm is applied again to the cases with $b_y = 0.5\ b_x$ and $\lambda_y = 0.2, 0.4, 0.8, 1.0$ and 2.0. The best broadcast frequency again turned out to be $\lambda_y = 1.0$, but with corresponding $J^* = 1.544 > 1.445$. However, as shown in Fig. 6.5, the true optimal broadcast frequency of the former should be greater than that of the latter case. These results are not surprising since the communication subsystem now needs to allocate more power to deliver the same state information and, thus, degrades the normal inter-task communications.

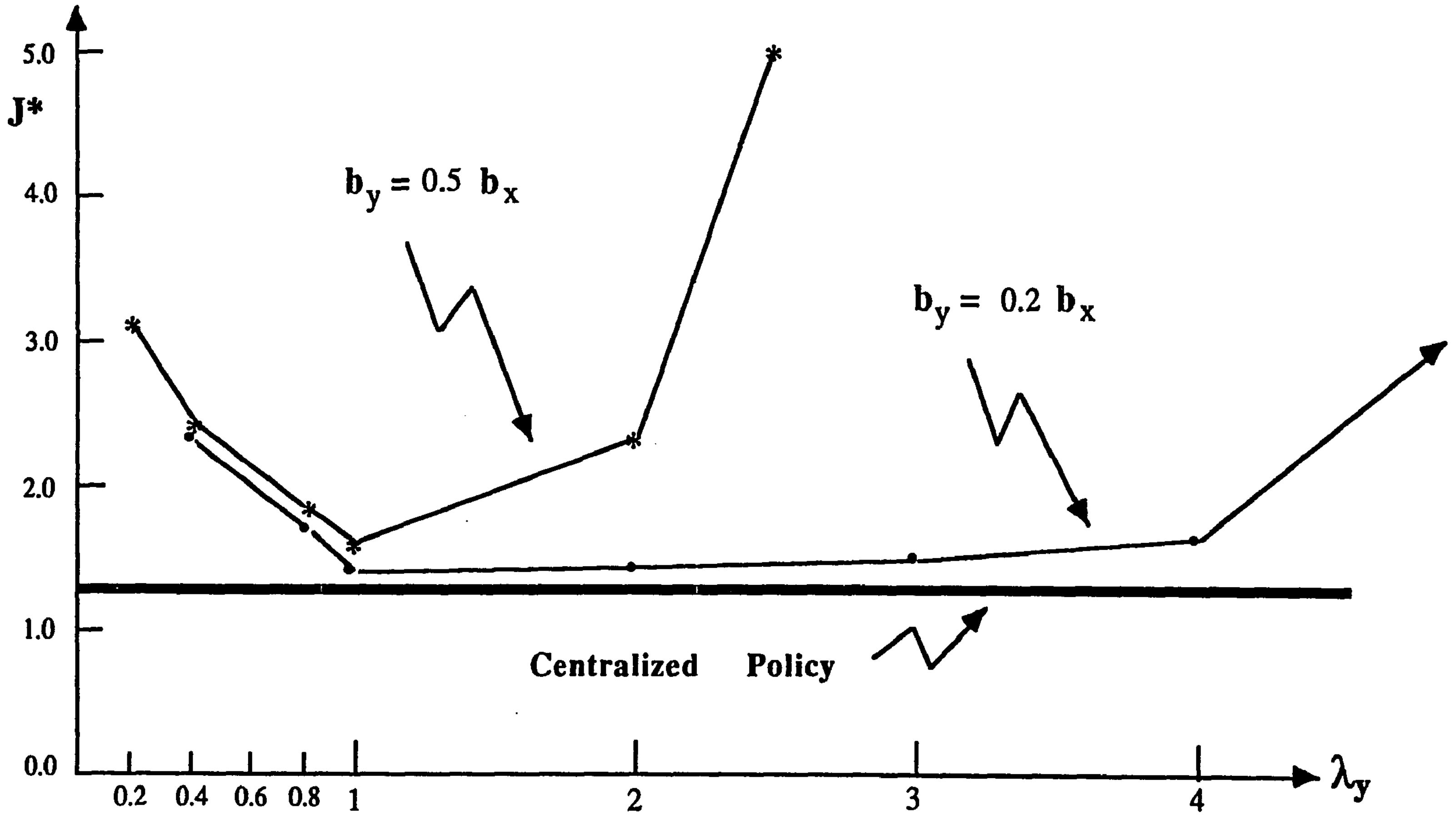


Figure 6.5. J^*_s for $b_y = 0.2\ b_x$ and $b_y = 0.5\ b_x$

CHAPTER 7
CONCLUSION

7.1. Summary

Real-time systems are characterized by the fact that the execution of computational tasks must not only be logically correct but also be completed in time to avoid severe consequences which might otherwise ensue. The major contribution of this dissertation is to solve the integrated problem of efficient execution of real-time tasks through modeling, assignment and scheduling of the tasks with respect to a new performance measure — the system hazard. The main results are summarized as follows:

- In Chapter 1, the new performance measure, system hazard, for scheduling real-time tasks of fixed execution times have been proposed and analyzed. The system hazard can be used not only for meeting deadlines but also for measuring how early each task can be completed.

  For a single processor with only independent periodic tasks, optimal static and dynamic scheduling algorithms are derived. In the static case, the RMS algorithm is shown to be optimal w.r.t. the system hazard. In the dynamic case however, the EDD scheduling algorithm is shown to be not optimal w.r.t the system hazard. More importantly, two best bounds of processor utilization are derived for these two cases. The optimal scheduling algorithms and the associated two utilization bounds can be used as a guide for task allocation where each task must be completed before its deadline.

Since optimal on-line algorithms w.r.t. the system hazard are shown to be non-existent for all but some special cases, simple mechanisms are derived to check whether or not an arriving aperiodic task can be completed with a system hazard not exceeding the prespecified value. These mechanisms are useful in a system where load-sharing strategies are adopted to improve system performance. An over-loaded PN may use the mechanisms to determine whether or not to transfer an arriving task to an under-loaded PN.

- For communicating (dependent) tasks in a distributed system, an optimal scheduling algorithm is derived in Chapter 3, where the scheduling problem is treated in a context called the *multi-project* scheduling problem (MPSP). Since the MPSP with precedence constraints is generally NP-hard, some form of enumeration is necessary to derive their solutions. In this chapter, we have presented a new approach to the problem using a B&B algorithm on the basis of (i) modeling the multi-project with an acyclic graph, (ii) identifying the dominance properties (DPs) w.r.t. all regular measures and the system hazard, (iii) developing a vertex expansion algorithm using the DPs into which the B&B algorithm is embedded, and (iv) deriving lower-bound costs for each non-terminal vertex so that the B&B algorithm may be guided more efficiently for an optimal schedule.

  Our computational experiences have indicated that this approach is very efficient for the MPSP. Because it depends on the B&B algorithm for a solution, extensions of this approach to similar problems with other types of resource constraints are also possible.

- Using the above results for an optimal scheduling, optimal static allocation of periodic tasks is derived in Chapter 4. Our task allocation method is different from others' because most of other allocation methods are to minimize the sum of total task execution and communication costs.

Task allocation problem is generally known to be NP-hard even without considering the precedence constraints among tasks. In this chapter, we have addressed the problem of allocating a set of periodic tasks with precedence constraints among the set of PNs of a distributed real-time system. First, the task system is modeled with a task graph (TG), which describes the computation and communication modules as well as the precedence constraints among them. Then, besides using the results of Chapter 3, lower-bound costs are computed such that a B&B algorithm can be established to find an optimal allocation.

Although more computational experience with the proposed algorithm needs to be gained, we believe that both the dominance properties and the lower-bound costs presented in this chapter can ease the computational difficulty significantly. This fact has been confirmed partially by our computational experiences obtained in Chapter 3 and the demonstrative example presented in Chapter 4.

- In Chapter 5, A Continuous-Time Markov Chain (CTMC) model is presented for the concurrent execution of periodic tasks assigned in a distributed real-time system. In the modeling process, activities are first identified by alternately applying combination and expansion phases on the original TFG's of the task system. Secondly, the activities and the precedence constraints among them are modeled by a system-wide GSPN. Finally, after considering time-driven task invocations, a sequence of CTMCs is built with the assumption of independently, exponentially distributed activity execution times.

  The proposed CTMC model has a finer granularity in describing the execution stages of tasks than most other models. Thus, the CTMC model has high potential use for resolving various design issues of distributed real-time systems, such as task execution time estimation, message handling, time-out, and task allocation. One of these issues is treated in Chapter 6.

- Scheduling tasks and messages is one of the most important issues in distributed real-time systems since time-critical tasks must be completed before their deadlines. In Chapter 6, we have presented both centralized and decentralized algorithms for the problem of optimally scheduling periodic tasks and their inter-task communication messages to minimize the average number of tasks missing deadlines.

  The CTMC model is first extended to incorporate the notion of default activities for message scheduling. Then, based on the extended CTMC model, the task/message scheduling problem is transfomed into a Semi-Markov Decision Process (SMDP) on which the DP technique can be applied for an optimal or sub-optimal policy for all PNs. For the centralized case, the optimal policy is computed by assuming the existence of a centralized monitor which dictates the scheduling decisions for each PN. For the decentralized case, however, we assume that all PNs periodically broadcast their local state information so that other PNs can make better scheduling decisions. Because of the difficulty of deriving an optimal solution, a sub-optimal solution is presented. The sub-optimal decentralized scheduling policy for each PN, and its associated optimal state broadcast frequency are also derived by using the DP technique.

7.2. Future Work

Many difficult problems need to be solved in the design and analysis of distributed real-time systems. In this dissertation, we have only looked into a few problems regarding the efficient execution of real-time tasks. Following are some of the closely related problems that warrant further research.

- Task allocation and scheduling with other resource constraints.

  In each of the five research problems, processors and communication bandwidth are assumed to be the only resource constraints. In practical systems other resources, such as memory and I/O processors, may also be needed by different tasks at the same time. However, considering more resource constraints will generally make task allocation and scheduling more difficult. Therefore, efficient algorithms with known performance bounds for the above problems as well as for the problems addressed in the thesis need to be derived.

- Load sharing for aperiodic tasks.

  A proper load sharing strategy for aperiodic tasks may improve performance significantly in a distributed real-time system. Any load sharing strategy consists of two components: (i) transfer policy, which decides whether and when to transfer an aperiodic task, and (ii) location policy, which determines where to transfer the aperiodic task. While the transfer policy has been partially addressed in Chapter 2, the research results to date on the location policy for real-time systems are far from being satisfactory. This is because the loading state of a PN is unavailable to other PNs without incurring extra communication costs. A different approach to the location policy could be to use the technique of information theory. That is, a location policy could be derived based on a different quantity such as entropy. Research issues on such a policy include static policies, which do not use information obtained through extra communication, and dynamic policies, which do.

- Task modeling, assignment and scheduling with fault-tolerance.

  Fault-tolerance is also essential in distributed real-time systems. Tasks for fault-tolerance are usually treated differently from those for pure system performance. Thus, a new set of schemes and optimal algorithms under those schemes for the modeling, assignment and scheduling of tasks may be needed such that both performance and reliability requirements of the real-time system can be met.

APPENDICES

APPENDIX A

AGGREGATION RULES FOR STRUCTURE POINTS

Denote the upper-end and lower-end points of a chain,[1] the fork and join points of an And-Subgraph, the fork and join points of an Or-Subgraph, and the collecting and branching points of a loop by C1, C2, A1, A2, R1, R2, L1 and L2, respectively.

There are a total of 39 cases to be considered, each of which is in the form of P1 → P2. The aggregation rule to reduce the number of redundant structure points for each case is formulated in the following format:

$$P1 \rightarrow P2 : AGGREGATION\ RULE,\quad CONTROL\ POINT\ ASSIGNMENT\ RULE$$

The AGGREGATION RULE indicates either of the following two actions:

(a) G => x means "aggregate P1 and P2, and replace them by the structure point x," where x is either P1 or P2.

(b) NG means "do not aggregate."

The CONTROL POINT ASSIGNMENT RULE also indicates either of the following two actions:

(i) S => y means "assign a control point to structure point y," where y is either P1 or P2.

(ii) NS means "no control point is assigned."

[1]The START and END points of the TFG are treated as upper-end and lower-end points, respectively.

The following are the rules for all 39 cases:

CL1: P2 is in the active layer, which is one layer inside the layer of P1

(1) C1 → A1: G => A1, NS

(2) A1 → C1: G => A1, NS

(3) C1 → R1: G => R1, NS

(4) R1 → C1: G => R1, NS

(5) C1 → L1: G => L1, NS

(6) L1 → C1: G => L1, NS

(7) A1 → A1: G => A1(outer layer), NS

(8) R1 → A1: NG, S=> A1

(9) A1 → R1: NG, S=> R1

(10) L1 → A1: NG, S=> A1

(11) A1 → L1: NG, S=> L1

(12) R1 → R1: G => R1(outer layer), NS (with proper adjustments of the branching probabilities of the inner layer Or-Subgraph.)

(13) L1 → R1: NG, S=> R1

(14) R1 → L1: NG, S=> L1

(15) L1 → L1: G => L1(outer layer), NS

CL2: P1 is in the active layer, which is one layer inside the layer of P2

(16) C2 → A2: G => A2, NS

(17) A2 → C2: G => A2, NS

(18) C2 → R2: G => R2, NS

(19) R2 → C2: G => R2, NS

(20) C2 → L2: G => L2, NS

(21) L2 → C2: G => L2, NS

(22) A2 → A2: G => A2(outer layer), NS

(23) A2 → R2: NG, S=> A2

(24) R2 → A2: NG, S=> R2

(25) A2 → L2: NG, S=> A2

(26) L2 → A2: NG, S=> L2

(27) R2 → R2: G => R2(outer layer), NS

(28) R2 → L2: NG, S=> R2

(29) L2 → R2: NG, S=> L2

(30) L2 → L2: G => L2(outer layer), NS (with proper adjustments of the branching probabilities of the outer layer loop.)

CL3: Both P1 and P2 are in the active layer

(31) A2 → A1: G => A2(or A1), S => A2(or A1)

(32) A2 → R1: NG, S => A2 and R1

(33) A2 → L1: NG, S => A2 and L1

(34) R2 → A1: NG, S => R2 and A1

(35) R2 → R1: NG, S => R2 and R1

(36) R2 $\rightarrow$ L1: NG, S => R2 and L1

(37) L2 $\rightarrow$ A1: NG, S => L2 and A1

(38) L2 $\rightarrow$ R1: NG, S => L2 and R1

(39) L2 $\rightarrow$ L1: NG, S => L2 and L1

APPENDIX B

SYSTEM STATES OF FIGURE 6.2 FOR $I_1 = [0, 5)$

A state s_i is represented as $i = \{j \mid \text{place } \phi_j \text{ has a token}\}$.

1	=	2 4 5 20 22 27
2	=	5 9 20 22 27
3	=	2 4 6 20 22 27
4	=	2 4 5 21 22 27
5	=	2 4 5 20 22 28
6	=	6 9 20 22 27
7	=	5 9 21 22 27
8	=	5 9 20 22 28
9	=	2 4 7 20 22 27
10	=	2 4 6 21 22 27
11	=	2 4 6 20 22 28
12	=	2 4 5 21 22 28
13	=	2 4 5 20 22 29
14	=	2 4 5 20 24 29
15	=	7 9 20 22 27
16	=	6 9 21 22 27
17	=	6 9 20 22 28
18	=	5 9 21 22 28
19	=	5 9 20 22 29
20	=	5 9 20 24 29
21	=	2 9 20 22 27
22	=	2 4 7 21 22 27
23	=	2 4 7 20 22 28
24	=	2 4 6 21 22 28
25	=	2 4 6 20 22 29
26	=	2 4 6 20 24 29
27	=	2 4 5 21 22 29
28	=	2 4 5 21 24 29
29	=	2 4 5 20 24 31
30	=	8 9 20 22 27
31	=	7 9 21 22 27
32	=	7 9 20 22 28
33	=	6 9 21 22 28
34	=	6 9 20 22 29
35	=	6 9 20 24 29
36	=	5 9 21 22 29
37	=	5 9 21 24 29
38	=	5 9 20 24 31
39	=	3 9 20 22 27
40	=	2 9 21 22 27
41	=	2 9 20 22 28
42	=	2 4 7 21 22 28
43	=	2 4 7 20 22 29
44	=	2 4 7 20 24 29
45	=	2 4 6 21 22 29
46	=	2 4 6 21 24 29
47	=	2 4 6 20 24 31
48	=	2 4 5 21 24 31
49	=	8 9 21 22 27
50	=	8 9 20 22 28
51	=	7 9 21 22 28
52	=	7 9 20 22 29
53	=	7 9 20 24 29
54	=	6 9 21 22 29
55	=	6 9 21 24 29
56	=	6 9 20 24 31
57	=	5 9 21 24 31
58	=	3 9 21 22 27
59	=	3 9 20 22 28
60	=	2 9 21 22 28
61	=	2 9 20 22 29
62	=	2 9 20 24 29
63	=	2 4 7 21 22 29
64	=	2 4 7 21 24 29
65	=	2 4 7 20 24 31
66	=	2 4 6 21 24 31
67	=	8 9 21 22 28
68	=	8 9 20 22 29
69	=	8 9 20 24 29
70	=	7 9 21 22 29
71	=	7 9 21 24 29
72	=	7 9 20 24 31
73	=	6 9 21 24 31
74	=	3 9 21 22 28
75	=	3 9 20 22 29
76	=	3 9 20 24 29
77	=	2 9 21 22 29
78	=	2 9 21 24 29
79	=	2 9 20 24 31
80	=	2 4 7 21 24 31
81	=	8 9 21 22 29
82	=	8 9 21 24 29
83	=	8 9 20 24 31
84	=	7 9 21 24 31
85	=	3 9 21 22 29
86	=	3 9 21 24 29
87	=	3 9 20 24 31
88	=	2 9 21 24 31
89	=	8 9 21 24 31
90	=	3 9 21 24 31

APPENDIX C

SYSTEM STATES OF FIGURE 6.2 FOR $I_2 = [5, 10)$

A state s_i is represented as $i = \{j \mid \text{place } \phi_j \text{ has a token }\}$.

1	=	11 13 14 20 22 32	37	=	11 13 14 20 24 32	73	=	12 18 20 24 32
2	=	11 13 14 21 22 32	38	=	11 13 14 21 24 32	74	=	11 18 20 24 33
3	=	14 18 20 22 32	39	=	14 18 20 24 32	75	=	11 13 16 20 25 34
4	=	11 13 15 20 22 32	40	=	11 13 15 20 24 32	76	=	11 13 15 20 25 35
5	=	11 13 14 20 22 33	41	=	11 13 14 20 24 33	77	=	17 18 21 24 32
6	=	14 18 21 22 32	42	=	14 18 21 24 32	78	=	16 18 21 24 33
7	=	11 13 15 21 22 32	43	=	11 13 15 21 24 32	79	=	15 18 26 34
8	=	11 13 14 21 22 33	44	=	11 13 14 21 24 33	80	=	14 18 26 35
9	=	15 18 20 22 32	45	=	15 18 20 24 32	81	=	12 18 21 24 32
10	=	14 18 20 22 33	46	=	14 18 20 24 33	82	=	11 18 21 24 33
11	=	11 13 16 20 22 32	47	=	11 13 16 20 24 32	83	=	11 13 16 26 34
12	=	11 13 15 20 22 33	48	=	11 13 15 20 24 33	84	=	11 13 15 26 35
13	=	15 18 21 22 32	49	=	11 13 14 20 25 34	85	=	17 18 20 24 33
14	=	14 18 21 22 33	50	=	15 18 21 24 32	86	=	16 18 20 25 34
15	=	11 13 16 21 22 32	51	=	14 18 21 24 33	87	=	15 18 20 25 35
16	=	11 13 15 21 22 33	52	=	11 13 16 21 24 32	88	=	12 18 20 24 33
17	=	16 18 20 22 32	53	=	11 13 15 21 24 33	89	=	11 18 20 25 34
18	=	15 18 20 22 33	54	=	11 13 14 26 34	90	=	11 13 16 20 25 35
19	=	11 18 20 22 32	55	=	16 18 20 24 32	91	=	17 18 21 24 33
20	=	11 13 16 20 22 33	56	=	15 18 20 24 33	92	=	16 18 26 34
21	=	16 18 21 22 32	57	=	14 18 20 25 34	93	=	15 18 26 35
22	=	15 18 21 22 33	58	=	11 18 20 24 32	94	=	12 18 21 24 33
23	=	11 18 21 22 32	59	=	11 13 16 20 24 33	95	=	11 18 26 34
24	=	11 13 16 21 22 33	60	=	11 13 15 20 25 34	96	=	11 13 16 26 35
25	=	17 18 20 22 32	61	=	11 13 14 20 25 35	97	=	17 18 20 25 34
26	=	16 18 20 22 33	62	=	16 18 21 24 32	98	=	16 18 20 25 35
27	=	12 18 20 22 32	63	=	15 18 21 24 33	99	=	12 18 20 25 34
28	=	11 18 20 22 33	64	=	14 18 26 34	100	=	11 18 20 25 35
29	=	17 18 21 22 32	65	=	11 18 21 24 32	101	=	17 18 26 34
30	=	16 18 21 22 33	66	=	11 13 16 21 24 33	102	=	16 18 26 35
31	=	12 18 21 22 32	67	=	11 13 15 26 34	103	=	12 18 26 34
32	=	11 18 21 22 33	68	=	11 13 14 26 35	104	=	11 18 26 35
33	=	17 18 20 22 33	69	=	17 18 20 24 32	105	=	17 18 20 25 35
34	=	12 18 20 22 33	70	=	16 18 20 24 33	106	=	12 18 20 25 35
35	=	17 18 21 22 33	71	=	15 18 20 25 34	107	=	17 18 26 35
36	=	12 18 21 22 33	72	=	14 18 20 25 35	108	=	12 18 26 35

REFERENCES

REFERENCES

[ADM87] M. Aicardi, F. Davoli and R. Minciardi, ''Decentralized Optimal Control of Markov Chains with a Common Past Information Set,'' *IEEE Trans. on Automat. Contr.*, Vol. AC-32, No. 11, November 1987, pp. 1028-1031.

[Bak74] K. R. Baker, *Introduction to Sequencing and Scheduling*, Wiley & Sons, 1974.

[BEN82] R. Bellman, A. O. Esogbue and I. Nabeshima, *Mathematical Aspects of Scheduling and Applications*, Pergamon Press, 1982.

[BLL83] K. R. Baker, *et al.*, ''Preemptive Scheduling of A Single Machine to Minimize Maximum Cost Subject to Release Dates and Precedence Constraints,'' *Operations Research*, Vol. 31, No. 2, Mar-Apr. 1983, pp. 381-386.

[BLR83] J. Blazewicz, J. K. Lenstra and A. H. G. Rinnooy Kan, ''Scheduling Subject to Resource Constraints: Classification and Complexity,'' *Discrete Applied Math.*, 5, 1983, pp. 11-24.

[BrW65] G. H. Brooks and C. R. White, ''An Algorithm for Finding Optimal or Near Optimal Solutions to the '' Production Scheduling Problems," *J. Indust. Eng.*, Vol. 16, 1965, pp. 34-40.

[CHL80] W. W. Chu, *et al.*, ''Task Allocation in Distributed Data Processing,'' *IEEE Computer*, Vol. 13, Nov. 1980, pp. 57-69.

[ChL84] W. W. Chu and K. K. Leung, ''Task Response Time Model and Its Applications for Real-Time Distributed Processing Systems,'' *Proc. IEEE, 5th Real-Time system Symposium*, Dec. 1984, pp. 225-236.

[ChL87] W. W. Chu and L. M. Lan, ''Task Allocation and Precedence Relations for Distributed Real-Time Systems,'' *IEEE Trans. on Computers*, Vol. C-36, No. 6, Jun. 1987, pp. 667-679.

[CMM67] R. W. Conway, W. L. Maxwell and L. W. Miller, *Theory of Scheduling*, Addison-Wesley, 1967, pp. 141-189.

[Cof76] E. G. Coffman, *Computer and Job-Shop Scheduling Theory*, New York, Wiley and Sons, 1976.

[CSR87] S. C. Cheng, J. A. Stankovic and K. Ramamrithan, "Scheduling Algorithms for Hard Real-Time Systems -- A Brief Survey," *IEEE Real-Time Systems Newsletter*, Vol. 3, No. 2, Summer 1987, pp. 1-24.

[DeG70] M. H. DeGroot, *Optimal Statistical Decisions*, McGraw-Hill, 1970, pp. 11-12.

[Dem81] M. A. H. Dempster, "A Stochastic Approach to Hierachical Planning and Scheduling," in *Deterministic and Stochastic Scheduling*, Dempster, *et al.* (eds), Reidel, Dordrecht, The Netherlands, 1981, pp. 271-296.

[Den82] E. Denardo, *Dynamic Programming: Models and Applications*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982.

[Der74] M. Dertouzos, "Control Robotics: The Procedural Control of Physical Processes," *Proc. of the IFIP Congress*, 1974, pp. 807-813.

[DhL78] S. K. Dhall and C. L. Liu, "On a Real-Time Scheduling Problem," *Opns Res*, Vol. 26, No. 1, 1978, pp. 127-140.

[Dij84] N. M. van Dijk, *Controlled Markov Processes: Time-Discretization*, CWI, 1984.

[Fre82] S. French, *Sequencing and Scheduling*, Halsted Press, 1982.

[GaJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability - A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, N.Y., 1979.

[GDB89] K. D. Gordon *et. al*, "Scheduling Aperiodic Tasks with Hard Deadlines in a Rate Monotonic Framework," *Proc. Sixth IEEE Workshop on Real-Time Operating Systems and Software*, 1989, pp. 1-5.

[GiT60] B. Giffler and G. L. Thompson, "Algorithms for Solving Production Scheduling Problems," *Opns Res.* 8(4), 1960, pp. 487-503.

[GoS78] T. Gonzalez and S. Sahni, "Flowshop and Jobshop Schedules: Complexity and Approximation," *Opns Res.* 26(1), 1978, pp. 36-52.

[GrS88] A. Griefer and R. Strong, "DCF: Distributed Communication With Fault Tolerance," *Proc. 7th Annual ACM Symp. on Principles of Distributed Computing*, Aug. 1988, pp. 18-27.

[HoC72] Y. C. Ho and K. C. Chu, "Team Decision Theory and Information Structures in Optimal Control Problems - Part 1," *IEEE Trans. on Automatic Control*, Vol.

AC-17, No. 1, Feb. 1972, pp. 15-22.

[Hor74] W. A. Horn, "Some Simple Scheduling Algorithms," *Naval Research Logistics Quarterly*, Vol. 21, 1974, pp. 177-185.

[HsM82] K. Hsu and S Marcus, "Decentralized Control of Finite State Markov Processes," *IEEE Trans. on Automatic Control*, Vol. AC-27, No. 2, Apr. 1982, pp. 426-431.

[Hua85] J. P. Huang, "Modeling of Software Partition for Distributed Real-Time Applications," *IEEE Trans. on Software Engineering*, Vol. SE-11, No. 10, Oct. 1985, pp. 1113-1126.

[KaN84] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing," *IEEE Trans. on Computers*, Vol. C-33, No. 11, Nov. 1984, pp. 1023-1029.

[Kle75] L. Kleinrock, *Queueing Systems, Volume I : Theory*, Wiley, 1975.

[KoS76] W. H. Kohler and K. Steiglitz, "Enumerative and Iteractive Computational Approach" in *Computer and Job-Shop Scheduling Theory*, Coffman *eds.*, Wiley and Sons, 1976, pp. 229-287.

[KuV86] P. R. Kumar and P. Varaiya, *Stochastic Systems: Estimation, Identification and Adaptive Control*, Prentice Hall, 1986.

[Lam84] L. Lamport, "Using time instead of timeout for fault-tolerant distributed systems," *ACM Trans. on Programming Languages and Systems*, vol. 6, no. 2, Apr. 1984, pp. 254-280.

[LaM85] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *Journal of the ACM*, vol. 32, no. 1, Jan. 1985, pp. 52-78.

[LeK78] J. K. Lenstra and A. H. G. R. Kan, "Complexity of Scheduling under Precedence Constraints," *Operations Research*, Vol. 26, No. 1, Jan-Feb. 1978, pp. 23-35.

[Leu85] K. K. Leung, "Tasks Response Time and Module Assignment for Real-Time Distributed Processing Systems," Ph.D dissertation, Computer Science Dept., UCLA, 1985.

[LiL73] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. of ACM*, Vol. 20, No. 1, 1973, pp. 46-61.

[LLR77] B. J. Lageweg, J. K. Lenstra and A. H. G. Rinnooy Kan, "Job-Shop Scheduling by Implicit Enumeration," *Management Science*, Vol. 24, No. 4, 1977, pp. 441-450.

[LLR81] E. L. Lawler, *et al.*, "Recent Developments in Deterministic Sequencing and Scheduling: A Survey," in *Deterministic and Stochastic Scheduling*, Dempster, *et al.* (eds), Reidel, Dordrecht, The Netherlands, 1981, pp. 35-74.

[LRB77] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker, "Complexity of Machine Scheduling Problems," *Ann. Discrete Math.*, 1, 1977, pp. 343-362.

[LSS87] J. P. Lehoczky, L. Sha and J. K. Strosnider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *IEEE Real-Time Systems Symposium*, Dec. 1987, pp. 261-270.

[MaR71] J. Marschak and R. Radner, *The Economic Theory of Teams*, New Haven, CT. Yale Univ. Press, 1971.

[Mar82] C. Martel, "Preemptive Scheduling with Release Times, Deadlines, and Due Times," *J. of ACM*, Vol. 29, No. 3, 1982, pp. 813-829.

[MBC84] M. Ajmone Marsan, G. Balbo and G. Conte, "A Class of Generalized Stochastic Petri Nets for the Performance Analysis of Multiprocessor Systems," *ACM Trans. on Computing Systems*, Vol. 2, No. 2, May. 1984, pp. 93-122.

[Mil68] B. L. Miller, "Finite State Continuous Time Markov Decision Processes with a Finite Planning Horizon," *SIAM J. Control*, vol. 6, 1968, pp. 266-280.

[MLT82] P. Y. R. Ma, *et al.* "A Task Allocation Model for Distributed Computing Systems," *IEEE Trans. on Computers*, Vol. C-31, No. 1, Jan. 1982, pp. 41-47.

[MoD78] A. K.-L. Mok and M. L. Dertouzos, "Multiprocessor Scheduling in a Hard Real-Time Environment," *Proc. 7th Texas Conf. on Compu. Sys.*, Nov. 1978.

[Mok83] A. K.-L. Mok, *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment*, Ph. D. Dissertation, M.I.T., Cambridge, 1983.

[Mol81] M. K. Molloy, "On the Integration of Delay and Throughput Measures in Distributed Processing Models," Ph.D thesis, UCLA, 1981, pp. 19-21 and 53-54.

[MoM84] A. Movaghar and J. F. Meyer, "Performability Modeling with Stochastic Activity Networks," Technical Rep., Communications and Network Lab., Industrial Technology Institute, Ann Arbor, Mich., 1984.

[PeS89] D.-T Peng and K. G. Shin, "Static Allocation of Periodic Tasks with Precedence Constraints in" Distributed Real-Time Systems," *Proc. IEEE, 9th Int. Conf. Distrib. Compu. Syst.*, Newport Beach, Calif., 1989, pp. 190-198.

[Pet81] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, 1981.

[PiS81] M. Pinedo and L. Schrage, "Stochastic Shop Scheduling: A Survey," in *Deterministic and Stochastic Scheduling*, Dempster, *et al.* (eds), Reidel, Dordrecht, The Netherlands, 1981, pp. 181-196.

[Ros70] S. M. Ross, *Applied Probability Models with Optimization Applications*, Holden-Day, 1970.

[Ros83] S. M. Ross, *Stochastic Processes*, Wiley, 1983.

[Sch70] L. Schrage, "Solving Resource-Constrained Network Problems by Implicit Enumeration - Nonpreemptive Case," *Opns Res*, **18**, 1970, pp. 263-278.

[Sch72] L. Schrage, "Solving Resource-Constrained Network Problems by Implicit Enumeration - Preemptive Case," *Opns Res*, **20**, 1972, pp. 668-677.

[Ser72] O. Serlin, "Scheduling of Time Critical Processes," *Proc. of AFIPS 1972 Spring Joint Computer Conf.*, AFIPS Press, Montvale, N. J., 1972, pp. 925-932.

[ShC88] K. G. Shin and Y.-C. Chang, "Load Sharing in Distributed Real-Time Systems with State-Change Broadcasts," *IEEE Trans. on Computers*, Vol. C-38, No. 8, Aug. 1989, pp. 1124-1142.

[ShE87] K. G. Shin and M. E. Epstein, "Intertask Communications in an Integrated Multi-Robot System," *Proc. of the 1985 IEEE Conf. on Robotics and Automation*, pp. 910-917. An improved version also appeared in *IEEE J. of Robotics and Automation*, Vol. RA-3, No. 2, Apr. 1987, pp. 90-100.

[ShR87] K. G. Shin and P. Ramanathan, "Clock synchronization of a large multiprocessor system in the presence of malicious faults," *IEEE Trans. on Comput.*, Vol. C-36, No. 1, Jan. 1987, pp. 2-12.

[ShT85] C. C. Shen and W. H. Tsai, "A Graph Matching Approach to Optimal Task Assignment in Distributed Computing Systems Using a Minimax Criterion," *IEEE Trans. on Computers*, Vol. C-34, No. 3, Mar. 1985, pp. 197-203.

[SKL85] K. G. Shin, C. M. Krishna and Y. H. Lee, "A Unified Method for Evaluating Real-Time Computer Controllers and Its Applications," *IEEE Trans. on Autumatic Control*, Vol. AC-30, No. 4, Apr. 1985, pp. 357-366.

[SLR86] L. Sha, J. P. Lehoczky and R. Rajkumar, "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *IEEE Real-Time Systems Symposium*, Dec. 1986, pp. 181-191.

[SSL89] B. Sprunt, L. Sha and J. P. Lehoczky, "Aperiodic Task Scheduling for Real-Time Systems," *The Journal of Real-Time Systems*, Kluwer Academic Publishers, 1989, pp. 27-60.

[Sta84] J. A. Stankovic, "A Perspective on Distributed Computer Systems," *IEEE Trans. on Computers*, Vol. C-33, No. 12, Dec. 1984, pp. 1102-1115.

[StB78] H. S. Stone and S. H. Bokhari, "Control of Distributed Processes," *IEEE Computer*, Vol. 11, Jul. 1978, pp. 97-106.

[Sto77] H. S. Stone, "Multiprocessor Scheduling with the Aid of Network Flow Algorithm," *IEEE Trans. on Software Engineering*, Vol. SE-3, No. 1, Jan. 1977, pp. 85-93.

[Tah76] H. A. Taha, *Operations Research: An Introduction*, Macmillan Publishing Co. NY, 1976, pp. 357-366.

[ThB83] A. Thomasian and P. F. Bay, "Analytic Queueing Network Models for Parallel Processing of Task Systems," *IEEE Trans. on Computers*, Vol. C-35, No. 12, Dec. 1986, pp. 1045-1054.

[TrM75] J. P. Tremblay and R. Manohar, *Discrete Mathematical Structures with Applications to Computer Science*. McGraw-Hill. 1975, pp. 149-162.

[VaW78] P. Varaiya and J. Walrand, "On Delayed Sharing Patterns," *IEEE Trans. on Automat. Contr.*,, Vol. AC-23, No. 3, June 1978, pp. 443-445.

[Vir84] M. L. Virginia, "Heuristic Algorithms for Task Assignment in Distributed Systems," *Proc. IEEE, 4th Int'l Conf. on Distributed Computing Systems*, May 1984, pp. 30-39.

[Vir85] M. L. Virginia, "Task Assignment to Minimize Completion Time," *Proc. IEEE, 5th Int'l Conf. on Distributed Computing Systems*, May 1985, pp. 329-336.

[WaM85] Y. T. Wang and R. J. T. Morris, "Load Sharing in Distributed Systems," *IEEE Trans. on Computers*, Vol. C-34, No. 3, Mar. 1985, pp. 204-217.

[War78] S. A. Ward, "An Approach to Real-Time Computation," *Proc. IEEE, 7th Texas Conf. on Computing Systems*, 1978, pp. 5.26-5.34.

[Web82] R. R. Weber, "Discussants' Report on Scheduling Stochastic Jobs on Parallel Machines," in *Applied Probability in Computer Science: The Interface*, Disney and Ott (eds.), Birkhausen, Boston, 1982, pp. 339-344.

[Wit71] H. S. Witsenhausen, "Separation of Estimation and Control for Discrete Time Systems," *Proc. of the IEEE*, Vol. 59, No. 11, 1971, pp. 1557-1566.

[Wit73] H. S. Witsenhausen, "A Standard Form for Sequential Stochastic Control," *Math. Sys. Th.*, vol. 7, No. 1, 1973, pp. 5-11.

[Woo86] M. H. Woodbury, "Analysis of the Execution Time of Real-Time Tasks," *Proc. IEEE 7th Real-Time System Symposium*, Dec. 1986, pp. 89-96.

[WoS74] R. E. D. Woolsey and H. S. Swanson, *Operations Research for Immediate Applications: A Quick and Dirty Mannual*, Harper and Row, 1974.

[YoK78] T. Yoshikawa and H. Kobayashi, "Separation of Estimation and Control for Decentralized Stochastic Control," *Proc. 1978 Inter. Fed. Automa. Contr. Congr.*, 1979, pp. 1857-1864.

[Yos75] T. Yoshikawa, "Dynamic Programming Approach to Decentralized Stochastic Control Problems," *IEEE Trans. on Automat. Contr.*, Vol. AC-20, December 1975, pp. 796-797.