Issues Related to Dynamic Scheduling in Real-Time Systems

Fuxing Wang
Computer Science Technical Report 94-21
Department of Computer Science
Lederle Graduate Research Center
University of Massachusetts
Amherst Ma. 01003-4610

ISSUES RELATED TO DYNAMIC SCHEDULING IN REAL-TIME SYSTEMS

A Dissertation Presented

by

FUXING WANG

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1993

Department of Computer Science

ISSUES RELATED TO DYNAMIC SCHEDULING IN REAL-TIME SYSTEMS

A Dissertation Presented

by

FUXING WANG

Approved as to style and content by:

Krithi Ramamritham, Co-Chair

John A. Stankovic, Co-Chair

James F. Kurose, Member

C. Mani Krishna, Member

W. Richards Adrion, Department Head
Department of Computer Science

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to Professor Krithi Ramamritham and Professor Jack Stankovic for their encouragement and guidance during the course of this work. I would like to thank Professor Jim Kurose and Professor Mani Krishna for their time and advice.

I would like to thank Professor Wei Zhao and Mr. Decao Mao for their substantial help in developing many structures and concepts of this work.

My special thanks to all members of Spring group, specially, Chia Shen, Goran Zlokapa, and Douglas Niehaus.

I am indebted to my parents, Shiyun and Jiaxiang, my wife, Zhimin and my relatives, Zhiying, Zhendong, and Zhicheng, for their caring and support.

ABSTRACT

ISSUES RELATED TO DYNAMIC SCHEDULING IN REAL-TIME SYSTEMS

SEPTEMBER 1993

FUXING WANG

B.S., EAST CHINA ENGINEERING INSTITUTE

M.S., BEIJING UNIVERSITY OF AERONAUTICS AND ASTRONAUTICS

PH.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Krithi Ramamritham and Professor John A. Stankovic

Dynamic scheduling in real-time systems involves dynamically making a sequence of decisions concerning the assignment of system resources to real-time tasks. Tasks may have arbitrary time constraints, different importance levels, and fault tolerance requirements. Unfortunately, making these scheduling decisions is difficult, partly because the decisions must be made without the full knowledge of the future arrivals of tasks and partly because scheduling has to deal with many complex issues, e.g., multiprocessors and fault tolerance. Many existing algorithms, such as the Earliest Deadline First algorithm, cannot provide performance predictability for this dynamic scheduling problem.

This dissertation presents a set of solutions, in the form of both theory and algorithms, with the goal of providing performance predictions and guarantees for three different scheduling problems.

First, we provide a worst case analysis for algorithms that dynamically schedule independent tasks, where tasks are assumed to have different values. We derive performance bounds for both uniprocessor and dual-processor on-line scheduling. A

set of threshold-based scheduling algorithms is found. These are shown to be better than the popular Earliest Deadline First algorithm.

Secondly, for scheduling tasks with additional resources, we study dynamic scheduling algorithms based on the ability to generate feasible schedules and the quality of the generated feasible schedules, expressed in terms of the schedule length. Based on the analysis of two known algorithms, namely, list scheduling and the Spring heuristic scheduling algorithm, a set of new algorithms is developed and shown to possess better performance.

Thirdly, for scheduling tasks with fault-tolerance requirements, assuming that reliability of task execution is achieved through task replication, we present an approach to mathematically determine the replication factor for tasks. The goal is to maximize the total performance index, which is a performance-related reliability measurement. We present a technique based on a continuous task model and show how it very closely approximates discrete models and tasks with varying characteristics.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Scheduling in Real-Time Systems

Real-time systems are defined as those systems in which the correctness and performance of the system depend not only on the logical results of the computation, but also on the time at which the results are produced. Examples of real-time systems are command and control systems, flight control systems, space shuttle avionics systems, and robotics systems. These real-time systems are large, complex, and adaptive. Designing and implementing *dependable* and *predictable* real-time systems are important research issues, because failure of these systems may result in high risks and enormous costs.

In a complex real-time system, there exist many time-constrained activities which control the behavior of the system. These activities are abstracted as *real-time tasks.* The main timing constraint for a real-time task is a deadline. Generally, tasks may share system-level resources, e.g., CPUs of a multiprocessor, and user-level resources, such as shared data structures; tasks may have precedence constraints; and tasks may have fault-tolerance requirements. It is a difficult problem to construct a *feasible schedule* for tasks with these complex constraints, such that each task will complete before its deadline, receive all requested resources, avoid potential conflicts in accessing the resources, and satisfy all precedence constraints and fault-tolerance requirements. Research into *real-time task scheduling* involves the study of various algorithmic approaches to resolve these problems.

Most theoretic results on real-time task scheduling are based on the early analysis work for periodic and aperiodic task scheduling. For periodic task scheduling theory, the paper by Liu and Layland [53] analyzes uniprocessor scheduling for the *rate monotonic scheduling algorithm*, an optimal fixed priority scheduling algorithm for periodic tasks, and for the *earliest deadline first scheduling algorithm*, an optimal dynamic priority scheduling algorithm for periodic tasks. For aperiodic task scheduling, the earliest deadline first (EDF) scheduling algorithm and the least laxity first (LLF) scheduling algorithm are two main algorithms found in the early analysis work. Both algorithms have been shown to be optimal for work conserving uniprocessor scheduling with independent tasks, in the sense that, for any task arrival pattern, if a feasible schedule exists, both EDF and LLF are guaranteed to find it [29, 76].

For periodic task scheduling, recent extensions to the basic algorithms include aperiodic task servers [78], task synchronization [65], imprecise computation [55], and multiprocessor scheduling [30]. For aperiodic task scheduling, recent extensions include the best-effort scheduling algorithm with a value function for each independent task [58], the "planning" based heuristic scheduling approach to handle real-time tasks with additional resource requirements on multiprocessors [70, 91], and the imprecise computation technique [52].

Most theoretical real-time scheduling results are based on a static system model. Very few results are based on the dynamic model. Real-time systems built using static scheduling approaches have difficulty dealing with the dynamics and uncertainties of the environment and adapting to changes in application requirements, hardware and software structures. Therefore, dynamic scheduling is an important component in designing adaptive and flexible real-time systems. This dissertation focuses on dynamic scheduling. The exact meaning of dynamic scheduling and the scope of this dissertation will be presented in the following section.

1.2 Dynamic Scheduling — Contributions

Algorithms for dynamic scheduling in real-time environments involve *dynamically* making a sequence of decisions concerning the assignment of system resources to real-time tasks. We assume that the system resources include processors and other additional resources, such as shared data structures, and the tasks are aperiodic and have timing constraints, values to represent importance, and other requirements, such as fault-tolerance. There are two types of scheduling algorithms, *best effort* and *guarantee-oriented* [73]. As new tasks arrive, a best effort approach simply puts the new tasks into a task ready queue and executes a task with the highest priority or with the best value among all ready tasks. On the other hand, a guarantee-oriented approach provides a schedulability analysis whenever new tasks arrive, and generates new feasible schedules accordingly. If a scheduling algorithm fails to find a feasible schedule, the system may take a number of actions to reduce the workload, such as, to reject some tasks. Compared to the best effort approach, a guarantee-oriented scheduling algorithm has higher overhead, but it provides a degree of predictability.

In this dissertation, we will first conduct a worst case analysis for dynamic best effort scheduling algorithms to determine what is the best level of performance guarantee which can be provided by the algorithms. By using worst case analysis, we study the ratio between the performance of an on-line algorithm and the performance of a clairvoyant algorithm, and determine the performance bound of the on-line algorithm. For example, the performance bounds for EDF and LLF are zero. As we will show in Chapter 3, some (best effort) algorithms can provide a higher performance bound than the performance bounds for EDF or LLF. We are also interested in finding the highest performance bound among all possible scheduling algorithms.

Turning to the guarantee-oriented approach, determining the feasibility for a set of tasks is a difficult problem, as the majority of the real-time scheduling problems have

been proven to be NP-hard or NP-complete in the strong sense [34, 37]. There are two ways to deal with the problem. One uses an exhaustive enumeration algorithm, such as branch and bound or dynamic programming. The other uses approximation algorithms. Due to their large execution overheads, exhaustive enumeration algorithms are not applicable in dynamic situations. It is an important research issue to find good approximation algorithms. In Chapter 4, we analyze the worst case performance characteristics for a set of heuristic algorithms that schedule tasks that have both time constraints and resource requirements.

In the guarantee-oriented dynamic scheduling approach, even with a good heuristic scheduling algorithm, we may not be able to guarantee all tasks, because it is possible that there exists no feasible schedule for the task set. Thus, another issue is to decide that which tasks will be favored and which tasks will be rejected when the system is overloaded. In particularly, if tasks have fault-tolerance requirements, we would like to increase tasks' redundancy to achieve higher reliability, which will easily cause system overload. Therefore, in Chapter 5, we present a technique for performance-related reliability analysis for determining tasks' redundancy levels. The result of this analysis provides us a way to balance both reliability and performance and to construct a proper workload.

To summarize, this dissertation deals with three issues related to dynamic scheduling:

- the worst case analysis of on-line scheduling,
- the bound analysis of heuristic algorithms for tasks with resource requirements, and
- the performance-related reliability analysis for determining tasks' redundancy levels.

Our results provide designers of dynamic real-time systems guidelines in designing scheduling algorithms that suit certain design and performance requirements and in evaluating their performance. In the remainder of this section, we discuss these three aspects in detail.

1.2.1 Worst Case Analysis of On-Line Scheduling

Consider the problem of on-line scheduling which makes a sequence of decisions dynamically by assigning system resources to real-time tasks. If a task request is successfully scheduled to completion, the value associated with the task is obtained; otherwise a value of zero is obtained. A real-time system is said to have a *loading factor* b if and only if it is guaranteed that there will be no interval of time $[t_x, t_y)$ such that the sum of the computation times of all tasks having deadlines within this interval is greater than $b \cdot (t_y - t_x)$. A real-time system is overloaded if the loading factor b is greater than 1. On-line scheduling is a difficult problem, where the main difficulty is caused by overload. Overloads cannot be avoided easily, especially in dynamic and uncertain environments such as in robotics, which requires its control subsystem to adapt to a dynamic environment. It would be too costly to construct a system such that overload will never occur and/or inefficient or even impossible to construct a schedule *a priori* in such a system.

During overloads, many known algorithms e.g., EDF and LLF, do not perform well. To study the performance of scheduling algorithms, we may compare the performance of an on-line algorithm with the performance of a clairvoyant algorithm. The clairvoyant algorithm is an ideal optimal off-line algorithm with full knowledge of the future arrivals of tasks. The *lower bound* on the performance of an on-line scheduling algorithm, A, can be defined in the following way: If over all task arrival sequences, the smallest value of the ratio of the values obtained by A and the values obtained by the clairvoyant algorithm is B_A, then B_A is the *lower bound* on the performance of

A. For example, the lower bound for EDF or LLF is zero. If, for a given scheduling problem, the largest value of B_A for all A is B, then B is the upper (performance) bound of the scheduling problem. In other words, the upper bound of the on-line scheduling problem is the highest lower bound among all on-line algorithms.

In this dissertation, we consider both uniprocessor on-line scheduling and dual-processor on-line scheduling. Tasks can have the same value density or different value densities, where value density is defined as the ratio between the value of a task and its computation time. We now summarize our main results in this area.

In the case of uniprocessor on-line scheduling, if tasks have the same value density, we show that the performance upper bound is 1/4. Another interpretation of the result is that, in the worst case, an on-line algorithm is only able to complete the amount of work which is 1/4 of the work completed by a clairvoyant algorithm. We also show that 1/4 is a tight bound by constructing an on-line algorithm, called Threshold-1 (TD_1) algorithm, which achieves the bound. TD_1 guarantees no less than 1/4 of the value obtained by a clairvoyant algorithm for any request sequence. This means that TD_1 has the best *lower bound* among all on-line algorithms. Therefore TD_1 is an *optimal on-line* scheduling algorithm under overload, and furthermore, it has the same performance as LLF and EDF when there is no overload.

The above result can be further extended to the cases in which tasks have different value densities. Let γ be the ratio of the highest and lowest value densities of tasks. We prove that the upper bound for the on-line scheduling problem is

$$\frac{1}{\gamma + 1 + 2\sqrt{\gamma}}.$$

As a special case, if γ is 1, the upper bound is 1/4, which is the result mentioned above. If γ is 2, the upper bound is 1/5.828.

For the dual-processor case, we prove an upper bound of 1/2. This bound is shown to be tight in the special case when all the tasks have the same value density and zero laxity.

1.2.2 Bound Analysis of Heuristic Algorithms for Tasks with Resource Requirements

Here, we consider dynamic scheduling for tasks with additional resources in a multiprocessor system. Whenever new tasks arrive at the system, the scheduler combines the new tasks with all existing tasks which have not been serviced yet and generates a new feasible (non-preemptive) schedule. This scheduling problem can be shown to be NP hard. So good heuristic scheduling algorithms with low overheads must be adopted in practice. We focus on two important performance measures for evaluating a heuristic scheduling algorithm. The first is the *ability* of an algorithm to generate a feasible schedule. This ability is commonly characterized by the mean behavior of the algorithm, which can be measured either by a probability model or by a simulation study. For simple systems, the probability model works well. For complex systems, simulation is commonly used as is the case here. The metric used is *mean success ratio (SR)*, defined in the following way. Given N task sets which are generated according to certain distributions where each task set is known to have at least one feasible schedule, let N_A be the total number of task sets for which algorithm A finds feasible schedules. Then

$$SR = \frac{N_A}{N}.$$

The second measure is the *quality* of the feasible schedules generated by an algorithm. Quality may refer either to the mean behavior or the worst case behavior of the algorithm, where behavior is characterized by metrics such as *schedule length* and *earliness* (the time interval between a task's completion time and its deadline). In this dissertation, we deal with the worst case behavior and use the schedule length metric. Schedule length is an important characteristic of a schedule for tasks in a real-time system for the following reasons:

- For real-time task scheduling, if one algorithm generates a shorter schedule than another algorithm, it means that, in general, the first algorithm can more easily accommodate tasks with shorter deadlines than the second algorithm.

- If the schedule length is shorter, it is a more efficient schedule and it is likely that there is more execution time remaining at the end of the schedule which can be used for executing other tasks, such as diagnostic tasks, or handling future tasks, in the case of dynamic arrivals.

We define a schedule length bound of one algorithm relative to an optimal algorithm in the following way. Let x be any instance of task sets to be scheduled, $L_A(x)$ be the schedule length of a scheduling algorithm A for instance x, and $L_O(x)$ be the schedule length of an optimal scheduling algorithm[1] O for x. If there exists a constant B, such that

$$\frac{L_A(x)}{L_O(x)} \leq B, \qquad \text{for all } x,$$

then B is called the bound for algorithm A. B is *tight* if it can be reached.

In this dissertation, we first show that list scheduling has a good bound on schedule lengths but it is not as successful in finding feasible schedules, while the so-called H scheduling algorithm is highly successful in finding feasible schedules but its schedule length bound is not as good as that of list scheduling. So we combine the good properties from both list scheduling and the H scheduling algorithm into a generalized heuristic scheduling algorithm, called H_k, where $2 \leq k \leq m$. The H_k algorithm schedules tasks according to their dynamically determined priorities while attempting to keep at least k processors busy whenever possible. If it is not possible to keep k processors busy, then it will keep as many processors busy as possible. Our main results include:

[1]The optimal algorithm produces feasible schedules with the minimum schedule length.

- The schedule length bound of H_k for tasks with the same computation time is

$$\frac{m}{k} + \sum_{j=2}^{k} \frac{1}{j},$$

where $2 \leq k \leq m$. The schedule length bound of H_k for tasks with arbitrary computation times is

$$\frac{m+1}{2},$$

where $2 \leq k \leq m$. In both cases, the complexity is $O(n^{k+1}r)$ for $2 < k \leq m$, and $O(n^2r)$ for $k = 2$, where n is the number of tasks;

- Using simulation studies of three algorithms, H_2, the H scheduling algorithm, and list scheduling, we show that the H_2 algorithm has almost the same mean behavior in the ability to find feasible schedules as the H scheduling algorithm, and both are much better than list scheduling. The scheduling overheads of H and H_2 are about the same, while the worst case schedule length bound for H_2 is about a half of the worst case schedule length bound for H. Therefore, H_2 is the best candidate for multiprocessor real-time task scheduling among these three algorithms.

1.2.3 Performance-Related Reliability Analysis for Determining Tasks' Redundancy Levels

Many real-time systems have both performance requirements and reliability requirements. Performance is usually measured in terms of success in completing tasks on time. Reliability is determined by hardware and software failure models. In many situations, there are tradeoffs between task performance and task reliability. Thus, a mathematical assessment of performance-reliability tradeoffs is necessary to evaluate the performance of real-time fault-tolerance systems.

Consider a system with m processors and n tasks. We must decide what level of redundancy should be assigned to the tasks such that both reliability and performance requirements are met. In particular, suppose a task T_i provides a reward

V_i if it completes successfully once it is guaranteed, a penalty P_i if it fails after being guaranteed, and a penalty Q_i if it is not guaranteed. Let R_i be the reliability of a guaranteed task T_i and F_i be its failure probability, with $R_i = 1 - F_i$. R_i is mainly affected by the redundancy level for a task T_i and the failure model for the processors. Then, we define a performance index for the system such that it takes the tasks' penalties, rewards, and reliabilities into account. The performance index PI_i for task T_i is defined as

$$PI_i = \begin{cases} V_i R_i - P_i F_i & \text{if } T_i \text{ is guaranteed} \\ -Q_i & \text{if } T_i \text{ is not guaranteed.} \end{cases}$$

The performance index PI for the task set is defined as

$$PI = \sum_{i=1}^{n} PI_i.$$

Given this definition of performance index, if we increase the redundancy level for T_i, we can increase its probability R_i of completing before its deadline, that is, decrease the probability that the task will fail after being guaranteed. But this will reduce the number of tasks that can be guaranteed in the first place and so will increase the penalties due to task rejection. Thus, clearly, there are tradeoffs involved between the fault tolerance of the system, the rewards provided by guaranteed tasks that complete successfully, and the penalties due to tasks that fail after being guaranteed or that fail to be guaranteed. Therefore, some way needs to be found that maximizes rewards while minimizing penalties.

In this dissertation, we will derive a new task configuration strategy with the goal of maximizing the performance index which considers deadline-related performance measures as well as fault-tolerance related requirements. Our main results are

- By using the powerful analysis tool of functional variations, we are able to reveal a simple and elegant principle called *iso-reliability* which says that the best task configuration strategy is to maintain a constant system reliability at

each point in time throughout a mission if tasks have the same reward and penalty rates and the mission length is much larger than the intervals in which tasks are expected to be executed.

- We show that the continuous model is a good approximation for the discrete model. Thus, applying the results for the continuous model, we can handle a discrete task model in which tasks have arbitrary computation times.
- We show that the same analysis approach can be further extended to handle tasks having different reward rates and penalty rates.
- Also, our model is not limited to hardware reliability given by constant failure rate functions as in most other models since we do not depend on the memoryless property.

1.3 Organization of the Thesis

This dissertation studies issues related to dynamic scheduling in real-time systems. In this section, we describe how the dissertation is organized. Chapter 1 presents the research problems, objectives, contributions, and the organization of the dissertation. A survey of various task scheduling approaches is presented in Chapter 2. In particular, we present the state of the art concerning independent task scheduling, task scheduling with resource requirements, and task scheduling with fault-tolerance requirements. Chapter 3 deals with the worst case analysis of on-line scheduling algorithms. In Chapter 4, we discuss the scheduling of tasks with additional resource requirements and analyze a set of heuristic scheduling algorithms with respect to the quality of the generated feasible schedules. We also discuss the insights gained from the analysis. In Chapter 5, we discuss the scheduling of tasks with fault-tolerance requirements. In particular, we focus on the issue of determining the redundancy levels of tasks. We present a technique based on a continuous task model and show

how it very closely approximates discrete models and tasks with varying characteristics. We summarize the dissertation in Chapter 6 by discussing our contributions and outlining some possibilities for future research.

CHAPTER 2

LITERATURE SURVEY

Scheduling is a very old research area and there are many results from the study of manufacturing systems, transportation systems, process control systems, and general computer systems without timing constraints. Two books on computer program scheduling theory [25, 37] provide excellent background readings. Here we only focus on real-time task scheduling. In the following sections, we survey the state of the art in real-time task scheduling in three sub-areas: independent task scheduling, task scheduling with resource requirements, and task scheduling with fault-tolerance constraints. Readers may refer to [85] for a background on real-time systems. [85] also covers real-time scheduling and fault-tolerance.

In a real-time system, the scheduling of tasks and resources involves two main components, feasibility analysis and schedule construction [73]. Feasibility analysis of a set of tasks is the process of verifying whether the timing requirements of these tasks can be satisfied, usually under a given set of constraints. Feasibility analysis can be performed off-line, on-line, or not performed at all. Schedule construction is the process of generating actual task schedules, which can be done either statically or dynamically.

2.1 Independent Task Scheduling

In this section, we discuss scheduling algorithms which use the off-line feasibility analysis first; we then discuss some on-line scheduling algorithms with the on-line feasibility analysis or without the feasibility analysis step. Both uniprocessor scheduling and multiprocessor scheduling are surveyed.

For uniprocessor scheduling algorithms using the off-line feasibility analysis, priority based algorithms have been extensively studied and analyzed. There are two types of priority based algorithms: static and dynamic. A static priority based algorithm assigns priorities to tasks *a priori* and tasks' priorities are fixed afterwards, while a dynamic priority based algorithm assigns priorities to tasks dynamically so that it is possible for a given task to have different priorities before it actually completes execution.

For periodic task scheduling, it has been shown that Rate Monotonic (RM) scheduling algorithm is optimal among all static priority scheduling algorithms [53] and Earliest Deadline First (EDF) and Least Laxity First (LLF) are optimal among all dynamic priority scheduling algorithms [53, 61]. Here optimality is defined as follows. If there exists a feasible schedule for a given set of tasks, the optimal algorithm will be guaranteed to find it. For all three algorithms, although the feasibility analysis is performed off-line, the schedule construction is performed either statically or dynamically. EDF or LLF guarantees to schedule a task set if its utilization is no greater than 1.0 by assigning tasks' priorities dynamically, assuming task deadlines are determined by task periods. RM assigns priorities inversely proportional to task periods, and guarantees to schedule a task set if its utilization is no greater than 0.693 [53], which is called as the performance bound, assuming task deadlines are determined by task periods. The average performance bound of RM is about 0.88 for periods drawn from a uniform distribution [50]. If the system load changes,

such as a mode-change in an avionics control system, the feasibility analysis must be performed on-line upon each mode change [72].

For aperiodic task scheduling, EDF and LLF are again optimal. Both algorithms use off-line feasibility analysis but construct schedules dynamically. For a given task set, if tasks have the same arrival times but different deadlines, EDF generates a non-preemptive schedule, while LLF requires preemptions. The optimality of both algorithms still holds if tasks have the same deadlines but different arrival times. If both arrival times and deadlines are arbitrary, both EDF and LLF may require preemptions. The performance bound for EDF or LLF is 1.0 [53].

Many extensions have been made to the RM algorithm. For example, aperiodic tasks can be handled by the RM algorithm with the deferrable server and the sporadic server [78]. Other important extensions will be discussed in the following sections.

For the multiprocessor scheduling algorithms using the off-line feasibility analysis, there are no simple optimal algorithms. The general approach is to first partition tasks among application processors and then have each processor apply a uniprocessor optimal scheduling algorithm [28, 30]. For periodic tasks, the partitioning problem is similar to a bin-packing problem, which is NP-complete, and there exist several approximation algorithms and their analysis can be found in the papers on bin-packing [33, 41].

Now let us consider some on-line scheduling algorithms, which have been studied extensively. Dertouzos and Mok studied multiprocessor on-line scheduling of real-time tasks [29][1]. They showed that, in the case of multiprocessors, no scheduling algorithm can be optimal and guarantee all tasks without *a priori* knowledge of task deadlines, computation times, and arrival times. Hence, researchers are developing new scheduling algorithms which try to maximize the total task value. It is again a

[1]Their results first appeared in 1978 [62].

difficult problem, because it can be shown that there exists no optimal algorithm for this new metric.

Locke developed an algorithm called Best-Effort (BE) for multiprocessor scheduling, by using time-dependent value functions to schedule real-time tasks [58]. BE does not perform the feasibility analysis. For uniprocessor scheduling, BE behaves the same as EDF when the system is not overloaded. During overloads, BE sheds tasks with the lowest-value-density-first approach. Although BE has been shown to have a good average performance, it does not perform well in the worst case.

Biyabani, Stankovic, and Ramamritham proposed two algorithms for uniprocessor scheduling in a real-time distributed environment and showed that they perform well through a simulation study [15]. They assumed that tasks have both timing constraints and importance values. The two algorithms use an on-line feasibility analysis. If the system is not overloaded, these two algorithms behave the same as EDF. If the system is overloaded, their algorithms shed tasks with less importance values. These two algorithms differ only in how they remove lower importance tasks. In the first algorithm, lower importance tasks are removed one at a time and in strict order from low to high importance. The second algorithm also removes tasks with the lower importance value, but does not follow the strict order found in the first algorithm. Again, the two algorithms do not perform well in the worst case.

Because no optimal algorithm exists for on-line scheduling to maximize the total task value, researchers have turned to a new analysis method, the worst case bound analysis, which provides very good insight into scheduling algorithms. To evaluate a particular on-line scheduling algorithm, the worst-case of a scheduling algorithm is compared with all possible competing algorithms, including the idealized clairvoyant algorithm (one that knows about the future task arrivals).

Baruah *et al.* designed an on-line scheduling algorithm called D* for uniprocessor scheduling [10], which is a modified version of EDF. All tasks are assumed to have

the same value density. The D* algorithm has been shown to have a low bound of 0.2, while the low bound for EDF or LLF is zero. If the system is never overloaded, D* will guarantee service to all tasks. Furthermore, the algorithm experimentally compares well with Locke's Best-Effort scheduler.

Recently, Baruah and Rosier studied the upper bound of uniprocessor scheduling problem [11]. Although they did not derive a tight bound, their work does provide an important base for others.

Besides the real-time on-line scheduling problem, there are many other on-line problems. The recent theoretical development of on-line algorithms has established a *Theory of On-Line Algorithms* which compares the relative power of on-line and off-line (or clairvoyant) algorithms. Sleator and Tarjan analyzed list searching and paging problems [75]. Manasse, McGeock, and Sleator presented results on the K-sever problem. Each on-line decision is to choose one server from K available servers to serve a request in a metric space. Other work on the theory of on-line algorithms can be found in [13, 14, 19, 26, 42].

Our work on dynamic scheduling of independent tasks is a further extension to the earlier results. We will establish the upper bound for on-line uniprocessor scheduling and consider some other extensions, such as, tasks with variable value densities and dual-processor scheduling.

2.2 Scheduling with Resource Requirements

Tasks may require both processor resources and other resources such as shared data structures. The shared resources can be protected by semaphores or locks, which we refer to as the *locking approach*, or by avoiding conflicts based on conflict-free schedules, which we refer to as the *lock-free approach*. The locking approach provides a model to simplify scheduling algorithm design, because the scheduling algorithm does not need to consider the shared data structures and resources explicitly. But the

locking approach may cause blocking during task execution. On the other hand, the lock-free approach complicates task scheduling, since the additional resources which represent the shared data structures protected by critical sections must be handled by a scheduling algorithm explicitly. In this section, we first consider the scheduling algorithms based on locking, which are mainly designed for scheduling of periodic tasks based on off-line feasibility analysis. Then we consider scheduling algorithms based on the lock-free approach, which are mainly designed for scheduling based on the on-line feasibility analysis.

The RM algorithm enhanced with a static priority ceiling protocol studied in [65] and the EDF algorithm enhanced with a dynamic priority ceiling protocol reported in [24] are examples of the locking approach. To achieve good performance with a priority ceiling protocol, the size of critical sections is assumed to be small compared to the size of tasks and the worst case execution times of critical sections are known. In general, the priority ceiling protocols are too pessimistic for some applications, because they must account the worst case blocking time for every task in a critical section. The basic RM with a priority ceiling protocol is not flexible for a dynamic environment, and hence a mode change protocol is discussed in [72].

In the lock-free approach, the scheduling problem is modeled as scheduling tasks with additional resource constraints. Without deadline constraints, scheduling tasks to minimize the schedule length [16, 37] on multiprocessor is a NP-hard problem. Therefore, many heuristic algorithms have been developed [32, 33, 34, 35, 45]. Most heuristic algorithms are similar to list scheduling, which orders tasks into a priority list and selects tasks based on their priorities. With deadline constraints added to tasks, the problem has been studied extensively by the Spring research group of University of Massachusetts at Amherst during last several years [67, 68, 69, 70, 86, 91, 92, 93]. The basic idea is the following. For a given set of processes, the Spring Software Generation System (SGS) [63] analyzes them and partitions each

process into a set of tasks. Tasks are non-preemptable scheduling entities in the Spring system. SGS may identify additional resources needed by a task, and these additional resources could be the data structures protected in critical sections. Then the Spring scheduler uses a heuristic function based on tasks' deadlines, resource requirements, and computation time. Hence, both timing constraints and resource constraints are taken into the consideration explicitly. The primary performance criterion is the *guarantee ratio*, which is defined as the number of times a heuristic scheduling algorithm finds feasible schedules among all tested task sets where it is known that a feasible schedule exists for each set.

There are some advantages with the lock-free approach. It can be applied to more complicated concurrent programming models besides critical sections, it does not require assumptions about the worst-case size of critical sections, and thus avoids the pessimistic worst case blocking time in the priority ceiling protocol.

Our work on scheduling with task synchronization is a further extension to the lock-free approach used in the Spring real-time system. We will develop new scheduling algorithms with the ability to find feasible schedules of good quality, measured in terms of the schedule length, of the generated feasible schedules.

2.3 Task Scheduling with Fault-Tolerance Constraints

In this section, we first survey fault-tolerant programming models. Then some real-time fault-tolerant task scheduling algorithms are discussed. Readers may read the paper by Stankovic, which surveys some real-time scheduling approaches with fault-tolerance constraints [83].

Many fault-tolerant program models can be applied to real-time systems. Fault-tolerance is an active research area. Several papers [2, 3, 27, 47, 57] provide general summaries of fault-tolerance research, while [1, 80] focus on software fault-tolerance

research. [40] provides the recent opinions of a group of experts in fault-tolerance. In the remaining parts of this chapter, we survey some popular fault-tolerant program models, task scheduling with fault-tolerance requirements, and performance-related reliability measurements.

Many fault-tolerance mechanisms have been developed in the last three decades. Each has different advantages and disadvantages based on different fault assumptions. Here we categorize these assumptions into three groups. The first group assumes that software is fault-free and the only faults are from hardware component failures. The second group assumes that faults may come from both software design faults and hardware component failures. The third group assumes that faults come from the events in which application programs miss their deadlines. Let us discuss each group in detail.

To tolerate hardware failures only, there are several fault-tolerant approaches to improve the delivered services:

- Retry,
- N-Modular-Redundancy (NMR), and
- Error Recovery (ER).

Retry involves reissuing the failed action in the hope that it will be successful. It makes use of temporal redundancy to recover from transient faults or after the system eliminates the consequence of errors. Although it is simple and cheap, it cannot recover from an error caused by a permanent hardware fault. Further it assumes that multiple runs of a program do not generate multiple effects to the system or the environment. A recent work using Retry is reported in [71].

NMR is a spatial redundancy approach. It replicates a program with N copies. Each copy is executed on a different processor. The results generated by N copies are voted upon. When N is three, NMR becomes Triple Modular Redundancy (TMR).

TMR is widely used in many fault-tolerance systems, such as Fault-Tolerant Multiprocessor (FTMP) for avionics [77], IBM 9020 [39], and JPL-STAR computer [5]. Some systems use more flexible voting schemes allowing more than three modules to participate in voting, such as Software Implemented Fault-Tolerance (SIFT) computer [77] and Multicomputer Architecture for Fault-Tolerance (MAFT) [87]. One special feature of SIFT is that the voting is done under program control, rather than through hard-wired circuitry [7]. Care must be taken with the NMR approach to avoid generating multiple external effects to other modules or the environment. Also, NMR consumes N times more system resources even if there are no faults in the system.

Recovery refers to the action of eliminating the effect of a fault or error [3]. Common techniques include backward error recovery [36], such as using log files and checkpoints, and forward error recovery. Recovery techniques are intimately related to the issues of fault latency and fault confinement. In general, the longer it takes to detect a fault, the more recovery work is required. Furthermore, fault confinement techniques must be used to block spreading of the fault as much as possible.

Backward error recovery involves the maintenance of recovery points, which introduce overheads during normal system functioning. Also, upon the occurrence of a failure, certain actions may have to be undone before the system state is restored to an earlier error-free state. Thus, in this case, recovery entails time and resource overheads. Since the system is restored to a previous state and some tasks are undone, this implies that any interaction with the environment by the "undone" tasks should be undoable. This will not always hold in real-time systems. A problem of an opposite nature occurs with forward error recovery. Forward error recovery may cause the system to move from the error state to a new correct state. This implies that certain expected interactions with the environment may not occur.

The next group of fault-tolerance mechanisms assumes that faults may come from both software design faults and hardware component failures. It is hard to distinguish the faults caused by software and the faults caused by hardware in many situations. But if provisions exist to deal with software design faults, the faults from hardware component failures can be handled in the same fashion using the fault-tolerant mechanisms we discussed below. There are three approaches to deal with software faults:

- Recovery Block (RB),
- N-Version Programming (NVP), and
- Resourceful Systems (RS).

All three approaches employ multiple versions of programs.

RB uses an on-line acceptance test to determine which version to believe [38, 44]. It has the following structure: *ensure T by B1 else by B2 ... else by Bn else error,* where T is the acceptance test, $B1$ is the primary block, and Bk, $2 \leq k \leq n$, are the alternative blocks. The execution order of blocks in actual implementation can be sequential or parallel.

NVP executes all versions and votes upon their results [4, 23]. It has three steps during the execution: (1) invoking each of the versions; (2) waiting for the versions to complete their execution; (3) comparing and acting upon the N sets of results. Some experiments of NVP are reported in [6, 43], which show several advantages of NVP. But there are shortcomings with this method, which are summarized in [1].

Resourceful System (RS) is a generalization of the RB approach to software fault-tolerance [1]. Systems are resourceful if they can determine whether they have achieved their goals and, if not, to develop and carry out alternative plans. RS possesses three properties: (1) functional richness, (2) an explicit testable goal, and (3)

an ability to develop and to carry out plans for achieving its goals. Some similar ideas can be found in other research areas, such as in real-time AI [84].

In the third group of fault-tolerance mechanisms, the timing properties of software modules are used. For example, a *Deadline Mechanism* can be used to select the proper version of a software module which meets a known deadline [3, 22]. The *Monotonic Algorithm* approach is based on an iterative method so that system scheduling has a certain range of selections to trade off time and precision of a software module with a deadline constraint [52, 55]. This same idea is used in the *Mandatory-Optionals* work [74], which does not require iteration. These imprecise computation models are summarized in [56]. The mechanisms in this group provide a basis to let the system scheduler select a proper version of the corresponding software module, such that the events of missing deadlines are avoided. However, to use these mechanisms, a cost and/or reward functions must be associated with the software modules. Then the system scheduler tries to minimize the total cost or maximize the total reward, which is a non-trivial problem.

The fault-tolerant mechanisms of the three groups discussed so far can be combined. For example, *Multi-Primary-Voting and Multi-Ghost-Backup* is used in [46]. *N Self-Checking Programming* combines both temporal and spatial redundancy [48]. *Multi-Language Versions* combines NVP and RB [64].

Now let us consider some real-time fault-tolerant task scheduling algorithms, with both static scheduling and dynamic scheduling.

For static scheduling, Krishna and Shin considered a multiprocessor scheduling problem which takes fault-tolerance constraints into consideration explicitly [46]. Each task consists of several primaries and several ghosts. Primaries vote upon their computational results. The ghosts are passive backups. They assume that there exists an optimal allocation algorithm to assign fault-tolerant tasks to different application processors, and that there exists an optimal scheduling algorithm for

each processor. They developed a new scheduling algorithm in conjunction with the optimal scheduling algorithm to obtain an optimal schedule when enough ghosts are incorporated into the schedule to sustain the required number of processor failures. A dynamic programming technique is used to derive fault-tolerant schedules. The whole scheme is supposed to apply to static fault-tolerant task scheduling, because the dynamic programming algorithm is an expensive algorithm.

Liestman and Campbell considered fault-tolerant task scheduling based on a deadline mechanism [51]. Their scheme guarantees that a primary algorithm will make its deadline if there is no failure, and that an alternative algorithm (of less precision) will complete by the deadline if there is a failure. For most of the tasks, both primary and alternative algorithms are guaranteed with resources reserved. Their scheduling algorithm tries to increase the system performance by re-using the resources assigned to alternatives.

Ramamritham investigated the techniques used for fault-tolerance in distributed systems and discussed their applicability to the Spring real-time system [66]. Some ideas about scheduling fault-tolerant tasks with respect to distribution, multiprocessing, and communication in Spring are presented.

In distributed systems, a node may die or lose a part of its computational power because the failures of components. It is important to rebalance the load across the whole system and keep the system stable. This problem has been studied by Stankovic [82]. He developed and compared various distributed scheduling/reallocation algorithms for tasks running on a distributed environment. The algorithms include Focused Addressing, Bidding, Local Preemption, and Global Preemption. The stability of the Bidding algorithm has also been analyzed in [81]. Distributed scheduling is not considered in this dissertation, although it is a very important issue.

Finally, it is important to quantify both performance and reliability in fault-tolerance systems. In general, performance-related reliability models use a state-based approach by assigning some kind of performance value or reward to a system's various working configurations. Using continuous-time Markov chain model, a degradable multiprocessor is expressed as an n-state process with state space as 0, 1, $\cdots$, n. State 0 represents the system failed state and state 1 through n represent various working configurations. Each working state i is associated with a reward rate r_i. Solving this Markov chain model would yield the probability that the system is in different working state at time t. For example, $p_i(t)$ is the probability that the system is in state i at time t. Let τ_i be the time spent by the system in configuration i over the mission length L. Then

$$\sum_{i=1}^{n} \tau_i = L.$$

Beaudry introduced measures such as computation reliability and computation availability for degradable multiprocessors [12]. Beaudry's model is built on the Markov reward process. If the reward r_i is interpreted as the amount of useful computation unit over time units e.g., instructions/second, in state i, then the expected computing power at time t is

$$\sum_{i=1}^{n} p_i(t) \cdot r_i.$$

The concepts in Beaudry's model were generalized by Meyer, who introduced performability [60], which is the probability distribution function of accumulated system performance. The performability of the system over the mission time t is the distribution of the accumulated reward

$$\sum_{i=1}^{n} r_i \tau_i.$$

Performability is an important performance metric and has been widely used to analyze fault-tolerant systems.

Lee and Shin introduced an active reconfiguration strategy for a degradable multimodule computing system [49]. They recognized that the system should reconfigure itself after a certain amount of mission time has passed, even without any failure. Their model is also a state-based approach which is represented as a Markov reward process. The rewards are determined by different system configurations, and the reliabilities are determined by two factors:

- the hardware modules' reliabilities where the modules are assumed to have constant failure rates, and
- the probabilities that the system cannot recover from module failures under certain configurations.

Their objective was to maximize the (expected) accumulated rewards, which can be considered as a specific instance of the performability measure introduced by Meyer. Their solution is composed of a set of feasible schedules organized in a two-dimension table. Each feasible schedule in the table corresponds to a different task configuration. The table is then used on-line by the system to choose at during the mission time. Such an active reconfiguration is a very good idea to increase performance.

For the state-based approaches discussed thus far, tasks and task scheduling are implicitly accounted for within a system state. The number of states would explode when considering all possible subsets of tasks, their redundancy levels, and all possible feasible schedules. This may not be a major problem for small static systems. But, in a dynamic system, we cannot afford to use any time-consuming algorithm such as dynamic programming to exhaustively search for a solution and we cannot generate task schedules off-line because we do not have enough task information to make these scheduling decisions. To reduce computational complexity, we must look for an alternative. Our main idea is to focus on the key factor which affects both system reliability and performance. This key factor here is task schedules. The ability to

construct feasible task schedules depends on tasks' redundancy levels which mainly affects system reliability and on the tasks themselves which mainly affects system performance.

Our work on task scheduling with fault-tolerance constraints is a further extension to the performance-related reliability assessment. We provide a fast method to determine the optimal redundancy levels of tasks without explicitly referring to states and without using any expensive algorithms for exhaustive search. Also our method is not limited to hardware reliability given by constant failure rate functions as in most other models, since we do not depend on the memoryless property.

CHAPTER 3

WORST CASE ANALYSIS OF ON-LINE SCHEDULING

3.1 Introduction

The problem of on-line scheduling in real-time environments is to *dynamically* make a sequence of decisions by assigning system resources to real-time tasks. This decision must be made without *a priori* knowledge of future tasks. System resources are processors, memory, and shared data structures[1], and the tasks are assumed to be independent and preemptable, and have arbitrary arrival times, computation times, deadlines, and importance values. If a task request is successfully scheduled to completion, a value equal to the task's execution time is obtained; otherwise a value of zero is obtained. Because a scheduling decision is made without *a priori* knowledge, the outcome of the decision is not fully predictable. So, the objective is to maximize the value accrued from tasks that complete on time.

If overloads are known to be impossible, then the Earliest-Deadline-First algorithm (EDF) can be applied. Intuitively, when a task is preempted, since we have a future knowledge that overloads will never occur, we have 100% confidence that the remaining portion of the task can be completed before its deadline. Many real-time systems do not have such future knowledge. One example is robotics, which requires its control subsystem to adapt to a dynamic environment. It would be too costly to assume that overload will never occur and/or inefficient to construct a schedule *a priori* in such a system. Therefore, on-line scheduling is important in such real-time

[1]Only processors are considered in this chapter.

systems, and on-line scheduling is more practical than off-line scheduling because overload *will* occur in many systems. Overload happens in many practical systems because

- the environment changes;
- there is a burst of task arrivals; or
- a part of the system fails.

Hence, on-line scheduling is necessary to shed task load. Without overload, simple algorithms, such as EDF and Least-Laxity-First (LLF), perform very well. However, with overload, it is more difficult to construct a good on-line algorithm to compete with a *clairvoyant* algorithm, as it will be clear from the following. A clairvoyant algorithm is an ideal optimal off-line algorithm with full knowledge of the future arrivals of tasks.

The *lower bound* on the performance of an on-line scheduling algorithm, A, can be defined in the following way: If over all task arrival sequences, the smallest value of the ratio of the performance of A and that of the clairvoyant algorithm is B_A, then B_A is the *lower bound* on the performance of A. If, for a given scheduling problem, the largest value of B_A for all A is B, then B is the upper (performance) bound of the scheduling problem.

In this chapter, we consider both uniprocessor on-line scheduling and dual-processor on-line scheduling. Tasks have either the same value density or different value densities, where value density is defined as the ratio between the value of a task and its computation time.

In the case of uniprocessor on-line scheduling, if tasks have the same value density, we show that the upper bound on performance is 1/4. Another interpretation of the result is that, in the worst case, an on-line algorithm is only able to complete the amount of work which is 1/4 of the work completed by a clairvoyant algorithm. We

also show that 1/4 is a tight bound by constructing an on-line algorithm, called Threshold-1 (TD_1) algorithm, to reach the bound. TD_1 guarantees no less than 1/4 of the value obtained by a clairvoyant algorithm for any task request sequence. This means that TD_1 has the best *lower bound* among all on-line algorithms. Therefore, TD_1 is an *optimal on-line* scheduling algorithm under overload, and furthermore, it has the same performance as LLF and EDF in case of non-overload.

The above result can be further extended to the cases in which tasks have different value densities. Let γ be the ratio of the highest and lowest value densities of tasks, we prove that the upper bound of the on-line scheduling problem is

$$\frac{1}{\gamma + 1 + 2\sqrt{\gamma}}.$$

As a special case, if γ is 1, the upper bound is 1/4, which is the result mentioned above. If γ is 2, the upper bound is 1/5.828.

For the dual-processor case, we prove an upper bound of 1/2. This bound is shown to be tight in the special case when all the tasks have the same density and zero laxity.

The remainder of the chapter is organized as follows. Section 3.2 presents some notations, assumptions, and examples. Section 3.3 presents some useful properties of a family of integer sequences. These are useful in deriving the upper bound of the on-line scheduling problem. In Section 3.4 we study uniprocessor on-line scheduling by assuming that all tasks have the same value-density. This assumption is removed in Section 3.5. Section 3.6 extends the discussion to systems with dual-processors. We show that the upper bound for the dual-processor on-line scheduling problem is 1/2 if all tasks have the same value density. This bound is tight if the tasks all also have zero laxity. We conclude the chapter in Section 3.7.

3.2 Assumptions and Example

A computer system is assumed to be either a uniprocessor or a dual-processor, which serves a sequence of tasks. Let R be an arbitrarily task request sequence, $\{T_1, T_2, ..., T_n\}$, where n can be arbitrary large. Task T_i is defined by (a_i, c_i, d_i, v_i), where

- a_i — its arrival time,
- c_i — its computation time,
- d_i — its deadline,
- v_i — the value obtained by the system if the task completes its execution before its deadline, otherwise its value is zero. The ratio between task's value v_i and its computation time c_i is called the value density, which is an important parameter used in the scheduling algorithms.

The laxity of task T_i is defined as $d_i - c_i - a_i$. We assume tasks are aperiodic, independent, and preemptable without penalty (it helps to calculate the bound even though it is not a realistic assumption). A preempted task can be resumed on any available processor. We also assume that $a_i \leq a_{i+1}$, where $1 \leq i < n$. Further, the system does not have the information about the tasks before they arrive.

Definition 3.1: Let R be an arbitrary task request sequence. A is an *on-line* scheduling algorithm if it knows T_i only at time a_i. A *clairvoyant* algorithm, C, is an ideal optimal off-line scheduling algorithm, which knows all tasks in R *a priori*.

Definition 3.2: Let R be an arbitrary task request sequence, A be an on-line scheduling algorithm, and C be a clairvoyant algorithm. $V_A(R)$ is the total value obtained by A. $V_C(R)$ is the total value obtained by C.

Definition 3.3: The *lower bound*, B_A, of an on-line scheduling *algorithm*, A, is defined as

$$\frac{V_A(R)}{V_C(R)} \geq B_A, \qquad \text{for all } R,$$

where $B_A \in [0,1]$ because $\forall R \{ V_A(R) \leq V_C(R)\}$.

A lower bound is *tight* if it can be reached. The *upper bound*, B, of an on-line scheduling *problem* is defined as

$$B \geq B_A, \qquad \text{for all } A.$$

An upper bound is *tight* if it can be reached.

Several examples will illustrate these terms and ideas.

Example 3.1: Let A be an on-line scheduling algorithm in a uniprocessor system. A uses a simple strategy to make scheduling decisions: it uses EDF when the system is underloaded, and it favors the tasks with *larger value density* during overload, (this is the best-effort algorithm [58]). Let a task request sequence be

$$R = \{T_1, T_2\},$$

with its parameters specified in Table 3.1.

Table 3.1 Task parameters for Example 3.1

Tasks	a_i	c_i	d_i	v_i
T_1	0	2	2	3
T_2	1	100	101	100

At time 0, T_1 arrives and gets service. At time 1, T_2 arrives and the system is overloaded. Algorithm A favors a task with a larger value density, which is T_1. Hence, T_2 is rejected and is lost. The total value obtained by A is 3, and the total

value obtained by a clairvoyant algorithm can be 100 (v_2). The performance ratio is

$$\frac{V_A(R)}{V_C(R)} = \frac{3}{100}.$$

If the computation time, the deadline, and the value of T_2 increase at the same rate, then the ratio between $V_A(R)$ and $V_C(R)$ goes to zero.

Example 3.2: Let A be an on-line scheduling algorithm in a uniprocessor system. A uses a simple strategy to make scheduling decisions: it uses EDF when the system is underloaded, and it favors a task with *larger value* during overload [15]. Let Let a task request sequence be

$$R = \{T_1, T_1', T_2, T_2', T_3, T_3', T_4, T_4', T_5, T_5', T_6, T_6', T_7, T_7', T_8, \},$$

with its parameters specified in Table 3.2.

Table 3.2 Task parameters for Example 3.2

Tasks	a_i	c_i	d_i	v_i	Tasks	a_i	c_i	d_i	v_i
T_1	0	10	10	10	T_1'	0	9	11	9
T_2	9	11	20	11	T_2'	9	10	21	10
T_3	19	12	31	12	T_3'	19	11	32	11
T_4	30	13	43	13	T_4'	30	12	44	12
T_5	42	14	56	14	T_5'	42	13	57	13
T_6	55	15	70	15	T_6'	55	14	71	14
T_7	69	16	85	16	T_7'	69	15	86	15
T_8	84	16	100	16					

Notice that the value densities of all tasks are the same, which is 1. The schedule of a clairvoyant algorithm is simply in the following order:

$$(T_1', T_2', T_3', T_4', T_5', T_6', T_7', T_8),$$

with the total value 100. The algorithm A works as follows: At time 0, the system is empty and T_1 and T_1' arrive. T_1 gets service and T_1' is discarded because A favors the larger valued task during overload. (A does not know that T_2 will

arrive, otherwise it will choose T_1'.) At time 9, T_2 and T_2' arrive and the system is overloaded again, and T_2 gets service because it is the task with largest value among the current task sets. This pattern continues until T_8 arrives at time 84. The current running task is T_7 with the same value as T_8, hence, algorithm A does not make the switch. The total value obtained by A is 16 because only T_8 makes its deadline and all other tasks are lost. The performance ratio is

$$\frac{V_A(R)}{V_C(R)} = \frac{16}{100}.$$

The above task pattern can be used to construct a scenario with a task arrival sequence with an arbitrary number of tasks, such that the ratio between $V_A(R)$ and $V_C(R)$ goes to zero.

From the above examples, we can observe a phenomenon which is common in on-line scheduling. That is, an on-line algorithm sometimes makes some mistakes because it lacks future knowledge. This is unavoidable. Also, both examples give a zero performance bound in the worst case. But there exist other algorithms which have non-zero performance bounds. Therefore, researchers are searching for on-line scheduling algorithms with good lower bounds. The best one can reach the upper bound of the problem, which will be shown in the following sections.

3.3 Constant-Ratio Sequences

In this section, we study a particular family of integer sequences, which will help us construct worst case situation for on-line scheduling. In particular, the computation times of a task sequence will correspond to such an integer sequence. The properties

of this family are used in the proof of the upper bound of the on-line scheduling problem in the next section. One example of the sequence in this family is

$$1, 3, 8, 20, 48, 112, \cdots \tag{3.1}$$

which is defined by a *recurrence relation*, or *difference equation*:

$$c_{k+2} = 4(c_{k+1} - c_k)$$

with $c_0 = 1$ and $c_1 = 3$.

In general, this family of integer sequences has a generic form:

$$c_{k+2} = \beta(c_{k+1} - c_k) \tag{3.2}$$

with

$$c_0 = 1, \quad \text{and} \quad c_1 = \beta - 1.$$

When $\beta = 4$, it gives sequence (3.1). This family has some interesting properties, which, to our knowledge, have not been studied in literature. We call these sequences *Constant-Ratio* (CR) sequences because of the next property:

Property 3.1: [Constant-Ratio Property]

$$\frac{c_{k-1}}{\sum_{j=0}^{k} c_j} = \frac{1}{\beta}. \tag{3.3}$$

Proof.

$$\begin{aligned}
\sum_{j=0}^{k} c_j &= c_0 + c_1 + \sum_{j=2}^{k} \beta(c_{j-1} - c_{j-2}) \\
&= 1 + (\beta - 1) + \beta \sum_{j=2}^{k} c_{j-1} - \beta \sum_{j=2}^{k} c_{j-2} \\
&= \beta + (\beta c_{k-1} + \beta \sum_{j=1}^{k-2} c_j) - (\beta + \beta \sum_{j=1}^{k-2} c_j) \\
&= \beta c_{k-1}.
\end{aligned}$$

Hence,

$$\frac{c_{k-1}}{\sum_{j=0}^{k} c_j} = \frac{1}{\beta}.$$

□

We will use a CR sequence to construct a task request pattern, such that, in the worst case, a clairvoyant algorithm is able to obtain a value which is close to $\sum_{j=0}^{k} c_k$ while an on-line scheduling algorithm can only obtain a value c_{k-1}. This is the main reason for studying CR sequences.

Next, we show that a CR sequence is monotonically increasing if $\beta \geq 4$, by using a rather standard method in the study of recurrence relations [54].

Property 3.2: [Monotonicity Property]

If $\beta > 4$,

$$c_k = (\frac{1}{2} + \frac{\beta - 2}{2\sqrt{\beta(\beta-4)}})(\frac{\beta + \sqrt{\beta(\beta-4)}}{2})^k + (\frac{1}{2} - \frac{\beta - 2}{2\sqrt{\beta(\beta-4)}})(\frac{\beta - \sqrt{\beta(\beta-4)}}{2})^k \tag{3.4}$$

If $\beta = 4$,

$$c_k = (\frac{k}{2} + 1)2^k. \tag{3.5}$$

Proof. Given the recurrence relation (3.2):

$$c_{k+2} = \beta(c_{k+1} - c_k)$$

the corresponding characteristic equation is

$$x^2 - \beta x + \beta = 0.$$

Part 1: When $\beta > 4$, the characteristic equation has two distinct roots

$$x_1 = \frac{\beta + \sqrt{\beta(\beta-4)}}{2} \qquad \text{and} \qquad x_2 = \frac{\beta - \sqrt{\beta(\beta-4)}}{2}.$$

It follows that

$$c_k = A_0(\frac{\beta + \sqrt{\beta(\beta-4)}}{2})^k + A_1(\frac{\beta - \sqrt{\beta(\beta-4)}}{2})^k. \tag{3.6}$$

With the boundary conditions

$$c_0 = 1 \qquad \text{and} \qquad c_1 = \beta - 1,$$

the two constants can be determined as:

$$A_0 = \frac{1}{2} + \frac{\beta - 2}{2\sqrt{\beta(\beta - 4)}} \qquad \text{and} \qquad A_1 = \frac{1}{2} - \frac{\beta - 2}{2\sqrt{\beta(\beta - 4)}}.$$

Substitute A_0 and A_1 into (3.6) deriving (3.4).

Part 2: When $\beta = 4$, the characteristic equation has two identical roots:

$$x_1 = x_2 = 2.$$

It follows that

$$c_k = (A_0 k + A_1)2^k. \tag{3.7}$$

With the boundary conditions

$$c_0 = 1 \qquad \text{and} \qquad c_1 = \beta - 1,$$

the two constants can be determined as:

$$A_0 = \frac{1}{2} \qquad \text{and} \qquad A_1 = 1.$$

Substitute A_0 and A_1 into (3.7) deriving (3.5). Furthermore, Equation (3.5) is clearly monotonically increasing while k increases. If $\beta > 4$, the sequence defined by (3.2) will increase faster than the case in which $\beta = 4$. This can be seen in Figure 3.1. Therefore, the sequence is monotonically increasing while k increases, when $\beta > 4$. This completes the proof. □

Finally, we show that a CR sequence has an oscillation property if $\beta < 4$.

Property 3.3: [Oscillation Property]

If $\beta < 4$,

$$c_k = \frac{2}{\sqrt{4-\beta}}(\sqrt{\beta})^{k+1}\cos(k\theta - \theta_1), \tag{3.8}$$

where

$$\theta = \tan^{-1}\sqrt{\frac{4-\beta}{\beta}} \qquad \text{and} \qquad \theta_1 = \tan^{-1}\frac{\beta-2}{\sqrt{\beta(4-\beta)}}.$$

Proof. The corresponding characteristic equation is

$$x^2 - \beta x + \beta = 0,$$

which has two distinct roots

$$x_1 = \frac{\beta + i\sqrt{\beta(4-\beta)}}{2} \qquad \text{and} \qquad x_2 = \frac{\beta - i\sqrt{\beta(4-\beta)}}{2},$$

where $i = \sqrt{-1}$. It follows that

$$c_k = A_0(\frac{\beta + i\sqrt{\beta(4-\beta)}}{2})^k + A_1(\frac{\beta - i\sqrt{\beta(4-\beta)}}{2})^k. \tag{3.9}$$

With the boundary conditions

$$c_0 = 1 \qquad \text{and} \qquad c_1 = \beta - 1,$$

we have

$$A_0 + A_1 = 1$$

and

$$A_0(\frac{\beta + i\sqrt{\beta(4-\beta)}}{2}) + A_1(\frac{\beta - i\sqrt{\beta(4-\beta)}}{2}) = \beta - 1.$$

The two constants can be determined as:

$$A_0 = \frac{1}{2} - \frac{i(\beta-2)}{2\sqrt{\beta(4-\beta)}} \qquad \text{and} \qquad A_1 = \frac{1}{2} + \frac{i(\beta-2)}{2\sqrt{\beta(4-\beta)}}.$$

But

$$(\frac{\beta + i\sqrt{\beta(4-\beta)}}{2})^k = (\sqrt{\beta})^k(\cos k\theta + i \sin k\theta) \tag{3.10}$$

and

$$(\frac{\beta - i\sqrt{\beta(4-\beta)}}{2})^k = (\sqrt{\beta})^k(\cos k\theta - i \sin k\theta), \tag{3.11}$$

where

$$\theta = \tan^{-1}\sqrt{\frac{4-\beta}{\beta}}.$$

Substitute A_0, A_1, (3.10), and (3.11) into (3.9):

$$c_k = (\frac{1}{2} - \frac{i(\beta-2)}{2\sqrt{\beta(4-\beta)}})(\sqrt{\beta})^k(\cos k\theta + i \sin k\theta)$$
$$+(\frac{1}{2} + \frac{i(\beta-2)}{2\sqrt{\beta(4-\beta)}})(\sqrt{\beta})^k(\cos k\theta - i \sin k\theta),$$

or simply

$$c_k = (\sqrt{\beta})^k(\cos k\theta + \frac{\beta-2}{\sqrt{\beta(4-\beta)}} \sin k\theta).$$

By defining

$$\theta_1 = \tan^{-1}\frac{\beta-2}{\sqrt{\beta(4-\beta)}},$$

we have

$$c_k = \frac{2}{\sqrt{4-\beta}}(\sqrt{\beta})^{k+1} \cos(k\theta - \theta_1).$$

c_k oscillates while k increases, because there exists a factor of the *cos* function of k and θ while $0 < \theta < 2\pi$. □

Observation 3.1: As mentioned before, CR sequences are used to construct a task request pattern, such that, in the worst case, a clairvoyant algorithm is able to obtain a value which is close to $\sum_{j=0}^{k} c_k$ while an on-line scheduling algorithm can only obtain a value c_{k-1}. The ratio between c_{k-1} and $\sum_{j=0}^{k} c_k$ is $1/\beta$ according to Property 3.1. Hence, the ratio decreases while β increases. But if $\beta \geq 4$, all

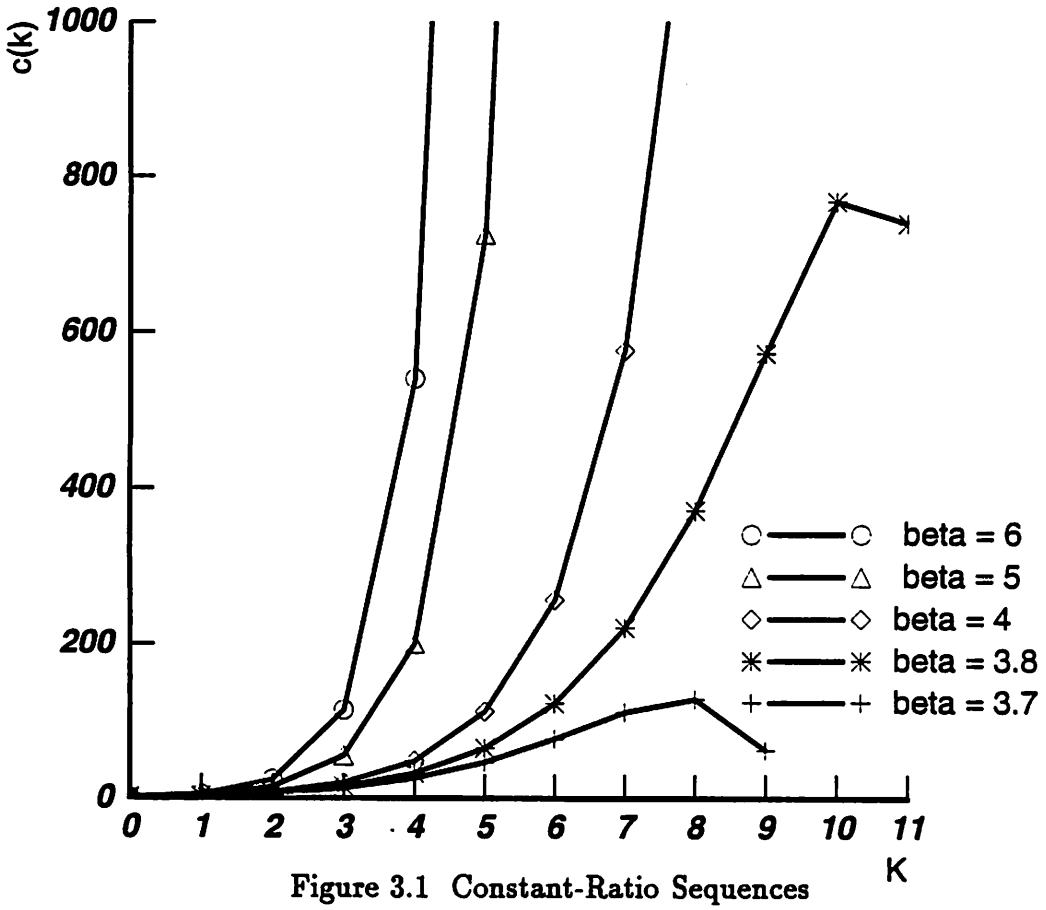

Figure 3.1 Constant-Ratio Sequences

sequences monotonically increase, which means that the value of tasks becomes larger and larger. An on-line scheduling algorithm will simply make a switch every time a more valuable task arrives. The ratio will be much better than $1/\beta$ because the on-line scheduling algorithm gets the most valuable task. On the other hand, when β is less than 4, the sequence begins oscillating. Whenever $c_k \geq c_{k+1}$, the on-line scheduling algorithm cannot make the switch because it is not worth it to switch to a less valuable task. Then, Property 3.1 can be used to measure the performance ratio. Intuitively, the worst case happens when β is very close to 4. Figure 3.1 shows the behavior of CR sequences with the different values of β (**beta** in Figure 3.1).

3.4 Uniprocessor On-Line Scheduling for Tasks with the Same Value Density

In this section, we assume that the value density of all tasks is a constant, and we simply use the computation time of a task as its value. Under this assumption, we prove that the upper bound of the uniprocessor on-line scheduling problem is 1/4, that is, no on-line algorithm has its lower bound better than 1/4. Then we present a simple threshold algorithm with a guaranteed performance ratio of at least 1/4 compared to a clairvoyant algorithm. This is thus the best among all on-line scheduling algorithms with respect to the lower bound.

We first consider a general framework for studying on-line scheduling algorithms. The on-line scheduling problem can be considered as a "game" played by a *player* and an *adversary*. Whenever the adversary posts certain tasks, with different values and deadlines, the player examines these tasks and makes an on-line decision by applying an on-line policy or algorithm, A, to pick some tasks and to reject others, such that the total value obtained is as high as possible.

To show the upper bound of the uniprocessor on-line scheduling problem, it is sometimes necessary to consider the behavior of all algorithms on all possible input patterns according to Definition 3.3. This is a very difficult job because it is not practical to scrutinize all on-line scheduling algorithms. To avoid this, we use the following approach in our proof. We first show that there exists a task request sequence pattern from the adversary, such that, no on-line scheduling algorithm can get a performance ratio higher than 1/4. Then we show that an on-line scheduling algorithm, TD_1, has its performance ratio at least 1/4 for all task request sequences. Consequently, the upper bound of the uniprocessor on-line scheduling problem is 1/4 by simply combining these two facts.

Lemma 3.1: There exists a task request sequence pattern, P, such that, no on-line scheduling algorithm can get a performance ratio higher than 1/4 compared to a clairvoyant algorithm.

Proof It is enough to prove that there exists P and an arbitrarily small δ, such that,

$$\forall R \in P \quad \forall A \quad \frac{V_A(R)}{V_C(R)} \leq \frac{1}{4} + \delta.$$

Let A be an arbitrary on-line scheduling algorithm used by the player of the "game". The adversary uses two types of tasks: τ-tasks and α-tasks with identical value density, represented by

(arrival-time, computation-time, deadline)

as follows:

$$\tau\text{-tasks} \quad : \quad T_t^\tau = (t, \tau, t+\tau),$$

and

$$\alpha\text{-tasks} \quad : \quad T_t^\alpha = (t, \alpha, t+\alpha),$$

where t is time, and α and τ are real to represent task size (computation time), while τ can be arbitrarily small and α will be specified in the following. These two

task types have zero laxity, so the player is forced to make a decision immediately whenever a task arrives.

At time 0, the adversary posts a τ-task and an α-task:

$$T_0^\tau = (0, \tau, \tau), \quad \text{and} \quad T_0^1 = (0, 1, 1).$$

The player has only two choices, T_0^τ or T_0^1. If the player chooses T_0^τ, T_0^1 will be lost. The adversary will stop the game by not providing more requests. In contrast, the clairvoyant algorithm simply chooses T_0^1, and the ratio between the value obtained by the on-line scheduling algorithm, which is τ, and the value obtained by the clairvoyant algorithm, which is 1, is equal to $\tau/1$ which is far less than $1/4$.

If the player chooses T_0^1, then the adversary posts another τ-task at time τ:

$$T_\tau^\tau = (\tau, \tau, 2\tau).$$

Again the player has only two choices: switch or not. If the player aborts T_0^1 for T_τ^τ, the adversary will stop the game and the ratio will be far less than $1/4$. If the player keeps T_0^1, the game continues.

In general, while the player serves an α-task, τ-tasks will keep coming one after another. The adversary will stop the game whenever the player aborts the current α-task for a τ-task.

Now we specify the arrival pattern of other α-tasks. The second α-task will be $T_{1-\tau}^{3-\epsilon}$ at time $1-\tau$. At that time, if the player does not abort the current α-task, T_0^1, the game stops. There are no τ-tasks arriving after T_0^1 completes. Therefore, the player obtains the total value 1, while the clairvoyant algorithm gets the values from $T_{1-\tau}^{3-\epsilon}$ and all τ-tasks between time 0 and $1-\tau$. The ratio is

$$\frac{1}{4-\tau-\epsilon} \leq \frac{1}{4} + \delta,$$

where δ is a function of τ and ϵ, and δ goes to zero as both τ and ϵ go to zero. On the other hand, if the player aborts T_0^1 for $T_{1-\tau}^{3-\epsilon}$, the game continues. τ-tasks keep coming one after another during the time the player serves $T_{1-\tau}^{3-\epsilon}$.

In general, if the player does not abort the current α-task for a τ-task or if the player aborts the current α-task for a new α-task, the adversary will keep posting more τ-tasks and α-tasks. Each τ-task follows the previous τ-task, and each α-task is posted at the time just when the previous α-task can be completed.

The computation time of α-tasks are defined by the following recurrence relation:

$$c_0 = 1,$$

$$c_1 = \beta - 1,$$

$$c_{k+2} = \beta(c_{k+1} - c_k),$$

where

$$\beta = 4 - \epsilon$$

and c_0 and c_1 correspond to T_0^1 and $T_{1-\tau}^{3-\epsilon}$ respectively.

According to Property 3.1 and Property 3.3, we have

$$\frac{c_k}{\sum_{j=0}^{k+1} c_j} = \frac{1}{4-\epsilon}. \tag{3.12}$$

and

$$c_k = \frac{2}{\sqrt{4-\beta}}(\sqrt{\beta})^{k+1}\cos(k\theta - \theta_1),$$

where $\beta = 4 - \epsilon$,

$$\theta = \tan^{-1}\sqrt{\frac{4-\beta}{\beta}},$$

and

$$\theta_1 = \tan^{-1}\frac{\beta - 2}{\sqrt{\beta(4-\beta)}}.$$

Because c_k has a factor of *cos* function of k and θ, and $\theta \neq 0$, $\theta \neq 2\pi$, therefore, there exists k', such that

$$c_{k'} > c_{k'+1}.$$

Hence, the size of α-tasks does not monotonically increase. Whenever the size of the next α-task, $c_{k'+1}$, is less than $c_{k'}$, the adversary changes $c_{k'+1}$ to be the same size

as $c_{k'}$, and stops posting more τ-tasks and α-tasks. At this moment, the player can only choose one of the last two α-tasks with the same size, and obtains a total value c'_k. The clairvoyant algorithm gets the total value of

$$(\sum_{j=0}^{k'} c_j) + c'_k - k'\tau.$$

By Equation (3.12), we have

$$\frac{c'_k}{(\sum_{j=0}^{k'} c_j) + c'_k - k'\tau} < \frac{c'_k}{(4-\epsilon)c'_k - k'\tau} \leq \frac{1}{4} + \delta,$$

where δ is a function of τ and ϵ, and δ goes to zero as both τ and ϵ go to zero.

In summary, the adversary uses a task request pattern, P, such that any on-line scheduling algorithm used by the player has a performance ratio no more than 1/4.

□

Next, we show the 1/4 bound is reachable, that is, there exists a particular on-line scheduling algorithm, TD_1, which has a performance ratio at least 1/4 for all task request sequences, including the worst case pattern we used in the proof of Lemma 3.1.

To introduce the TD_1 algorithm, we define some notation. Then we consider several examples in order to understand the properties of TD_1. Finally, we present the pseudo code of TD_1 in three versions. Each version has a set of different restrictions on the tasks' parameters. In the first version, we assume that all tasks have zero laxity. In the second version, we assume that tasks may have laxities but all preempted tasks are discarded. In the last version, the above two restrictions are removed. For the purpose of showing that the 1/4 bound is reachable, our first version of the TD_1 algorithm is enough. But, tasks do have laxities in many real-time applications. For the practical purpose, we extend the simple algorithm to the second version by considering the scheduling decisions at tasks' lastest start times during overloads. Although these two versions have their performance bound as 1/4 they cannot complete 100% of the workload when the system is underloaded. As

we mentioned before, EDF can complete the 100% of the workload when the system is underloaded. Therefore, we would like to have an algorithm which has the same performance as EDF during underloads but also has the same performance as the second version of TD_1 during overloads. This is our main motivation to design the third version of the TD_1 algorithm.

Technically, to prove the 1/4 bound is reachable for a given algorithm, we must compare the schedule generated by the algorithm to the schedule generated by an optimal algorithm. Instead of considering the whole schedule, which may continue infinitely, we partition the schedule into small pieces called time intervals which are defined below. Using the notion of time intervals, the proof is simply to show that the algorithm can complete the 25% of the work completed by an optimal algorithm in any time interval.

Let t denote current time. Let T_i be an arbitrary task, $T_i = (a_i, c_i, d_i, v_i)$. a_i, c_i, d_i, and v_i are the arrival time, computation time, deadline, and value of T_i respectively as defined in Section 3.2. Let l_i be the *latest start time* of T_i:

$$l_i = d_i - c_i.$$

Definition 3.4: A *time interval*, or simply *interval*, at t is a time segment $[t_b, t_e)$ which consists of a busy subsegment $[t_b, t_f]$ followed by an optional idle subsegment $[t_f, t_e)$, where t_b ($t_b \leq t$) is the time the system transits from an idle state to a running state, t_f ($t \leq t_f$) is the time the system transits (or is expected to transit) back from a running state to an idle state because a task completes (or is expected to complete), and

$$t_e = \max(t_f, \max(\{dl_{discarded}\}))$$

where $\{dl_{discarded}\}$ are the deadlines of all tasks discarded during the time subsegment $[t_b, t_f]$.

Definition 3.5: An interval is *closed* at t whenever a task has completed in the interval, otherwise, it is *open.*

Intervals may be separate, cascaded, or partially overlapping each other. We present two examples to illustrate these terms and ideas.

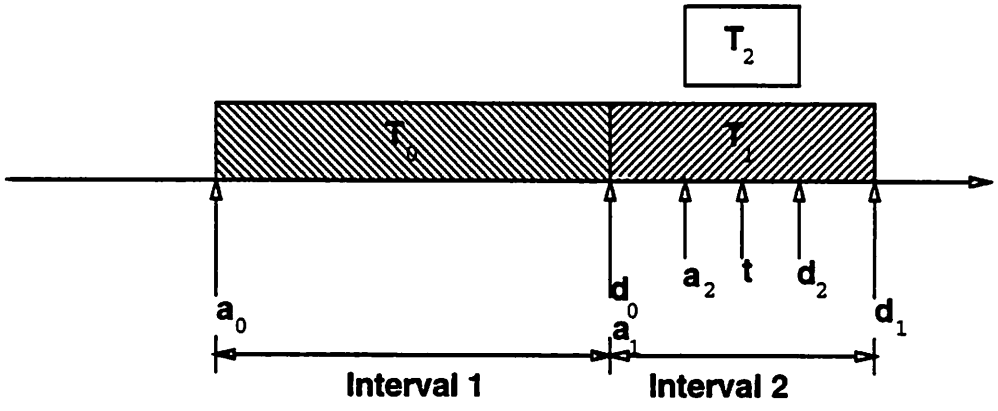

Figure 3.2 A closed interval and an open interval

Example 3.3: Assume T_0, T_1, and T_2 are three tasks with arrival times a_0, a_1, and a_2 and deadlines d_0, d_1, and d_2 as showing in Figure 3.2. These three tasks have zero laxity. Let t be the current time. At time a_0, T_0 arrives and is served at once, which opens Interval 1. Interval 1 is closed at time d_0. At time a_1, the next task T_1 arrives and it is also served at once. The system opens the second interval, Interval 2 $[a_1, d_1)$. At time a_2, T_2 arrives. Let us assume T_2 is rejected at the time a_2. At current time t, T_1 is still running, so Interval 2 remains open, and is expected to end at the time d_1. In Figure 3.2, the shadowed areas represent tasks which are either complete or running, and un-shadowed areas represent tasks that missed their deadlines.

Example 3.4: Assume T_0, T_1, T_2, and T_3 are four tasks with arrival times a_0, a_1, a_2, and a_3 and deadlines d_0, d_1, d_2, and d_3 as showing in Figure 3.3. Let t be

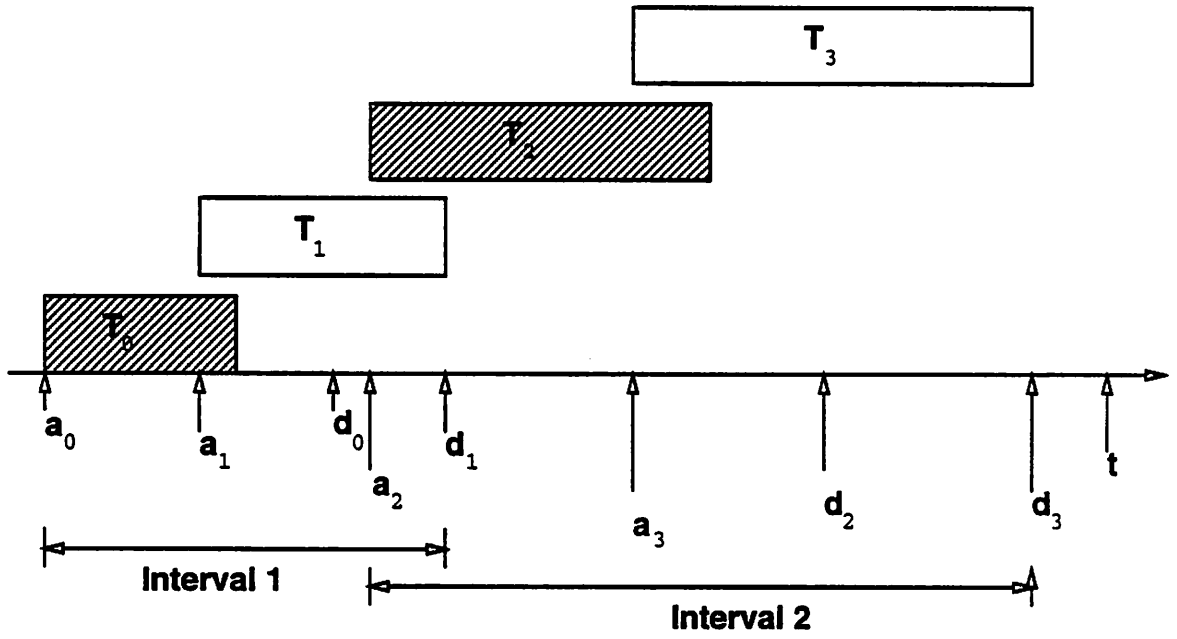

Figure 3.3 Two partially overlapping intervals

the current time. At the time a_0, T_0 arrives and is served at once, which opens an interval $[a_0, a_0 + c_0)$, where c_0 is the computation time of T_0. At the time a_1, T_1 arrives. Assume T_1 is discarded in favor of T_0. The interval $[a_0, a_0 + c_0)$ is expanded to $[a_0, d_1)$ to account for the effect of T_1. At the time $a_0 + c_0$, T_0 completes and the interval $[a_0, d_1)$ is closed. This is Interval 1 in Figure 3.3. At the time a_2, T_2 arrives and is served at once, which opens an interval $[a_2, a_2 + c_2)$, where c_2 is the computation time of T_2. At the time a_3, T_3 arrives. Assume T_3 is discarded in favor of T_3. The interval $[a_2, a_2 + c_2)$ is expanded to $[a_2, d_3)$ to account for the effect of T_3. At the time $a_2 + c_2$, T_2 completes and the interval $[a_2, d_3)$ is closed. This is Interval 2 in Figure 3.3. At the time t, the system is idle and there is no open interval. Interval 1 and Interval 2 are overlapped each other in this example.

In the proofs of the TD_1 algorithm, the following sequences are used. Let Δ be an arbitrary interval. The size of Δ depends on the tasks involved. The arrival pattern

of these involved tasks determines how a sequence of intermediate open intervals grows to the final closed interval Δ. Let

$$(T_{a_1}, T_{a_2}, T_{a_3}, ..., T_{a_n}) \tag{3.13}$$

be a list of n tasks considered in Δ by the algorithm according the time sequence and

$$(\Delta_{a_1}, \Delta_{a_2}, \Delta_{a_3}, ..., \Delta_{a_n}) \tag{3.14}$$

be the corresponding interval list, where each Δ_{a_i} is an interval when T_{a_i} is considered by the algorithm and $\Delta_{a_n} = \Delta$. Let

$$(T_1, T_2, T_3, ..., T_k) \tag{3.15}$$

be a list of tasks executed by the system where each T_i aborts another task, $1 < i \leq k$, and

$$(\Delta_1, \Delta_2, \Delta_3, ..., \Delta_k) \tag{3.16}$$

be the corresponding interval list. Both sequence in (3.14) and sequence in (3.16) are monotonically increasing.

Now we are ready to describe the TD_1 algorithm, which can guarantee a performance ratio of 1/4. TD_1 is used to prove that there is an on-line algorithm with a lower bound of 1/4, which, in turn, is used to show that 1/4 is the tight bound of the uniprocessor on-line scheduling problem. Hence, TD_1 is very simplified. But the algorithm can be easily expanded to further improve its average performance.

In version 1 of TD_1, we assume that all tasks have zero laxity. Therefore, a scheduling decision must be made whenever a new task arrives. Its pseudo code is shown in Figure 3.4, where Δ_{run} records the current interval and v_{run} is the value of a running task. v_{run} is set to zero when the system is idle. The correctness of version 1 of TD_1 guaranteeing a performance bound of 1/4 is based on the following lemma which shows that TD_1 obtains at least 1/4 of the value obtained by a clairvoyant algorithm in each interval.

```
whenever T_next arrives {
    update(Δ_run);
    if (v_run < Δ_run/4) { T_run = T_next; }
}
```

Figure 3.4 On-Line Scheduling Algorithm TD_1 (version 1)

Lemma 3.2: In any interval, version 1 of TD_1 obtains at least 1/4 of the value obtained by a clairvoyant algorithm.

Proof. Let Δ be an arbitrary interval. Let sequences in (3.13), (3.14), (3.15), and (3.16) be defined as before, where the sequence in (3.14) is corresponding to the values of Δ_{run} in the algorithm. We first use mathematical induction on the number of task involved in the task sequence in (3.15) to prove

$$v_k > \Delta_k/2. \tag{3.17}$$

1. *Basis of induction.* For $k = 1$, it is trivial, because $v_1 = \Delta_1$ $(> \Delta_1/2)$.
2. *Induction step.* Assume that $k = i$,

$$v_i > \Delta_i/2. \tag{3.18}$$

Because T_i is aborted by T_{i+1},

$$v_i < \Delta_{i+1}/4 \tag{3.19}$$

and

$$\Delta_{i+1} < \Delta_i + v_{i+1}. \tag{3.20}$$

Applying (3.20), (3.19), and (3.18) in order,

$$\begin{aligned} 2v_{i+1} &> v_{i+1} + (\Delta_{i+1} - \Delta_i) \\ &> v_{i+1} + 4v_i - \Delta_i \\ &> v_{i+1} + 2\Delta_i - \Delta_i \\ &= v_{i+1} + \Delta_i > \Delta_{i+1}. \end{aligned}$$

Thus the inequality (3.17) is true for any integer k.

To show that version 1 of TD_1 obtains at least 1/4 of the value obtained by a clairvoyant algorithm in each interval, we consider two cases.

- Case 1: $T_{a_n} = T_k$, which means that no other tasks arrive after T_k. By the inequality (3.17),

$$v_k > \Delta_k/2 = \Delta/2 > \Delta/4.$$

- Case 2: $T_{a_n} \neq T_k$, which means that there are tasks which arrived after T_k and were discarded. By the threshold rule,

$$v_k \geq \Delta_{a_n}/4 = \Delta/4.$$

□

In the next version, tasks may have laxities, but all preempted tasks are discarded. A queue, Q, is used to hold tasks waiting for service and they are sorted by latest start times in non-decreasing order. Figure 3.5 shows the pseudo code of version 2 of TD_1. In version 2 of TD_1, whenever a new task arrives, it is inserted in Q first. If the system is idle, it will execute the first task in Q and starts a new interval. Otherwise, a scheduling decision is made when the latest start time of the first task in Q is equal to the current time (or when the task's laxity reaches zero). Hence, only the first task in an interval may have laxity and all other tasks have zero laxity because they are invoked at their latest start times. In a given interval, if the first task has non-zero-laxity and is executed fully or partially by TD_1, then a clairvoyant algorithm can execute this non-zero-laxity task after the interval while version 2 of TD_1 cannot, because the task has been executed inside of the interval even though its deadline may be larger than the end point of the current interval. So the clairvoyant algorithm may obtain the value of the first task by scheduling it after the end of the current interval and the all values inside of the interval by packing it fully with small tasks. The maximum value which is obtainable after the current interval is bounded

by the value of the first task. If we do not consider this situation in version 2 of TD_1, the algorithm cannot guarantee 1/4 of the value obtained by the clairvoyant algorithm. To deal with this problem, the algorithm uses a variable, *potential_loss* (*p_loss*), as a compensation. *p_loss* records the value of the first task in an interval, which is the only task that may have laxity. The clairvoyant algorithm may obtain the value which is proportional to the interval size plus the value of *p_loss* which is bounded by the value of the first task.

The correctness of version 2 of TD_1 with respect to a performance bound of 1/4 is based on the following lemma which shows that TD_1 obtains at least 1/4 of the value obtained by a clairvoyant algorithm in each interval when a compensation is added.

```
whenever (idle && not-empty(Q)) {
    T_run = dequeue(Q); p_loss = v_run;
}
whenever (running && alarm(Q)) {
    T_next = dequeue(Q); update(Δ_run);
    if (v_run < (Δ_run + p_loss)/4)
        {T_run = T_next; }
}
```

Figure 3.5 On-Line Scheduling Algorithm TD_1 (version 2)

Lemma 3.3: Version 2 of TD_1 obtains at least 1/4 of the value obtained by a clairvoyant algorithm in any interval with considering the compensation.

Proof. Let Δ be an arbitrary interval. Let sequences in (3.13), (3.14), (3.15), and (3.16) be defined as before, where the sequence in (3.14) corresponds to the values of

Δ_{run} in the algorithm. We first use mathematical induction on the number of tasks involved in the task sequence in (3.15) to prove

$$v_k \geq (\Delta_k + p\_loss)/2. \tag{3.21}$$

1. *Basis of induction.* For $k = 1$, it is trivial, because

$$v_1 = \Delta_1 = (\Delta_1 + p\_loss)/2,$$

where $p\_loss = \Delta_1$.

2. *Induction step.* Assume that $k = i$,

$$v_i > (\Delta_i + p\_loss)/2. \tag{3.22}$$

Because T_i is aborted by T_{i+1},

$$v_i < (\Delta_{i+1} + p\_loss)/4 \tag{3.23}$$

and

$$\Delta_{i+1} < \Delta_i + v_{i+1}. \tag{3.24}$$

Applying (3.24), (3.23), and (3.22) in order,

$$\begin{aligned} 2v_{i+1} &> v_{i+1} + (\Delta_{i+1} - \Delta_i) \\ &> v_{i+1} + 4v_i - p\_loss - \Delta_i \\ &> v_{i+1} + 2(\Delta_i + p\_loss) - p\_loss - \Delta_i \\ &= v_{i+1} + \Delta_i + p\_loss \\ &> \Delta_{i+1} + p\_loss. \end{aligned}$$

Thus the inequality (3.21) is true for any integer k.

To show that version 2 of TD_1 obtains at least 1/4 of the value obtained by a clairvoyant algorithm in each interval, we again consider two cases.

- Case 1: $T_{a_n} = T_k$, which means that no other tasks compete with T_k. By the inequality (3.21),

$$\begin{aligned} v_k &> (\Delta_k + p\_loss)/2 \\ &= (\Delta + p\_loss)/2 \\ &> (\Delta + p\_loss)/4. \end{aligned}$$

- Case 2: $T_{a_n} \neq T_k$, which means that some tasks compete with T_k and are discarded. The threshold rule guarantees that

$$v_k \geq (\Delta_{a_n} + p\_loss)/4.$$

□

It is enough to use version 2 of TD_1 to demonstrate the performance bound of 1/4 is a tight bound for the uni-processor on-line scheduling problem. However, when it is compared with EDF, there is still a small problem. If the system is underloaded, EDF has a performance bound of 1 while TD_1 has 1/4. It is better to design an algorithm having a performance bound of 1 under non-overloads and 1/4 under overloads. This is the motivation for the design of version 3 of TD_1.

The new algorithm uses the Earliest Deadline First (EDF) rule when the system is not overloaded. Therefore, TD_1 guarantees a performance bound of 1 under non-overloads. A queue, Q, is used to hold tasks waiting for service and they are sorted by deadlines. Because of the EDF rule, some tasks in Q may have been executed partially. These tasks are called *fragmented tasks*, while other tasks are called *regular tasks*. The interval definition is accordingly expanded as follows.

Definition 3.6: A *time interval* or simply *interval* at t is a time segment $[t_b, t_e)$ and consists of a busy subsegment $[t_b, t_f]$ followed by an optional idle subsegment $[t_f, t_e)$. It satisfies all the following conditions:

- t_b ($t_b \leq t$) is the time the system transits from an idle state to a running state or the time the system switches from a running task to a regular task with both tasks feasible;
- In $[t_b, t_f]$, there is no such switch from a running task to a regular task with both tasks feasible;
- t_f ($t \leq t_f$) is the time the system transits (or is expected to transit) back from a running state to an idle state, because some tasks complete (or is expected to complete);
- All fragmented tasks in Q can completed before their deadlines if they start after t_e; ***and***
- $t_e = \max(t_f, \max(\{dl_{fragmented}\}), \max(\{dl_{discarded}\}))$, where $\{dl_{fragmented}\}$ are the deadlines of all fragmented tasks which are either discarded or completed and $\{dl_{discarded}\}$ are the deadlines of all tasks discarded during the time subsegment $[t_b, t_f]$.

An interval is *closed* when a task completes and all involved fragmented tasks are either completed or discarded, otherwise it is *open.*

There are some features in the new algorithm. Whenever a task from Q starts execution, the algorithm guarantees that remaining fragmented tasks in the Q are feasible if they start after the end point of the current interval. If an interval is underloaded, only one task is actually involved in an interval. The tasks are completed in the order of their deadlines (the EDF rule). An open underloaded interval may be aborted when the system switches to a new task with a smaller deadline and both tasks are feasible. The preempted task becomes a fragmented task and is put back to Q. The time consumed in the aborted interval will be counted as the compensation in a future interval. (The compensation will be recorded in a variable, *p_loss*, as in version 2 of TD_1.) If an interval is overloaded, more than one task is

involved in the interval, with their deadlines located within the interval. Let T_{run} be a running task and T_{next} be a regular task in the queue. If $l_{next} < t_f$, the algorithm makes a scheduling decision at l_{next}. Some fragmented tasks, $\mathbf{T}_{frag}$, may also conflict with T_{next}, which means that $\mathbf{T}_{frag}$ is a minimum subset of the fragmented tasks removed from Q such that the T_{next} and the remaining fragmented tasks in Q are feasible. Therefore, the algorithm chooses either T_{run} and $\mathbf{T}_{frag}$ or T_{next} based on a threshold rule. If T_{next} is discarded, the algorithm is expected to complete T_{run} and $\mathbf{T}_{frag}$ in the current interval. If T_{next} wins, then T_{run} and $\mathbf{T}_{frag}$ are discarded. The algorithm maintains a conflict task set, $\mathbf{T}_{conflict}$, which consists of all $\mathbf{T}_{frag}$ involved in the interval that have not yet been discarded or executed. When a running task completes, the system executes a fragmented task from $\mathbf{T}_{conflict}$ immediately if it is not empty, otherwise, the interval is closed at this moment. A clairvoyant algorithm may get extra values outside of an interval with, the total amount being bounded by *p_loss*. So we must add this factor into the threshold rule as a compensation.

Figure 3.6 is the pseudo code of version 3. It applies the following threshold rule under overloads:

$$\frac{gain + v_{run} + Value(\mathbf{T}_{conflict})}{\Delta_{run} + p\_loss} \geq \frac{1}{4}, \tag{3.25}$$

where *gain* is the value obtained from the tasks which have been completed in the interval, $Value(\mathbf{T}_{conflict})$ is the value obtained by executing $\mathbf{T}_{conflict}$, Δ_{run} is the current interval size, and *p_loss* is the sum of the previously aborted intervals which relates to all involved fragmented tasks in the current interval.

The correctness of version 3 of TD_1 with respect to guaranteeing a performance bound of 1/4 under overloads and 1 under non-overloads is based on the following lemma.

Lemma 3.4: In any interval, version 3 of TD_1 has a performance bound 1 under non-overloads and 1/4 under overloads, assuming all tasks have the same value density.

```
choose_one(T_run, T_next, Q)
{
    "compute T_conflict;"
    update(p_loss); update(Δ_run);
    if ("Rule in (3.25) is false")
        { T_run = T_next; discard(T_conflict); }
    return(T_run);
}
/* The event-triggered routines */
whenever (idle && not-empty(Q)) {
    T_next = remove_edf(Q);
    T_run = choose_one(Null, T_next, Q);
}
whenever (running && alarm(Q)) {
    T_next = remove_alarmed_task(Q);
    T_run = choose_one(T_run, T_next, Q);
}
whenever (finish) {
    if (empty(T_conflict))
        "set system state to idle";
    else T_run = remove_edf(T_conflict);
}
whenever (arrival) {
    if ( (d_run ≤ d_arr) || idle )
        insert(T_arr, Q);
    else if (feasible (T_run, T_arr, "all fragments")) {
        insert(T_run, Q);
        T_run = T_arr;
        "start a new interval, Δ_run";
    } else
        T_run = choose_one(T_run, T_arr, Q);
}
```

Figure 3.6 On-Line Scheduling Algorithm TD_1 (version 3)

Proof. In underloads, it is trivial to see that the algorithm follows the Earliest-Deadline-First rule, which guarantees a performance bound of 1. The remaining part of the lemma is proved as follows.

Let Δ be an arbitrary interval. Let sequences in (3.13), (3.14), (3.15), and (3.16) be defined as before, where the sequence in (3.14) corresponds to the values of Δ_{run} in the algorithm. It is possible that there are more than one task completions in the task sequence in (3.15). We use mathematical induction on the number of tasks involved in the task sequence in (3.15) to prove:

$$v_k > (\Delta_k + p\_loss_k)/2, \tag{3.26}$$

1. *Basis of induction.* For $k = 1$, it is trivial, because $v_1 = \Delta_1 + p\_loss_1$.
2. *Induction step.* Assume that $k = i$,

$$v_i > (\Delta_i + p\_loss_i)/2. \tag{3.27}$$

Because the system switches to T_{i+1}, the threshold rule in (3.25) is false, which implies:

$$v_i + V_{i+1} < (\Delta_{i+1} + p\_loss_{i+1})/4, \tag{3.28}$$

where V_{i+1} corresponds to $Value(\mathbf{T}_{conflict})$ in (3.25). With the definition of the interval and the property of the compensation, we have

$$\Delta_{i+1} < \Delta_i + v_{i+1} + V_{i+1} \tag{3.29}$$

and

$$p\_loss_i + V_{i+1} \geq p\_loss_{i+1} \tag{3.30}$$

Applying (3.29), (3.28), (3.27) and (3.30) in order,

$$\begin{aligned} 2v_{i+1} &> v_{i+1} + (\Delta_{i+1} - \Delta_i - V_{i+1}) \\ &> v_{i+1} + (4(v_i + V_{i+1}) - p\_loss_{i+1}) - \Delta_i - V_{i+1} \end{aligned}$$

$$
\begin{aligned}
&> v_{i+1} + 2(\Delta_i + p\_loss_i) - p\_loss_{i+1} - \Delta_i + 3V_{i+1} \\
&= (\Delta_i + v_{i+1} + V_{i+1}) + 2(p\_loss_i + V_{i+1}) - p\_loss_{i+1} \\
&\geq \Delta_{i+1} + 2p\_loss_{i+1} - p\_loss_{i+1} \\
&= \Delta_{i+1} + p\_loss_{i+1}.
\end{aligned}
$$

Thus the inequality (3.26) is true for any integer k.

To show that version 3 of TD_1 obtains at least 1/4 of the value obtained by a clairvoyant algorithm in each interval with the compensation p_loss, we consider two cases.

- Case 1: $T_{a_n} = T_k$, which means that no other tasks arrive after T_k. By the inequality (3.26),

$$v_k > (\Delta_k + p\_loss_k)/2 > (\Delta_k + p\_loss_k)/4.$$

- Case 2: $T_{a_n} \neq T_k$, which means that the system does not make any preemption after T_k in this interval. The performance bound of 1/4 is guaranteed by the threshold rule in (3.25). □

Combining the above results, we have the following theorem.

Theorem 3.1: If all tasks have the same value density, then the upper bound of the uniprocessor on-line scheduling problem is 1/4.

We have the following comments about the theorem.

- The assumption that the same value density on all tasks may not be practical. However, it is an important special case of the model considered in the next section. It provides a basis for analyzing more sophisticated models.

- If we interpret the task's computation time as its value, then this theorem says that, in the worst case, any on-line algorithm can only complete 1/4 of the work completed by a clairvoyant algorithm.

- The theorem has another implicit assumption, which is that the computation time of tasks can be arbitrarily small or large. This may not be true in practice. The bound can be improved if the ratio between the largest and the smallest computation times is bounded. For example, if the ratio is arbitrarily close to 1, the bound is also close to 1. The bound decreases while the ratio increases. The bound converges to 1/4, as the ratio goes to infinity.

- Although TD_1 is used to prove the theorem, version 3 of TD_1 can be applied in many real-time systems to substitute for the EDF scheduling algorithm. The difference between EDF and TD_1 is that EDF uses only information about tasks' deadlines while TD_1 uses more information, such as, the size of the current interval, the value of a running task, the value of the task that competes for the processor resource, and tasks' deadlines.

3.5 Uniprocessor On-Line Scheduling for Tasks with Arbitrary Value Densities

In this section, we generalize the result of the last section to the case in which tasks do not have the same value density.

Let γ be the ratio between the highest value density and the lowest value density of tasks. The actual value densities of tasks can be mapped to $[1, \gamma]$. As in the proof of Lemma 3.1 in last section, an adversary uses both α-tasks and τ-tasks to build a worst case pattern of task request sequence. The adversary assigns a value density

1 to all α-tasks and a value density γ to all τ-tasks. The computation times of the α-tasks are defined by the following recurrence relation:

$$c_{k+2} = (\beta - \gamma + 1)c_{k+1} - \beta c_k \tag{3.31}$$

with the boundary conditions as

$$c_0 = 1$$

and

$$c_1 = \beta - \gamma.$$

The constant ratio property mentioned in Section 3.4 can be generalized to the following form:

Property 3.1: [Constant Ratio Property]

$$\frac{c_{k-1}}{\gamma \sum_{j=0}^{k-1} c_j + c_k} = \frac{1}{\beta}. \tag{3.32}$$

Proof. We use mathematical induction to prove

$$\gamma \sum_{j=0}^{k-1} c_j + c_k = \beta c_{k-1}.$$

1. *Basis of induction.* For $k = 1$, we have

$$\gamma c_0 + c_1 = \gamma + (\beta - \gamma) = \beta = \beta c_0.$$

2. *Induction step.* Assume that

$$\gamma \sum_{j=0}^{k-1} c_j + c_k = \beta c_{k-1}.$$

We have

$$\begin{aligned}
\gamma \sum_{j=0}^{k} c_j + c_{k+1} &= (\gamma \sum_{j=0}^{k-1} c_j + \gamma c_k) + ((\beta - \gamma + 1)c_k - \beta c_{k-1}) \\
&= (\gamma \sum_{j=0}^{k-1} c_j + c_k) + \beta c_k - \beta c_{k-1} \\
&= \beta c_{k-1} + \beta c_k - \beta c_{k-1} \\
&= \beta c_k.
\end{aligned}$$

Hence, Equation (3.32) is true for all $k \geq 1$. □

Lemma 3.5: Given γ, there exists task request sequence pattern, P', such that, no on-line scheduling algorithm can get a performance ratio higher than $1/(\gamma + 1 + 2\sqrt{\gamma})$ compared to a clairvoyant algorithm.

Proof. The adversary uses a sequence of the α-tasks defined by the recurrence relation (3.31) and presents them to the on-line scheduling algorithm in the similar way as before, so that whenever the "game" is stopped, the performance ratio will never be larger than $1/\beta$.

The adversary wants the α-task sequence to have the following two features:

1. the "game" always stops; and

2. β is as large as possible.

The first feature requires that the value of the α-task sequence should oscillate. The characteristic equation for (3.31) is

$$x^2 - (\beta - \gamma + 1)x + \beta = 0.$$

The oscillation occurs when the characteristic roots are complex numbers, which happens if

$$(\beta - \gamma + 1)^2 - 4\beta < 0. \tag{3.33}$$

The inequality of (3.33) can be written as

$$(\beta - (\gamma + 1 + 2\sqrt{\gamma}))(\beta - (\gamma + 1 - 2\sqrt{\gamma})) < 0,$$

which gives

$$\gamma + 1 - 2\sqrt{\gamma} < \beta < \gamma + 1 + 2\sqrt{\gamma} \tag{3.34}$$

and

$$\gamma + 1 - 2\sqrt{\gamma} > \beta > \gamma + 1 + 2\sqrt{\gamma}. \tag{3.35}$$

(3.35) is a contradiction and is discarded.

Therefore the largest possible value of β is $(\gamma + 1 + 2\sqrt{\gamma}) - \epsilon$, which will be used by

the adversary to construct the α-task sequence. Thus, the adversary has a strategy to force every on-line algorithm to stop with a performance ratio not higher than $1/(\gamma + 1 + 2\sqrt{\gamma})$ compared to a clairvoyant algorithm. □

Lemma 3.6: Given γ, there exists an on-line scheduling algorithm, TD_1', for all input sequences of tasks, it has a performance ratio at least $1/(\gamma + 1 + 2\sqrt{\gamma})$ compared to a clairvoyant algorithm.

Proof. Similar to the proofs of Lemma 3.2, 3.3, and 3.4 in last section, we may construct an algorithm TD_1' from TD_1 by using a new threshold, which is

$$\frac{1}{\gamma + 1 + 2\sqrt{\gamma}},$$

to substitute to the old threshold 1/4. The basic idea is as follows. TD_1' computes the ratio between the total value obtained or obtainable by the algorithm in a time interval and the total value obtained or obtainable by the clairvoyant algorithm in the same time interval, and then compares this ratio to the above threshold to made the decisions of task switches. For example, corresponding to version 1 of TD_1, in a given interval, the value obtained or obtainable by TD_1' is v_{run} and the value obtained or obtainable by the clairvoyant algorithm is bounded by $\gamma(t_f - t_b) + Value([t_e - t_f])$, where $Value([t_e - t_f])$ is the maximum value obtainable in the time segment of $[t_e - t_f]$. Using the same argument as before, it is easy to show that TD_1' has the desired lower bound. □

Combining the above two lemmas, we derive the following theorem.

Theorem 3.2: If γ is the ratio between the highest and the lowest value density of tasks, then the upper bound of the uniprocessor on-line scheduling problem is

$$\frac{1}{\gamma + 1 + 2\sqrt{\gamma}}.$$

We have the following comments about the theorem.

- It is easy to verify that the upper bound is 1/4 when γ is 1, which is the same as the first result.

- When γ increases, the upper bound decreases. This means that a clairvoyant algorithm has a greater advantage over an on-line scheduling algorithm, because, in the worst case, the clairvoyant algorithm works on the highest value density tasks while the on-line algorithm works on the lowest value density tasks.

3.6 On-Line Scheduling on Dual Processors

We further generalize the previous results from uniprocessor to dual-processor on-line scheduling in this section. We first show that the upper bound for the dual-processor on-line scheduling problem is 1/2. Then we show that 1/2 is tight for tasks without laxity.

There is a special feature of the dual-processor on-line scheduling problem which distinguishes it from the uniprocessor on-line scheduling problem. With dual processors, the two processors can co-operate with each other. This kind of co-operation may allow a dual-processor system to perform better than two separate uni-processor systems.

Lemma 3.7: If tasks have uniform value density, then 1/2 is an upper bound for the dual-processor on-line scheduling problem.

Proof. We show that there exists a worst-case sequence of task requests such that no on-line scheduling algorithms may acquire a value more than 1/2 that obtained by a clairvoyant algorithm.

We use a player-adversary model with the player as an on-line scheduling algorithm. The adversary uses a task sequence pattern, P, to force the player to stop the "game" within a limited time and to get a performance ratio no higher than

$(1/2+\delta)$ compared to a clairvoyant algorithm, where δ can be arbitrarily small. The adversary's strategy is described as follows.

The adversary uses two types of tasks: τ-tasks and unit-tasks with identical value density, represented by

(arrival-time, computation-time, deadline)

as follows:

$$\tau\text{-tasks} \quad : \quad T_t^\tau = (t, \tau, t+\tau),$$

and

$$\text{unit-tasks} \quad : \quad T_t^1 = (t, 1, t+1),$$

where t is time. Both task types have zero laxity. Hence, the player is forced to make a decision immediately whenever a task arrives.

At time 0, the adversary first posts two unit-tasks and two τ-tasks. The player has six choices:

$$\Phi, \{\tau\}, \{1\}, \{\tau, \tau\}, \{\tau, 1\}, \{1, 1\},$$

where $\{\tau, 1\}$ represents the assignment of a τ-task and a unit-task to the two processors and Φ represents an empty set. If the player chooses one from the first five choices, the adversary stops posting new tasks, and the performance ratio can be easily checked to be less than or equal to $(1/2+\delta)$ where δ goes to zero as τ goes to zero.

If the player chooses the last choice, the "game" continues. The adversary will post more pairs of τ-tasks in the time segment of $[\tau, 1-\tau]$. The "game" will stop whenever the player makes a switch from a unit-task to a τ-task, and the ratio will be less than or equal to $(1/2+\delta)$ when τ goes to zero.

At time $(1-\tau)$, if the "game" is not stopped, which means that the two processors execute the two unit-tasks, the adversary posts two more unit-tasks and then stops posting more tasks afterwards. This time, the player has three choices: not switching,

switching one, and switching two tasks. In any of the three choices, the player obtains a total value of 2, but a clairvoyant algorithm obtains a total value $(4-2\tau)$ from all τ-tasks and two unit-tasks. The ratio is equal to $(1/2+\delta)$ as τ goes to zero.

Therefore, there is no chance for the player to obtain a performance ratio higher than 1/2.

Next, we show that 1/2 is tight for tasks without laxity by presenting a particular dual-processor on-line scheduling algorithm to reach the 1/2 bound.

We need some notation first. Whenever the system is not idle, one of the two processors is arbitrarily designated the "primary processor"(PP), and the other is designated the "secondary processor"(SP). A task running on PP is guaranteed to complete and will be called a primary task, denoted as T_p. T_p starts at time t_{s_p} and is expected to complete at time t_{f_p}. A task running on SP is not guaranteed to complete and will be called a secondary task, denoted as T_s. T_s starts at time t_{s_s} and is expected to complete at time t_{f_s}. Let T_{next} denote a newly arrived task.

Lemma 3.8: If tasks have the same value density and zero laxity, then 1/2 is tight for dual-processors.

Proof. We show that, under the conditions that all tasks have the same value density and zero laxity, there is a dual-processor on-line scheduling algorithm that achieves the 1/2 bound.

In fact, the following algorithm is such an example. We simply call it DPS (Dual-Processor Scheduling).

1.1 If a new task T_{next} arrives and both processors are idle, assign T_{next} to an arbitrary processor which becomes PP and the other processor becomes SP. Accordingly, the task becomes primary task T_p, and will be served in $[t_{s_p}, t_{f_p})$. Furthermore, set both t_{s_s} and t_{f_s} to the current time, t.

1.2 During the period of $[t_{s_p}, t_{f_p})$, if a new task T_{next} arrives, then assign T_{next} to SP if $d_{next} > t_{f_s}$ and discard T_{next} otherwise.

1.3 At time t_{f_p}, T_p is completes on PP. If SP is busy, switch the roles of PP and SP.

We notice that, when PP works on a primary task, SP is treated as a helper, or an assistant to PP. During the period of the execution of the primary task, SP always executes the task with the latest deadline, so PP never worrys about a large task that will be lost. Therefore, at the time the primary task completes, algorithm DPS guarantees that (a) all tasks rejected in the period of $[t_{s_p}, t_{f_p})$ have their deadlines less than or equal to t_{f_s}, which is the finish time of the secondary task; (b) both the primary task and the secondary task will be completed in the period of $[t_{s_p}, t_{f_s})$ because SP changes to PP at time t_{f_p}. Hence, DPS obtains at least 1/2 of the value obtained by a clairvoyant algorithm in the period of $[t_{s_p}, t_{f_s})$. The above argument can be used repeatedly. If all tasks in the system are projected on a time axis, and all tasks completed by DPS are projected on another time axis, these two projections will be identical. A more rigorous mathematical induction on the number of tasks can be easily applied here.

Therefore, under the conditions of identical value density and zero laxity, 1/2 is tight, since it is both an upper bound and achievable. □

Combining the above two lemmas, we prove the following theorem:

Theorem 3.3: If all tasks have the same value density, then the upper bound of the dual processor on-line scheduling problem is 1/2.

Compared to uniprocessor on-line scheduling, dual-processor on-line scheduling has a higher performance bound 1/2, which is a 200% improvement over the bound for uniprocessor on-line scheduling. The reason is that two processors can cooperate with each other to guarantee that at least one processor does some productive work

compared to a clairvoyant algorithm. For the designers of real-time systems, this will be an important reason to choose a dual-processor based system instead of a uniprocessor based system.

There are a number of important extensions which are still open, such as, on-line scheduling for three or more processors and on-line scheduling for dual-processor with tasks having arbitrary value densities.

3.7 Conclusions

In this chapter we have discussed the upper bound for any on-line scheduling algorithm in a real-time environment, in which the overload must be handled quickly and effectively. If all tasks have the same value density, the upper bound for the uniprocessor on-line scheduling problem is 1/4. If tasks have different value densities and the ratio between the highest and the smallest value density is γ, the upper bound for the uniprocessor on-line scheduling problem is $1/(\gamma + 1 + 2\sqrt{\gamma})$. We have also presented the on-line scheduling algorithms, TD_1 and TD_1', to reach these two upper bounds respectively. TD_1 and TD_1' use a simple threshold rule to make on-line decisions during the system overload periods. They can be easily implemented and further optimized. The upper bound is doubled from 1/4 in uniprocessors to 1/2 in dual-processors, which means that, in the worst case, the value obtained from a dual-processor system is twice the value obtained from two separate uniprocessor systems. For the designers of real-time systems, this will be an important reason to choose a dual-processor system instead of a uniprocessor system.

CHAPTER 4

BOUND ANALYSIS OF HEURISTIC ALGORITHMS FOR TASKS WITH RESOURCE REQUIREMENTS

4.1 Introduction

From the last chapter, we have known that a best on-line scheduling algorithm can only complete the 25% of the work completed by an optimal algorithm in the worst case, assuming tasks have the same value density and the system has only one processor. The performance can be much lower if tasks are not preemptable and they have additional resource requirements. In this chapter, we focus a different issue and we consider guarantee-oriented multiprocessor scheduling algorithms for tasks with additional resources. A guarantee-oriented algorithm generates a new feasible schedule whenever there is a task arrives. To determine feasibility for a set of tasks is a difficult problem and it has been shown to be NP-complete [32, 37] even for much simpler models. There are two ways to deal with the problem. One uses an exhaustive enumeration algorithm, such as branch and bound or dynamic programming. The other uses heuristic algorithms. Due to their large execution overheads, exhaustive enumeration algorithms are not applicable in dynamic situations. Hence, it is an important research issue to find good heuristic algorithms.

Two important performance aspects must be considered to evaluate a heuristic scheduling algorithm for hard real-time tasks. The first is the *ability* of an algorithm to generate a feasible schedule. This ability is commonly characterized by the mean behavior of the algorithm, which can be measured either by a probability model or by a simulation study. For simple systems, the probability model works well. For

complex systems, simulation is commonly used. In this chapter, we use simulation to measure the ability of a scheduling algorithm to find a feasible schedule. The metric used is *mean success ratio (SR)* defined in the following way. Given N task sets which are generated according to certain distributions where each task set is known to have at least one feasible schedule, let N_A be the total number of task sets for which algorithm A finds feasible schedules. Then

$$SR = \frac{N_A}{N}.$$

The second aspect is the *quality* of the feasible schedules generated by an algorithm. Quality may refer to either the mean behavior or the worst case behavior of the algorithm, where behavior is characterized by metrics such as *schedule length* and *earliness* (the time interval between a task's completion time and its deadline). In this chapter, we deal with the worst case behavior and use the schedule length metric. Schedule length is an important characteristic of a schedule for tasks in a real-time system for the following reasons:

- For real-time task scheduling, if one algorithm generates a shorter schedule than another algorithm, it means that, in general, the first algorithm can more easily accommodate tasks with shorter deadlines than the second algorithm.
- If the schedule length is shorter, it is a more efficient schedule and it is likely that there is more execution time remaining at the end of the schedule which can be used for executing other tasks, such as diagnostic tasks, or handling future tasks, in the case of dynamic arrivals.

We define a schedule length *bound* of one algorithm relative to an optimal algorithm in the following way. Let x be any instance of task sets to be scheduled, $L_A(x)$ be the schedule length of a scheduling algorithm A for instance x, and $L_O(x)$ be the schedule

length of an optimal scheduling algorithm[1] O for x. If there exists a constant B, such that

$$\frac{L_A(x)}{L_O(x)} \leq B, \qquad \text{for all } x,$$

then B is called the bound for algorithm A. B is *tight* if it can be reached asymptotically. We simply use $\frac{L_A}{L_O} \leq B$ to represent the above inequality if there is no ambiguity.

The remainder of this chapter is organized as follows. Section 4.2 describes our problem, task model, terminology, and assumptions. Two known scheduling approaches, list scheduling and the H scheduling algorithm, are discussed in Section 4.3. In Section 4.4, we present the heuristic algorithm, H_k, and analyze its schedule length bounds for both uniform tasks, i.e., tasks with the same computation time and non-uniform tasks, i.e., tasks with the arbitrary computation times. In Section 4.5, H_2, which is a special case of H_k, is compared via simulation studies with the two known algorithms: the H scheduling algorithm and list scheduling, with respect to the ability to find feasible schedules. The practical insights gained by the analysis are discussed in Section 4.6. We summarize the chapter in Section 4.7.

4.2 Task and Resource Models

The scheduling problem is to assign a set of real-time tasks to processors and additional resources, such that, all tasks meet their resource requirements and timing requirements. More formally, the problem is characterized by a processor-resource-task model given by $\{P_1, P_2, ..., P_m\}$, $\{R_1, R_2, ..., R_r\}$, and $\{T_1, T_2, ..., T_n\}$.

$\{P_1, P_2, ..., P_m\}$ is a set of m identical processors in a homogeneous multiprocessor system. Each processor is capable of executing any task.

$\{R_1, R_2, ..., R_r\}$ is a set of r resources such as data structures and buffers. In general, resources are discrete and each resource may have multiple instances. To

[1]The optimal algorithm produces feasible schedules with the minimum schedule length.

make the analysis easier, we assume that resources are *continuous and renewable.* A resource is continuous if a task can request any portion of it. A resource is renewable if its total amount is always fixed and the resources are not consumed by the tasks. Resources can be used by tasks in two different modes: when in *shared mode*, several tasks can use the resource simultaneously; when in *exclusive mode*, only one task can use it at a time.

$\{T_1, T_2, ..., T_n\}$ is a set of n tasks. Task T_i is characterized by the following:

- c_i — its worst case computation time,
- d_i — its deadline,
- $\vec{r_i} = (r_1, r_2, ..., r_r)$ — its resource requirement vector. Each r_j has two components: resource usage information and the amount requested. The resource usage information has one of three values: *not used, shared use*, and *exclusive use.* Besides these resource requirements, each task requires one processor.

We assume tasks are aperiodic, independent, nonpreemptable, and ready to start execution at the time of invocation of the scheduling algorithm. We also assume that the resources requested by a task are used throughout the task's execution time. If all tasks have the same computation time, we call them *uniform tasks*, otherwise they are *non-uniform tasks.*

With respect to schedule length, the above scheduling problem can be represented as $P|res\cdots, d_i|C_{max}$, by using the notation defined in [37], where P denotes that processors are identical and the number of processors is a free parameter, $res\cdots$ denotes that the number of resources, the resource sizes, and resource requirements are free parameters, d_i denotes that the task deadline is a free parameter, and C_{max} denotes that the performance criterion is to minimize the maximum completion time.

In general, scheduling tasks with these characteristics is a hard problem. If there are three processors, one resource, and a set of tasks with the same computation time

and deadline, it is NP-complete to find a feasible schedule [34] and it is NP-hard to find a minimum length feasible schedule [16]. Hence, only heuristic scheduling algorithms are considered here.

4.3 List Scheduling and the H Scheduling Algorithm

A very common heuristic algorithm used to determine feasible non-preemptive schedules on multiprocessors is list scheduling [25, 37]. In list scheduling, every task has a priority defined in some way (e.g., it can be based on deadline, value, resource needs, or some weighted formula that combines several or all these features). In list scheduling, tasks are arranged in some order on a list and when a processor is idle, the list is scanned from the beginning to the end, the first task which does not violate resource constraints is assigned to a free processor. We show later that while list scheduling has a good worst case bound on schedule length, it does not have good average case performance, because moving a low priority task up in the schedule when higher priority tasks are blocked by resource constraints may sometimes cause more important tasks to miss their deadlines.

Recently, another approach for determining feasible non-preemptive schedules on multiprocessors has been developed [70, 91]. This algorithm, called the H scheduling algorithm in this chapter, defines the priority of a task as

$$h(T_i) = d_i + W \cdot b_i \tag{4.1}$$

where $W(> 0)$ is the weight and b_i is the earliest time at which task T_i can be serviced with respect to the current partial schedule, (i.e., at b_i all the resources needed by T_i are available). (How to choose W is discussed in [70].) The smaller the value of $h(T_i)$, the higher its priority. The sophistication of this priority calculation is greater than is typically found in list scheduling (although nothing prevents a list scheduling

algorithm from using the same definition of priority). However, it is important to note that the H scheduling algorithm is *not* list scheduling, because given the priority calculation, the H scheduling algorithm simply selects the highest priority task to execute next. It does *not*, as list scheduling does, try to be greedy about the usage of processor resources by searching the entire remaining list of tasks to be scheduled to determine if any of them can be moved up and start to run in an earlier time slot. Hence, the H scheduling algorithm focuses on maintaining the priority order rather than being greedy about the usage of processor resources. This results in good average case performance with respect to meeting deadlines but a poor worst case schedule length bound.

Example 4.1: We consider the H scheduling algorithm reported in [91]. H uses the heuristic function (4.1) to assign priorities to tasks. Let $W = 6$. Whenever a task is added to the current partial schedule, H_1 re-computes the priority of all remaining tasks because their b_i's may change. Here we assume that the algorithm does not backtrack, and we only consider the worst case behavior of H with respect to schedule length.

Assume there are two processors, three tasks, and one resource which is used only in exclusive mode. A task can request use of a fraction of the resource. The task parameters are listed in Table 4.1.

Table 4.1 Task parameters for Example 4.1

task	T_1	T_2	T_3
computation time	9	10	1
resource request	0.5	0.5	1.0
deadline	9	74	11

H starts with an empty partial schedule. It computes the tasks' priority: $h(T_1) = 9, h(T_2) = 74, h(T_3) = 11$, because $b_i = 0, 1 \leq i \leq 3$. So T_1 is the highest priority task and it is added to the (empty) partial schedule. Then H

re-computes the priority for each remaining task: $h(T_2) = 74$, $h(T_3) = 65$, because $b_2 = 0$ and $b_3 = 9$. Hence, T_3 is the next task added to the current partial schedule. Now T_2 is the only unscheduled task and is appended to the current partial schedule. Thus, H_1 generates a schedule in the order

$$(T_1, T_3, T_2)$$

in which the three tasks are executed sequentially. The optimal schedule however has the following order

$$(T_1, T_2, T_3),$$

i.e., the first two tasks run in parallel. Thus, the schedule length ratio for this example is 20/11. While this example produces a ratio of 20/11, in the worst case, the schedule length bound for H is m (the number of processors) [89].

To further clarify the difference between the H scheduling algorithm and list scheduling, we consider a particular list scheduling algorithm which uses the same priority assignment as the one used in H_1. For the above task set, the first task scheduled by list scheduling is T_1, because it is the highest priority task among all tasks which can start at time zero. Then, because one processor is still idle at time zero, list scheduling tries to find a task that can start at time zero (this is the main difference between the H scheduling algorithm and list scheduling). It re-computes the priorities of the remaining two tasks. Although T_3 has the higher priority, only T_2 can start at time zero and so list scheduling chooses T_2. Finally, T_3 is scheduled. Hence, in this example, the final feasible schedule is the same as the optimal schedule mentioned above. In the worst case, the schedule length bound for list scheduling is the minimum of $(m + 1)/2$ and $r + 2 - (2r + 1)/m$ [32], where m is the number of processors and r is the number of resources.

In principle, given a heuristic function and a task set, the task priority list depends on the tasks' deadlines and resource needs. In performing a worst case analysis, a particular priority ordering can be produced to generate a particularly bad situation by adjusting the deadlines. In the rest of the chapter, we do not explicitly specify deadlines, since we know that theoretically a feasible deadline assignment exists that is consistent with any given priority ordering.

In this chapter we develop a variation of the H algorithm called H_k which combines features of both the H algorithm and list scheduling, such that, it performs well in finding feasible schedules and has a good schedule length bound.

4.4 A Generalized Heuristic Algorithm and Its Schedule Length Bound

We consider a generalized heuristic scheduling algorithm in this section. This algorithm is called H_k, where $2 \leq k \leq m$. It combines features from both list scheduling and the H scheduling algorithm. First, the H_k algorithm is presented, then its schedule length bounds are derived for both uniform tasks and non-uniform tasks.

4.4.1 H_k Scheduling Algorithm

H_k uses the same priority list as the H scheduling algorithm (H_1), which means that it uses the same heuristic function as in Equation (4.1), but tries to be greedy to a certain degree with respect to processor usage, just like list scheduling. Hence, H_k schedules by priority while attempting to keep at least k processors busy whenever possible. If it is not possible to keep k processors busy, then it will keep as many processors busy as possible.

The H_k algorithm maintains a variable called t_{c_k} which divides a partial schedule into two portions: one portion is before t_{c_k} and the other is after t_{c_k}. The first portion satisfies the property that, in each sub-interval of this portion, either at least

k processors are busy or less than k processors are busy; for those intervals where there are less than k processors busy there will be a resource conflict if another task is added. Formally, t_{c_k} is set to the maximum possible value that satisfies the following: In any sub-interval $[x, y]$ of $[0, t_{c_k})$, (1) at least k processors are busy, or (2) less than k processors are busy, but no other tasks can be scheduled in $[x, y]$ because of resource conflicts. Only in case (1) can a new task be added to the partial schedule which completes before t_{c_k}. Therefore, the H_k algorithm simply applies the highest priority first rule to schedule a task which can fit into the partial schedule before t_{c_k}. The pseudo code of the H_k scheduling algorithm is presented in Figure 4.1.

Let n be the number of tasks, r be the number of resources, and m be the number of processors. The time complexity of H_k is $O(n^{k+1}r)$, where $2 < k \leq m$, because, if all processors are idle at t_{c_k}, the algorithm may need to find up to k tasks from S_2 which defined in Figure 4.1. This will require examination of up to $\sum_{i=1}^{k} C_{n-1}^{i}$ subsets[2], where k is bounded by m. Each task can request up to r resource types and their availability has to be considered in constructing the schedule. So the time complexity is $O(n^k r)$. The algorithm may loop n times. This leads to an $O(n^{k+1}r)$ complexity. However, the time complexity of H_2 is $O(n^2 r)$: At time t_{c_2}, either all processors are idle or there is only one processor busy (by the definition of t_{c_2}). If all processors are idle, the if-condition (all processors are idle at t_{c_k} and $k = 2$) is satisfied and one task will be selected. If there is only one processor busy, $k' = 1$ and only one task will be selected. So the complexity for each round is $O(nr)$. The algorithm may loop n times, so the complexity of H_2 is $O(n^2 r)$.

Example 4.2: Let us consider H_2 and H_3 and show how they differ. Assume that there are three processors, two resources, which are used only in exclusive

---

[2] $$C_{n-1}^{i} = \frac{i!}{(n-i)!\,i!},$$ which is the total number of combinations of n objects taken at i at a time. The complexity of C_{n-1}^{i} is $O(n^i)$.

Initialize current partial schedule to null;
Consider all tasks unscheduled;
While (more tasks remain to be scheduled) do

Compute tasks' priorities by a function $H(T_i) = d_i + W \cdot b_i$,

where W is an adjustable weight and b_i is the earliest time at which task T_i can be serviced [70, 91]; (The lower the value of h, the higher the priority.)

Compute t_{c_k} for the current partial schedule;

Partition unscheduled tasks into three sets:

$$\begin{aligned} S_1 &= \{T_i : b_i + c_i \leq t_{c_k}\}, \\ S_2 &= \{T_i : (b_i \leq t_{c_k}) \& (b_i + c_i > t_{c_k})\}, \\ S_3 &= \{T_i : b_i > t_{c_k}\}, \end{aligned}$$

where S_1 is the set of tasks which can complete before t_{c_k}; S_2 is the set of tasks which can start at or before t_{c_k} but complete after t_{c_k}, and S_3 is the set of tasks which can start only after t_{c_k}; (S_1 and S_2 are not empty at the same time by the definition of t_{c_k}.)

Sort S_1 and S_2 by priority;
If (S_1 is not empty)

then select a task with the highest priority and add it to the current partial schedule;

Else if (all processors are idle at t_{c_k} and $k == 2$)

then add a task with the highest priority in S_2 to the current partial schedule;

Else

Let k' be the number of processors busy at t_{c_k}. Select the first subset of tasks of S_2 from the following order, such that, all tasks in the subset can start at their earliest start times, and add the subset to the current partial schedule:

lexicographically ordered $(k - k')$-subsets (i.e., subsets of size $(k - k')$) of tasks,

lexicographically ordered $(k - k' - 1)$-subsets of tasks,
.....
lexicographically ordered 1-subsets of tasks.

(Here the lexicographic order is just the "dictionary" order according to task priority. Such an order can be generated using the algorithm in [31, Page 31];)

Check the current partial schedule for feasibility;

If (the current partial schedule is not feasible)
then abort the algorithm;

end of while;

Figure 4.1 H_k algorithm

mode, and five tasks. We use the heuristic function (4.1) to determine the priority. The task parameters are shown in Table 4.2.

Table 4.2 Task parameters for Example 4.2

task	T_1	T_2	T_3	T_4	T_5
comp. time	10	10	10	20	20
request for R_1	0.2	0.4	0.4	0.8	0.2
request for R_2	0.2	0.4	0.4	0.2	0.8
deadline	10	40	40	30	35

Let us consider H_2 first. In the beginning, $t_{c_2} = 0$ and $S_2 = \{T_1, T_2, T_3, T_4, T_5\}$. H_2 schedules T_1 first because T_1 has the highest priority among the five tasks. Then each one of the remaining four tasks can be chosen to run in parallel with T_1, and H_2 chooses T_4, because it has the highest priority among them. Now $t_{c_2} = 10$, $S_2 = \{T_5\}$, and $S_3 = \{T_2, T_3\}$. H_2 schedules T_5 and it starts at 10. Next, t_{c_2} is 30, because no task can start before 30, and $S_2 = \{T_2, T_3\}$. So H_2 schedules T_2 first and then it schedules T_3, both tasks start at 30. The schedule length is thus 40 and the final schedule is:

$$(T_1, T_4, T_5, T_2, T_3).$$

The schedule for H_3 is simple. Let us consider the different subsets of tasks in lexicographic ordering — given their priority order: $(T_1, T_4, T_5, T_2, T_3)$. We are seeking the largest subset up to size $(k - k')$. Here $k = 3$ and at $t_{c_3} = 0$, $k' = 0$. T_1, T_2, T_3 is this subset. H_3 schedules them first and they all start at 0. Then $t_{c_3} = 10$ and $S_2 = \{T_4, T_5\}$. So H_2 schedules T_4 and T_5 next and they start at 10. The final schedule length is 30 and the final task order is:

$$(T_1, T_2, T_3; T_4, T_5).$$

4.4.2 Analysis of the H_k Algorithm for Uniform Tasks

The problem of scheduling uniform tasks with additional resources is similar to a generalized bin packing problem studied in [33]. The difference between two problems is that the number of "tasks" is unbounded in the bin packing problem, but it is bounded by $m \cdot L_O$ in the multiprocessor scheduling problem, where m is number of processors and L_O is the optimal schedule length. Let us look at an example first.

Example 4.3: Let $m = 4$, $k = 3$, and $r = 4$. Let the resources be: R_1, R_2, R_3, and R_4, which are used only in exclusive mode. There are 24 tasks labeled as follows:

$$\{T_{i,j} : 1 \leq i \leq 4, 1 \leq j \leq 6 \}$$

All tasks request one unit of computation time. For $1 \leq j \leq 6$,

each $T_{1,j}$ requests 1/3 of R_1, 2/9 of R_2, 1/6 of R_3, and 1/9 of R_4;

each $T_{2,j}$ requests 2/9 of R_1, 1/3 of R_2, 1/6 of R_3, and 1/9 of R_4;

each $T_{3,j}$ requests 2/9 of R_1, 2/9 of R_2, 1/2 of R_3, and 1/9 of R_4;

each $T_{4,j}$ requests 2/9 of R_1, 2/9 of R_2, 1/6 of R_3, and 2/3 of R_4.

The deadlines of the tasks are such that H_3 schedules the tasks in the following order:

$$(T_{1,1}, T_{1,2}, \cdots, T_{1,6}; T_{2,1}, T_{2,2}, \cdots, T_{2,6}; \cdots; T_{4,1}, T_{4,2}, \cdots, T_{4,6}).$$

In total, the T(1,i's) take 2 units of time. The same is true of the $T_{2,i}$'s. These together take 4 units of time. Because there is no other subset of three tasks left in the remaining unscheduled tasks such that the subset can be scheduled in parallel, $T_{3,j}$'s are scheduled next, 2 tasks in parallel each time. This takes 3 units. Finally, $T_{4,j}$'s can only be scheduled in sequential order. This takes 6 units. Thus the length of the schedule generated by the H_3 algorithm is 13 time units.

The optimal algorithm uses the following order:

$$(T_{1,1}, T_{2,1}, T_{3,1}, T_{4,1}; T_{1,2}, T_{2,2}, T_{3,2}, T_{4,2}; \cdots; T_{1,6}, T_{2,6}, T_{3,6}, T_{4,6}).$$

The length of the schedule generated by an optimal scheduling algorithm is 6 time units, since it is able to keep all 4 processors busy all the time. Thus we have:

$$\frac{L_{H_3}}{L_O} = \frac{13}{6}.$$

This is the worst case schedule length indicated by Theorem 4.1 below.

Before we state and prove Theorem 4.1, certain lemmas are in order.

For any feasible schedule generated by the H_k algorithm, we define:
$\mathcal{B}_i$ = {all time intervals with i processors busy} and
I_i = the sum of the length of all time intervals in $\mathcal{B}_i$,
where $1 \leq i \leq m$. In our bound analysis, we add the length of each I_i such that $\sum_{i=1}^{m} I_i$ reaches the maximum. The bound will depend on m, k and r. We assume that there are enough non-processor resources to construct a worst-case situation, because it gives us the maximum schedule length bound. The analysis for other cases is reported in [89].

Lemma 4.1: Under the uniform task model, any feasible schedule generated by the H_k algorithm has the following property:

$$\sum_{i=1}^{j} \frac{i}{j} I_i \leq L_O$$

where $j < k$.

Proof: Assume that all tasks have unit computation time. Examining the number of tasks in $\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_j$, where $2 \leq j < k$, we have $\sum_{i=1}^{j} i \cdot I_i \leq j \cdot L_O$.

Suppose this is not true. Then in the sets $\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_j$, the number of tasks is more than $j \cdot L_O$. Apply the *pigeonhole principle*, which says, for $j \cdot L_O + 1$ balls

and L_O boxes, there must exist one box with $j+1$ or more balls. This means that there must exist a subset of tasks with $j+1$ or more tasks which can be executed in parallel without any resource conflict. From the pseudo code for the H_k algorithm we see that it would have scheduled the subset in one of the time interval sets $\mathcal{B}_{j+1}$, $\mathcal{B}_{j+2}$, $\cdots$, $\mathcal{B}_m$ and not in $\mathcal{B}_1$, $\mathcal{B}_2$, $\cdots$, $\mathcal{B}_j$. Therefore, this subset of tasks would not be in the sets $\mathcal{B}_1$, $\mathcal{B}_2$, $\cdots$, $\mathcal{B}_j$. This is a contradiction.

The lemma is derived by dividing both sides of the above inequality by j. □

Using the above result, the following lemma can be derived, which gives us a bound for H_k.

Lemma 4.2: For uniform tasks,

$$\frac{L_{H_k}}{L_O} \leq \frac{m}{k} + \sum_{j=2}^{k} \frac{1}{j}$$

where m is the number of processors and $2 \leq k \leq m$.

Note that for $k = 2$, the above bound is $(m+1)/2$.

Proof: Partition any schedule of H_k into $\mathcal{B}_1$, $\mathcal{B}_2$, $\cdots$, $\mathcal{B}_m$, and use I_1, I_2, $\cdots$, I_m to represent their lengths as defined before. We have

$$\begin{aligned} \sum_{i=1}^{k-1} i \cdot I_i + k \cdot \sum_{i=k}^{m} I_i \;&\leq\; \sum_{i=1}^{k-1} i \cdot I_i + \sum_{i=k}^{m} i \cdot I_i \leq \sum_{i=1}^{m} i \cdot I_i \\ &=\; \sum_{i=1}^{n} c_i \leq m \cdot L_O \end{aligned}$$

where n is the number of tasks, c_i is the computation time of task T_i, which is same for all tasks. Noticing that $\sum_{i=k}^{m} I_i = L_{H_k} - \sum_{i=1}^{k-1} I_i$, we have

$$\begin{aligned} (\sum_{i=1}^{k-1} i \cdot I_i) + k \cdot \sum_{i=k}^{m} I_i \;&=\; (\sum_{i=1}^{k-1} i \cdot I_i) + k(L_{H_k} - \sum_{i=1}^{k-1} I_i) \\ &\leq\; m \cdot L_O. \end{aligned}$$

Rewriting the above inequality:

$$k \cdot L_{H_k} \leq m \cdot L_O + \sum_{i=1}^{k-1}(k-i) \cdot I_i.$$

We first analyze the term $\sum_{i=1}^{k-1}(k-i) \cdot I_i$:

$$\sum_{i=1}^{k-1}(k-i) \cdot I_i = \sum_{i=1}^{k-1} i \cdot k \cdot (\frac{1}{i} - \frac{1}{k}) \cdot I_i$$

Since

$$(\frac{1}{i} - \frac{1}{k}) = \sum_{j=i}^{k-1} \frac{1}{j \cdot (j+1)},$$

$$\begin{aligned}
\sum_{i=1}^{k-1} i \cdot k \cdot (\frac{1}{i} - \frac{1}{k}) \cdot I_i &= \sum_{i=1}^{k-1}\sum_{j=i}^{k-1} \frac{i \cdot k}{j \cdot (j+1)} \cdot I_i \\
&= \sum_{j=1}^{k-1}\sum_{i=1}^{j} \frac{i \cdot k}{j \cdot (j+1)} \cdot I_i \\
&= \sum_{j=1}^{k-1} \frac{k}{j+1} \cdot (\sum_{i=1}^{j}(\frac{i}{j}) \cdot I_i) \\
&\leq \sum_{j=1}^{k-1} \frac{k}{j+1} \cdot L_O, \text{ by lemma 4.1.}
\end{aligned}$$

So,

$$k \cdot L_{H_k} \leq m \cdot L_O + \sum_{j=1}^{k-1} \frac{k}{j+1} \cdot L_O$$

By rewriting the above inequality and changing the index bound, the lemma is derived. □

To show that the above bound is tight, we need the following lemma.

Lemma 4.3: For the H_k scheduling algorithm, there exists a case where, with uniform tasks, the schedule length bound of

$$\frac{m}{k} + \sum_{j=2}^{k} \frac{1}{j}$$

can be reached. m is the number of processors, and $2 \leq k \leq m$.

Proof: For any given m, we can construct a case such that the bound is reached. Consider a set of $(d * m)$ tasks and m additional resources, where $d = LCM(2, 3, \cdots, k)$, ($LCM()$ is the least common multiple function). The tasks are

$$T_{1,1}, T_{1,2}, \cdots, T_{1,d}; T_{2,1}, T_{2,2}, \cdots, T_{2,d}; \cdots; T_{m,1}, T_{m,2}, \cdots, T_{m,d},$$

and the m resources are

$$R_1, R_2, \cdots, R_m,$$

which are used only in exclusive mode. Let

$$\{a_1, a_2, \cdots, a_k, a_{k+1}, \cdots, a_m\} = \{\frac{1}{k}, \frac{1}{k}, \cdots, \frac{1}{k}, \frac{1}{k-1}, \cdots, \frac{1}{3}, \frac{1}{2}, 1-\epsilon\}.$$

Each $T_{i,j}$ requests a_i of R_i and the remaining R_i is uniformly divided among $T_{l,j}$, where $1 \leq l \leq m$, $l \neq i$, and $1 \leq j \leq d$.

Suppose the task deadlines are set[3], such that, H_k generates a feasible schedule and selects the tasks in the following order:

$$(T_{1,1}, T_{1,2}, \cdots, T_{1,d}; T_{2,1}, T_{2,2}, \cdots, T_{2,d}; \cdots; T_{m,1}, T_{m,2}, \cdots, T_{m,d}). \qquad (4.2)$$

The optimal algorithm gives the following schedule:

$$(T_{1,1}, T_{2,1}, \cdots, T_{m,1}; T_{1,2}, T_{2,2}, \cdots, T_{m,2}; \cdots; T_{1,d}, T_{2,d}, \cdots, T_{m,d}). \qquad (4.3)$$

Let us compute the schedule length for H_k and the optimal algorithm. Let list (4.2) be divided evenly into m segments of d tasks each. Each of the first $(m - k + 1)$ segments of tasks is scheduled by H_k keeping k processors busy. The schedule length for these segments is $(m - k + 1) * d/k$. The i-th segment of the remaining $(k - 1)$

[3]See the penultimate paragraph in Section 4.2.

segments is scheduled by H_k with a length of $d/(k-i)$ by keeping $(k-i)$ processors busy. Hence the length of the schedule formed by the H_k algorithm is:

$$L_{H_k} = \frac{(m-k+1)\cdot d}{k} + \frac{d}{k-1} + \frac{d}{k-2} + \cdots + \frac{d}{2} + \frac{d}{1}.$$

If list (4.3) is divided evenly into d segments of m tasks each, then each segment can fit in one time unit and the length of the schedule generated by the optimal algorithm is $L_O = d$. So the bound is reached, because

$$\begin{aligned} \frac{L_{H_k}}{L_O} &= \frac{m-k+1}{k} + \frac{1}{k-1} + \frac{1}{k-2} + \cdots + \frac{1}{2} + 1 \\ &= \frac{m}{k} + \sum_{j=2}^{k} \frac{1}{j}. \end{aligned}$$

□

By combining lemmas 4.2 and 4.3, we have the following theorem.

Theorem 4.1: If all tasks have the same computation time, then the tight bound for H_k is

$$\frac{m}{k} + \sum_{j=2}^{k} \frac{1}{j},$$

where m is the number of processors and $2 \leq k \leq m$.

For any given m, the above theorem gives us a sequence of bounds for different values of k. So, if a scheduler is willing to spend more time, a better bound can be guaranteed. In one extreme, let $k = m$. H_m has a bound of:

$$\sum_{j=1}^{m} \frac{1}{j} \approx \ln m + \gamma$$

where $\gamma = 0.57721\cdots$ is Euler's constant.

4.4.3 Analysis of the H_k Algorithm for Non-Uniform Tasks

Lemma 4.4: For non-uniform tasks,

$$\frac{L_{H_k}}{L_O} \leq \frac{m+1}{2}.$$

where m is the number of processors and $2 \leq k \leq m$.

Proof: For the schedule generated by H_k, let $\mathcal{B}_i$ and I_i be as defined in the previous subsection. Because I_1 accounts for one processor being busy and the remaining I_i account for at least two processors being busy,

$$I_1 + 2 \cdot \sum_{i=2}^{m} I_i \;\leq\; \sum_{i=1}^{n} c_i.$$

The schedule length of H_k is L_{H_k}. So $\sum_{i=2}^{m} I_i = L_{H_k} - I_1$. Using an argument similar to the one used in the proof of lemma 4.1, it can be shown that $I_1 \leq L_O$. Since, $\sum_{i=1}^{n} c_i \leq m \cdot L_O$,

$$I_1 + 2 \cdot (L_{H_k} - I_1) \;\leq\; \sum_{i=1}^{n} c_i \leq m \cdot L_O.$$

Rewriting the above inequality:

$$\begin{aligned} 2 \cdot L_{H_k} - I_1 &\leq m \cdot L_O, \\ 2 \cdot L_{H_k} - L_O &\leq m \cdot L_O, \\ 2 \cdot L_{H_k} &\leq (m+1) \cdot L_O, \\ \frac{L_{H_k}}{L_O} &\leq \frac{m+1}{2}. \end{aligned}$$

□

Lemma 4.5: For the H_k scheduling algorithm, there exists a case where, with non-uniform tasks, the schedule length bound of $\frac{m+1}{2}$ can be reached. m is the number of processors, and $2 \leq k \leq m$.

Proof: For any given m, we can construct a case such that the bound is reached. Let the tasks be

$$\{T_i, T_i' : 1 \leq i \leq m-1\} \cup \{T''\} \cup \{T_i^j : 1 \leq i \leq m-1,\ 1 \leq j \leq k-2\}$$

and the resources be

$$\{R_1, R_2, ..., R_m\},$$

which are used only in exclusive mode. Further let C be a constant. Each T_i and T_i' has the same computation time C and requests $1/2$ of R_i and ϵ of all other resources,

where $1 \leq i \leq m-1$. T'' has a computation time of $2C$ and needs $2/3$ of R_m and ϵ of all other resources. Each T_i^j has $c_i^j = \epsilon$, and requests ϵ of every resource except R_i, where $1 \leq i \leq m-1$, $1 \leq j \leq k-2$. These small tasks are used for running in parallel with two large tasks in the schedule generated by the H_k algorithm to keep k processors busy.

Suppose the task deadlines are set[4], such that, H_k generates a feasible schedule and selects the tasks in the following order:

$$(T_1, T_1', T_1^1, T_1^2, ..., T_1^{k-2}, T_2, T_2', T_2^1, T_2^2, ..., T_2^{k-2},$$

$$..., T_{m-1}, T_{m-1}', T_{m-1}^1, T_{m-1}^2, ..., T_{m-1}^{k-2}, T'').$$

The optimal algorithm selects tasks from in the following order:

$$(T'', T_1, T_2, ..., T_{m-1}, T_1', T_2', ..., T_{m-1}', T_1^1, T_1^2, ..., T_1^{k-2},$$

$$T_2^1, T_2^2, ..., T_2^{k-2}, ..., T_m^1, T_m^2, ..., T_m^{k-2}).$$

In the optimal schedule, all T_i tasks are scheduled to run in parallel and all T_i' tasks are also scheduled to execute in parallel; both are scheduled to run in parallel with T''. All the remaining tasks can be scheduled within a time interval $(m-1) \cdot \epsilon$. Thus the schedule length for the optimal schedule is $L_O = 2 \cdot C + (m-1) \cdot \epsilon$. In the schedule generated by the H_k algorithm, the two large tasks, T_i and T_i', are scheduled to execute in parallel with $(k-2)$ small tasks, T_i^j, where $1 \leq j \leq k-2$ and $1 \leq i \leq m-1$. Then T'' follows because it is blocked by every pair of T_i and T_i'. So its schedule length is $L_{H_2} = (m-1) \cdot C + 2 \cdot C$.

We have

$$\frac{L_{H_k}}{L_O} = \frac{(m-1) \cdot C + 2 \cdot C}{2 \cdot C + (m-1) \cdot \epsilon} \rightarrow \frac{m+1}{2}, \; as \; \epsilon \rightarrow 0. \; \square$$

Lemmas 4.4 and 4.5 lead to the following theorem.

[4]See the penultimate paragraph in Section 4.2.

Theorem 4.2: If tasks have arbitrary computation times, then the tight bound for H_k is $(m+1)/2$, where m is the number of processors and $2 \leq k \leq m$.

4.5 Comparing H_2, H and List Scheduling with respect to Success Ratio

The time complexity of H_k is relatively high when $k > 2$. Only H_2 has a time complexity comparable to the H scheduling algorithm and list scheduling (both are $O(n^2r)$, where n is the number of tasks and r is the number of resources). Also, for non-uniform tasks, the schedule length bound does not improve when $k > 2$. Therefore, we mainly focus on H_2 in this section. In particular, we use simulation to evaluate the performance given by the success ratios of the three algorithms: H_2, the H scheduling algorithm and list scheduling, where the H scheduling algorithm is based on [70] and the list scheduling algorithm applies the earliest deadline first rule to construct its priority list. Clearly, what we are striving for is a scheduling algorithm that is able to find a feasible schedule for a set of tasks, if such a schedule exists. Obviously, a heuristic algorithm cannot be guaranteed to achieve this. However, one heuristic algorithm can be considered better than another, if given a number of task sets for which feasible schedules exist, the former is able to find feasible schedules for more task sets than the latter. This is the basis for our simulation study. Ideally, we would like to come up with a number of task sets, each of which is known to have a feasible schedule. Unfortunately, given an arbitrary task set, only an exhaustive search can reveal whether the tasks in this task set can be feasibly scheduled.

Therefore we take a different approach in our study here. We use the task generator which is used in [70], which can generate schedulable task sets where the number of tasks in a task set can be very large. Also the tasks are generated to guarantee the (almost) total utilization of the processors. The schedule generated by the task generator is used only for the purpose of generating a feasible set of tasks which

is then input to the scheduling algorithm, i. e., the scheduling algorithms have no knowledge of the schedule itself but are only given the tasks and their requirements. The following are the parameters used to generate the task set:

- Probability that a task uses a resource, Use_P.
- Probability that a task uses a resource in shared mode, Share_P.
- The minimum computation time of tasks, Min_C.
- The maximum computation time of tasks, Max_C.
- The schedule length, L, which controls the length of a simulation.

The schedule generated by this task set generator is in the form of a matrix which has $m + r$ columns and L rows. The first m columns represent m processors. The remaining columns represent r resource types with one instance for each resource type. Each row represents a time unit. The task set generator starts with an empty matrix, then generates a task by selecting one of the m processors with the earliest available time and then requests the r resources according to the probabilities specified in the generation parameters. The generated task's processing time is randomly chosen using a uniform distribution between the minimum processing time and the maximum processing time. The task set generator marks on the matrix that the processor and resources required by the task are used for a number of time units equal to the task's computation time starting from the aforementioned earliest available time of the processor. The task set generator generates tasks until the remaining unused time units of each processor, up to L, is smaller than the minimum processing time of a task, which means that no more tasks can be generated to use the processors. Then, the largest finish time of a generated task becomes the task set's *shortest completion time*, SC. As a result, we generate tasks according to a very tight schedule without leaving any usable time units on the m processors between 0 and SC. However, there may be some empty time units in the r resources. So the generated matrix can be

approximated to be the schedule generated by an optimal scheduling algorithm with respect to the feasibility and the schedule length.

So far we have discussed how task resource requirements and computation times are determined. The issue of choosing task deadlines without any bias is addressed now. In order to exercise the scheduling algorithms in scenarios that have different levels of scheduling difficulty, we choose the deadline of a task in the task set randomly between $(1 + R) \cdot f_i$ and $(1 + R) \cdot SC$, where f_i is the task's completion time in the above matrix and R (the Relaxation Factor) is a simulation parameter indicating the tightness of the deadlines. As we increase the value of R, it is not difficult to see that the scheduler has a better and better chance to guarantee a task set. The simulation program uses 10 seeds to generate task sets. The number of processors is 5 and the number of other types of resources is 12. There is one instance for each type of resource. A task's computation time is randomly chosen between Min_C (10) and Max_C (40). There are two parameters to control task's resource requests. One is resource use probability, Use_P, which may vary from 0.1 to 0.7. The other is shared probability, Shared_P, set to 0.5. There are about 40 tasks in each task set. Our requirement on the statistical data is to generate 95% confidence intervals for the success ratio whose half width is less than 6% of the point estimate of the success ratio.

Figure 4.2 shows the effect of R on the success ratio. The performance of the H scheduling algorithm and H_2 is much better than the performance of list scheduling, e.g., at $R = 0.2$ and Use_$P = 0.3$, SR increases from 57.6% for list scheduling to about 80.1% for the H scheduling algorithm and H_2, and at $R = 0.2$ and Use_$P = 0.7$, SR increases from 24.8% for list scheduling to 57.3% for the H scheduling algorithm and H_2. Figure 4.3 shows the effect of resource contention by changing the probability of task's resource request. Again, the performance of the H scheduling algorithm and H_2 are close and far better than the performance of list scheduling, e.g., when

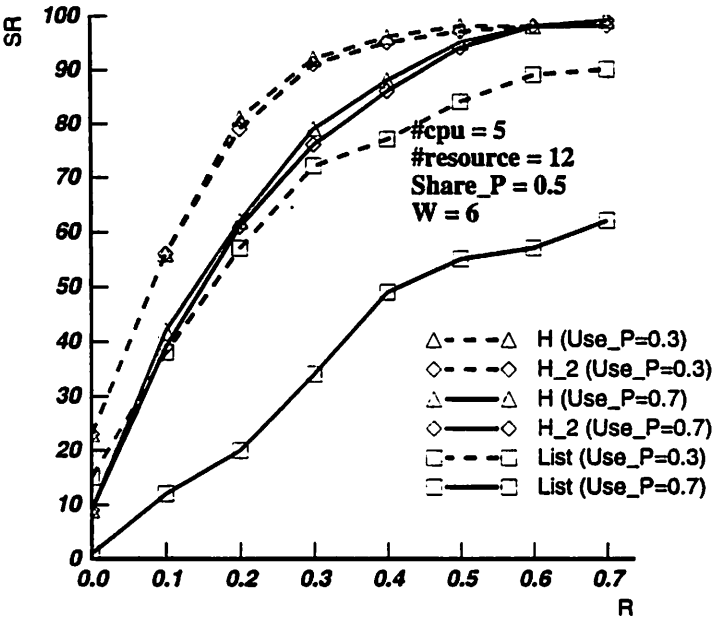

Figure 4.2 Effect of R on three algorithms

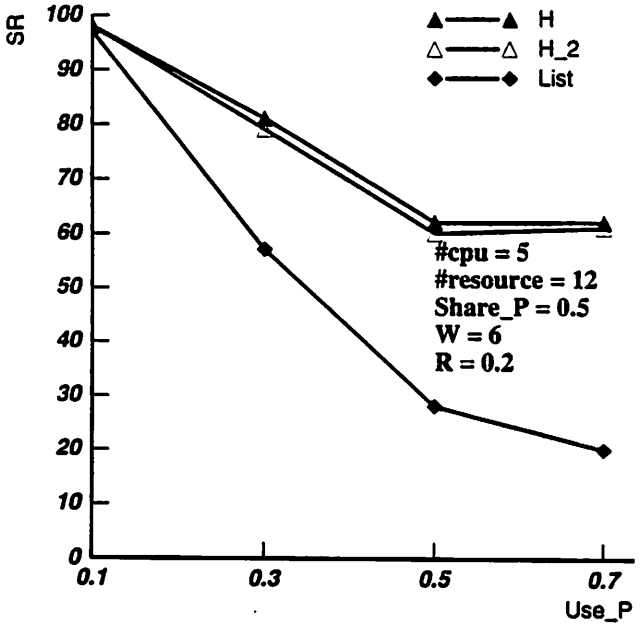

Figure 4.3 Effect of resource contention on SR

Use_$P = 0.5$, SR increases from 30.0% for list scheduling to 63.0% for the H scheduling algorithm and H_2. In both figures, the differences between the H scheduling algorithm and H_2 are within the 6% of confidence interval. These show that whereas the H_2 algorithm has performance close to the H scheduling algorithm, it is much better than the deadline-driven list scheduling algorithm.

4.6 Insights Gained from the Analysis

An initial goal of this work was to analyze the performance characteristics of the H scheduling algorithm, especially its schedule length bound. The fact that this analysis led us to other algorithms, namely H_2 and then H_k, indicates that a number of practical alternatives to the H scheduling algorithm were discovered primarily because we undertook the analysis. In this section we consolidate the practical insights gained from the analysis work.

Analysis showed that in the worst case, the length of the schedules produced by the H scheduling algorithm will be m times of the length of an optimal schedule [89], although on the average, the H scheduling algorithm is highly successful in finding feasible schedules [89, 91]. On the other hand, list scheduling has a much better worst case schedule length bound, almost half of the worst case schedule length bound of the H scheduling algorithm. But list scheduling is not as successful in finding feasible schedules.

The two algorithms differ mainly in the task selections during the scheduling process. A task's $h()$ value consolidates the task's deadline, and resource requirement information. The H scheduling algorithm selects a task with the minimum $h()$ value among all unscheduled tasks, while list scheduling selects a task with the minimum $h()$ value among the unscheduled tasks which have the same earliest start time. Thus, in list scheduling the task's deadline is only a secondary factor. H_2 uses a task selection procedure that has elements from both the H scheduling algorithm and list

scheduling. It selects a task to keep at least two processors busy whenever possible, otherwise its behavior is similar to the H scheduling algorithm. This change was motivated by the the fact that, in general, list scheduling keeps more processors busy at a given time than the H scheduling algorithm. This selection procedure of H_2 helps produce a better worst-case schedule length bound for H_2 compared to the H scheduling algorithm. At the same time, it has almost the same success as the H scheduling algorithm in finding feasible schedules.

Can the bound of H_2 be reduced further by increasing the (polynomial time) complexity of the scheduling algorithm? The development of H_k was motivated by this question. H_k is a generalized version of H_2 which keeps at least k processors busy whenever possible. It turns out that the added time complexity of H_k does not help to reduce the bound $((m+1)/2)$. The reason is that, in the worst case, there may be some tasks with very small computation times which contribute significantly to the number of tasks running in parallel at scheduling decision points, but they do not contribute much to the schedule length. If the ratio between the shortest task and the longest task in a task set is closer to 1, it is likely that the bound of H_k will be smaller than $(m+1)/2$. In the extreme case, the ratio is one, which corresponds to the uniform task model. The bound of H_k then reduces from $(m+1)/2$ to $m/k+\sum_{j=2}^{k} 1/j$ as stated in Theorem 4.1. In practice, it is also very important to choose a proper k to balance the time overhead, which increases with k, and the achieved bound.

One topic for future research is the characterization of the worst-case task characteristics that produce the above bounds. This will help avoid the pessimistic bounds.

4.7 Conclusions

In this chapter we discussed the performance of heuristic algorithms for hard real-time scheduling. Both the ability to generate feasible schedules and the quality of the generated feasible schedules, expressed in terms of the schedule length bound,

are important performance criteria. We studied the heuristic scheduling algorithm H_k which attempts to keep k processors busy if possible. For tasks with the same computation time, we showed that the bound is $\frac{m}{k} + \sum_{j=2}^{k} \frac{1}{j}$, where $2 \leq k \leq m$ and m is the number of processors. For tasks with arbitrary computation times, the bound is $(m+1)/2$, where $2 \leq k \leq m$. In both cases, the complexity is $O(n^{k+1}r)$ for $2 < k \leq m$, and $O(n^2r)$ for $k = 2$, where n is the number of tasks and r is the number of resources. Simulation studies of three algorithms, H_2, the H scheduling algorithm, and list scheduling, are reported. Results show that the H_2 algorithm has almost the same mean behavior in the ability to find feasible schedules as the H scheduling algorithm, and both are much better than list scheduling.

CHAPTER 5

PERFORMANCE-RELATED RELIABILITY ANALYSIS FOR DETERMINING TASKS' REDUNDANCY LEVELS

5.1 Introduction

In real-time systems, static or dynamic schedulability analysis is used to *guarantee* that tasks will meet their time constraints. A task is guaranteed subject to a set of assumptions, for example, about its worst case execution time and the nature of faults in the system. If these assumptions hold, once a task is guaranteed it can be assumed to meet its timing requirements. Thus, the probability of a task's successful completion is affected by the probability with which the assumptions hold. In this chapter, we focus on the fault-tolerance assumptions about the tasks in systems where decisions are made statically, i.e., at system design time, even though the approach developed in this chapter can be tailored to apply to systems that perform dynamic schedulability analysis.

Consider a system with m processors and n tasks. We must decide what level of redundancy should be assigned to the tasks such that both reliability and performance requirements are met. In particular, suppose a task T_i provides a reward V_i if it completes successfully once it is guaranteed, a penalty P_i if it fails after being guaranteed, and a penalty Q_i if it is not guaranteed. Let R_i be the reliability of a guaranteed task T_i and F_i be its failure probability, with $R_i = 1 - F_i$. R_i is mainly affected by the redundancy level for a task T_i and the failure model for the processors. Then, we define a performance index for the system such that it takes

the tasks' penalties, rewards, and reliabilities into account. The performance index PI_i for task T_i is defined as

$$PI_i = \begin{cases} V_i R_i - P_i F_i & \text{if } T_i \text{ is guaranteed} \\ -Q_i & \text{if } T_i \text{ is not guaranteed.} \end{cases}$$

The performance index PI for the task set is defined as

$$PI = \sum_{i=1}^{n} PI_i.$$

Thus PI accounts for both performance requirements and reliability requirements of real-time tasks. It provides us a base for achieving a mathematical assessment of performance-reliability tradeoffs.

Given this definition of performance index, we need to know the reliability for each scheduled task. Tasks' reliabilities are affected by faults in both software and hardware. Many fault-tolerance structures have been developed to tolerate these two types of faults. For example, *task replication* and *N-modular-redundancy* are commonly used to tolerate hardware faults, and *recovery block* and *N-version programming* are commonly used to tolerate software faults. The quantitative models for hardware reliability are well established [21], while the quantitative models for software reliability are still not fully understood. So we focus on hardware faults and the related quantitative reliability models. Furthermore, we only consider task replication as the fault-tolerance approach.

If we increase the redundancy level for T_i, we can increase its probability R_i of completing before its deadline, that is, decrease the probability that the task will fail after being guaranteed. But this will reduce the number of tasks that get guaranteed in the first place and so will increase the penalties due to task rejection. Thus, clearly, there are tradeoffs involved between the fault tolerance of the system, the rewards provided by guaranteed tasks that complete successfully, and the penalties due to tasks that fail after being guaranteed or that fail to be guaranteed. Therefore, some way needs to be found that maximizes rewards while minimizing penalties.

We use the term *task configuration* or *configuring tasks* to refer to the problem of determining a task's redundancy level. Once a task redundancy level is determined, a task is said to be guaranteed if the given number of replicas of the task are all scheduled to complete before the task's deadline. The question we address in this chapter is the following: What should be the level of redundancy associated with each task so that the system's performance, given by PI, is maximized, i.e., how should a task redundancy level be determined for optimal performance?

The remainder of the chapter is organized as follows. Section 5.2 presents notations, assumptions, and the system model. We derive the optimal task configuration strategy for a continuous model in Section 5.3. We discuss how to deal with a discrete model with tasks having different computation times in Section 5.4. In Section 5.5 we discuss the effects of using integer functions to approximate the optimal task configuration function which is a real valued function. In Section 5.6, we consider the task configuration strategy with tasks having different reward/penalty parameters. In Section 5.7, we discuss how to apply the task configuration theory in practice. We conclude the chapter in Section 5.8.

5.2 System Model and Assumptions

In this section, we present the processor-task model first, followed by a discussion of task configuration and task scheduling.

Formally, the problem is characterized by a processor-task model given by $\{P_1, P_2, ..., P_m\}$ and $\{T_1, T_2, ..., T_n\}$.

$\{P_1, P_2, ..., P_m\}$ is a set of m identical processors in a homogeneous multiprocessor system[1]. Each processor is capable of executing any task. Processors may fail during

[1]The method developed in this chapter could be extended to a distributed system with m identical processor nodes. The main difficulty dealing with the distributed system is that both communication bandwidth and communication reliability should be considered as we compute task's reliability. The results of this chapter are based on a simpler system model which may provide a base to deal with the issues related to systems involving communication among nodes.

a mission and the failed processors are assumed to be fail-stop with failures being independent. Processors are associated with the reliability function, $R(t)$, and the failure function $F(t)$, where t is the time variable and

$$R(t) = 1 - F(t). \tag{5.1}$$

There are no restrictions on the reliability function. A simple example of the reliability function is an exponential function which is widely used to model many fault-tolerance systems:

$$R(t) = 1 - e^{-\lambda t},$$

where λ is a constant representing the failure rate.

$\{T_1, T_2, ..., T_n\}$ is a set of n aperiodic tasks to be configured and scheduled on m processors in a time interval [0, L], where L is the largest deadline of the tasks. Task T_i is characterized by the following:

- e_i — its ready time, which is the earliest time the task can start,
- c_i — its computation time,
- d_i — its deadline,
- V_i — its reward, if it is serviced successfully,
- v_i — its reward rate, derived as V_i/c_i,
- P_i — its failure penalty, if it is scheduled and fails because of processor failures,
- p_i — its failure penalty rate, derived as P_i/c_i,
- Q_i — its rejection penalty, if it is rejected,
- q_i — its rejection penalty rate, derived as Q_i/c_i.

If task T_i is accepted, it gets a reward V_i if it succeeds, and gets a failure penalty P_i if it fails. If task T_i is rejected, it gets a rejection penalty Q_i. The scheduling window for task T_i is the time interval from its ready time e_i to its deadline d_i. To simplify the analysis, we assume that tasks' scheduling windows are relatively small compared to L. Further, tasks are assumed to be independent.

With the above processor-task model, our task configuration strategy assigns a redundancy level, u_i, to task T_i, for $1 \leq i \leq n$. Redundant copies of the same task are assumed to be scheduled on different processors. So u_i is bounded from above by the number of processors, m, where $1 \leq i \leq n$. A task is considered to have failed only if all its redundant copies fail.

After the task set is configured, a task scheduling algorithm attempts to generate a feasible schedule. Let u_i be the number of redundant copies of task T_i and $\vec{\mathbf{f}}_i$ be its scheduled finish time vector made up of finish times of each copy of T_i:

$$\vec{\mathbf{f}}_i = (f_1, f_2, ..., f_{u_i}), \text{ where } f_j \leq d_i,\ 1 \leq j \leq u_i.$$

Then its reliability and failure probability are

$$R_i = 1 - F(f_1)F(f_2)\cdots F(f_{u_i}) \tag{5.2}$$

and

$$F_i = F(f_1)F(f_2)\cdots F(f_{u_i}). \tag{5.3}$$

Now we can define the *performance index* PI_i for task T_i. If the task is feasibly scheduled, i.e., guaranteed, it contributes a reward V_i with a probability R_i and a failure penalty P_i with a probability F_i. Thus, we have

$$\begin{aligned} PI_i &= V_iR_i - P_iF_i \\ &= c_iv_i - c_i(v_i + p_i)F_i. \end{aligned} \tag{5.4}$$

On the other hand, if T_i is rejected, the task contributes a rejection penalty Q_i. In this case, we have

$$PI_i = -Q_i = -c_i q_i. \tag{5.5}$$

Our goal is to maximize the total performance index PI,

$$PI = \sum_{i=1}^{n} PI_i. \tag{5.6}$$

PI is mainly determined by tasks' reliabilities and reward/penalty parameters. Tasks' reliabilities are determined by their redundancy levels which can be controlled within the task configuration phase, but we cannot change tasks' reward/penalty parameters. We present a simple example to demonstrate the relationship of PI and task redundancy level.

Example 5.1: Assume there is a multiprocessor with ten processors and there are ten tasks with their parameters listed in Table 5.1. All scheduled tasks will finish at time 10. If we assume each processor has a reliability of 0.9, then the reliability of a task is 0.9 when its redundancy is one and it is 0.99 when its redundancy is two, etc. Table 5.2 shows the values of PI for different redundancy levels (u). The maximum PI is reached when all scheduled tasks have their redundancy levels at two ($u = 2$). According to PI, the redundancy level is not enough if $u < 2$ and it is too much if $u > 2$. So the idea we are following in the remainder of the chapter is to derive the optimal redundancy level required at each time instance within L. Then, knowing this time-dependent optimal redundancy level, we can determine how much load to shed and based on this we can configure the task set accordingly (see Section 5.7).

Table 5.1 Task parameters for Example 5.1

Task	e_i	c_i	d_i	V_i	P_i	Q_i
$T_1, T_2, \cdots T_{10}$	0	10	10	10	100	1

Table 5.2 Relations between u and PI for Example 5.1

u	$PI = \sum_{i=1}^{10} PI_i$
1	$10(10 * 0.9 - 100 * 0.1) = -10$
2	$5(10 * 0.99 - 100 * 0.01) - 5 \approx 40$
3	$3(10 * 0.999 - 100 * 0.001) - 7 \approx 23$
4	$2(10 * 0.9999 - 100 * 0.0001) - 8 \approx 12$
10	$1(10 * (1 - 10^{-10}) - 100 * 10^{-10})) - 9 \approx 1$

In the next section, we discuss how tasks' redundancy levels can be derived as a closed form formula. To achieve this, we will use a continuous model to represent discrete tasks. Here we briefly present the basic idea. Consider a task T_i with only one copy scheduled to start at t_1 and to finish at t_2. Its failure probability is $F(t_2)$, because task T_i can be executed successfully only if the processor does not fail up to t_2. Its performance index PI_i is

$$c_i r_i - c_i(r_i + p_i)F(t_2), \tag{5.7}$$

where $c_i = t_2 - t_1$. In a continuous model, the performance index of the same task is represented as an integral from $t1$ to $t2$:

$$\int_{t_1}^{t_2} (r_i - (r_i + p_i)F(t))dt, \tag{5.8}$$

which is slightly larger than the one computed by (5.7) if $F(t)$ is a monotonically increasing function (which is true in general because of hardware aging process). Typically, the reliability function $R(t)$ or, equivalently, the failure probability function $F(t)$ changes very slowly. Hence,

$$c_i F(t_2) \approx \int_{t_1}^{t_2} F(t)dt, \tag{5.9}$$

and the value computed by (5.7) is about the same as the one computed by (5.8). A similar argument applies for tasks with multiple copies. Once we have such a continuous model, instead of considering the redundancy level for each task, we can consider the redundancy level required at a particular time t. Later, we show that

while this simplifies analysis, it does not result in any loss of accuracy in determining task redundancy levels.

5.3 Basic Task Configuration Strategy

In this section, we assume that all tasks have the same computation time c and the same v, p, and q. We discuss a way to derive task configuration function $u(t)$, where $u(t)$ represents the required redundancy level at time t, so as to optimize the performance index. It is important to note that all these assumptions are relaxed in the sections that follow. Specifically,

- In Section 5.4, a discrete task model is considered with tasks having different computation times and it is shown that the continuous model is a good approximation for the discrete model.
- In Section 5.5, we study the effects of converting $u^*(t)$, a real valued function, into integer values of $u(t)$ since, in practice, redundancy levels will be integers.
- In Section 5.6, we present an approach to handle tasks having different reward rates and penalty rates, i.e., different values of v, p, and q for different tasks.

Because tasks' scheduling windows are assumed to be small, all redundant copies of task T_i will be scheduled to finish around the same time. Let t be the task finish time. Its performance index PI_i given in (5.4) becomes

$$PI_i = c(v - (v+p)F(t)^{u(t)}), \tag{5.10}$$

where $u_i = u(t)$, which is the redundancy level for T_i.

When c becomes very small, we can use a continuous model and use dt to represent c. Equation (5.10) then becomes

$$PI_i = (v - (v+p)F(t)^{u(t)})dt. \tag{5.11}$$

On average, the number of tasks that can be scheduled in the time interval $[t - dt, t]$ is $m/u(t)$. So the total performance index for the time interval $[t - dt, t]$ is

$$\frac{m}{u(t)}(v - (v+p)F(t)^{u(t)})dt. \tag{5.12}$$

Thus, performance index for the tasks that can be accommodated is

$$\int_0^L \frac{m}{u(t)}(v - (v+p)F(t)^{u(t)})dt, \tag{5.13}$$

and the penalty due to all rejected tasks is

$$q(C - \int_0^L \frac{m}{u(t)}dt), \tag{5.14}$$

where C is the total computation times of all tasks without counting their redundant copies,

$$C = \sum_{i=1}^{n} c_i. \tag{5.15}$$

Therefore, the total performance index is

$$PI = \int_0^L \frac{m}{u(t)}(v + q - (v+p)F(t)^{u(t)})dt - qC. \tag{5.16}$$

The task configuration problem is translated into a form of calculus of variations, and we want to find the best $u(t)$ which maximizes PI. Let us define

$$G(t, u(t)) = \frac{m}{u(t)}(v + q - (v+p)F(t)^{u(t)}). \tag{5.17}$$

Then, the maximum PI is determined by the following Euler equation according to the theory of the calculus of variations [18]:

$$\frac{\partial G}{\partial u} = 0, \tag{5.18}$$

with the boundary conditions of $1 \leq u(t) \leq m$. Equation (5.18) is the same as:

$$F(t)^{u(t)}(1 - lnF(t)^{u(t)}) = \frac{v+q}{v+p}, \tag{5.19}$$

where $0 < F(t) < 1$.

Define

$$\frac{v+q}{v+p} = \alpha. \tag{5.20}$$

Let us explain the physical meaning of α. Suppose the rejection penalty rate $q = 0$. Assume that the failure penalty rate p is much larger than the reward rate v. The latter is a reasonable assumption for many fault-tolerant systems because that is the main reason we need redundancy. Then α is roughly the ratio of the reward rate v and the failure penalty rate p. However, if $q > p$, then $\alpha > 1$ and it means that the penalty for rejecting tasks is too high, so we should accept more tasks and reduce the task redundancy. In this case, there exists no solution for Equation (5.19) and the best configuration strategy is $u^*(t) = 1$ by using one of the boundary conditions.

Table 5.3 shows the relations between v, p, q, and α. Case 1 corresponds to a relatively low failure penalty rate while Case 5 corresponds to a relatively high failure penalty rate.

Table 5.3 Relations between v, p, q, and α

Case	v	p	q	α
1	1	19	1	0.1
2	1	199	1	0.01
3	1	1999	1	0.001
4	1	19999	1	0.0001
5	1	199999	1	0.00001

If Equation (5.19) has a solution, it must satisfy the *iso-reliability principle*:

$$F(t)^{u(t)} = A_\alpha, \tag{5.21}$$

where A_α is a constant mainly dependent on α. It is easy to verify that this is indeed the solution if we substitute $F(t)^{u(t)}$ by a constant (A_α) in Equation (5.19) and observe that both sides of the equation become constant, although we must choose A_α properly. The iso-reliability principle is the most interesting feature of this task configuration problem. Its name was chosen to suggest the fact that A_α represents

a level of tasks' failure probability which should be kept as a constant. Thus, the tasks' reliability is also a constant with respect to $(1 - A_\alpha)$.

Substituting (5.21) into (5.19) to determine A_α and we have

$$A_\alpha(1 - lnA_\alpha) = \alpha. \tag{5.22}$$

To search for the root, we may use a binary search algorithm such as the Bisection algorithm or a fast converging algorithm such as the Newton-Raphson algorithm [59].

We can then derive the optimal task configuration strategy $u^*(t)$ by rewriting (5.21),

$$u^*(t) = \frac{lnA_\alpha}{lnF(t)}, \tag{5.23}$$

where $0 < F(t) < 1$ and $1 \leq u^*(t) \leq m$.

In practice, we cannot control the failure function $F(t)$, but we can control the function $u(t)$ during the task configuration stage. We make two observations that are of significance from a practical viewpoint.

Observation 5.1: $u^*(t)$ changes slower than the failure function $F(t)$, because, in Equation (5.23), $u^*(t)$ is inversely proportional to $lnF(t)$, where $0 < F(t) < 1$.

In practice, $F(t)$ is likely to be a very slow function of t, so $u^*(t)$ is likely to be an even slower function of t.

Observation 5.2: If $F(t)$ is a monotonically increasing function, then $u^*(t)$ is a non-decreasing function, where $0 < F(t) < 1$ and $1 \leq u^*(t) \leq m$.

To see that the observation is right, we show that $(u^*(t))' > 0$. If $F(t)$ is a monotonically increasing function and $0 < F(t) < 1$, then $F'(t) > 0$, $lnF(t) < 0$ because $F(t) < 1$, and $lnA_\alpha < 0$ because

$$0 < F(t)^{u^*(t)} = A_\alpha < 1.$$

From Equation (5.23),

$$(u^*(t))' = (\frac{lnA_\alpha}{lnF(t)})' = lnA_\alpha \frac{(-1)}{(lnF(t))^2} \frac{F'(t)}{F(t)} > 0.$$

$F(t)$ is likely to be a monotonically increasing function because of the hardware aging process. Therefore, we can expect $u^*(t)$ to increase with time. This is demonstrated in the following example.

Example 5.2: We plot $u^*(t)$ in Figure 5.1, 5.2, and 5.3 for three different L by assuming that $m = 10$, $F(t) = 1 - e^{-\lambda t}$, $\lambda = 0.0001$. In these figures, each curve corresponds to a different α. Here are some conclusions we can derive from these figures:

- u^* increases with t,
- u^* increases slowly when α becomes larger,
- u^* is flat for large α when t is small, e.g., $\alpha = 0.1$, because u^* hits the lower bound 1. This means that, when the failure penalty rate is low, we do not need any redundancy for tasks.

In this section, we have built the basic task configuration strategy, based on a continuous model assuming tasks have the same computation time and reward/penalty

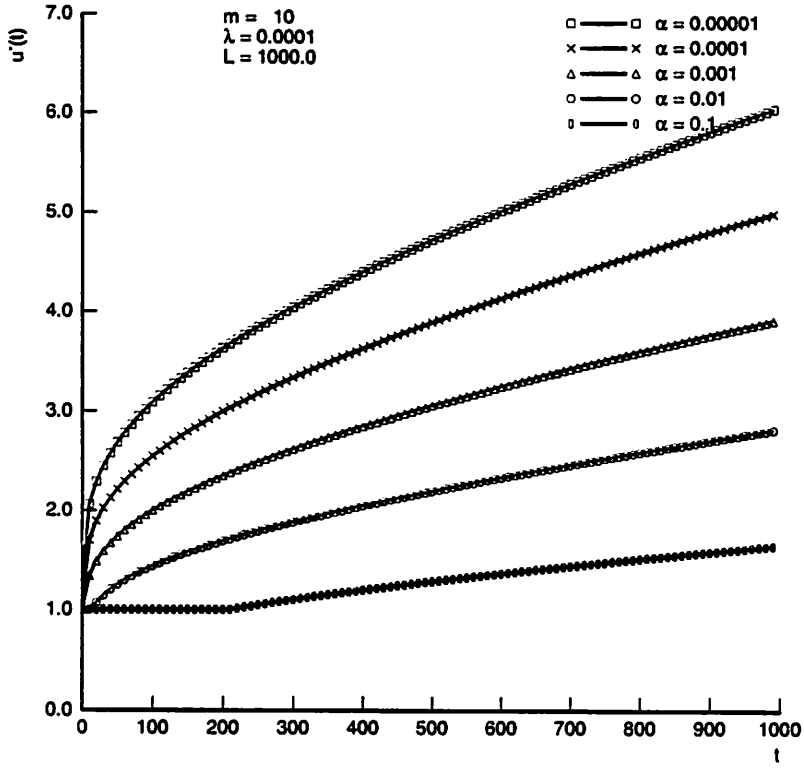

Figure 5.1 Optimal configuration strategy $u^*(t)$ with $L = 1000$

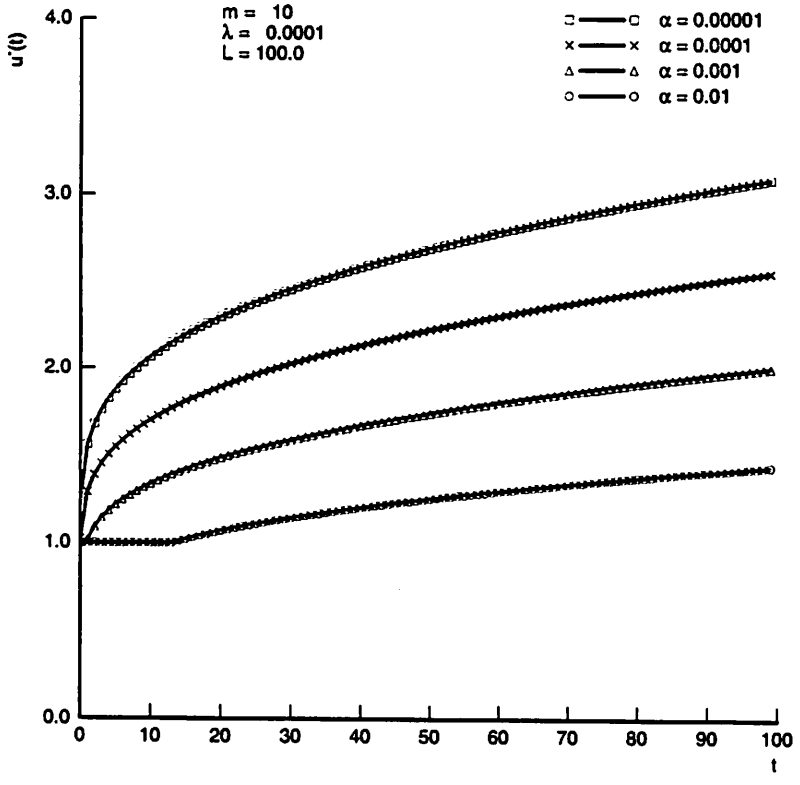

Figure 5.2 Optimal configuration strategy $u^*(t)$ with $L = 100$

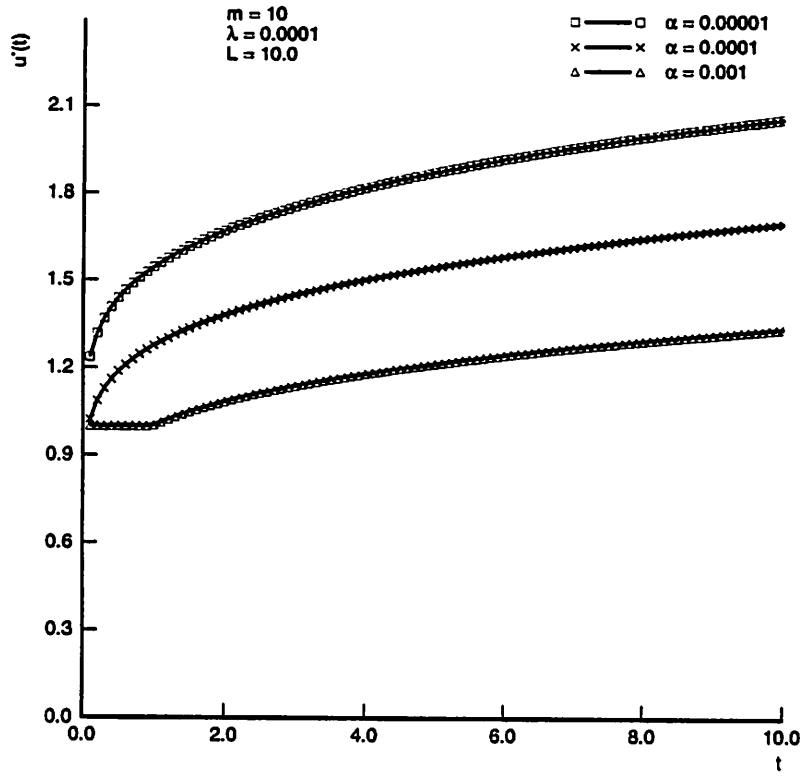

Figure 5.3 Optimal configuration strategy $u^*(t)$ with $L = 10$

rates v, p, and q. Specifically, we derived the optimal task configuration strategy $u^*(t)$ in a simple closed form:

$$u^*(t) = \frac{B}{lnF(t)}$$

where B is a constant, $B = lnA_\alpha$, and $F(t)$ is the failure function. In order to relax these assumptions, several extensions to this basic model are discussed in the following sections.

5.4 Discrete Model

In this section, we extend the continuous model to a discrete model. This relaxes the assumption that task computations are infinitely small as assumed in the last section. We discuss two cases:

1. Tasks have the same computation time c;
2. Tasks have different computation times.

Tasks are assumed to have the same v, p, and q. This is relaxed in Section 5.6.

We consider case 1 first. Let L be divided equally into k equal sized intervals of size c:

$$[t_0, t_1], [t_1, t_2], \cdots\cdots, [t_{k-1}, t_k],$$

with $t_0 = 0$ and $t_k = L$. Let $u(t_i)$ be the average redundancy in the interval $[t_{i-1}, t_i]$. Then, Equation (5.16) becomes

$$PI = -qC + \sum_{i=1}^{k} \frac{m}{u(t_i)}(v + q - (v + p)F(t_i)^{u(t_i)})c. \tag{5.24}$$

Using the same analysis method, we can derive the optimal configuration strategy as

$$u^*(t_i) = \frac{lnA_\alpha}{lnF(t_i)} \tag{5.25}$$

where $0 < F(t_i) < 1$, $i = 1, 2, \cdots, k$, and A_α is the same as defined in (5.20). Table 5.4 shows the relations between c versus PI under three different L, where PI is computed with the optimal configuration strategy, $u^*(t_i)$, for $1 \leq i \leq k$. We assume that $m = 10$, $\alpha = (r + q)/(r + p) = 0.0001$, and $\lambda = 0.0001$. The table shows that, for different values of c, the performance index PI is about the same for a particular L, especially when L is large compared to c. This implies that the continuous model

which assumed very small values for c ably represents the discrete model with respect to the performance index. So

$$\int_0^L \frac{m}{u(t)}(v+q-(v+p)F(t)^{u(t)})dt \approx \sum_{i=1}^{k} \frac{m}{u(t_i)}(v+q-(v+p)F(t_i)^{u(t_i)})c. \quad (5.26)$$

For the case 2, where tasks have different computation times, it is difficult to to extend Equation (5.16) directly by using the similar method as in the case 1, because tasks may have different finish times in any subinterval. But, notice that in Table 5.4, for a given L, PI is almost the same for tasks with different computation times. So, given the approximation in (5.26), we can use the continuous model to approximate this case also to obtain the performance index.

Table 5.4 Relations between c and PI

c	$PI\|_{L=1000}$	$PI\|_{L=100}$	$PI\|_{L=10}$
10.0	2583.36988	423.31917	54.15487
1.0	2605.02754	437.06966	60.36285
0.1	2607.82060	439.25115	61.73948
0.01	2608.36467	439.49504	61.93850
0.001	2608.40129	439.55622	61.95905

5.5 Converting $u^*(t)$ into integer values of $u(t)$

The optimal configuration strategy $u^*(t)$ in (5.23) is a real valued function. In practice, tasks' redundancies are integers. In this section, we show that the optimal task configuration strategy $u^*(t)$ can be approximated by an integer function with a very small loss with respect to the performance index PI.

We compare the performance index using $u^*(t)$ with the performance index using the following integer functions which approximate $u^*(t)$:

- $u\_ceil(t)$ — the integer equal to or greater than $u^*(t)$;
- $u\_rint(t)$ — rounding $u^*(t)$ to an integer;

- $u\_int(t)$ — choosing one of the two neighboring integers of $u^*(t)$ which gives the better performance.

Let $PI(u(t))$ be the performance index using strategy $u(t)$. Comparing the performance index PI using $u^*(t)$ to the performance index PI using $u\_ceil(t)$, $u\_rint(t)$, and $u\_int(t)$ respectively, it is not difficult to see that

$$PI(u^*(t)) \geq PI(u\_int(t)) \geq \{PI(u\_ceil), PI(u\_rint)\}.$$

In Figure 5.4, 5.5, and 5.6, we plot $u^*(t)$ and these three functions with $L = 100$. Table 5.5 lists the ratios of the performance indices based on integer functions and the the performance index using the optimal configuration strategy $u^*(t)$, for three different L. Here, we assume that $m = 10$, $\alpha = (v+q)/(v+p) = 0.0001$, $\lambda = 0.0001$, and $c = 1$.

Table 5.5 Ratios of the performance indices using integer functions and $PI(u^*)$

L	$PI(u\_ceil)/PI(u^*)$	$PI(u\_rint)/PI(u^*)$	$PI(u\_int)/PI(u^*)$
1000	0.929	0.921	0.960
100	0.859	0.716	0.906
10	0.825	0.080	0.825

From Table 5.5, we conclude that $u\_int(t)$ is the best candidate to represent $u^*(t)$ with respect to the performance index PI. Also Figure 5.6 shows that $u\_int(t)$ has the redundancy values 2 and 3 in relatively large time windows. This validates the assumption we made earlier that the optimal task redundancy is highly likely to be a constant within tasks' scheduling windows.

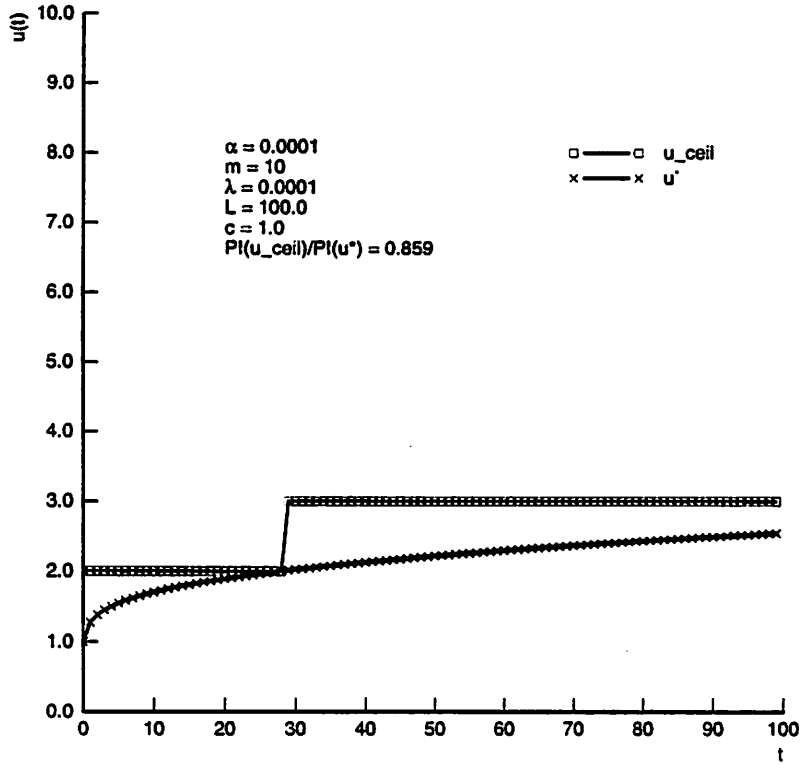

Figure 5.4 u^* versus *u_ceil* with $L = 100$

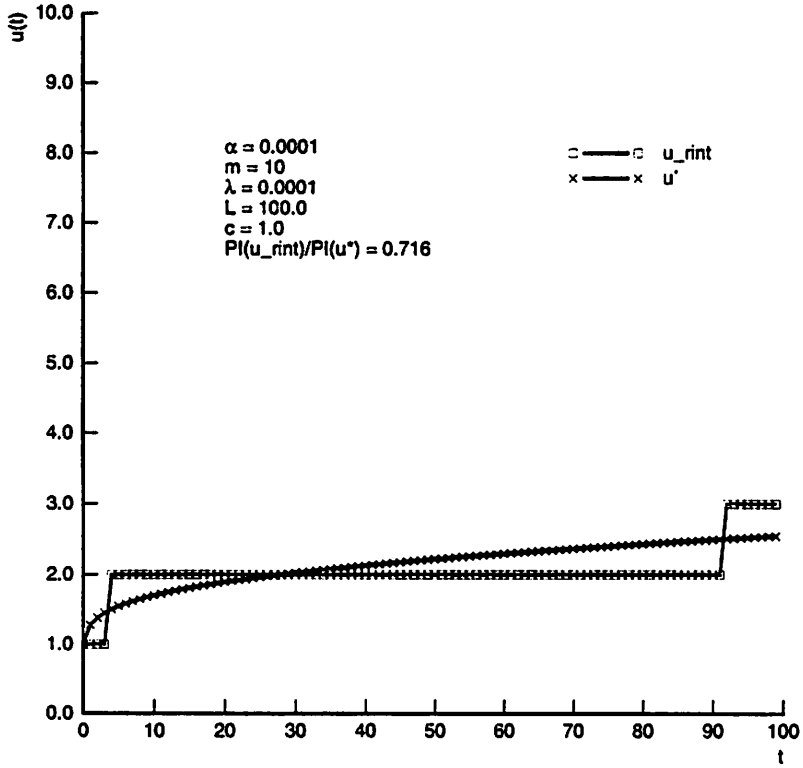

Figure 5.5 u^* versus *u_rint* with $L = 100$

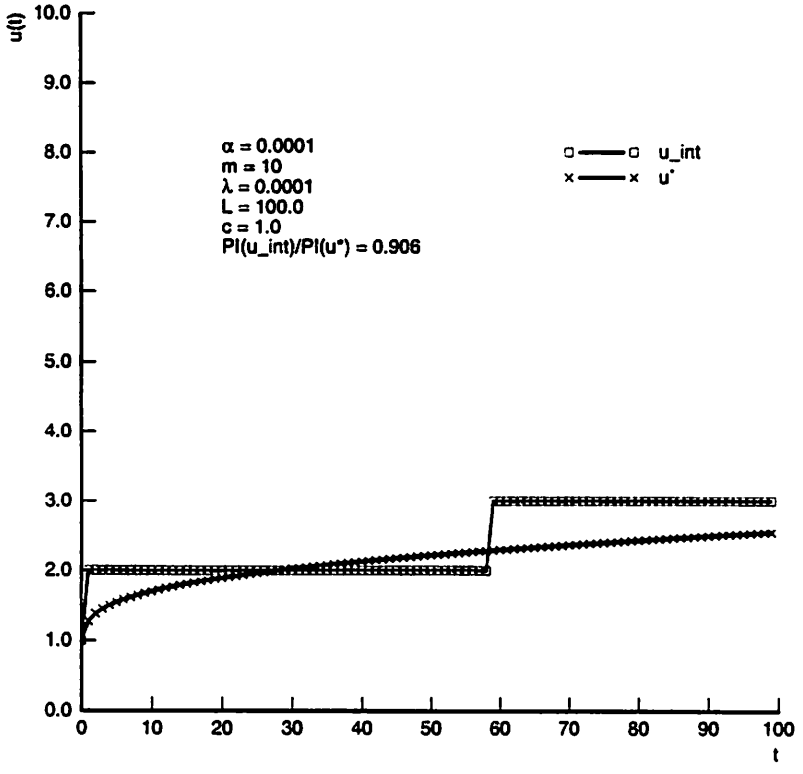

Figure 5.6 u^* versus u_int with $L = 100$

5.6 Configuring Tasks with Different Reward/Penalty Parameters

In this section, we extend the basic task configuration model to allow tasks with different v, p, and q. Let $v(t)$, $p(t)$, and $q(t)$ be the average reward rate and the

average penalty rates at t, computed from tasks whose scheduling windows include t. The total performance index becomes

$$PI = \int_0^L \{\frac{m}{u(t)}[v(t) + q(t) - (v(t) + p(t))F(t)^{u(t)}] - \frac{q(t)C}{L}\}dt. \tag{5.27}$$

Define

$$G(t, u(t)) = \frac{m}{u(t)}[v(t) + q(t) - (v(t) + p(t))F(t)^{u(t)}] - \frac{q(t)C}{L}. \tag{5.28}$$

Then, the maximum of PI is again determined by Euler equation:

$$\frac{\partial G}{\partial u} = 0, \tag{5.29}$$

with the boundary conditions of $1 \leq u(t) \leq m$. Equation (5.29) is the same as:

$$F(t)^{u(t)}(1 - lnF(t)^{u(t)}) = \alpha(t), \tag{5.30}$$

where $0 < F(t) < 1$, $1 \leq u(t) \leq m$, and

$$\alpha(t) = \frac{v(t) + q(t)}{v(t) + p(t)}. \tag{5.31}$$

The solution for (5.30) is the optimal configuration strategy $u^*(t)$, which must satisfy

$$F(t)^{u^*(t)} = A_\alpha(t), \tag{5.32}$$

where $0 < F(t) < 1$, $1 \leq u^*(t) \leq m$, and $A_\alpha(t)$ depends on $\alpha(t)$. Rewriting the above equation, we have

$$u^*(t) = \frac{lnA_\alpha(t)}{lnF(t)}. \tag{5.33}$$

To compute function $A_\alpha(t)$, we substitute (5.32) into (5.30):

$$A_\alpha(t)(1 - lnA_\alpha(t)) = \alpha(t). \tag{5.34}$$

Thus, given t, $0 \leq t \leq L$, we can compute $\alpha(t)$. From $\alpha(t)$, we can determine the corresponding $A_\alpha(t)$. Table 5.6 lists 320 pairs of $\alpha(t)$ and $A_\alpha(t)$, and from that and (5.33), we can determine $u^*(t)$.

Table 5.6 Relationship between α and A_α

α	A_α	α	A_α	α	A_α	α	A_α
0.00000001	0.00000000044	0.00000081	0.00000004522	0.00000161	0.00000009369	0.00000241	0.00000014384
0.00000002	0.00000000092	0.00000082	0.00000004581	0.00000162	0.00000009431	0.00000242	0.00000014448
0.00000003	0.00000000140	0.00000083	0.00000004641	0.00000163	0.00000009493	0.00000243	0.00000014511
0.00000004	0.00000000190	0.00000084	0.00000004700	0.00000164	0.00000009555	0.00000244	0.00000014575
0.00000005	0.00000000240	0.00000085	0.00000004759	0.00000165	0.00000009617	0.00000245	0.00000014638
0.00000006	0.00000000290	0.00000086	0.00000004818	0.00000166	0.00000009679	0.00000246	0.00000014702
0.00000007	0.00000000342	0.00000087	0.00000004878	0.00000167	0.00000009741	0.00000247	0.00000014765
0.00000008	0.00000000393	0.00000088	0.00000004937	0.00000168	0.00000009803	0.00000248	0.00000014829
0.00000009	0.00000000445	0.00000089	0.00000004997	0.00000169	0.00000009865	0.00000249	0.00000014893
0.00000010	0.00000000497	0.00000090	0.00000005056	0.00000170	0.00000009927	0.00000250	0.00000014956
0.00000011	0.00000000549	0.00000091	0.00000005116	0.00000171	0.00000009989	0.00000251	0.00000015020
0.00000012	0.00000000602	0.00000092	0.00000005175	0.00000172	0.00000010051	0.00000252	0.00000015083
0.00000013	0.00000000655	0.00000093	0.00000005235	0.00000173	0.00000010113	0.00000253	0.00000015147
0.00000014	0.00000000708	0.00000094	0.00000005295	0.00000174	0.00000010175	0.00000254	0.00000015211
0.00000015	0.00000000762	0.00000095	0.00000005354	0.00000175	0.00000010237	0.00000255	0.00000015275
0.00000016	0.00000000815	0.00000096	0.00000005414	0.00000176	0.00000010299	0.00000256	0.00000015338
0.00000017	0.00000000869	0.00000097	0.00000005474	0.00000177	0.00000010361	0.00000257	0.00000015402
0.00000018	0.00000000923	0.00000098	0.00000005534	0.00000178	0.00000010424	0.00000258	0.00000015466
0.00000019	0.00000000977	0.00000099	0.00000005593	0.00000179	0.00000010486	0.00000259	0.00000015529
0.00000020	0.00000001031	0.00000100	0.00000005653	0.00000180	0.00000010548	0.00000260	0.00000015593
0.00000021	0.00000001086	0.00000101	0.00000005713	0.00000181	0.00000010610	0.00000261	0.00000015657
0.00000022	0.00000001141	0.00000102	0.00000005773	0.00000182	0.00000010673	0.00000262	0.00000015721
0.00000023	0.00000001195	0.00000103	0.00000005833	0.00000183	0.00000010735	0.00000263	0.00000015785
0.00000024	0.00000001250	0.00000104	0.00000005893	0.00000184	0.00000010797	0.00000264	0.00000015849
0.00000025	0.00000001305	0.00000105	0.00000005954	0.00000185	0.00000010860	0.00000265	0.00000015913
0.00000026	0.00000001360	0.00000106	0.00000006014	0.00000186	0.00000010922	0.00000266	0.00000015976
0.00000027	0.00000001416	0.00000107	0.00000006074	0.00000187	0.00000010984	0.00000267	0.00000016040
0.00000028	0.00000001471	0.00000108	0.00000006134	0.00000188	0.00000011047	0.00000268	0.00000016104
0.00000029	0.00000001527	0.00000109	0.00000006194	0.00000189	0.00000011109	0.00000269	0.00000016168
0.00000030	0.00000001582	0.00000110	0.00000006254	0.00000190	0.00000011172	0.00000270	0.00000016232
0.00000031	0.00000001638	0.00000111	0.00000006315	0.00000191	0.00000011234	0.00000271	0.00000016296
0.00000032	0.00000001694	0.00000112	0.00000006375	0.00000192	0.00000011297	0.00000272	0.00000016360
0.00000033	0.00000001750	0.00000113	0.00000006436	0.00000193	0.00000011359	0.00000273	0.00000016424
0.00000034	0.00000001806	0.00000114	0.00000006496	0.00000194	0.00000011422	0.00000274	0.00000016488
0.00000035	0.00000001862	0.00000115	0.00000006556	0.00000195	0.00000011484	0.00000275	0.00000016552
0.00000036	0.00000001918	0.00000116	0.00000006617	0.00000196	0.00000011547	0.00000276	0.00000016616
0.00000037	0.00000001974	0.00000117	0.00000006677	0.00000197	0.00000011609	0.00000277	0.00000016680
0.00000038	0.00000002031	0.00000118	0.00000006738	0.00000198	0.00000011672	0.00000278	0.00000016744
0.00000039	0.00000002087	0.00000119	0.00000006798	0.00000199	0.00000011735	0.00000279	0.00000016809
0.00000040	0.00000002144	0.00000120	0.00000006859	0.00000200	0.00000011798	0.00000280	0.00000016873
0.00000041	0.00000002201	0.00000121	0.00000006920	0.00000201	0.00000011860	0.00000281	0.00000016937
0.00000042	0.00000002257	0.00000122	0.00000006980	0.00000202	0.00000011923	0.00000282	0.00000017001
0.00000043	0.00000002314	0.00000123	0.00000007041	0.00000203	0.00000011986	0.00000283	0.00000017065
0.00000044	0.00000002371	0.00000124	0.00000007102	0.00000204	0.00000012048	0.00000284	0.00000017129
0.00000045	0.00000002428	0.00000125	0.00000007163	0.00000205	0.00000012111	0.00000285	0.00000017193
0.00000046	0.00000002485	0.00000126	0.00000007223	0.00000206	0.00000012174	0.00000286	0.00000017258
0.00000047	0.00000002542	0.00000127	0.00000007284	0.00000207	0.00000012237	0.00000287	0.00000017322
0.00000048	0.00000002599	0.00000128	0.00000007345	0.00000208	0.00000012299	0.00000288	0.00000017386
0.00000049	0.00000002657	0.00000129	0.00000007406	0.00000209	0.00000012362	0.00000289	0.00000017450
0.00000050	0.00000002714	0.00000130	0.00000007467	0.00000210	0.00000012425	0.00000290	0.00000017515
0.00000051	0.00000002772	0.00000131	0.00000007528	0.00000211	0.00000012488	0.00000291	0.00000017579
0.00000052	0.00000002829	0.00000132	0.00000007589	0.00000212	0.00000012551	0.00000292	0.00000017643
0.00000053	0.00000002887	0.00000133	0.00000007650	0.00000213	0.00000012614	0.00000293	0.00000017707
0.00000054	0.00000002944	0.00000134	0.00000007711	0.00000214	0.00000012677	0.00000294	0.00000017772
0.00000055	0.00000003002	0.00000135	0.00000007772	0.00000215	0.00000012740	0.00000295	0.00000017836
0.00000056	0.00000003060	0.00000136	0.00000007833	0.00000216	0.00000012803	0.00000296	0.00000017901
0.00000057	0.00000003118	0.00000137	0.00000007894	0.00000217	0.00000012866	0.00000297	0.00000017965
0.00000058	0.00000003175	0.00000138	0.00000007955	0.00000218	0.00000012929	0.00000298	0.00000018029
0.00000059	0.00000003233	0.00000139	0.00000008017	0.00000219	0.00000012992	0.00000299	0.00000018094
0.00000060	0.00000003291	0.00000140	0.00000008078	0.00000220	0.00000013055	0.00000300	0.00000018158
0.00000061	0.00000003349	0.00000141	0.00000008139	0.00000221	0.00000013118	0.00000301	0.00000018222
0.00000062	0.00000003408	0.00000142	0.00000008200	0.00000222	0.00000013181	0.00000302	0.00000018287
0.00000063	0.00000003466	0.00000143	0.00000008262	0.00000223	0.00000013244	0.00000303	0.00000018351
0.00000064	0.00000003524	0.00000144	0.00000008323	0.00000224	0.00000013308	0.00000304	0.00000018416
0.00000065	0.00000003582	0.00000145	0.00000008384	0.00000225	0.00000013371	0.00000305	0.00000018480
0.00000066	0.00000003641	0.00000146	0.00000008446	0.00000226	0.00000013434	0.00000306	0.00000018545
0.00000067	0.00000003699	0.00000147	0.00000008507	0.00000227	0.00000013497	0.00000307	0.00000018610
0.00000068	0.00000003758	0.00000148	0.00000008568	0.00000228	0.00000013561	0.00000308	0.00000018674
0.00000069	0.00000003816	0.00000149	0.00000008630	0.00000229	0.00000013624	0.00000309	0.00000018738
0.00000070	0.00000003875	0.00000150	0.00000008692	0.00000230	0.00000013687	0.00000310	0.00000018803
0.00000071	0.00000003933	0.00000151	0.00000008753	0.00000231	0.00000013750	0.00000311	0.00000018868
0.00000072	0.00000003992	0.00000152	0.00000008815	0.00000232	0.00000013814	0.00000312	0.00000018932
0.00000073	0.00000004051	0.00000153	0.00000008876	0.00000233	0.00000013877	0.00000313	0.00000018997
0.00000074	0.00000004109	0.00000154	0.00000008938	0.00000234	0.00000013940	0.00000314	0.00000019061
0.00000075	0.00000004168	0.00000155	0.00000008999	0.00000235	0.00000014004	0.00000315	0.00000019126
0.00000076	0.00000004227	0.00000156	0.00000009061	0.00000236	0.00000014067	0.00000316	0.00000019191
0.00000077	0.00000004286	0.00000157	0.00000009123	0.00000237	0.00000014131	0.00000317	0.00000019255
0.00000078	0.00000004345	0.00000158	0.00000009184	0.00000238	0.00000014194	0.00000318	0.00000019320
0.00000079	0.00000004404	0.00000159	0.00000009246	0.00000239	0.00000014257	0.00000319	0.00000019385
0.00000080	0.00000004463	0.00000160	0.00000009308	0.00000240	0.00000014321	0.00000320	0.00000019450

5.7 Applying the Results in Practice

In this section, we discuss how to apply this task configuration theory in a real-time system. The basic idea is to derive the optimal task redundancy function $u^*(t)$ first. Note that $u^*(t)$ is determined only by the processor reliability function and task reward/penalty rates. $u^*(t)$ is then approximated by a corresponding integer function $u\_int(t)$. Finally, by using $u\_int(t)$, we can determine how many copies of each task can be need to be scheduled.

If tasks to be configured have the same reward and penalty rates, e.g., the same v, p, and q, the task configuration procedure becomes relatively easy, even if their computation times are different. A_α is computed by solving Equation (5.22) and $F(t)$ is determined from the failure properties of the hardware. We can then compute $u^*(t)$ using Equation (5.23). Next, we use $u\_int(t)$ to approximate $u^*(t)$ as presented in Section 5.5. In general, $u\_int(t)$ has a shape similar to the one plotted in Figure 5.6, which is a step function. Thus, the computation for $u\_int(t)$ can be easily speeded up, by just computing each turning point of the step function $u\_int(t)$. For example, in Figure 5.6, the redundancy levels are 1 for [0, 1), 2 for [1, 59), and 3 for [59, 100], and the turning points occur at 1 and 59. Also, we may compute $u\_int(t)$ off-line to form a table for on-line use. After $u\_int(t)$ is derived, tasks are assigned the redundancy levels given by $u\_int(t)$ in the following way. If a task's scheduling window covers two different values of $u\_int(t)$, we assign the task the higher redundancy level. Otherwise, we assign the task with a redundancy level determined by $u\_int(t)$. Note that this is only an approximation because the optimal redundancy level required by a task is determined by its scheduled finished time in the final schedule.

We can then schedule the tasks. If a scheduling algorithm can shed tasks during scheduling, the task set is directly handed to the scheduling algorithm. Here we either apply a heuristic-based algorithm using a heuristic function to determine which task

to select next as in [91] or use a bin-packing-based algorithm such as the one in [33]. In both cases, the remaining tasks are rejected after all available system resources are consumed. If a scheduling algorithm cannot shed tasks during scheduling, e.g., the H_2 heuristic scheduling algorithm, then we may shed some tasks from the system before the task set is handed to the scheduling algorithm. We should avoid repeated failures of the scheduling algorithm for finding a feasible schedule. Failures are mainly caused by the following factors:

- The task set for the scheduling algorithm has an overload in a time interval $[t_x, t_y]$ (for $t_y \leq L$), such that the sum of the computation times of all tasks having deadlines within this interval is greater than $m \cdot (t_y - t_x)$, where m is the number of processors available.

- The heuristic scheduling algorithm may fail to find a feasible schedule even if the task set is feasible, because the heuristic scheduling algorithm is only an approximation to the optimal algorithm which always find the feasible schedule if it exists.

- Tasks may have more complex constraints, e.g., additional resource requirements, which may reduce the system utilization because of the resource contentions among tasks.

Possible solutions to avoid these failures are (1) to avoid overloads in all sub-intervals and (2) to reduce task workload further because a portion of the system utilization will be wasted because of the resource contentions among tasks.

If tasks to be configured have different reward and penalty rates, the task configuration procedure becomes a bit more complicated. Here is a high level description of one way to solve the problem. We divide L into K equal intervals of size Δ. The value of Δ will depend on how closely we would like the redundancy levels to reflect optimal values. For example, assuming that $\bar{c}$ is the average computation time for

the tasks, we set Δ to be $\bar{c}/2$. For each t, where $t = i\Delta$, $i = 1 \cdots K$, we do the following: We compute $\alpha(t)$ based on Equation (5.31), with $v(t)$, $p(t)$, and $q(t)$ being the average reward rate and the average penalty rate at t, computed from tasks whose scheduling windows cover t. From $\alpha(t)$, we can derive $A_\alpha(t)$ either by solving Equation (5.34) or by using Table 5.6. After knowing $A_\alpha(t)$ and $F(t)$, we compute $u^*(t)$ based on Equation (5.33). Next, we use $u\_int(t)$ to approximate $u^*(t)$ as presented in Section 5.5. Knowing $u\_int(t)$, tasks are then assigned the redundancy with the values specified by $u\_int(t)$. Finally, we can start to schedule tasks. Many issues related to task scheduling discussed above also apply here.

Note that, in both cases, the values of $u\_int(t)$ represent the lower bound on task redundancy to maximize performance without being jeopardized by too high redundancy levels. Therefore, some tasks could be assigned higher redundancy levels than the ones specified by $u\_int(t)$, if there are not enough tasks to fill the processor resources in a given time interval. This will improve reliability without affecting the schedulability of tasks. Also, how well all this works in practice must still be determined by simulations or actual system implementations.

5.8 Conclusions

In this chapter, we have developed a new task configuration strategy with the goal of maximizing the preformance index which considers deadline-related performance measures as well as fault-tolerance related requirements. Our analysis shows that the basic continuous task model very closely approximates the discrete models, and the optimal task configuration function $u^*(t)$ can be substituted by an integer function with very minor effects on the total performance index. Also, we showed how our basic model can be extended to handle tasks with different reward/penalty parameters and computation times.

There are several avenues that are worthy of further investigation to determine how well this works in practice. Interactions between the configuration phase and the scheduling phase need to be explored. Also, it is important to understand the implications of applying such a configuration strategy in a dynamic system with tasks arriving and then being configured (with existing tasks being reconfigured). Such a dynamic reconfiguration can help improve performance further [49].

CHAPTER 6

SUMMARY

6.1 Contributions

The research presented in this dissertation focuses on three issues related to dynamic scheduling arising in many real-time applications. (1) the worst case analysis of on-line scheduling algorithms, (2) the bound analysis of heuristic algorithms for tasks with resource requirements, and (3) the performance-related reliability analysis for determining tasks' redundancy levels. By studying these problems of interest to dynamic real-time systems using both mathematical as well as simulation tools and by providing algorithmic solutions to the problems, we have contributed in significant ways to both the theoretical aspects and practical aspects of real-time systems.

For the worst case analysis of on-line scheduling algorithms, we consider tasks with different values, and consider the performance bound to be the ratio of the value obtained by an on-line scheduling algorithm to the value obtained by an ideal optimal off-line "clairvoyant" algorithm. The main contributions of our analysis are:

- If all tasks have the same value density, i.e., the value per unit computation time, we show that the tight upper bound of the uniprocessor on-line scheduling problem is 1/4;

- If tasks have different value densities and the ratio between the highest and the lowest value density is γ, we show that the upper bound for the uniprocessor on-line scheduling problem is

$$\frac{1}{\gamma + 1 + 2\sqrt{\gamma}};$$

- Two on-line scheduling algorithms, TD_1 and TD_1', are designed to reach the two upper bounds mentioned above;

- If all tasks have the same value density, we show that the tight upper bound of the dual-processor on-line scheduling problem is 1/2.

For the bound analysis of heuristic algorithms for tasks with resource requirements, we focus on heuristic algorithms which have both a good ability to generate feasible schedules and can produce feasible schedules of high quality, expressed in terms of the schedule length. The heuristic algorithms must handle tasks' real-time constraints and some additional resource requirements. Particularly, we consider a generalized heuristic scheduling algorithm, called H_k, where $2 \leq k \leq m$. It combines features from both list scheduling and the H algorithm which is the basic heuristic scheduling algorithm of the Spring real-time system. The H_k algorithm schedules tasks according to their dynamically determined priorities while attempting to keep at least k processors busy whenever possible. If it is not possible to keep k processors busy, then it will keep as many processors busy as possible. The main contributions in this part of the dissertation are:

- We show that the schedule length bound of H_k for tasks with the same computation time is

$$\frac{m}{k} + \sum_{j=2}^{k} \frac{1}{j},$$

where $2 \leq k \leq m$. We show that the schedule length bound of H_k for tasks with arbitrary computation times is

$$\frac{m+1}{2},$$

where $2 \leq k \leq m$. In both cases, the complexity is $O(n^{k+1}r)$ for $2 < k \leq m$, and $O(n^2r)$ for $k = 2$, where n is the number of tasks;

- Using simulation studies of three algorithms, H_2, the H scheduling algorithm, and list scheduling, we show that the H_2 algorithm has almost the same mean behavior in the ability to find feasible schedules as the H scheduling algorithm, and both are much better than list scheduling. The scheduling overheads of H and H_2 are about the same, while the worst case schedule length bound for H_2 is about a half of the worst case schedule length bound for H.

For the performance-related reliability analysis for determining tasks' redundancy levels, we study algorithms for assigning redundancy levels to tasks such that the total performance index is maximized. The main contributions include:

- If tasks have the same reward/penalty parameters, and the task scheduling windows are relatively small compared to the length of the schedules, we show that the optimal task configuration strategy follows the iso-reliability principle and there exists a closed form solution to compute the optimal strategy, by using a continuous model.

- We show that the continuous model is a good approximation for the discrete model. Thus, applying the results for the continuous model, we can handle a discrete task model where tasks have arbitrary computation times.

- We show that the same analysis approach can be further extended to handle tasks having different reward rates and penalty rates.

6.2 Future Research in Dynamic Scheduling

A number of possible extensions are:

- In the worst case analysis for on-line scheduling algorithms, if we have some partial knowledge of future tasks, we could improve the performance bound further. For example, in many real-time systems, all task parameters are known beforehand except that their arrival times are unknown. It is highly desirable if we can determine the worst case performance bound for a given set of tasks. This is not going to be an easy problem, because to derive a worst case request sequence for an arbitrary task set is extremely difficult if not impossible. Alternatively, we may consider some general situations: (1) tasks have a certain amount of laxity and (2) the computation times of tasks are bound within a certain range. For these two cases, it is likely that improved performance bounds can be derived.

  Other open problems for on-line scheduling include on-line scheduling for three or more processors and on-line scheduling for dual-processor with tasks having arbitrary value densities.

- The heuristic algorithms studied in this dissertation are limited to independent tasks with additional resource requirements. In practice, tasks may have more complex structures, such as precedence constraints. It will be useful to extend the H_2 heuristic algorithm to accommodate the new constraints.

- In the performance-related reliability analysis, interactions between the configuration phase and the scheduling phase need to be explored to better understand the practical implication of the configuration strategy. Also, it is important

to understand the implications of applying such a configuration strategy in a dynamic system with tasks arriving and then being configured (with existing tasks being reconfigured). Such a dynamic reconfiguration can help improve performance further [49].

BIBLIOGRAPHY

[1] Abbott, R. J. Resourceful systems for fault tolerance, reliability, and safety. *ACM Computing Surveys*, 22(1):35–68, 1990.

[2] Anderson, T., editor. *Resilient Computer Systems*, volume 1. John Wiley & Sons, Inc., 1985.

[3] Anderson, T. and Lee, P. A. *Fault-Tolerance – Principles and Practice.* Prentice/Hall, 1981.

[4] Avizienis, A. The N-version approach to fault tolerant software. *IEEE Trans. on Software Engineering*, SE-11(12):1491–1501, 1985.

[5] Avizienis, A. et al. The STAR (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design. *IEEE Trans. on Computer*, C-20(11):1312–1321, 1971.

[6] Avizienis, A. et al. In search of effective diversity: a six-language study of fault-tolerant flight control software. In *Digest of Papers FTCS-18*, pages 15–22, 1988.

[7] Bannister, J. A. and Trivedi, K. S. Task allocation in fault-tolerant distributed systems. *ACTA Information*, 22:261–81, 1983.

[8] Baruah, S., Koren, G., Mao, D., Mishra, B., Rosier, A. R. L., Shasha, D., and Wang, F. On the competitiveness of on-line real-time tasks scheduling. *Proceedings Real-Time Systems Symposium*, 1991.

[9] Baruah, S., Koren, G., Mao, D., Mishra, B., Rosier, A. R. L., Shasha, D., and Wang, F. On the competitiveness of on-line real-time tasks scheduling. *Real-Time Systems*, 4(2), June 1992.

[10] Baruah, S., Koren, G., Mishra, B., Raghunathan, A., Rosier, L., and Shasha, D. On-line scheduling in the presence of overload. In *1991 IEEE Symposium on Foundations of Computer Science*, 1991.

[11] Baruah, S. K. and Rosier, L. E. Limitations concerning on-line scheduling algorithms for overloaded real-time systems. *8th IEEE Workshop on Real-Time Operating Systems and Software*, pages 128–132, 1991.

[12] Beaudry, M. D. Performance-related reliability measures for computing systems. *IEEE Transactions on Computers*, pages 540–547, June 1978.

[13] Ben-David, S., Borodin, A., Karp, R. M., Tardos, G., and Wigderson, A. On the power of randomization in online algorithms. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 379–386, 1990.

[14] Bern, M., Greene, D. H., Raghunathan, A., and Sudan, M. Online algorithms for locating checkpoints. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 359–368, 1990.

[15] Biyabani, S., Stankovic, J. A., and Ramamritham, K. The integration of deadline and criticalness in hard real-time scheduling. In *Proc. Real-Time Systems Symposium*, 1988.

[16] Blazewicz, J., Lenstra, J. K., and Kan, A. H. G. R. Scheduling subject to resource constraints: Classification and complexity. *Discrete Applied Mathematics*, 5:11–24, 1983.

[17] Bodson, M., Stankovic, J., and Strigini, L. Adaptive fault tolerance for real-time systems. *Proc. of Third International Workshop on Responsive Computer Systems*, 1993.

[18] Bolza, O. *Lectures on the Calculus of Variations.* Chelsea Publishing Company, 1960.

[19] Borodin, A., Linial, N., and Saks, M. An optimal online algorithm for metrical task system. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 373–382, 1987.

[20] Butazzo, G. and Stankovic, J. RED: Robust earliest deadline scheduling. *Proc. of Third International Workshop on Responsive Computer Systems*, 1993.

[21] Calabro, S. R. *Reliability Principles and Practices.* McGraw-Hill, New York, 1962.

[22] Campbell, R. H., Horton, K. H., and Belford, G. G. Simulations of a fault-tolerant deadline mechanism. In *Digest of Papers FTCS-9*, pages 95–101, 1979.

[23] Chen, L. and Avizienis, A. N-version programming: A fault tolerance approach to reliability of software operation. In *Digest of Papers FTCS-8*, pages 3–9, 1978.

[24] Chen, M.-I. and Lin, K.-J. Dynamic priority ceiling: a concurrency control protocol for handling real-time tasks in unpredictable environments. *The Journal of Real-Time Systems*, 2(4):347–364, 1990.

[25] Coffman, Jr., E. G., editor. *Computer and Job-Shop Scheduling Theory.* John Wiley & Sons, 1976.

[26] Coppersmith, D., Doyle, P., Raghavan, P., and Snir, M. Random walks on weighted graphs, and applications to on-line algorithms. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 369–378, 1990.

[27] Cristian, F. Understanding fault-tolerant distributed systems. *CACM*, February 1991.

[28] Davari, S. and Dhall, S. K. An on-line algorithm for real-time tasks allocation. *Proceedings Real-Time Systems Symposium*, pages 194–200, 1986.

[29] Dertouzos, M. L. and Mok, A. K. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Trans. on Software Engineering*, SE-15(12):1497–1505, 1989.

[30] Dhall, S. K. and Liu, C. L. On a real-time scheduling problem. *Operations Research*, 26(1):127–40, February 1978.

[31] Even, S. *Algorithmic Combinatorics.* The Macmillan Company, New York, 1973.

[32] Garey, M. R. and Graham, R. L. Bounds for multiprocessor scheduling with resource constraints. *SIAM J. on Computing*, 4:187–200, 1975.

[33] Garey, M. R., Graham, R. L., Johnson, D. S., and Yao, A. C.-C. Resource constrained scheduling as generalized bin packing. *J. Combinatorial Theory Ser. A*, 21:257–298, 1976.

[34] Garey, M. R. and Johnson, D. S. Complexity results for multiprocessor scheduling under resource constraints. *SIAM J. on Computing*, 4:397–411, 1975.

[35] Gillies, D. W. and Liu, J. W.-S. Greed in resource scheduling. *Proceedings Real-Time Systems Symposium*, pages 285–294, 1989.

[36] Gregory, S. T. and Knight, J. C. On the provision of backward error recovery in production programming languages. In *Digest of Papers FTCS-19*, pages 506–11, 1989.

[37] Hammer, P. L., editor. *Scheduling under Resource Constraints – Deterministic Models*, volume 7. J. C. Baltzer AG, 1986.

[38] Horning, J. J., Lauer, H. C., Melliar-Smith, P. M., and Randell, B. A program structure for error detection and recovery. In Gelenbe, E. and Kaiser, C., editors, *Lecture Notes in Computer Science 16*, pages 171–187. Springer-Verlag, Berlin, 1974.

[39] Hunt, V. R. and Zellweger, A. The FAA's Advance Automation System: Strategies for future air traffic control systems. *IEEE Computers*, February 1987.

[40] Jalote, P. and Tripathi, S. K. Workshop on integrated approach for fault tolerance — current state and future requirements. *Operating Systems Review*, 24(1), Jan. 1990.

[41] Johnson, D. S., Demers, A., Ullman, J. D., Garey, M. R., and Graham, R. L. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM J. on Computing*, 3(4):299–325, 1974.

[42] Karp, R. M., Vazirani, U. V., and Vazirani, V. V. An optimal algorithm for on-line bipartite matching. In *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing*, pages 352–369, 1990.

[43] Kelly, J. P. J. et al. A large scale second generation experiment in multi-version software: description and early results. In *Digest of Papers FTCS-18*, pages 9–14, 1988.

[44] Kim, K. H. and Welch, H. O. Distributed Execution of Recovery Blocks: An approach for uniform treatment of hardware and software faults in real-time applications. *IEEE Trans. Computers*, 38(5):626–36, May 1989.

[45] Krause, K. L., Shen, V. Y., and Schwetman, H. D. Analysis of several task-scheduling algorithms for a model of multiprogramming computer systems. *J. ACM*, 22:522–550, 1975.

[46] Krishna, C. M. and Shin, K. G. On scheduling tasks with a quick recovery from failures. *IEEE Trans. Computers*, 35(5):448–55, May 1986.

[47] Laprie, J. C. Dependable computing and fault tolerance: concepts and terminology. In *IEEE 1985 Fault Tolerant Computing*, 1985.

[48] Laprie, J.-C. et al. Hardware- and software-fault tolerance: Definition and analysis of architectural solutions. In *Digest of Papers FTCS-17*, pages 116–21, 1987.

[49] Lee, Y.-H. and Shin, K. G. Optimal reconfiguration strategy for a degradable multimodule computing system. *Journal of the ACM*, pages 326–348, April 1987.

[50] Lehoczky, J. P., Sha, L., and Ding, Y. The rate monotonic scheduling algorithm: exact characterization and average case behavior. *Proceedings Real-Time Systems Symposium*, pages 166–171, 1989.

[51] Liestman, A. L. and Campbell, R. H. A fault-tolerant scheduling problem. *IEEE Trans. on Software Engineering*, 12(11):1089–95, November 1986.

[52] Lin, K.-J., Natarajan, S., and Liu, J. W.-S. Imprecise results: utilizing partial computations in real-time systems. *Real-Time System Symposium*, pages 210–217, December 1987.

[53] Liu, C. and Layland, J. Scheduling algorithms for multi-programming in a hard-real-time environment. *J. ACM*, 20:46–61, 1973.

[54] Liu, C. L. *Introduction to Combinatorial Mathematics.* McGraw-Hill, Inc., 1968.

[55] Liu, J. W.-S., Lin, K.-J., and Natarajan, S. Scheduling real-time, periodic jobs using imprecise results. *Real-Time System Symposium*, pages 252–260, December 1987.

[56] Liu, J. W. S., Lin, K.-J., Shih, W.-K., Yu, A. C.-S., Chung, J.-Y., and Zhao, W. Algorithms for scheduling imprecise computations. *IEEE Computer*, May 1991.

[57] Load, A. M. Computer system dependability: An introduction. Technical Report 140, Department of Computer Science, University of Warwick, April 1989.

[58] Locke, C. D. *Best-effort decision making for real-time scheduling.* PhD thesis, Carnegie Mellon University, 1986.

[59] Mathews, J. *Numerical Methods for Mathematics, Science, and Engineering.* Prentice Hall, 1992.

[60] Meyer, J. F. On evaluating performability of degradable computing systems. *IEEE Trans. on Computers*, C-29:720–31, 1980.

[61] Mok, A. K. *Fundamental design problems of distributed systems or hard real-time environment.* PhD thesis, M.I.T., 1983.

[62] Mok, A. K. and Dertouzos, M. L. Multiprocessor scheduling in a hard real-time environment. In *Proceedings of the Seventh Texas Conference on Computing System*, 1978.

[63] Niehaus, D. and Kuan, C.-H. Spring Project Software Generation System. Spring project documentation, University of Massachusetts, June 1990.

[64] Purtilo, J. M. and Jolote, P. A system for supporting multi-language versions for software fault tolerance. In *Digest of Papers FTCS-19*, pages 268–74, 1989.

[65] Rajkumar, R. *Task synchronization in real-time systems.* PhD thesis, Carnegie Mellon University, 1989.

[66] Ramamritham, K. Fault tolerance in Spring: Issues and possibilities. Technical report, Department of Computer Science, University Massachusetts, 1989.

[67] Ramamritham, K. and Stankovic, J. A. Dynamic task scheduling in distributed hard real-time systems. *IEEE Software*, 1(3), July 1984.

[68] Ramamritham, K. and Stankovic, J. A. Scheduling strategies adopted in Spring: An overview. In Tilborg, A. M. V. and Koob, G. M., editors, *Foundation of Real-Time Computing Scheduling and Resource Management*, pages 277–305. Kluwer Academic Publishers, 1991.

[69] Ramamritham, K., Stankovic, J. A., and Shiah, P. O(n) scheduling algorithms for real-time multiprocessor systems. *Proceedings of the International Conference on Parallel Processing*, August 1989.

[70] Ramamritham, K., Stankovic, J. A., and Shiah, P. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Trans. on Parallel and Distributed Systems*, 1, April 1990.

[71] Ramos-Thuel, S., Strosnider, J. K., and Lehoczky, J. P. Performance impact of time redundancy for backward error recovery in real-time workloads. Unpublished, 1990.

[72] Sha, L., Rajkumar, R., Lehoczky, J. P., and Ramamritham, K. Mode change protocols for priority-drive preemptive scheduling. *Real-Time Systems*, Dec. 1989.

[73] Shen, C. and Ramamritham, K. Scheduling in real-time systems. Technical report, Department of Computer Science, University Massachusetts, 1992.

[74] Shih, W.-K., Liu, J. W., and Chung, J.-Y. Fast algorithms for scheduling imprecise computations. *Real-Time System Symposium*, pages 12–19, December 1989.

[75] Sleator, D. and Tarjan, R. Amortized efficiency of list update and paging rules. *CACM*, 28(2):202–208, 1985.

[76] Smith, E. W. Various optimizers for single-stage production. *Naval Research Logistics Quarterlies*, 3:59–66, 1956.

[77] Smith III, T. B. et al., editors. *The fault tolerant multiprocessor computer.* Noyes Publications, 1986.

[78] Sprunt, B., Sha, L., and Lehoczky, J. P. Aperiodic task scheduling for hard real-time systems. *The Journal of Real-Time Systems*, 1:27–60, 1989.

[79] Stankovic, J. and Wang, F. The integration of scheduling and fault tolerance in real-time systems. *Workshop on Imprecise and Approximate Computation*, 1992.

[80] Stankovic, J. A., editor. *Reliable Distributed System Software.* IEEE Computer Society Press, Silver Spring Md., 1985.

[81] Stankovic, J. A. Stability and distributed scheduling algorithms. *IEEE Trans. on Software Engineering*, SE-11(10), 1985.

[82] Stankovic, J. A. Decentralized decision making for task reallocation in a hard real-time system. *IEEE Trans. on Computers*, 38(3), 1989.

[83] Stankovic, J. A. The integration of scheduling and fault tolerance in real-time systems. Technical Report 92-49, Department of Computer Science, University Massachusetts, 1992.

[84] Stankovic, J. A. et al., editors. *Real-Time Systems*, volume 2. Kluwer Academic Publishers, 1990. Special Issue: Real-Time Artificial Intelligence.

[85] Stankovic, J. A. and Ramamritham, K., editors. *Hard real-time systems.* IEEE Computer Society Press, 1988.

[86] Stankovic, J. A. and Ramamritham, K. The design of the Spring kernel. *Real-Time System Symposium*, pages 146–57, December 1989.

[87] Walter, C. J., Kiechhafer, R. M., and Finn, A. M. MAFT: A multicomputer architecture for fault-tolerance in real-time control systems. *Real-Time System Symposium*, pages 133–40, 1985.

[88] Wang, F. and Mao, D. Worst case analysis for on-line scheduling in real-time systems. Technical report. COINS Technical Report 91-54, University of Massachusetts, September 1991.

[89] Wang, F., Ramamritham, K., and Stankovic, J. A. Bounds on the schedule length of some heuristic scheduling algorithms for hard real-time tasks. Technical report. COINS Technical Report 90-67, University of Massachusetts, September 1990, 43 pages.

[90] Wang, F., Ramamritham, K., and Stankovic, J. A. Bounds on the performance of heuristic algorithms for multiprocessor scheduling of hard real-time tasks. *Proceedings Real-Time Systems Symposium*, 1992.

[91] Zhao, W. and Ramamritham, K. Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints. *The Journal of System and Software*, 7:195–207, 1987.

[92] Zhao, W., Ramamritham, K., and Stankovic, J. A. Preemptive scheduling under time and resource constraints. *IEEE Trans. on Computers*, C-36(8):949–960, August 1987.

[93] Zhao, W., Ramamritham, K., and Stankovic, J. A. Scheduling tasks with resource requirements in hard real-time system. *IEEE Trans. on Software Engineering*, SE-13(5), May 1987.