2001

Controlling schedulability-reliability trade-offs in real-time systems

Ra'ed Mohammad S. Al-Omari
Iowa State University

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Controlling schedulability-reliability trade-offs in real-time systems

by

Ra'ed Mohammad S. Al-Omari

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Major Professor: Arun K. Somani

Iowa State University

Ames, Iowa

2001

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Graduate College
Iowa State University

This is to certify that the Doctoral dissertation of

Ra'ed Mohammad S. Al-Omari

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Major Professor

Signature was redacted for privacy.

For the Major Program

Signature was redacted for privacy.

For the Graduate College

DEDICATION

This thesis would be incomplete without mentioning the support that has been given to me by my wife, Maisaa, to whom this thesis is dedicated. She was my own soul, who kept my spirits up when the muses failed me. Without her lifting me up when this thesis seemed interminable, I doubt it should ever have been completed.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Real-time systems are defined as those systems in which the system's performance depends not only on the logical correctness of the result, but also on the time at which the results are produced. Due to the importance of meeting tasks' deadlines in real-time systems most of the existing real-time scheduling algorithms base their decisions on the worst-case estimates of task and system parameters. Since deadline misses occur due to system faults or uncertainties in task parameters, time redundancy is an essential technique to tolerate such misses. Being that time is the most crucial resource in real-time systems, the tendency would be to design and develop new techniques to customize the amount of time redundancy depending on the frequency of faults and the criticality of tasks. Therefore, the main focus of this thesis is to introduce new techniques that offer trade-offs between schedulability (ability of the system to accept more tasks) and reliability (ability of the system to tolerate more faults) in real-time systems. Different techniques have been introduced for different system models. The mechanisms that are used to offer trade-offs in these techniques differ based on the deadline strictness for each model. In this thesis, three different real-time models (hard, soft, and firm real-time systems) have been studied.

For hard real-time systems, the amount of time redundancy can be varied based on the number and type of faults which can be tolerated. Therefore, we propose two new techniques to accommodate more tasks and/or tolerate more faults effectively in hard real-time systems. In the first technique, called *dynamic grouping*, the processors are dynamically grouped into logical groups in order to achieve efficient overloading of resources, thereby improving the schedulability and the reliability of the system. In the second technique, called *Primary-Backup (PB) overloading*, the primary of a task can share/overlap in time with the backup of another task

on a processor. The intuition is that, for a primary (backup), the PB-overloading can assign an earlier *start time* than that of the BB-overloading, thereby increasing the schedulability. The efficiency of these techniques have been measured in terms of the percentage of incoming tasks that they can schedule and the frequency of faults that they can tolerate. For soft real-time systems, the type of redundancy can be varied dynamically based on the frequency of faults and the arriving task's laxity. Therefore, we propose an adaptive scheme that controls the overlap interval between the primary and backup versions of a task based on an estimate of primary fault probability in the system and its (task) laxity. Two variations of the adaptive scheme are proposed by varying the adaptation mechanism. The adaptation can be done in a continuous manner which leads to an approach called primary-backup overlap continuous (PB-OVER-CONT), or it can be in a discrete manner which leads to an approach called primary-backup overlap switch (PB-OVER-SWITCH). The efficiency of these techniques have been measured in terms of the system's utilization and the output value of tasks. For firm real-time systems, the execution time for the incoming tasks can be dynamically estimated based on the system's performance. Therefore, we propose three closed-loop scheduling algorithms that use feedback from *(i)* the deadline miss ratio in the first approach which is called *CL-OVER-MISS*; *(ii)* the task rejection ratio in the second approach which is called *CL-OVER-REJ*; *(iii)* both the miss ratio and rejection ratio in the final approach which is called *CL-OVER-MISSREJ*. These feedbacks are used to efficiently estimate the execution time for the arriving tasks. The efficiency of these techniques have been measured in terms of the percentage of incoming tasks they can schedule and the percentage of scheduling tasks that meet their deadlines.

CHAPTER 1. INTRODUCTION

The consistent decrease in the cost of hardware has led to the employment of computers in many application areas. As a consequence, the complexity of modern computer systems has increased proportionally and more effort is needed to maintain the performability of these computer systems. Those computer systems in which the complexity exists in the dimension of time are called real-time systems. Real-time systems are defined as systems in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the results are produced [84]. Examples of current real-time systems range from very simple micro-controllers in embedded systems to highly sophisticated and complex systems such as air traffic control and avionics.

Real-time systems are different from general purpose computing systems in several ways. Their processes have time related attributes such as ready times, deadlines, computation times and periods. A real-time system must provide predictable response times. Therefore, this requirement makes the worst case behavior of real-time systems more important than the average response time or user convenience, which are important issues for general purpose computing systems. The inter-process communication and synchronization of real-time systems must be bounded and predictable. For example, if a process has to wait before entering a critical section, its waiting time must be bounded. Even the maximum time taken to complete I/O has to be predetermined. Other components of general purpose computing systems are not as important in real-time systems. For example, memory management in real-time systems is often static having all the processes residing in main memory all the time. As a result, memory management is not very important since processes do not have to be swapped into a secondary storage. The file system is also not considered to be a high priority, since disk access times

are too high for most real-time systems. All the main components of real-time systems use resources that have to be managed within time constraints, and thus new scheduling policies are needed for that purpose.

The operating systems community have extensively studied scheduling of processes and tasks without time constraints. However, the popular scheduling policies used in general purpose systems such as first in first out, shortest job next, or round robin with time slice are not appropriate for real-time systems. These scheduling policies attempt to reduce the average response time and do not deal with time constraints. Scheduling policies in real-time systems need to guarantee that tasks will meet their deadlines in all circumstances. The problem of meeting task deadlines is critical in real-time systems because failure to meet task deadlines may result in severe consequences, possibly loss of human life. Scheduling of tasks involves the allocation of processors (including resources) and time in such a way that certain performance requirements are met [70]. The scheduling algorithms have to satisfy not only the time constraints of tasks, but also the resource constraints and/or precedence constraints among tasks, if any. These real-time scheduling schemes can be used for the processors, communications (e.g., for reserving network bandwidth), and other resources used by real-time processes such as sensors, actuators, etc.

Real-time scheduling algorithms fall into two categories: static and dynamic scheduling. In static scheduling [48], the assignment of tasks to processors and the time at which the tasks start execution are determined a priori. Static algorithms are often used to schedule periodic tasks and are not applicable to aperiodic tasks whose arrival times and deadlines are not known *a priori*. Scheduling such tasks requires a dynamic scheduling algorithm. In dynamic scheduling, the scheduling algorithm does not require the complete knowledge of the task set and its constraints. Among the dynamic scheduling algorithms, some of them operate under resource-sufficient environments and others operate under resource-insufficient environments. The former are systems in which the resources are sufficient to a priori guarantee that all the tasks are schedulable [48]. While the system designers try to design the system with sufficient resources, because of unpredictable environments and cost, it is sometimes impossible

to guarantee that system resources are sufficient. Dynamic planning based schedulers, such as the Spring scheduling [82], can dynamically guarantee incoming tasks via on-line admission control and planning.

Due to the importance of meeting task deadlines in real-time systems most of the existing real-time scheduling algorithms base their scheduling decisions on the worst-case estimates of task parameters. When accurate workload models are not available, such an approach can result in a highly underutilized system based on an extremely pessimistic estimation of the workload. Unfortunately, many real world complex applications, such as robotics and agile manufacturing, are dynamic and operate in a non-deterministic environment wherein the workload cannot be accurately modeled. In this case, it is preferable to base scheduling decisions on average execution time and to be ready to deal with bounded transient overloads dynamically. This approach is especially preferable in firm/soft real-time systems as it provides a firm performance guarantee in terms of deadline misses while achieving high utilization and throughput (guarantee ratio) at the same time. In an unpredictable environment, it is impossible for a system to achieve 100% utilization and a 0% deadline miss ratio all the time and a trade-off between deadline miss ratio (reliability) and utilization (schedulability) is unavoidable.

Moreover, real-time tasks may miss their deadline due to the effects of the operating environment on the system. Since the environment may generate various kinds of faults, it is essential to incorporate fault tolerance when a real-time system is designed. A system is fault-tolerant if it continues to perform its specified tasks in the presence of hardware failures or software errors [30]. A fault-tolerant system has to ensure that faults in the system (which are defects in hardware or software) do not lead to failure (which is the non-performance of some action that is due or expected). Fault tolerance is achieved through the use of redundancy, which is the addition of information, resources, or time beyond what is needed for normal system operation [30, 88]. The reliable execution of a task is usually achieved by scheduling multiple versions of the task (time redundancy) [32, 35, 37, 54, 63, 66, 88]. Thus, a trade-off between the schedulability (the number and the value of the accepted tasks) and the reliability

(the number and the frequency of faults tolerated) in the system is unavoidable. Most of the existing fault-tolerant scheduling algorithms for real-time systems tend to favor schedulability over reliability or vise versa. This may not be desirable when tasks have wide-varying utility values. This motivates the need for new techniques that offer trade-offs between schedulability and reliability depending on the frequency of faults and the criticality of the tasks. So the main focus of this thesis is to introduce these techniques for real-time systems.

Specifically, in this thesis, we first propose two techniques to improve the schedulability and/or the reliability of hard real-time systems. These techniques offer trade-offs between the percentage of tasks being scheduled and the frequency of faults being tolerated in hard real-time systems. Secondly, we propose an adaptive primary-backup scheduling scheme that makes use of an estimate of primary fault probability and a task soft laxity to control the degree of overlap between its (task) versions in soft real-time systems. This adaptive technique offers trade-offs between the percentage of tasks being scheduled and the total utility of their output to the soft real-time system. Finally, we use the feedback control theory to design a closed-loop scheduling algorithm for firm real-time systems. This closed-loop scheduling algorithm offers a trade-off between the percentage of tasks being rejected and the percentage of tasks that missed their deadline.

The rest of this thesis is organized as follows. In Chapter 2, real-time systems, scheduling in real-time, and fault-tolerance in real-time are discussed and their classification is stated. In Chapter 3, the goals, motivation, scope, and system models of this thesis are stated. In Chapter 4, efficient overloading techniques for primary-backup scheduling in hard real-time systems are proposed and analyzed. In Chapter 5, an adaptive scheme for fault-tolerant scheduling of tasks in soft real-time system is proposed and analyzed. In Chapter 6, new algorithms for open-loop and closed-loop scheduling of firm real-time tasks based on execution time estimation are proposed and analyzed. Finally in Chapter 7, the concluding remarks are made.

CHAPTER 2. BACKGROUND

This chapter discusses previous research done in areas relevant to this thesis. In Section 2.1, real-time system is defined and its classifications are stated. In Section 2.2, scheduling in real-time is defined, its classifications are stated, and the previous research done in real-time scheduling is discussed. In Section 2.3, fault-tolerance in real-time is defined, the classifications of real-time fault-tolerant scheduling algorithm are stated, and the previous research done in real-time fault-tolerant scheduling is discussed.

2.1 Real-Time Systems

Real-time systems are computing systems that must react within precise time constraints to events in the environment. A real-time system usually has a mission to achieve. A typical real-time system consists of a *controlling system*, a *controlled system*, and the *environment* as shown in Figure 2.1. The controlling system is a computer(s) which acquires information about the environment through input devices (sensors), performs certain computations on the

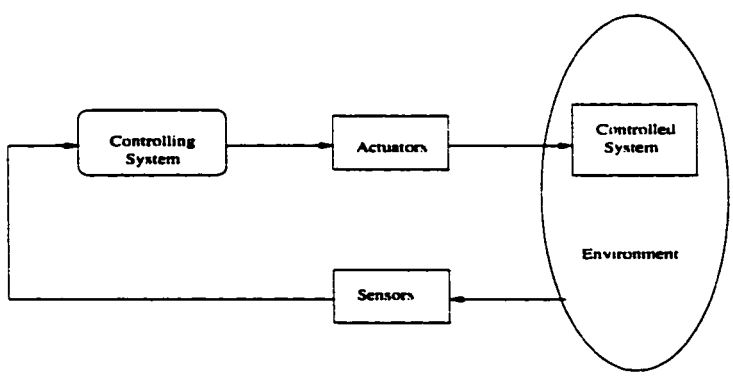


Figure 2.1 A typical real-time system

data, and activates the actuators through some controls. Since a mismatch between the state of the environment that is perceived by the controlling system and the actual state of the environment can lead to disastrous results, the controlling system must periodically monitor the environment as well as timely process the sensed information. Thus, *time* is the most precious resource to manage in real-time systems.

An example for a typical real-time system, is a mission to reach an intended destination safely by a car. Let us assume a computer is driving the car. Therefore, the controlling system is the computer, the controlled system is the car, the operating environment is the road conditions and other cars on the road, the sensors are the cameras, infrared receiver, and laser telemeter, and the actuators are wheels, engines, and brakes of the car. The various controls through which the actuators can be controlled are the accelerator, steering wheel, and break-pedal.

2.1.1 Real-Time Tasks

The computations that occur in a real-time system that have timing constraints are called *real-time tasks*. The computer should execute the real-time tasks so that each task will meet its timeliness requirement. In other words, a real-time task should finish its execution before a certain time, called the *deadline*. A real-time application is usually combined from a set of cooperating tasks, which are activated at regular intervals and/or on particular events. A task typically senses the state of the system, performs certain computations, and if necessary sends commands to change the state of the system. Tasks in real-time systems are divided into two types, *periodic* tasks and *aperiodic* tasks [79].

2.1.1.1 Periodic tasks

Periodic tasks are time-driven and reoccur at regular intervals called *period*. Characteristics of a periodic task such as its *period* and *worst-case computation time* are known a *priori*. Most of the sensory processing tasks are periodic. Some periodic tasks exist from the point of system initialization while others may come into existence dynamically. The temperature monitor of

a nuclear reactor is an example of a permanent periodic task. An example of a dynamically created periodic task is the computation in a radar, which monitors a particular flight. This task is activated when the aircraft enters an air traffic control region and is terminated when the aircraft leaves the region.

2.1.1.2 Aperiodic tasks

Aperiodic tasks are event-driven and activated only when certain events occur. Characteristics of aperiodic task such as its *ready time*, worst-case computation time, and *deadline* are known only when it arrives. Aperiodic tasks with known minimum inter-arrival time are known as *sporadic tasks*. Most of the dynamic processing tasks are aperiodic. For example, in an aircraft control system the controllers often activate tasks, which are aperiodic, depending on what appears on their monitor. Similarly in an industrial control system, the robot that monitors and controls various processes may have to perform path planning dynamically which results in activation of aperiodic tasks. Another example of a system having aperiodic tasks is the system that monitors the condition of several patients in an intensive care unit of a hospital, in which an action has to be taken as soon as the condition of the patient changes.

In addition to timing constraints, tasks in a real-time application can also have other requirements that are common to tasks in traditional non-real-time applications [79]. A task may require accessing certain resources other than the processor such as I/O devices, communication buffers, data structures, and files. Accesses to a resource can either be in a shared mode or in an exclusive mode. In the shared mode, more than one task are allowed to access a resource, whereas in the exclusive mode, at most one task can access the resource at any point of time. A *resource constraint* exists between two tasks if both of them require the same resource and one of the accesses is exclusive. A task can have *precedence constraints* with other tasks, in which it requires the results of the other tasks before its execution can be started or the other tasks require the result of this task before their execution can be started. Another important requirement of tasks is the *fault-tolerant requirement* which is necessary for the continuous operation of the system even in the presence of faults.

2.1.2 Real-Time Systems Classifications

There are various real-time system classifications that depend on the following system dimensions: (i) deadlines strictness; (ii) tasks characteristics; and (iii) system size.

2.1.2.1 Deadlines strictness

Real-time systems are broadly classified into three categories based on the nature of deadline as shown in Figure 2.2, namely *(i) hard real-time systems* in which the consequences of not executing a task before its deadline may be catastrophic, *(ii) firm real-time systems* in which the results produced by the corresponding task ceases to be useful as soon as the deadline expires but the consequences of not meeting the deadline are not very severe, and *(iii) soft real-time systems* in which the utility of the results produced by a task decreases over time after the deadline expires [79].

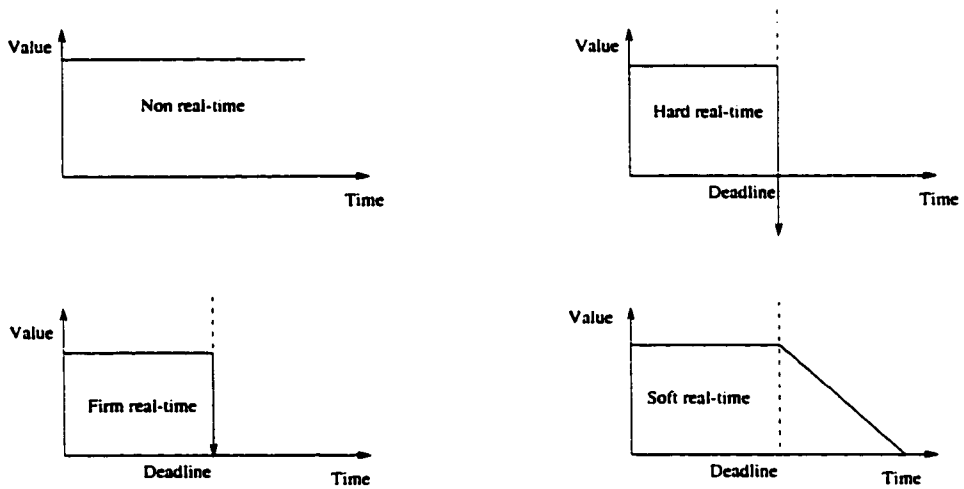


Figure 2.2 Classification of real-time systems based on deadlines strictness

Avionics control and nuclear plant control are examples of hard real-time systems. Online transaction processing applications such as airline reservation and banking are examples of firm real-time systems, and the telephone switching system and image processing applications are examples of soft real-time systems.

2.1.2.2 Tasks' characteristics

Real-time systems can be classified as *static* or *dynamic* based on the arrival times of the tasks. If the arrival times of all the tasks are known before the system starts its operation, then the system is *static*. However, if tasks arrive into the system during its operation and arrival times are not known in advance, then the system is said to be *dynamic*. Most often static systems consist of periodic tasks performing control operations while dynamic systems need to schedule periodic or aperiodic tasks which are generated by external events.

2.1.2.3 System size

Real-time systems can be classified as *uniprocessor* or *multiprocessor* based on the size of the system. The uniprocessor system, as shown in Figure 2.3a, uses only one processor that runs a scheduling algorithm and tasks. The multiprocessor model uses n identical processors that communicate through shared memory (parallel system) as shown in Figure 2.3b or message passing (distributed system) as shown in Figure 2.3c. The parallel multiprocessor systems usually has a separate task scheduling processor. This processor is informed of all task arrivals. It determines whether or not a new task can be scheduled, and maintains a global schedule. In a distributed system, nodes are connected through an interconnection network and each node can either be a uniprocessor or a multiprocessor.

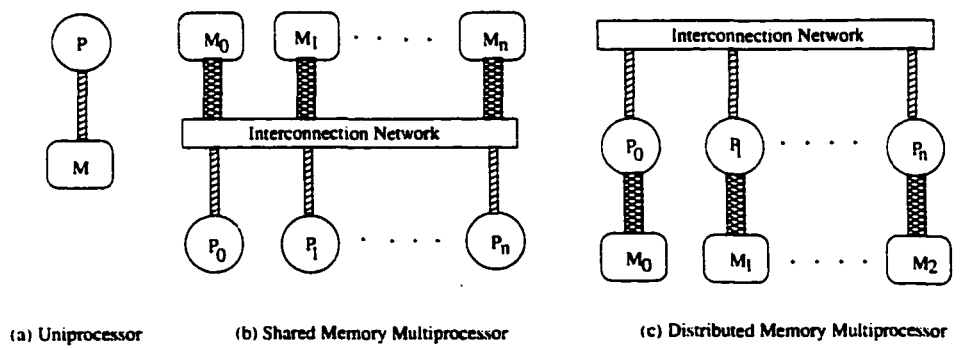


Figure 2.3 Classification of real-time systems based on system size

2.2 Scheduling Algorithms in Real-Time Systems

Scheduling and resource allocations in real-time systems are difficult problems due to the timing constraints of the tasks involved. The order in which the tasks are scheduled or dispatched has a large effect on the chances of the tasks meeting their timing constraints. Many of the real-time scheduling problems are known to be NP complete [21]. A great deal of research has been conducted for scheduling tasks in a variety of real-time system models. In [70] and [10], good reviews of scheduling in real-time systems can be found, while a summary of scheduling results for real-time systems can be found in [81].

2.2.1 Classifications of Real-Time Scheduling Algorithms

Among the great variety of algorithms proposed for scheduling real-time tasks, the following main classes can be identified [13], as shown in Figure 2.4.

- **Preemptive.** With preemptive algorithms, the running task can be interrupted any time to assign the processor to another active task, according to a predefined scheduling policy.

- **Non-preemptive.** With non-preemptive algorithms, the processor executes a task until completion, once its started. In this case, all scheduling decisions are taken as a task terminates its execution.

- **Static.** Static algorithms are those in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation.

- **Dynamic.** Dynamic algorithms are those in which scheduling decisions are based on dynamic parameters that may change during system evolution.

- **Off-line.** We say that a scheduling algorithm is used off-line if it is executed on the entire task set before actual task activation begins. The schedule generated in this way is stored in a table and later executed by a dispatcher.

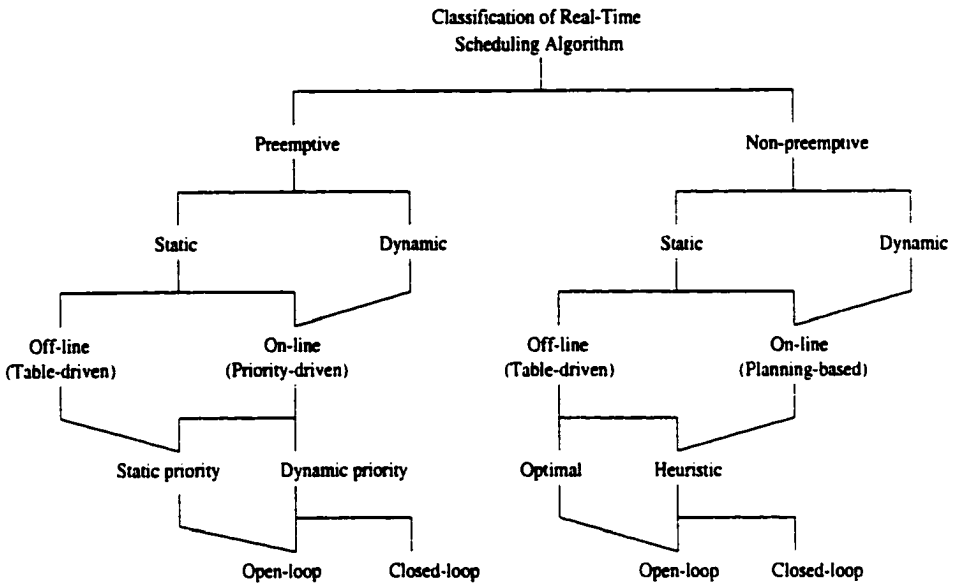


Figure 2.4 Classifications of real-time scheduling algorithms

- **On-line.** We say that a scheduling algorithm is used on-line if scheduling decisions are taken at runtime every time a new task enters the system or when a running task terminates.

- **Optimal.** An algorithm is said to be optimal if it minimizes some given cost function defined over the task set. When no cost function is defined and the only concern is to achieve a feasible schedule, then the algorithm is said to be optimal if when it fails to meet a deadline, there are no other algorithms of the same class that can meet it.

- **Heuristic.** An algorithm is said to be heuristic if it tends toward but does not guarantee to find the optimal schedule.

- **Open-loop.** An algorithm is said to be open-loop if its scheduling decisions are simply based on the worst-case estimates of task's parameters.

- **Closed-loop.** An algorithm is said to be closed-loop if its scheduling decisions are based on nominal values of the task parameter and if it uses a feedback mechanism to dynamically adjust the task and/or scheduler parameters in case of bounded transient overloads.

2.2.2 Preemptive vs Non-preemptive Scheduling

Real-time scheduling algorithms can be classified into two major distinct models: preemptive, and non-preemptive. Preemptive algorithms assume that any task can be interrupted during its execution, while non-preemptive algorithms do not allow a running task to be interrupted. Both classes of algorithms have certain advantages and disadvantages. Mathematical tools are available for the analysis of preemptive algorithms [33] and these algorithms can be implemented on most modern computer systems [34]. In such algorithms, a higher priority task is never delayed by a lower priority task. However, preemption adds to the overhead of the system, and also adds an aspect of unpredictability that might make the system harder to validate. An example for real-time systems that have implemented preemptive scheduling are RT-Mach [87] and LynxOS [53]. Non-preemptive algorithms are usually based on heuristics, but are easy to implement [34] and can be used as a technique for concurrency control [10]. Non-preemptive algorithms can also be used more easily when tasks have resource requirements besides the processor. An example for real-time systems that use non-preemptive scheduling are Spring [82], Maruti [74] and MAFT [31].

2.2.3 Static vs Dynamic Scheduling

In static algorithms [68], the assignment of tasks to processors and the time at which the tasks start execution are determined a priori. Static algorithms are often used to schedule periodic tasks with hard deadlines. The main advantage is that, if a solution is found then, one can be sure that all deadlines will be met. However, this approach is not applicable to aperiodic tasks whose arrival times and deadlines are not known a priori. Scheduling such tasks in a real-time system requires a dynamic scheduling algorithm [56, 71]. Two static scheduling approaches was proposed in the literature:

- **Static table-driven approaches:** These perform static schedulability analysis and the resulting schedule, usually stored in the form of a table, is used at run time to decide when a task must begin execution. The main motivation for static table-driven scheduling is the fact that the resources needed to meet the deadlines of safety-critical tasks must be

pre-allocated so that each individual task can meet its deadline even under the worst case conditions. This is mainly applied to periodic tasks. For a set of periodic tasks, there exists a *feasible schedule* if and only if all the task instances are feasible in the schedule for a time duration equal to the Least Common Multiple (LCM) of the task periods (also known as the planning cycle) [43].

- **Static priority-driven preemptive approaches:** These perform static schedulability analysis but no explicit schedule is constructed. At run time, tasks are executed in highest-priority-first order. In this approach, each task is assigned a priority. The priority assignment is related to the time constraints associated with the task and this can be either static or dynamic. Period, deadline, and laxity of the tasks are commonly used to assign priorities to them. In [48], two ways of assigning priorities have been proposed: (i) static priorities based on the task period, where the task with the shortest period has the highest priority and (ii) dynamic priorities based on the task deadline, where the task with shortest deadline has the highest priority. Static priorities, once assigned will not change, whereas the dynamic priorities may change with new tasks arrival. Thus, the dynamic priorities may have to be recomputed with every new task arrivals. This makes dynamic priority assignment more expensive compared to static priority assignment in terms of run time overheads. The Rate Monotonic Scheduling (RMS) [41, 42, 48] is an example of static priority scheme. Moreover, Earliest Deadline First (EDF) [48] and Least Laxity First (LLF) are examples of dynamic priority schemes.

In dynamic scheduling, when a new set of tasks arrives at the system, the scheduler dynamically determines the feasibility of scheduling these new tasks without jeopardizing the guarantees that have been provided for the previously scheduled tasks. The objective of a dynamic real-time scheduling algorithm is to improve the *guarantee ratio* which is defined as the percentage of tasks, that arrived into the system, whose deadlines are met. Two dynamic scheduling approaches were proposed in the literature:

- **Dynamic planning based approaches:** Unlike the static approaches, schedulability of a task is checked at run time, i.e., dynamically arriving task is accepted for execution

if it is found to be schedulable. Such a task is guaranteed to meet its performance requirements. Dynamic planning combines the flexibility of dynamic scheduling with the predictability offered by schedulability checking [70]. When a task arrives, an attempt is made to guarantee the task by constructing a plan for its execution without violating the guarantees of the previously scheduled tasks. The schedulability check for a task takes into account the worst-case computation time, timing and/or precedence constraints, and resource and/or fault-tolerant requirements of the task. Once a task passes the schedulability check, it is admitted into the system, and if the assumptions about its characteristics hold, it will meet its deadline. Thus, predictability is checked for each task on arrival. If the schedulability check for a task fails, then the task is not feasible and a *timing fault* is forecasted. If this is known sufficiently ahead of the deadline, there may be time to take alternative actions.

- **Dynamic best effort approaches:** In dynamic best effort scheduling, schedulability check is not performed while admitting a new task into the system. In some sense, every task arrived in the system is admitted and the scheduler tries its best to guarantee the execution of the task. Therefore, best effort scheduler cannot ensure predictability of the tasks' executions. In this approach, tasks are assigned priorities based on RMS, EDF, or LLF policies. Task execution occurs in priority order, and currently executing low priority task is preempted by a high priority task. A task is being executed, possibly with preemption, until it completes or its deadline has reached, whichever happens first. Another major disadvantage of the dynamic best effort approach is its poor performance under overload conditions [11].

2.2.4 Optimal vs Heuristic Scheduling

The general problem of optimal scheduling (finding a schedule whenever one exists) of non-preemptive tasks on a uniprocessor or a multiprocessor system is NP-complete [21]. Therefore, different heuristics have been used to schedule non-preemptive real-time tasks with the aim of maximizing performance measures such as schedulability and processor utilization.

Research has been conducted in developing heuristics and conditions for scheduling real-time tasks both on uniprocessor and multiprocessor systems. For a uniprocessor, non-preemptive versions of EDF and LLF scheduling policies have been studied in [92]. A heuristic approach for solving the problem of dynamically scheduling tasks in a real-time system where tasks have deadlines and resource requirements is described in [94]. The heuristic function used to select the next task to be scheduled uses three factors that consider information about real-time constraints of tasks and their utilization of resources. For multiprocessor systems, various heuristics for scheduling non-preemptive tasks have been developed in [71]. The tasks are scheduled one at a time, with a heuristic function directing the search for a feasible schedule helping to choose the next task to be added to the schedule. The heuristics include functions for minimum deadline first, minimum computation time first, minimum ready time first, and other combinations of these attributes.

2.2.5 Open-Loop vs Closed-Loop Scheduling

Despite the significant body of results in these paradigms of real-time scheduling, many real world problems are not easily supported. While algorithms such as EDF, RMS and Spring scheduling can support sophisticated task set characteristics, they are all "open-loop" scheduling algorithms. Open-loop refers to the fact that the scheduling decisions are simply based on the worst-case estimates of task parameters. They neither continuously observe the operation of the system in terms of tasks' execution nor adjust the system parameters accordingly. In recent years, the "closed-loop" scheduling [83, 51, 7] has gained importance due to its applicability to many real-world problems wherein the feedback information can be exploited efficiently to adjust the task and/or scheduler parameters, thereby improving system's performance.

Even though, open-loop scheduling algorithms perform well in static or dynamic systems in which the workloads (i.e., task sets) can be accurately modeled, they perform poorly in unpredictable dynamic systems, i.e., systems whose workloads cannot be accurately modeled. Systems with open-loop schedulers are usually designed based on *worst-case* workload parame-

ters. When accurate workload models are not available, such an approach can result in a highly underutilized system based on an extremely pessimistic estimation of workload. Unfortunately, many real world complex applications, such as robotics and agile manufacturing, are dynamic and operate in a non-deterministic environment wherein the workload cannot be accurately modeled. In many cases, it is preferable to base scheduling decisions on average execution time and to be ready to deal with bounded transient overloads dynamically. This approach is especially preferable in firm/soft real-time systems as it provides a firm performance guarantee in terms of deadline misses while achieving high utilization and throughput (guarantee ratio) at the same time.

2.3 Fault-Tolerance in Real-Time Systems

As mentioned earlier, tasks in real-time systems are, by definition, critical in nature. In applications such as space shuttles and nuclear power plant controllers, it is vitally important that all tasks meet their deadlines under all circumstances. These circumstances include faults generated by various factors including electromagnetic radiation in space, power fluctuations, etc. Fault-tolerance is informally defined as the ability of a system to deliver the expected service even in the presence of faults. A common misconception about real-time computing is that fault-tolerance is orthogonal to real-time requirements [79]. It is often assumed that the dependability requirements of a system can be addressed independent of its timing constraints. This assumption, however, does not consider the distinguishing characteristic of real-time systems which is the *timeliness* of correct results. In other words, a real-time system may fail to function correctly either because of faults in its hardware and/or software (physical faults) or because of not responding in time (timing faults) due to overload conditions. Hence, to avoid the catastrophic consequences of missing deadlines, it is essential that real-time tasks meet their deadlines even in the presence of faults and/or overload conditions.

2.3.1 Tolerating Physical Faults in Real-Time Systems

Researchers of the real-time and fault tolerance communities have recognized the need for fault tolerance in real-time systems. In [79], the authors stated that "a real-time system can be viewed as one that must deliver the expected service in a timely manner even in the presence of faults". In [39], the authors articulated the need for fault tolerance in real-time systems by saying that real-time systems "must be sufficiently fault-tolerant to withstand losing large portions of the hardware or the software and still perform critical functions". The interest in this area of research is also demonstrated by the collections of papers on *responsive systems* [20] and *ultra-dependable systems* [86], which are systems incorporating both real-time and fault-tolerance aspects. Examples for real-time systems that can tolerate faults are the MAFT system [31], the MARS system [35], the Maruti operating system [74], and the HARTS [77]

2.3.1.1 Fault and redundancy

A fault is a defect or a flaw that occurs within some hardware or software component. Hardware faults can be of three kinds: *permanent*, *transient* or *intermittent*. Permanent faults are caused by total failure of a computing unit. Transient faults are temporary malfunctions of the computing unit or any other associated components, which cause an incorrect result to be computed. Intermittent faults are repeated occurrences of transient faults. Software faults are caused by errors in algorithm logic, or by an incorrect input.

Faults are tolerated by using redundancy [80], which are of three kinds: *hardware redundancy*, *software redundancy* and *time redundancy*. Hardware redundancy is the addition of extra hardware to the system, such as spare processors that are used if one of the running processors fail. Software redundancy is the use of extra software modules to verify the results, or to use multiple versions of a program. Time redundancy is the use of additional time to perform the functions of a system. This time might be used to re-execute a faulty task or to execute a different version of the task (thus combining software and time redundancy). Permanent faults are tolerated by using hardware redundancy, while transient and intermittent faults can be handled using hardware or time redundancy. Software faults are tolerated using

software redundancy.

Even though various kinds of faults can be tolerated by adding redundancy to the system, simply adding redundancy is not sufficient. The additional resources have to be managed such that all timing constraints are met, and faults are guaranteed to be tolerated. To manage the available resources (processors) in order to achieve timing and fault tolerance guarantees, specialized scheduling algorithms are required.

2.3.2 Real-Time Fault-Tolerant Scheduling Algorithms

The goal of a fault-tolerant scheduling algorithm is to guarantee the recovery of real-time tasks that do not generate correct outputs due to faults in the system. This is done, while scheduling a task, by ensuring the availability of sufficient time to allow for the re-execution of the faulty task within its deadline without violating the timing constraints of any other task. Such time can be reserved on the same processor if only transient faults are to be tolerated, and has to be scheduled on a different processor if permanent faults are also to be tolerated. To tolerate software faults, instead of re-executing the same task within the reserved time, the system runs a different version of the task. Two major techniques have evolved for fault-tolerant scheduling of real-time tasks: N-Version Programming (NVP) [18], and Recovery Blocks (RB) [72].

2.3.2.1 N-version programming

In an N-Version Programming (NVP) approach [25, 57, 88, 91, 95], multiple identical processors execute concurrently different versions of the same task, and the results produced by these processors are voted on. The voter compares the outputs to determine the correct output using, for example, the majority vote. This technique is a combination of software redundancy and time redundancy since extra time is needed in addition to extra software to schedule the different versions. Due to the time redundancy incorporated in N version programming, this technique can also be used to tolerate transient faults. The voter in the NVP approach is assumed to be reliable, possibly by employing redundancy. This approach is

based on the principle of design diversity, which means that multiple versions of the same task are created by employing different algorithms, languages, and/or programming teams.

2.3.2.2 Recovery blocks

Recovery Blocks (RB) uses multiple alternates to perform the same function. One version of a task is primary, and the others are secondary. When the primary task completes execution, its outcome is checked by an acceptance test. If the output is not acceptable, a secondary task executes after undoing the effects of the primary. The secondary versions are executed until either an acceptable output is obtained or the alternates are exhausted, or the deadline of the task is missed. The acceptance tests are usually sanity checks. In other words, they make sure that the output is within a certain acceptable range or that the output does not change more than the allowed maximum rate. Selecting the range for the acceptance test is crucial. If the allowed ranges are too small, the acceptance tests will label the correct outputs as bad. If they are too large, the probability that incorrect outputs will be accepted as correct will be more [38]. In the RB scheme, the versions of tasks are executed serially as opposed to the concurrent execution in the NVP. In the RB scheme, when the number of secondary (backup) copies of every task is one, it is called Primary-Backup (PB) approach. Several implementations of this approach for preemptive and non-preemptive scheduling algorithms can be found in [4, 24, 37, 46, 47, 56, 59, 63, 89, 96].

2.3.3 Tolerating Timing Faults in Real-Time Systems

Timing faults in real-time systems are the faults caused by tasks missing their deadlines. Timing faults usually occur due to transient overload in the system. Overloading in real-time system occurs due to uncertainties in task parameters, component failures, or unpredictable dynamic arrival of tasks. In this section we will discuss different ways in which task scheduling can be made flexible and adaptive to changes and uncertainties in task parameters. A key issue for this is the ability to dynamically adjust task parameters. Reasons for the adjustments could for example be to improve the performance in overload situations or to dynamically optimize

control performance. Examples of task parameters that could be modified are periods and deadlines. One could also allow the execution time for a task to be varied. In order for this to be realistic, the controllers must be capable of supporting dynamically changing execution times. Changes in the task period and in the execution time both have the effect of changing the utilization that the task requires. Four meager techniques have evolved for avoiding timing faults during transient overloads in real-time systems; period skipping [67]; period adjustments [12, 40, 61, 75], imprecise computations [49]; and feedback scheduling [51, 83].

2.3.3.1 Period skipping

A simple task attribute adjustment is to skip an instantiation of a periodic task. This is equivalent to require that the task period should be doubled for this particular instantiation, or that the maximum allowed execution time should be zero. Scheduling in systems that allow skips is treated in [36] and [67]. The latter paper considers scheduling which guarantees that at least m out of k instantiations will execute. (m,k)-firm deadline model provides scheduling flexibility by trading off the quality of the result in order to meet task (message) deadlines [28]. A slightly different motivation for skipping samples is presented in [15, 16]. Here the main objective is to use the obtained execution time to enhance the responsiveness of aperiodic tasks.

2.3.3.2 Period adjustments

Adjustment of task periods has been suggested by many authors. For example, in [40] the authors proposed a load-scaling technique to gracefully degrade the workload of a system by adjusting the task periods. Tasks are assumed to be equally important and the objective is to minimize the number of fundamental frequencies to improve schedulability under static priority assignments. In [61], a system is presented that increases the period of a task whenever the deadline of the task is missed. In [44], a number of policies to dynamically adjust task rates in overload conditions are presented. In [62], it is shown how a multimedia activity can adapt its requirements during transient overloads by scaling down its rate or its computational

demands.

One of the recently proposed models for period adjustment is the *elastic task model* for periodic tasks [12]. In which, each task is characterized by five parameters: computation time C_i, a nominal period T_{i_0}, a minimum period $T_{i_{min}}$, a maximum period $T_{i_{max}}$, and an elasticity coefficient $e_i \geq 0$. A task may change its period within its bounds. When this happens the periods of the other tasks are adjusted so that the overall system is kept schedulable. An analogy with a linear spring is used, where the utilization of a task is viewed as the length of a spring that has a given rigidity coefficient $(1/e_i)$ and length constraints. The elasticity coefficient is used to denote how easy or difficult it is to adjust the period of a given task (compress the string). Another recent work that allows task-period changes in multiprocessor systems has been done in [78] by making on-line use of the proposed off-line method in [75]. A performance index for the control tasks is used to determine the value to the system of running a given task at a given period. The index is weighted for the task's importance to the overall system.

2.3.3.3 Imprecise computations

The imprecise computation model [8, 12, 49, 52, 78] provides scheduling flexibility by trading off the quality of the results in order to meet the task deadlines. This model is suitable for the case when the tasks can be described as "any-time algorithms", i.e., algorithms that always generate a result (provide some QoS) but where the quality of the result (the QoS level) increases with the execution time of the algorithm. In this model, a task is divided into mandatory and optional parts. The mandatory part must be completed before the task's deadline for acceptable quality of results. The optional part, which can be skipped in order to conserve system's resources, refines the result.

A task is said to have produced precise results if it has executed its mandatory as well as optional parts before its deadline. Otherwise it is said to have produced imprecise (i.e., approximate) results when it executes the mandatory part alone. There are two types of imprecise computational tasks, namely, monotone tasks and 0/1 constraint tasks. A task is

monotone if the quality of its intermediate result does not decrease as it executes longer. An imprecise task with 0/1 constraint requires that the optional part is either fully executed or not at all.

2.3.3.4 Feedback scheduling

The idea of feedback has been used informally in scheduling algorithms for applications where the dynamics of the computation workload cannot be characterized accurately for a long time. For example, the VMS operating system [9] uses multi-level feedback queues to improve system throughput. Internet protocols use feedback to help solve the congestion problems. Recently, in [85] the authors presented a feedback-based scheduling scheme that adjusts CPU allocation based on application-dependent progress monitors (e.g., the fill-level of buffers). In the area of real-time databases, in [29] the authors proposed *Adaptive Earliest Deadline* (AED), a priority assignment policy based on EDF. In order to stabilize the performance of EDF under overload conditions, AED features feedback control loop that monitors transactions' deadline miss ratio and adjusts transaction priority assignments accordingly. Recently, under the title of quality of service in multimedia scheduling, the idea of feedback has been used for traffic flow control (e.g. ABR service in ATM networks). Several papers [1, 3, 45] presented feedback control architectures and algorithms for QoS control in communication systems. In [1], the authors used a PI-controller and web-content adaptation mechanism for web server QoS resource allocation (in terms of hit rate and bandwidth).

Thus far very little work has been done in the area of real-time feedback scheduling. Both in [75] and [73], the authors proposed to integrate the design of the system controller with the scheduling of real-time control systems. Both papers aim at providing design tools that enable control engineers to take into consideration scheduling in the early stage of the control systems design. In [17, 49], the elastic task model is used to design an elastic control approach. This approach integrates the *continuous design with digitization* (CDD) [17] and the *direct digital design* (DDD) [49] with adjustable frequencies. A notable work in real-time feedback scheduling is [51, 83] where the authors propose the use of a PID controller to design an on-line scheduler,

called Feedback Control-EDF (FC-EDF). Here, the measured signal (the controlled variable) is the deadline miss ratio for tasks and the control signal is the requested CPU utilization. Changes in the requested CPU utilization are effectuated by two mechanisms (actuators). In the first mechanism, an admission controller is used to control the flow of workload into the system. In the second mechanism a service level controller is used to adjust the workload into the system. A simple liquid tank model is used as an approximation for the scheduling system. Recently, in [11] the authors proposed an approach in which task periods can be dynamically adjusted based on the current load. Where the load is estimated by monitoring the actual computation time of each task. If the estimated load is found to be greater than a certain threshold (e.g. 1 under EDF), then the elastic theory is used to enlarge the task periods to find a feasible configuration.

CHAPTER 3. OVERVIEW OF THE DISSERTATION RESEARCH

This chapter presents an overview of the thesis. It includes a general description of various concepts, terms and definitions applicable through the rest of this thesis.

In Section 3.1, the motivations for this thesis are stated. In Section 3.2, the scope of this research and its justification are discussed. In Section 3.3, the goals and the approaches of this research are stated. Finally, in section 3.4 the system models are presented.

3.1 Motivations of the Dissertation Research

In the previous chapter, the terminology and classifications related to the areas of scheduling in real-time systems and fault tolerance were defined. From this discussion, it is clear that *time redundancy* is a major and an important technique for tolerating faults in real-time systems. Since time is the most crucial resource in real-time systems, the tendency would be to design and develop new techniques to customize the amount of time redundancy depending on the frequency of occurrence of faults and the criticality of the tasks. Therefore, the main focus of this thesis is to introduce new techniques that offer trade-offs between schedulability and reliability in real-time systems. This thesis consists of three major parts. In the first part (Chapter 4), we propose techniques to improve the schedulability and/or the reliability of hard real-time systems. In the second part (Chapter 5), we propose an adaptive scheme to control the degree of overlap between task versions in soft real-time systems. Finally, in the third part (Chapter 6), we use feedback control theory to design a closed-loop scheduling algorithm for firm real-time systems.

One major advantage of the techniques presented in this thesis is that they can be used in the existing dynamic scheduling algorithms without any major change in the scheduling

strategy. Also these techniques do not increase the complexity of the scheduling algorithms. Therefore, the cost of adding these techniques to an existing system is affordable, and can be readily accomplished in a wide range of real-time systems currently in use.

3.1.1 Motivation for Grouping and Overloading in Hard Real-Time Systems

Due to the critical nature of tasks in a hard real-time system, it is essential that every task admitted in the system completes its execution even in the presence of faults. One of the approaches that is used for fault-tolerant scheduling of real-time tasks is the Primary-Backup (PB) model, in which two versions of a task are scheduled on two different processors and an acceptance test is used to check the correctness of the execution result [24, 32, 57, 72]. The backup version is executed only if the output of the primary version fails the acceptance test, otherwise it is deallocated from the schedule. In the context of scheduling, the term "overloading" refers to scheduling of more than one task (version) on the same/overlapping time interval on a processor. Fault-tolerant scheduling algorithms [23, 24, 57] have employed overloading techniques as a mean to conserve system resources, thus improving the schedulability of the system. For PB-based fault-tolerant scheduling, a technique called *backup overloading*, which we call it Backup-Backup (BB) overloading was proposed in [23, 24] and was improved in [57]. In BB-overloading, two or more backups can be overloaded.

In [23, 24], a backup has $n - 1$ choices for overloading with another backup (except the processor onto which its primary is scheduled), where n is the number of processors in the system. Thus, the number of backups that could potentially be overloaded (in a time slot) is $n - 1$. Although this algorithm has a potential to offer higher schedulability due to its maximum flexibility in overloading, it can tolerate at most one failure at any point of time. This is too optimistic and becomes unrealistic for larger n as $MTTF$ becomes smaller.

In [57], an algorithm was proposed to statically divide the system processors into disjoint logical groups and allow backup overloading to take place only within the group. That is, the processors onto which the backups are overloaded/scheduled belong to the same logical group where their primaries are scheduled. Note that, the number of backups that can be overloaded

in a time slot is $k-1$, where k is the number of processors in a logical group. Although this algorithm restricts the flexibility of overloading, it allows at most g faults in the system (where g is the number of logical groups) one fault in each logical group, thus improving the reliability of the system. In other words, static grouping introduces a trade-off between the system's utilization (schedulability) and reliability. Here, since the group size is taken to be 3 or 4, the assumption of having only one fault at a time in a group is reasonable.

To remove the assumption of one fault at a time as in [23] and the disadvantages of static grouping [57], we propose, in Chapter 4, a new technique called *dynamic grouping* which dynamically divides the processors into groups as tasks are scheduled and tasks finish executing. We will show that dynamic grouping offers better schedulability and reliability than static grouping due to its flexible and dynamic reconfiguration of groups in the system. However, dynamic grouping involves more scheduling overhead compared to static and no-grouping techniques [5, 93]. We also propose another technique called *Primary-Backup overloading* (PB-overloading) in which the primary of a task can be scheduled onto the same or overlapping time interval with the backup of another task on a processor. We will show that PB-overloading offers better schedulability than the original BB-overloading and offers reliability comparable to that of BB-overloading [6].

3.1.2 Motivation for Adaptive Overlapping in Soft Real-Time Systems

In soft real-time systems, there exists a trade-off between schedulability and reliability. Most of the existing fault-tolerant scheduling algorithms for real-time systems tend to favor schedulability over reliability. This may not be desirable when tasks have wide-varying utility values. This motivates the need for performance-oriented reliability measures, such as performability which inherently captures both schedulability and reliability in a single metric [91]. The reliable execution of a task is usually achieved by scheduling multiple versions of the task [32, 35, 37, 54, 63, 66, 88]. There are three types of PB-based fault-tolerant approaches that have been proposed in the literature: primary-backup exclusive (PB-EXCL), primary-backup concurrent (PB-CONCUR), and primary-backup overlap (PB-OVER). These PB types capture

a trade-off between the execution interval of a task and the total processor time used by the task for its execution. We call the latter as *consumption time* which is inversely proportional to the processor utilization. The lower the consumption time, the better the schedulability. The execution interval of a task is critical in soft real-time systems. The lower the execution interval, the higher the utility of the task output. In other words, the trade-off is between the percentage of tasks being scheduled and the total utility of their output to the system. This trade-off depends on the primary fault probability in the system and the task's laxity.

PB-EXCL approach is the most widely used approach in which the primary and backup versions of a task are excluded in time as well as in space (processors). The backup version is executed only if the output of the primary fails the acceptance test, otherwise it (backup) is deallocated from the schedule [24]. On the positive side, this approach uses less resources if faults rarely occur, because backups are mostly deallocated. On the negative side, it requires the execution interval (i.e., the interval between start time of primary and finish time of backup) of a task to be at least twice that of the task execution time. This not only increases the chances of a task (backup) not being scheduled before its deadline, but also introduces *holes* in the schedule [26], thus resulting in poor schedulability. In PB-CONCUR approach, the primary and the backup versions of each task are executed concurrently. This approach obviously uses more resources than required when faults rarely occur, because of concurrent execution of versions. Therefore, the consumption time of a task is always twice that of its worst case execution time. Nevertheless, the execution interval for the task never exceeds its worst case execution time [26]. This increases the chances of a task (backup) being scheduled before its deadline. The PB-OVER [26] is a flexible approach in which the primary and the backup versions of a task is allowed to overlap in execution. The overlap interval is critical in determining the schedulability of the system and the utility of the task's output. This approach has the potential to exploit the advantages of the other two approaches by carefully estimating the overlap interval.

It is evident from the above discussion that the PB approaches have a trade-off between the percentage of tasks being scheduled and the total utility of their output to the system.

This trade-off depends on the primary fault probability in the system and task's laxity. For example, the PB-EXCL offers lower consumption time than PB-CONCUR when the fault probability is low; PB-CONCUR offers less execution interval for the task than PB-EXCL when the fault probability is high. Thus, the PB-CONCUR offers better performance when the fault probability is high and tasks have tight laxity. On the other hand, when the fault probability is low or tasks have sufficient laxity the PB-EXCL offers better performance. It can be noted that the reduction in both consumption time and execution interval will contribute towards better performance.

This trade-off can be captured by an adaptive scheduling scheme by controlling the overlap interval between the versions based on the primary fault probability and task's laxity. The adaptive scheme operates between two extremes: behaves like PB-EXCL approach when this interval is zero, behaves like PB-CONCUR approach when the interval is the execution time of the task. Towards achieving this objective, we use a fault monitoring system to observe the fault rate in the system. The observed fault rate is used to estimate the fault probability of the tasks. This probability is used as a feedback to the scheduler to be used with the arriving tasks' laxity in determining the overlap interval between the versions of each task. Since the fault probability and task's soft laxity are used to adjust one of the scheduler parameters (i.e., the overlap interval), the scheduler becomes an adaptive scheduler. We would like to point out here that the feedback based adaptiveness is different from non-feedback based adaptiveness, such as the one used in [25], wherein tasks specify the adaptiveness strategy as opposed to the system deciding it.

In Chapter 5, we propose an adaptive PB scheduling scheme that makes use of an estimate of the primary fault probability and task's laxity to control the degree of overlap between its (task) versions. Two variations of the adaptive scheme are proposed by varying the adaptation mechanism. The adaptation can be done in a continuous manner which leads to an approach called PB-OVER continuous (PB-OVER-CONT), or it can be in a discrete manner which leads to an approach called PB-OVER switch (PB-OVER-SWITCH). In PB-OVER-CONT, the overlap interval varies from no overlap to full overlap in a continuous manner as the fault

probability varies from 0 to 1. In PB-OVER-SWITCH, the scheduler uses a threshold value of fault probability to switch from PB-CONCUR to PB-EXCL, i.e., if the probability is less than the threshold, the adaptive scheduler behaves like PB-EXCL, otherwise it behaves like PB-CONCUR. In PB-OVER-SWITCH, the threshold value is adapted with the task's laxity.

3.1.3 Motivation for Closed-Loop Scheduling in Firm Real-Time Systems

Most real-time scheduling algorithms are based on estimations of the worst-case execution time *(WCET)* of tasks. In practice, it is very difficult to obtain a tight bound on *WCET*. Very few timing analysis tools are available [65], and the ones that exist often require that the users deliberately slow down the processor in order to be deterministic (e.g., by turning off caches). Therefore, most of the analysis tools that are used for estimating the *WCET* depend on over-estimation strategies. Also, the computer market is dominated by general purpose computing where the average case performance, not the worst case performance, matters the most. As a result, the modern hardware architectures increasingly rely on multi-level cache, deep pipelines and speculative branching. Theses techniques optimize average case performance at the cost of worst case performance. As a result, obtaining a tight bound on *WCET* is becoming increasingly irrelevant.

The above discussion makes it evident that the scheduling algorithm that is designed solely based on estimation of the *WCET* will result in an extremely underutilized system. In many cases, it is preferable to base scheduling decisions on average execution time and to be ready to deal with bounded transient overloads dynamically. This approach is especially preferable in firm/soft real-time systems as it provides a firm performance guarantee in terms of deadline misses while achieving high utilization and throughput (guarantee ratio) at the same time.

The requirements of an ideal firm real-time scheduling algorithm are to (1) provide (firm) performance guarantees to admitted tasks, i.e., maintain low deadline miss ratio among admitted tasks; and (2) admit as many tasks as possible, i.e., achieve high guarantee ratio. In an unpredictable environment, it is impossible for a system to achieve 100% utilization and a 0% miss ratio all the time and a tradeoff between miss ratio and utilization is unavoidable.

Two approaches can be used to deal with this tradeoff. The first approach, which uses an admission control based on worst case estimation, represents the pessimistic approach, where deadline misses are avoided at the cost of low utilization and throughput. This approach has been widely used in hard real-time systems. The second approach, which uses the closed-loop scheduling based on nominal estimations, represents the optimistic approach, where a low (but possibly non-zero) miss ratio is maintained with high utilization. When high misses happen due to underestimation of the execution time, the scheduler corrects the system's state back to the satisfactory state, i.e., a state with low miss ratio and high utilization and throughput. This optimistic approach would work as follows: start with a schedule based on the nominal assumptions of the incoming tasks. The system continuously monitors the actual performance of the schedule, compares it to the system requirements, and applies correction to keep the system within an acceptable range of performance.

The key issue addressed in Chapter 6 is the relaxation of the requirement on a known *worst-case* workload parameters. Our main approach to this is to design a closed-loop scheduling algorithm, in which, the tasks are scheduled based on an estimation for their *actual execution time (AET)*. A feedback of the system performance is used to generate an error term. This error is the input to a control unit that adjusts the estimation value. We will model and analyze the closed-loop scheduling algorithm using existing control theory. The result is that this scheduling paradigm has a low miss ratio while maintaining a high guarantee ratio thereby improving the productivity of the firm/soft real-time systems.

Most of the previous work for feedback scheduling in real-time (see Section 2.3.3.4) assumed the execution times are known and focused on how to reassign the periods for tasks to satisfy the utilization constraints. Instead, our work will focus on using feedback control loops to maintain satisfactory system performance when the task execution times change dynamically. Moreover, in both work [11, 51] the authors assume that each task has multiple versions that differ in their execution time, the higher the execution time the better the quality (similar to imprecise computation). Instead, our work assumes only one version for each task which is more general and realistic. Also, in [51] the feedback mechanism is used to reject tasks

in order to keep the total number of missed deadlines below a desired value. Instead, in our work the feedback mechanism is used to maximize both the guarantee ratio and the hit ratio. Nevertheless, in this work the admission controller uses the $WCET$ of task versions to perform the schedulability test. Instead, in our work we use an estimated execution time for each task to perform the schedulability test. The estimated execution time of the tasks are adjusted using the feedback mechanism. In [11], the control mechanism has not been analyzed to prove its correctness and stability. Instead, in our work we analyze the control mechanism and we present the block diagram for the system and the analytical equations that connect the miss ratio and the rejection ratio with the estimated execution time of tasks.

Therefore, in Chapter 6, we first design an open-loop dynamic scheduling algorithm that employs a notion of task overlap in the schedule in order to provide some flexibility in task execution times. This algorithm dynamically guarantees incoming firm tasks via on-line admission control and planning. Secondly, we use feedback control theory to design three closed-loop scheduling algorithms derived from the open-loop algorithm. The loop is closed by feeding back: *(i)* the deadline miss ratio in the first approach; *(ii)* the task rejection ratio in the second approach; *(iii)* both the miss ratio and rejection ratio in the final approach.

3.2 Scopes and Justification of the Dissertation Research

The scopes of the proposed techniques that are introduced in this thesis are the following:

1. The proposed solutions are applicable for multiprocessor systems: Multiprocessor systems are natural candidates for real-time applications due to their potential for high performance and reliability. Many future real-time systems would be consisting of multiprocessors. These systems operate for long periods in non-deterministic fault-inducing environments under rigid timing constraints. These systems need to be robust while delivering high real-time performance. Thus, there is a need for developing adaptive fault-tolerant techniques that dynamically address real-time constraints, and provide both a priori acceptable system-level performance guarantees and a graceful degradation in the presence of failures and time constraints.

2. The proposed techniques are applicable for dynamic planning based scheduling algorithms. In real-time systems, dynamic scheduling of tasks is interesting for the following reasons [55]:

   - Most of the complex real-time systems have aperiodic tasks (in addition to the periodic tasks), the scheduling of such tasks requires a dynamic scheduling algorithm.
   - Dynamic scheduling increases the real-time system's adaptability and reconfigurability in response to failures of external events, and it permits the dynamic activation of exception handling tasks.
   - In contrast to static scheduling, dynamic scheduling can take advantage of additional up-dated information regarding the system state.

3. The proposed fault-tolerant techniques come under the category of primary-backup (PB) based fault-tolerance model: PB is one of the most approaches that are used for fault-tolerant scheduling of real-time tasks. In this approach, the resources reserved for backup copies could be re-utilized by using the following two techniques to achieve high system utilization while providing fault tolerance:

   - Backup overloading, which is the scheduling of more than one backup in the same time slot on the same processor (overlapping of multiple backups).
   - Backup de-allocation, which is the reclamation of resources reserved for backup tasks when the corresponding primaries complete successfully.

4. The proposed techniques are applicable for non-preemptive systems: In multiprocessor, preemption adds to the overhead of the system and also adds an aspect of unpredictability that might make the system harder to validate.

3.3 Goals and Approaches

The main goal of this thesis is to introduce new techniques that offer trade-offs between schedulability and reliability in a dynamic multiprocessor real-time systems. The goals are as

follows:

Goal 1: Provides guarantees required for fault-tolerant execution in hard real-time systems.

The basic approach used in this thesis for providing fault tolerance through scheduling in real-time systems is the primary/backup approach. The backup is executed only if the primary fails. Note that this scheme of adding time redundancy also allows different versions of the task to be executed as primary and backup, thus facilitating the provision of tolerating software faults. When a new task is scheduled, certain conditions are tested to ensure that faulty tasks can be re-executed to be completed within their deadlines.

Given a task and fault model, it is possible to prove properties about the fault tolerance capabilities of the system. For example, it may be possible to prove that if certain assumptions or conditions hold, then one fault can be tolerated within a specific interval of time. When a new task is being considered for addition into the system, set of conditions is tested to check whether the fault tolerance guarantees can be provided. If the conditions are met, then the task is accepted, otherwise it is rejected. Thus, these conditions constitute the schedulability tests for new tasks.

The proposed fault-tolerant techniques check that both primary and backup copies of the task can execute before its hard deadline. A set of conditions is tested, and if satisfied, the task is guaranteed to finish before its hard deadline despite of faults. The type and frequency of faults that can be tolerated depends on the system, fault-tolerant technique, and fault model.

Goal 2: Proposes a technique to improve system utilization and reliability in hard real-time systems.

In this work, we propose a technique called *dynamic grouping* to be used with backup overloading in a PB-based dynamic scheduling algorithm in multiprocessor hard real-time systems. In the dynamic grouping technique, the processors are dynamically grouped into logical groups as tasks are either scheduled or finish execution. In dynamic grouping, the

number of groups and the size of the groups will vary dynamically in contrast to static grouping [57] where these quantities are fixed. Moreover, in static grouping, a processor can be a member of only one group. Whereas in dynamic grouping, a processor can be a member of more than one group which allows efficient overloading. We defined a set of rules and conditions that control the creation, deletion, expansion, and shrinking of these logical groups. From our analytical and simulation studies, we quantified the performance gain/loss of this technique (dynamic grouping) over the existing techniques (no grouping and static grouping) for all the performance metrics (see Chapter 4).

Goal 3: Proposes a technique to improve system performance in hard real-time systems.

In this work, we propose a technique called Primary-Backup (PB) overloading to be used in a PB-based dynamic scheduling algorithm in multiprocessor hard real-time systems. In PB-overloading, the primary of a task can be scheduled onto the same or overlapping time interval with the backup of another task on a processor. We defined a set of rules and conditions that control the PB-overloading technique. From our analytical and simulation studies, we quantified the performance gain/loss of this technique (PB-overloading) over the existing techniques (no overloading and BB-overloading) for all the performance metrics (see Chapter 4).

Goal 4: Uses a fault monitoring system to estimate the primary fault probability.

A fault monitoring system periodically (with period p) monitors the completion of tasks in the system. For each period the monitoring system counts the number of faulty primaries ($n^f(t)$) and the total number of primaries completed ($n(t)$) during that period.

According to the *frequency interpretation* probability concepts, the probability of an event (primary fail) is the proportion of the time that events of the same kind will occur in the long run. Hence, we define the primary fault probability (f) as the ratio of the finished

faulty primaries to the total finished primaries in an interval ($f(t) = \frac{n^f(t)}{n(t)}$). This probability is calculated every p time units, where p is the sample period for the monitor. This fault probability is used to control the degree of overlap between the versions of the tasks in the soft real-time system.

Goal 5: Proposes an adaptive technique to improve system performance in soft real-time systems.

In this work, we propose an adaptive scheduling scheme to be used in a PB-based dynamic scheduling algorithm in multiprocessor soft real-time systems. The adaptive PB scheduling scheme makes use of an estimate of the primary fault probability and task's laxity to control the degree of overlap between its (task) versions. Two variations of the adaptive scheme are proposed by varying the adaptation mechanism. The adaptation can be done in a continuous manner which leads to an approach called PB-OVER continuous (PB-OVER-CONT), or it can be in a discrete manner which leads to an approach called PB-OVER switch (PB-OVER-SWITCH). In PB-OVER-CONT, the overlap interval varies from no overlap to full overlap in a continuous manner as the fault probability varies from 0 to 1. In PB-OVER-SWITCH, the scheduler uses the threshold value of fault probability to switch from PB-CONCUR to PB-EXCL, i.e., if the probability is less than the threshold, the adaptive scheduler behaves like PB-EXCL, otherwise it behaves like PB-CONCUR. In PB-OVER-SWITCH, the threshold value is adapted with the task's laxity. From our analytical and simulation studies, we quantified the performance gain/loss of the adaptive techniques (PB-OVER-CONT and PB-OVER-SWITCH) over the existing non-adaptive techniques (PB-EXCL and PB-CONCUR) for all the performance metrics (see Chapter 5).

Goal 6: Proposes an open-loop scheduling algorithm to improve system performance in firm real-time systems.

In this work, we design an open-loop dynamic scheduling algorithm that employs a notion of task overlap in the schedule in order to provide flexibility in task execution times. This algo-

rithm, dynamically guarantees incoming firm tasks via on-line admission control and planning. This algorithm is called *OL-OVER-AvCET* (Open-Loop OVERlap with *AvCET* estimation). In which, *WCET* is used by the scheduler to perform the schedulability check for tasks. However, each task is assigned a time slot equal to its *AvCET* when the schedule is constructed and it is overlapped with its two neighbors by a time slot equal to $\frac{WCET-AvCET}{2}$. From our simulation studies, we quantified the performance gain/loss of the *OL-OVER-AvCET* scheduling algorithm over the existing no overlap scheduling algorithms (*OL-NO-OVER-WCET* and *OL-NO-OVER-AvCET*) for all the performance metrics (see Chapter 6).

Goal 7: Proposes a close-loop scheduling algorithm to improve the performance in firm real-time systems.

The key issue addressed in this work is the relaxation of the requirement on a known *worst-case* workload parameters. Our main approach to this is to design a closed-loop scheduling algorithm, in which, the tasks are scheduled based on an estimation for their *actual execution time (AET)*. A feedback of the system performance is used to generate an error term. This error is the input to a control unit that adjusts the estimation value. Specifically, in this work, we use feedback control theory to design three closed-loop scheduling algorithms derived from the open-loop algorithm. The loop is closed by feeding back *(i)* the deadline miss ratio in the first approach; *(ii)* the task rejection ratio in the second approach; *(iii)* both the miss ratio and rejection ratio in the final approach. We modeled and analyzed the closed-loop scheduling algorithm using control theory. From our simulation and analytical studies we quantified the performance gain/loss of the closed-loop scheduling over the open-loop scheduling for all the performance metrics (see Chapter 6).

Goal 8: Uses the feedback control theory to model and analyze the closed-loop scheduling algorithm.

To apply feedback control techniques in closed-loop scheduling, we restructured the schedulers based on the feedback control framework. We identified the set points, control variables,

regulated variables, and measured variables. Once these variables are identified, we obtained a mathematical abstraction that provides the effect of the inputs on the outputs. Moreover, we designed the control law that dictates how to alter the control inputs in response to information gathered about the system through measured variables.

Goal 9: Defines new performance metrics that are used to evaluate the proposed techniques.

In soft real-time systems, the efficiency of a scheduling algorithm is usually measured as the sum of the values contributed by the admitted tasks. For a given system capacity, there are two options: (i) admit less number of tasks with a higher value for each task or (ii) admit more number of tasks with a lower value for each task.

The adaptive fault-tolerant techniques that we proposed, for the soft real-time system, had a trade-off between the processor utilization and the execution interval for each task. This trade-off is effectively captured by the schedulability-reliability index (SR) given in Equation (5.3). The effect on the number of admitted tasks is inherently captured by the processor utilization of each task. The higher the processor utilization of a task is, the higher the number of admitted tasks and hence the better the performance.

In hard real time systems, the efficiency of a scheduling algorithm is directly proportional to the percentage of incoming tasks it can schedule in the system. In other words the efficiency of the algorithm is estimated by the guarantee ratio, which is the percentage of incoming tasks scheduled in the system.

The fault-tolerant techniques that we proposed, for the hard real-time system, had a trade-off between the system's utilization and its fault-tolerant capability. The capability of a system to tolerate faults is based on the number and frequency of faults it can tolerate. If a system includes more redundancy to tolerate a larger number of faults, its fault tolerance capability increases. However, the fault tolerance capability is achieved at the cost of system utilization which is the percentage of system resources used for actual operations. If there is a large amount of redundancy in the system, the percentage of resources being used for actual computing

purposes is small, and thus the system utilization becomes low.

The capability of a system to tolerate faults can be measured in terms of the minimum separation between two successive faults that can be tolerated. The smaller this separation the higher the resilience of the system. The resilience of a system is defined as the ability of a system to tolerate a second fault after recovering from the first one. In static systems, the resilience of the system can be measured exactly, because the static schedule determines how far apart two successive faults can be tolerated. On the other hand in dynamic systems, fault injection and simulations can determine the average resilience of the system. In Section 4.5.2.2, the average time at which a second fault can be tolerated, which is called Time To Second Fault (TTSF) is defined.

In a firm real-time system, the efficiency of a scheduling algorithm is directly proportional to the percentage of arrived tasks that met their deadlines. For a given system capacity, there are two options: (i) admit less tasks with a higher estimated execution time or (ii) admit more tasks with a lower estimated execution time for each task. The performance index that is used to compare the closed-loop scheduling algorithm with the open loop scheduling algorithms captures the trade off between the *guarantee ratio* (GR) and the *hit ratio* (HR) (percentage of the admitted tasks that meet their deadlines). The performance index that is used is the *effective ratio* (ER) which is equal to $GR * HR$. (see Chapter 6).

Goal 10: Helps real-time system designers make appropriate design choices.

The real-time system designer has to make several design choices while building a system. One of the goals of this thesis is to answer some of the trade-off questions related to reliability and schedulability. The most important question while building a real-time system is how much redundancy is appropriate and which fault-tolerant technique to use? This question is the main focus of each technique studied in this thesis. For example, the most appropriate overloading and grouping techniques can be selected given a certain $MTTF$ (Mean Time To Failure) in hard real-time systems. Also, the most appropriate overlap interval between the primary and the backup for a task can be dynamically determined given the estimated primary

fault probability and task's laxity in soft real-time systems. Finally, the most appropriate estimation for task's execution time can be estimated based on a feedback from the system performance.

Goal 11: Uses various analysis tools to study the performance of the proposed techniques.

Different kinds of analysis tools are used to study the propose techniques. Simulations have been used extensively to evaluate all the techniques studied in this thesis. Markov analysis has also been used to analyze the dynamic grouping and the primary-backup overloading techniques for hard real-time systems. Mathematical probability has also been used to analyze the adaptive fault-tolerant techniques for soft real-time systems. The feedback control theory has been used to analyze the scheduling techniques for firm real-time systems.

3.4 System Models

In this section, we state the task, and scheduler models that are used in the different parts of the thesis.

3.4.1 Task Models

1. Tasks are aperiodic, i.e., task arrivals are not known a priori
2. Tasks are non-preemptable, i.e., when a task starts execution on a processor, it finishes until its completion.
3. Tasks may have resource and/or precedence constraints.
4. For the fault-tolerant techniques in the hard and soft real-time systems (Chapters 4 and 5), each task T_i has two versions, namely primary copy (Pr_i) and backup copy (Bk_i). The versions of a task have identical attributes and resource requirements.
5. For the hard real-time tasks, every task T_i has the following attributes: arrival time (a_i), ready time (r_i), worst case computation time (c_i) and a hard deadline (d_i).

6. For the soft real-time tasks, each task T_i has the following attributes: arrival time (a_i), ready time (r_i), worst case computation time (c_i), a relative soft deadline (d_i^s), and a relative firm deadline (d_i^f). $d_i^s = k_1 \times c_i$ and $d_i^f = k_2 \times c_i$, where $k_2 \geq k_1$. The smallest value of k_2 is assumed to be 2 in order to ensure that the primary and the backup versions of each task can be scheduled within the firm deadline in the case of PB-EXCL.

7. For the firm real-time tasks, every task T_i has the following attributes: arrival time (a_i), ready time (r_i), worst-case execution time ($WCET_i$), best-case execution time ($BCET_i$), and firm deadline (d_i).

3.4.2 Scheduler Model

In our dynamic multiprocessor scheduling, all the tasks arrive at a central processor called the scheduler, from where they are dispatched to other processors in the system for execution. These processors are identical and are connected through a shared medium. The communication between the scheduler and the processors is through the dispatch queues. Each processor has its own dispatch queue as shown in Figure 3.1. The scheduler will be running in parallel with the processors to schedule the newly arriving tasks and periodically update the dispatch queues.

3.4.2.1 The dynamic scheduling algorithm

To demonstrate the effectiveness of the proposed new techniques in this thesis, we extend the Spring scheduling [82], a well known dynamic scheduling algorithm, incorporating these techniques. The Spring scheduling algorithm is a heuristic search algorithm that dynamically schedules arriving real time tasks with resource constraints. Figure 3.2 shows how the algorithm operates to schedule a set of tasks. A vertex in the search tree represents a partial schedule. The schedule from a vertex is extended only if the vertex is strongly feasible. A vertex is strongly feasible if a feasible schedule can be generated by extending the current partial schedule with each task of the feasibility check window. The feasibility check window is a subset of first w unscheduled tasks. The larger the size of the feasibility check window, the higher the scheduling

cost and the more is the look-ahead nature. If the current vertex is strongly feasible, the algorithm computes a heuristic function (H) for each task within the feasibility check window based on deadline and earliest start time of the task. It then extends the schedule by including the task that has the best (smallest) heuristic value. Otherwise, it back-tracks to the previous vertex and then the schedule is extended from there using a task which has the next best heuristic value.

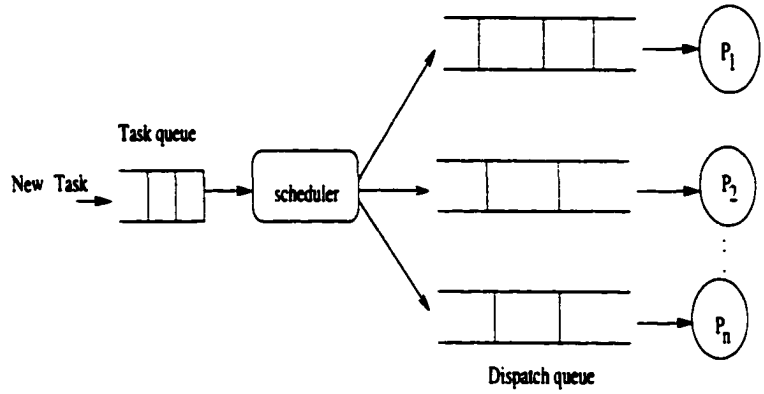


Figure 3.1 Scheduler model

Spring_schedule()

1. Order the tasks in the task queue in non-decreasing order of deadlines.
2. Compute EST(T_i) for the first w tasks, where w is the size of the feasibility check window.
3. Check for strong feasibility: check whether EST(T_i) + $c_i \leq d_i$ is true for all w tasks.
4. If strongly feasible or no more backtracking is possible then
   (a) Compute the heuristic function (H= d_i + EST(T_i)) for the first w tasks.
   (b) Choose the task (T_i) with the best (smallest) H value to extend the schedule.
   (c) If T_i is schedulable then Schedule T_i.
   (d) Else reject task T_i.

   Else backtrack to the previous search level and try to extend the schedule with a task having the next best H value.
5. Repeat steps(2-4) until termination condition is met:
   (a) All the tasks are scheduled or
   (b) All the tasks are considered for scheduling and no more backtrack is possible.

Figure 3.2 The Spring scheduling algorithm.

CHAPTER 4. A CASE OF SCHEDULABILITY-RELIABILITY TRADE-OFFS IN HARD REAL-TIME SYSTEMS

In this chapter, we propose two techniques to improve the schedulability and/or the reliability of hard real-time systems: (i) a technique called *dynamic grouping* that is used with BB and PB over-loadings and (ii) a technique called Primary-Backup (PB) overloading that allows overloading of a primary with one backup on the same/overlapping time interval(s) on a processor.

In Section 4.1 we define the terminology that is used in this chapter. In Section 4.2, we define the fault model. In Section 4.3, backup overloading is discussed. In Section 4.4, we present the dynamic logical grouping along with BB and PB overloading techniques. In Section 4.5, we present the simulation studies of the proposed techniques. In Section 4.6, we present the analytical studies of the proposed techniques. Finally in Section 4.7, we summarize the results.

4.1 Terminology

Definition 1: $st(T_i)$ is the scheduled start time of task T_i, which satisfies $r_i \leq st(T_i) \leq d_i - c_i$. $ft(T_i)$ is the scheduled finish time of task T_i, which satisfies $r_i + c_i \leq ft(T_i) \leq d_i$.

Definition 2: $Proc(Pr_i)$ is the processor on which the primary version of task T_i is scheduled. Similarly $Proc(Bk_i)$ denote the same for the backup version of T_i.

Definition 3: $st(Pr_i)$ is the scheduled start time and $ft(Pr_i)$ is the scheduled finish time of primary version of a task T_i. Similarly $st(Bk_i)$ and $ft(Bk_i)$ denotes the same for the backup version of T_i.

Definition 4: $S(Pr_i)$ is the time interval (slot) in which primary version of a task T_i is scheduled (i.e., $S(Pr_i) = [st(Pr_i), ft(Pr_i)]$). Similarly $S(Bk_i)$ denotes the same for the backup version.

Definition 5: The primary and backup copies of task T_i are said to be mutually exclusive in time, denoted as time exclusion(T_i), if $st(Bk_i) \geq ft(Pr_i)$.

Definition 6: The primary and backup copies of task T_i are said to be mutually exclusive in space, denoted as space exclusion(Ti), if $Proc(Bk_i) \neq Proc(Pr_i)$.

Definition 7: A task is said to be feasible in the fault-tolerant schedule if it satisfies the following constraints.

- The primary and backup versions of a task should satisfy the deadline and the time exclusion constraints.

  $r_i \leq st(Pr_i) \leq ft(Pr_i) \leq st(Bk_i) \leq ft(Bk_i) \leq d_i$.

- Primary and backup versions of a task should be mutually exclusive in space in the schedule. This is necessary to tolerate permanent processor failures.

4.2 Fault Model

Each processor, except the scheduler[1], may fail due to hardware fault which results in tasks failure. The faults can be transient or permanent and are independent. Each independent fault results in the failing of only one processor. The following assumptions form the fault model.

Assumption 1: The maximum number of processors that are expected to fail at any instant of time in a group of processors is assumed to be one (The concept of forming processors into groups will be explained later). This is a reasonable assumption given that the group size is very small, which is the case in our techniques wherein the maximum group size is three.

Assumption 2: The occurrence of faults in each processor follows a Poisson distribution with parameter μ_i.

[1]For example, the scheduler can be made fault-tolerant by executing it on more than one processor.

Assumption 3: Mean time to failure ($MTTF$) of a group is defined as the expected time for which the group operates before the first failure occurs. Time to second fault ($TTSF$) is the time at which a second fault can occur without affecting the operation of the system [24].

Note that, $MTTF$ is imposed on the system by the occurrence of a fault, whereas $TTSF$ is the ability of the system to react to fault. The smaller the $TTSF$, the better the fault-tolerant operation of the system. In a PB-based fault-tolerant scheduling, the minimum required value of $TTSF$ (see Section 4.5.2.2) is always greater than or equal to $(d_i - r_i)\ \forall\ T_i$. Obviously, $TTSF$ must be much smaller than $MTTF$ to enhance the probability that the system remains continuously operational. To satisfy this condition we assume that the $(d_i - r_i)\ \forall\ T_i$ is much smaller than the typical $MTTF$ value of the system. This assumption is used to enhance the probability of successfully execution for the backup of a task, if its primary fails.

Assumption 4: There exists fault-detection mechanisms such as fail-signal and acceptance test to detect processor and task failures, respectively. The scheduler will not schedule tasks to a known faulty processor.

4.3 Backup Overloading

Backup overloading [23, 24] is defined as scheduling backups of multiple primaries onto the same or overlapping time interval on a processor. Figure 4.1 shows two primaries (Pr_1 and Pr_2) that are scheduled on processors 1 and 3, respectively, and their backups (Bk_1 and Bk_2) are scheduled in an overloading manner on processor 2. The following are the conditions under which backups can be overloaded on a processor:

Condition 1: The primaries of the backups being overlapped must be scheduled onto different processors.

Condition 2: At most one of these primaries is expected to encounter a fault.

Condition 3: At most one version of a task is expected to encounter a fault. In other words, if the primary of a task fails, its backup is expected to succeed.

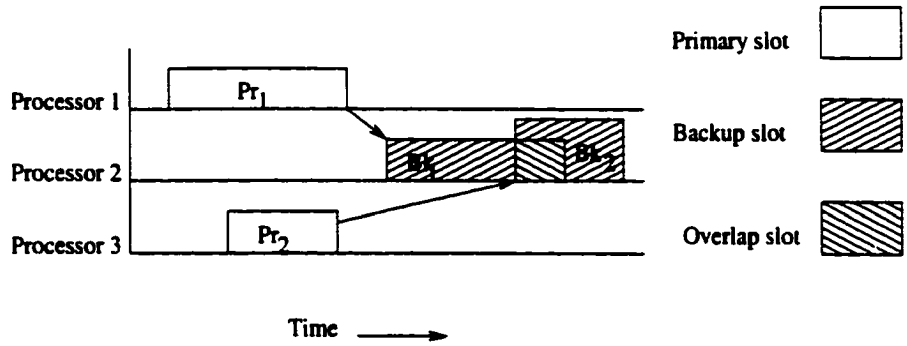


Figure 4.1 Backup overloading

Condition 1 is needed to handle permanent faults. Condition 2 is needed to ensure that at most one backup is required to be executed among the overloaded backups. Condition 3 is needed to ensure that at least one version of each task is executed without any fault. Condition 2 can be satisfied by Assumption 1. Condition 3 is satisfied by Assumption 3 which states that the $MTTF$ of the system is much greater than the $TTSF$ for the fault-tolerant techniques.

4.4 Dynamic Logical Grouping

Dynamic logical grouping is defined as the process of dynamically dividing the processors of the system into logical groups as tasks are scheduled and tasks finish executing. The number of groups and the size of the groups will vary dynamically in contrast to static grouping where these quantities are fixed. Moreover, in static grouping, a processor can be a member of only one group. Whereas in dynamic grouping, a processor can be a member of more than one group which allows efficient overloading. The creation and expansion of a group take place when a task is scheduled, and removing and shrinking of a group take place when a task finishes its execution.

This technique has a potential to offer better schedulability than static and comparable to that of no-grouping, and offers better reliability than static grouping. This potential is based on the following two observations.

First observation: In dynamic grouping, a backup has $n-1$ choices for overloading

(group are formed after scheduling the backup). This, in turn, helps the backup to find an earlier start time, thus resulting in a better chance of meeting the task deadline, and improving the schedulability of tasks.

Second observation: In dynamic grouping, the groups are dynamically reconfigured as tasks are scheduled and tasks finish executing. If a permanent fault occurs at time t in p_i, any task arriving after time t will be scheduled (both primary and backup) on the $n-1$ non-faulty processors (Assumption 4 of the fault model). Thus, this processor will not be part of the future group unless it is repaired. This, in turn, improves the groups utilization (increases the chance of overloading) thus, offers a graceful degradation in the performance as the number of faults increases.

The creation, deletion, expansion, and shrinking of logical groups with BB and PB overloading techniques are explained in the next two sub sections.

4.4.1 Backup-Backup Overloading (BB-overloading)

BB-overloading is defined as scheduling two backups onto the same or overlapping time interval on a processor. Figure 4.2 shows two primaries (Pr_1 and Pr_2) of tasks T_1 and T_2 that are scheduled on processors 1 and 3 and their backups (Bk_1 and Bk_2) are scheduled to be overlapped on processor 2. The conditions for BB-overloading (stated in Section 3) are restated below in a more precise form.

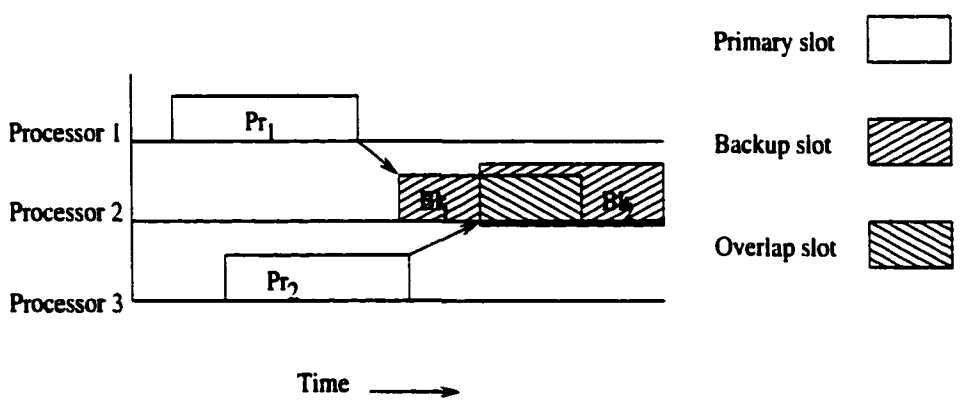


Figure 4.2 BB-overloading

1. Pr_1 and Pr_2 should be scheduled on different processors.

2. $ft(Pr_2) - st(Pr_1) \leq$ the expected minimum interval between two faults.

3. $Proc(Pr_1)$, $Proc(Pr_2)$, and $Proc(Bk_1, Bk_2)$ should be in the same group.

Condition 1 is needed to handle permanent faults. Conditions 2 and 3 are need to ensure that at most one version among these tasks is likely to encounter a fault. We claim, using the following propositions, that these conditions satisfy the fault model.

Proposition 1: At most one version of a task is likely to encounter a fault.

Proof: Since the two versions (Pr_i and Bk_i) of a task T_i are scheduled on two different processors in the same group in the interval $[r_i, d_i]$. Assumptions 1 and 3 (of the fault model) enhance the probability that at most one version of the task is likely to encounter a fault.

Proposition 2: At most one of the primaries is likely to encounter a fault.

Proof: Since these three processors ($Proc(Pr_1)$, $Proc(Pr_2)$), and $Proc(Bk_1, Bk_2)$) are within the same group, this group expected to have only one fault at a time (Assumption 1), and since $ft(Pr_2) - st(Pr_1) \leq$ the expected minimum interval between two faults, so at most the fault is likely to be in one of these primaries.

4.4.1.1 Group dynamics with BB-overloading technique

In the BB-overloading, a backup can be in one of two states. In the first state (green), another backup is allowed to overlap with it. In the second state (red), no other backup is allowed to overlap with it. The states are controlled by the following rules:

Rule 1: All the backups are assigned a green state when they are initial scheduled to execute.

Rule 2: A green backup is changed to red if its primary has failed to execute successfully.

Rule 3: A red backup stays red until it finishes execution.

These rules ensure that tasks are scheduled into fault-free groups (i.e., groups that did not encounter a fault previously), which will increase the probability of satisfying the fault model. Note that, the proposed dynamic grouping technique can be incorporated into any dynamic scheduling algorithm. Here, we discuss the procedures that are relevant to BB-overloading with dynamic grouping in such algorithms.

The first procedure in Figure 4.3 shows the creation and expansion of groups when a task T_i is scheduled. In it, if a primary is schedulable, the algorithm tries in the following order to schedule its backup : without overloading (Step 3b: creating a new group with 2 processors) or with overloading (Step 4b: expanding an existing group to have 3 processors). This ordering implicitly favors reliability over schedulability. This is because, under low loads, most of the tasks are schedulable even without overloading, resulting in high reliability; under high load, the tasks are overloaded anyway to meet the deadlines.

The second procedure in Figure 4.3 shows the removing and shrinking of a group when a version of a task T_i finishes its execution. If the version is primary, the outcome of the acceptance test determines whether the group remains the same or needs to be shrunk or deleted: no change in the group - acceptance test failed (Step 2); shrink - acceptance test passed and the group size was 3 (Step 1c); remove - acceptance test passed and the group size was 2 (Step 1b). If the finished version was a backup then the group is removed.

4.4.1.2 Example for group dynamics in BB-overloading

Figure 4.4 shows an example that illustrates how the groups are formed and removed dynamically, with the BB-overloading algorithm, as tasks are scheduled and finish execution. Figure 4.4a shows that primary and backup copies of task T_1 are scheduled on processors 1 and 2, respectively. Then, these two processors will form a logical group (group 1) that will stay until one of the copies executes successfully. Figure 4.4b shows the same situation as in Figure 4.4a, but this time the primary of T_2 is scheduled on processor 3, and its backup is overloaded with Bk_1 on processor 2. This results in expanding the group to have three processors. Figure

Group_Creation_Expansion(group G_k)
{

- If the primary Pr_i can be scheduled then /* any dynamic scheduling algorithm (e.g. Spring scheduling algorithm) */

    1. Schedule Pr_i.
    2. Set ready time(Bk_i) = $ft(Pr_i)$. /* time exclusion */
    3. If Bk_i can be schedule without overloading then
        (a) Schedule Bk_i in a green state.
        (b) Form_group($Proc(Pr_i)$, $Proc(Bk_i)$).
    4. Else if Bk_i can be overloaded with another green backup Bk_j in group G_k then /* condition 1 and 2 must be valid */
        (a) Schedule Bk_i in a green state. /* overload it with Bk_j.*/
        (b) Expand_group(G_k, $Proc(Pr_i)$).
    5. Else unscheduled primary Pr_i, and reject the task.

- Else reject the task T_i.

}

Group_Deletion_Shrinking(group G_k)
{

- If the version is a primary (Pr_i) then

    1. If its acceptance test is successful then
        (a) De-allocate the backup (Bk_i).
        (b) if the group size = 2 then Remove_group(G_k).
        (c) Else Shrink_group(G_k,Proc(Pr_i)) to contain only two processors: $Proc(Pr_j)$ and $Proc(Bk_j)$, where Bk_j is the backup that was overloaded with Bk_i.
    2. Else convert Bk_i to red state and keep the group until Bk_i finishes execution.

- Else Remove_group(G_k).

}

Figure 4.3 Group creation, expansion, deletion, and shrinking with BB-overloading

4.4c shows the same situation as in Figure 4.4b, but now the scheduler has decided to schedule the primary of T_3 on processor 4, and its backup on processor 3, then processors 3 and 4 will form a logical group (group 2). Figure 4.4d shows the situation when Pr_1 has executed successfully. Therefore Bk_1 is de-allocated, which results in shrinking group 1 to have two processors 2 and 3.

On the other hand, if static grouping is employed for the same example processors 1, 2, and 3 will form group 1 and processor 4 will be in group 2. In this case, the situations shown in Figures 4.4c and 4.4d cannot occur because the groups are disjointed and their sizes are fixed.

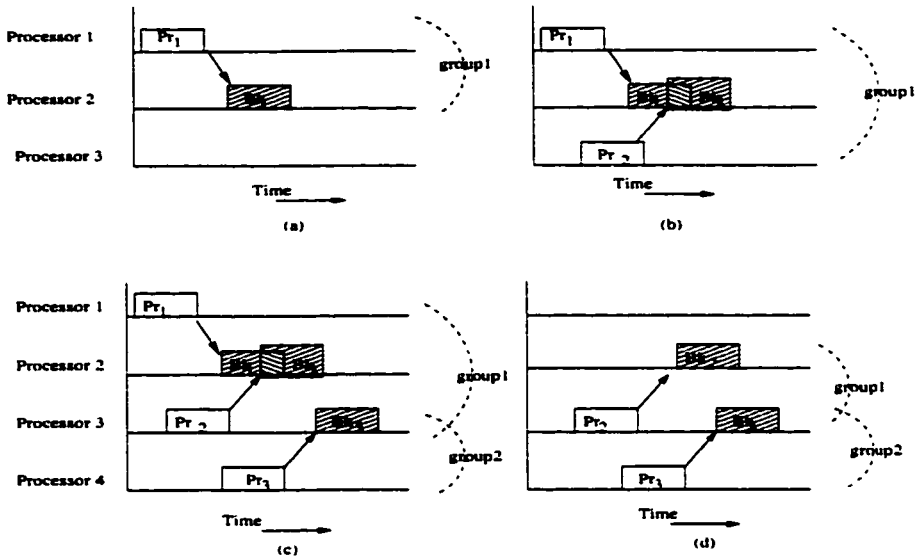


Figure 4.4 Dynamic grouping

It can be seen that dynamic grouping results in higher utilization for processor 3 (in Figure 4.4c), because in static grouping Bk_3 can not be scheduled to processor 3 since Pr_3 is in group 2. Also, dynamic grouping enhances the reliability of the system as tasks T_2 and T_3 can be executed successfully even in the presence of faults in processors 2 and 4 (in Figure 4.4d).

Similarly, if no-grouping was employed for the same example, all the processors would be in one group. In this case, the situation shown in Figure 4.4c may be changed since more than two backups can be overlapped at the same time slot. This will enhance the system's utilization, but the system's reliability will decrease since the probability of satisfying the assumption of having only one fault at a time in the system is decreased.

4.4.2 Primary-Backup Overloading (PB-overloading)

In this section, we propose a new overloading technique called primary-backup overloading (PB-overloading) in which the primary of a task can be scheduled onto the same or overlapping time interval (i.e., overloaded) with the backup of another task on a processor. This technique has a potential to offer better schedulability than BB-overloading and is based on the following observation.

For a primary, PB-overloading can assign an earlier *start time* than that of BB-overloading, because the primary can be overloaded on an already scheduled backup. This, in turn, helps its backup to find an earlier start time, thus resulting in a better chance of meeting the task deadline.

In other words, PB-overloading has a better chance of making a task feasible (refer to Definition 7) in the schedule compared to BB-overloading. Figure 4.5 shows a primary Pr_1 that is scheduled on processor 1 and its backup Bk_1 is scheduled on processor 2. Also it shows another primary Pr_2 that is overloaded with Bk_1 on processor 2 and its backup Bk_2 is scheduled in processor 3. With respect to this example, we state the following conditions that govern the PB-overloading, which are equivalent to the conditions governing BB-overloading.

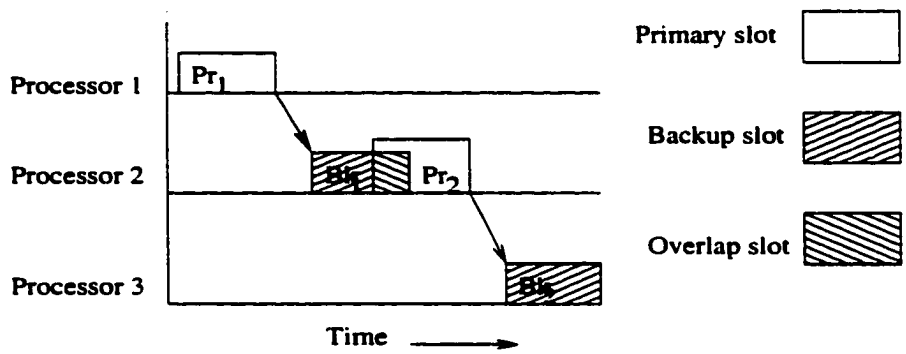


Figure 4.5 PB-overloading

1. Pr_1 and Bk_2 should be scheduled onto different processors.

2. $ft(Bk_2) - st(Pr_1) \leq$ the expected minimum interval between two faults.

3. $Proc(Pr_1)$, $Proc(Bk_1) = Proc(Pr_2)$, and $Proc(Bk_2)$ should be in the same group.

Condition 1 is needed to handle permanent faults. Conditions 2 and 3 are need to enhance the probability that at most one version among these tasks is likely to encounter a fault. We claim, using the following proposition, that these conditions satisfy the fault model.

Proposition 1: At most one version among these tasks is likely to encounter a fault.

Proof: Since these three processors ($Proc(Pr_1)$, $Proc(Pr_2) = Proc(Bk_1)$, and $Proc(Bk_2)$) are in the same group, this group is expected to have only one fault at a time (Assumption 1), and since $ft(Bk_2) - st(Pr_1) \leq$ the expected minimum interval between two faults, so at most only one fault is likely among these tasks.

4.4.2.1 Group dynamics with PB-overloading

In the PB-overloading, as in BB-overloading, a backup can be in one of two states. In the first state (green), a primary is allowed to overlap with it. In the second state (red), a primary is not allowed to overlap with it. The states are controlled by the following rules.

Rule 1: If a primary is scheduled without overloading with another backup, its backup is said to be in the green state.

Rule 2: If a primary is scheduled with overloading on another backup, its backup is said to be in the red state.

Rule 3: A red backup changes to green if the backup that was overloaded with its primary is de-allocated (i.e., the primary for that backup has executed successfully). Otherwise, the backup stays red.

Rule 4: A green backup changes to red if its primary has failed to execute successfully. Otherwise, it stays green.

These rules ensure that tasks are scheduled onto fault-free groups (i.e., groups that did not encounter a fault previously), which will increase the probability of satisfying the fault model. Also, these rules ensure that the PB-overloading chain will not contain more than two tasks at the same time.

Noted that, both BB- and PB-overloading techniques can co-exist in a single scheduling algorithm. However, we believe that this will significantly increase the cost (complexity) of scheduling, which may not offer a proportionate increase in schedulability. Therefore, in this work, we do not explore the co-existence of BB- and PB-overloading techniques.

Group_Creation_Expansion(group G_k)
{

1. If the primary Pr_i can schedule without overloading then /* any dynamic scheduling algorithm (e.g. Spring scheduling algorithm) */
   (a) Schedule Pr_i.
   (b) Set ready time(Bk_i) = $ft(Pr_i)$. /* time exclusion */
   (c) If Bk_i can be schedule then
      i. Schedule Bk_i in a green state.
      ii. Form_group($Proc(Pr_i)$, $Proc(Bk_i)$).
   (d) Else unscheduled primary Pr_i, and reject the task.
2. Else if Pr_i can be overloaded with another green backup Bk_j in group G_k then /* condition 1 and 2 must be valid */
   (a) Schedule Pr_i . /* overload it with Bk_j.*/
   (b) Set ready time(Bk_i) = $ft(Pr_i)$. /* time exclusion */
   (c) If Bk_i can be schedule then
      i. Schedule Bk_i in a red state.
      ii. Expand_group(G_k, $Proc(Bk_i)$).
   (d) Else unscheduled primary Pr_i, and reject the task.
3. Else reject the task T_i.

}

Group_Deletion_Shrinking(group G_k)
{

- If the version is a primary (Pr_i) then
  1. If its acceptance test is successful then
     (a) De-allocate the backup (Bk_i).
     (b) if the group size = 2 then Remove_group(G_k).
     (c) Else if the group size = 3 then
        i. Shrink_group(G_k,Proc(Pr_i)) to contain only two processors: Proc(Pr_j) and Proc(Bk_j), where Pr_j is the primary that was overloaded with Bk_i.
        ii. Convert Bk_j to green state.
  2. Else If its acceptance test is failed then
     (a) De-allocate the primary (Pr_j).
     (b) convert Bk_i to red state and keep the group until Bk_i and Bk_j finish execution.
- Else if the version is a backup (Bk_i) then
  1. if the group size = 2 then Remove_group(G_k).
  2. Else if the group size = 3 then Shrink_group(G_k,Proc(Pr_i))

}

Figure 4.6 Group creation, expansion, deletion, and shrinking with PB-overloading

The first procedure in Figure 4.6 shows the creation and expansion of groups when a task T_i is scheduled. In it, the primary of a task is attempted for scheduling in the following order: (i) schedule it without overloading (Step 1: create a new group with 2 processors) or (ii) schedule it with overloading (Step 2: expand an existing group to have 3 processors). This ordering implicitly favors reliability over schedulability. This is because, under low loads, most of the tasks are schedulable even without overloading, resulting in high reliability; under high load, the tasks are overloaded anyway to meet the deadlines.

The second procedure in Figure 4.6 shows the removing and shrinking of a group when a version of a task T_i finishes its execution. If the version is primary, the outcome of the acceptance test determines whether the group remains the same or needs to be shrunk or deleted: no change in the group - acceptance test failed (Step 2); shrink the group - acceptance test passed and the group size was 3 (Step 1c); remove the group - acceptance test passed and the group size was 2 (Step 1b). If the finished version was a backup: the group is removed if its size was 2, otherwise it is shrunk.

4.4.2.2 Example PB-overloading dynamics

Figure 4.7 shows an example that illustrates the working of PB-overloading. Figure 4.7a shows that the primary of task T_1 is scheduled on processor 1, and its backup (green state) is scheduled on processor 2. Then, these two processors will form a logical group (group 1) that will stay until one of these versions executes successfully. Figure 4.7b shows the same situation as in Figure 4.7a, but this time the primary of T_2 is scheduled to overload with Bk_1 on processor 2, and its backup (red state) is scheduled on processor 3. This results in expanding the group to have three processors. Figure 4.7c shows the situation when Pr_1 has executed successfully. Therefore, Bk_1 is de-allocated which results in shrinking group 1 to have two processors (processors 2 and 3), and Bk_2 becomes green (state). On the other hand, Figure 4.7d shows the situation after Pr_1 has failed to execute successfully. Therefore, Pr_2 is de-allocated and Bk_1 becomes red (state). The group will stay until Bk_1 finishes its execution which results in shrinking group 1 to have two processors (processors 2 and 3) which will stay

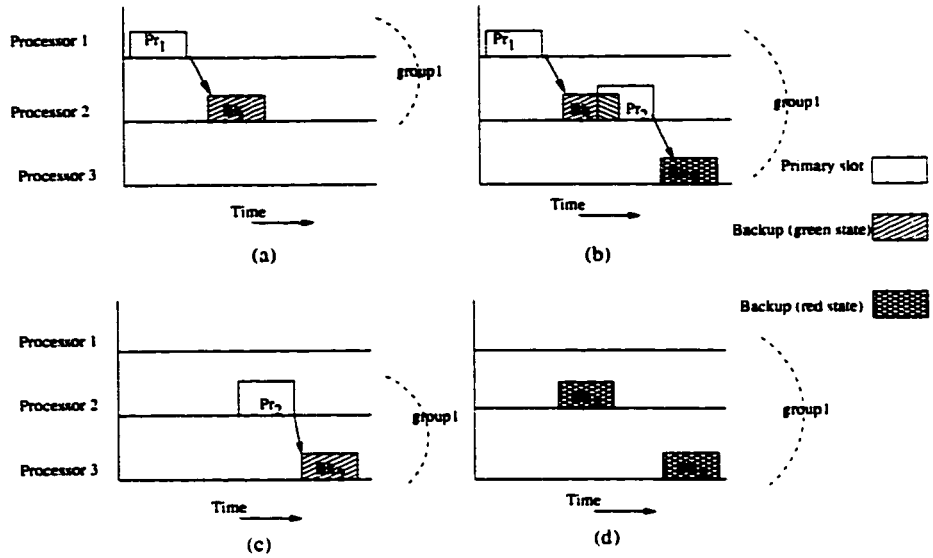


Figure 4.7 Dynamics of PB-overloading

until Bk_2 also finishes its execution.

4.5 Performance Studies

We have conducted schedulability and reliability analysis of the proposed techniques through extensive simulation and analytical studies.

4.5.1 Simulation Studies

For our simulation studies, we have used Spring scheduling [82], a well known dynamically scheduling algorithm, to incorporate the proposed techniques. In this section, we first compare the performance of the proposed dynamic grouping technique with that of the static grouping [57] and no-grouping techniques [23, 24] under BB-overloading. Then, we compare the performances of the PB-overloading technique with that of the BB-overloading and no-overloading techniques under static grouping.

The guarantee ratio (GR) is used as the performance metric, which is defined as:

$$GR = \frac{the\ number\ of\ tasks\ guaranteed}{the\ number\ of\ tasks\ arrived}. \tag{4.1}$$

For each point in the performance curves in Figures 4.8-4.15, the total number of tasks arrived in the system is 20,000. This number of task have been chosen to cancel the effect of the warm-up period of the simulation and to have approximately 95% confidence interval within ±0.005 around the results. The parameter and overhead costs used in the simulation studies are given in Table 4.1. In the table, the $sched_{cost}$ is the cost for each invocation of the scheduler, and the $group_{cost}$ is the overhead cost needed to create or update groups in the dynamic grouping technique. The tasks for the simulation are generated as follows:

Table 4.1 Simulation parameters.

Parameter	Explanation	Values
Min_c	minimum computation time of tasks	20 sec.
Max_c	maximum computation time of tasks	40 sec.
L	the offered task load to the system	0.25... 1
R	laxity parameter	2 ... 10
N	number of processors	3 ... 24
μ	mean time between faults	80... 800 sec.
$sched_{cost}$	scheduler's cost for each invocation of the scheduler.	4 sec.
$group_{cost}$	overhead cost need to make a group	3 sec.

- The worst case computation time of a task (primary version) is chosen uniformly between Min_c and Max_c.
- The deadline of a task T_i (primary version) is uniformly chosen between $r_i + 2 * c_i$ and $r_i + R * c_i$, where R $\geq$ 2.
- The lead time δ, which is the difference between the ready time and the arrival time (i.e., $r_i = a_i + \delta$), is uniformly chosen between 0 and Min_c.
- The inter-arrival time of tasks follows exponential distribution with mean θ.
- The backup versions are assumed to have identical characteristics of their primary versions.

- The inter-arrival time of faults follows exponential distribution with mean μ. and a minimum value of $2R * Max_c$. This value is chosen to be greater than both $TTSF_{PB}$ and $TTSF_{BB}$ to satisfy Assumption 3.

- The task load L is defined as the expected number of task arrivals per mean service time and its value is approximately equal to $\frac{1}{\theta}C$, where C is the mean computation time.

4.5.1.1 Comparison of grouping techniques

In this section, we compare the performances of the no grouping, static grouping, and dynamic grouping under BB-overloading. In Figures 4.8 and 4.10, we have restricted the number of faults to be one at a time in order to compare the grouping based algorithms with the no-grouping algorithm.

4.5.1.1.1 *Effects of task load*

The task load (L) has been varied in Figure 4.8. The figure shows that increasing L decreases the guarantee ratio for all the algorithms. From the figure, it can be seen that dynamic grouping performs better than static grouping and the difference in performance is maximum for medium task loads. This can be explained as follows: For higher L, the guarantee ratio is lower for all the algorithms because the task load is much more than the capacity of the system. On the other hand, for lower L, the guarantee ratio is higher for all the techniques because the task load is less than the system capacity, which means that most of the tasks are schedulable by all the techniques. Also, note that the difference in performance between no-grouping and dynamic grouping is small which means that the dynamic grouping technique increases the system utilization to a point equal to the no-grouping technique and the difference in performance is partly due to the overhead cost associated with dynamic grouping.

4.5.1.1.2 *Effects of task laxity*

The effects of task laxity (R) is studied in Figure 4.9. As the laxity increases, the guarantee ratio also increases. For lower laxities, the difference in guarantee ratio between the dynamic

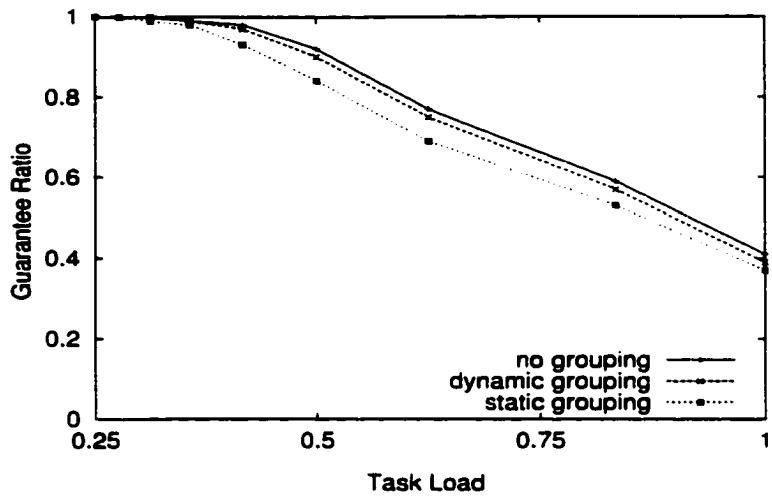


Figure 4.8 Effects of task load on GR for the grouping techniques ($N = 12$, $R = 6$)

and static grouping techniques is higher and decreases with increasing laxity. This is because, for lower values of laxity, the deadlines of tasks are very tight and hence the chances of rejecting the tasks is very high in both techniques. Dynamic grouping increases system utilization by allowing more backup overloading, which increases the guarantee ratio. From the figure, it can be seen that the dynamic grouping performs better than static grouping for all laxity values.

4.5.1.1.3 *Effects of number of processors*

Figure 4.10 shows the effects of varying the number of processors on the performance of the techniques. Increasing the number of processors for the same task load increases the guarantee ratio because adding more processors increases the capacity of the system. In other words, the average load per processor decreases with more processors for the same load. From the figure, we can see that dynamic grouping always offer more guarantee ratio than static grouping for all system size.

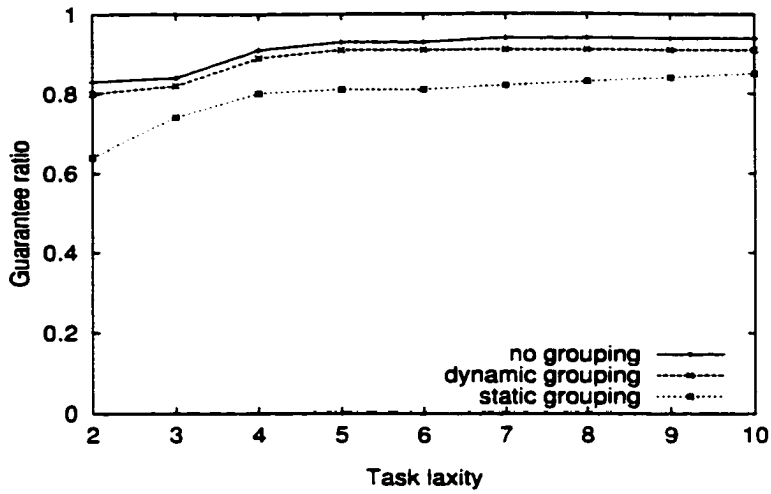


Figure 4.9 Effects of Task laxity on GR for the grouping techniques ($N = 12$, $L = 0.5$)

4.5.1.1.4 *Effects of number of faults*

Figure 4.11 shows the effects of varying number of fault occurrences in the system for task loads of 0.25, 0.5 and 1. The faults are generated following an exponential distribution with mean $\mu = 480$ *sec.* and minimum value equals to 320 *sec.* and they are randomly distributed in the system. The figure shows that the difference in the guarantee ratio between the techniques is significant at low load ($L = 0.25$) and medium load ($L = 0.5$) compared to at high load ($L = 1$). This is explained as follows: For light load (L=0.25), the dynamic grouping can compensate the degradation in guarantee ratio, due to faults, by rearranging the groups which is not possible in the static grouping. For full load (L=1), the dynamic grouping tends to behave similar to static grouping as all the processors are heavily loaded. The figure also shows that the guarantee ratio offered by the static grouping algorithm decreases more rapidly than the dynamic grouping as the number of faults increases due to the same reason.

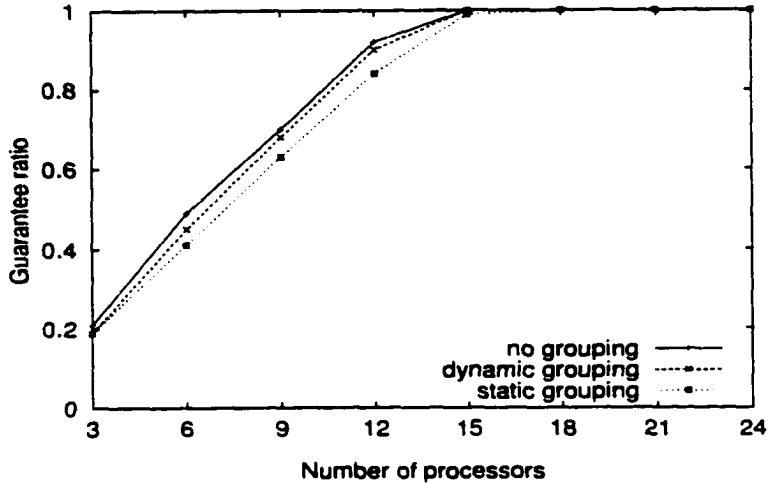


Figure 4.10 Effects of number of processors on GR for the grouping techniques ($R = 6$, $L = 0.5$)

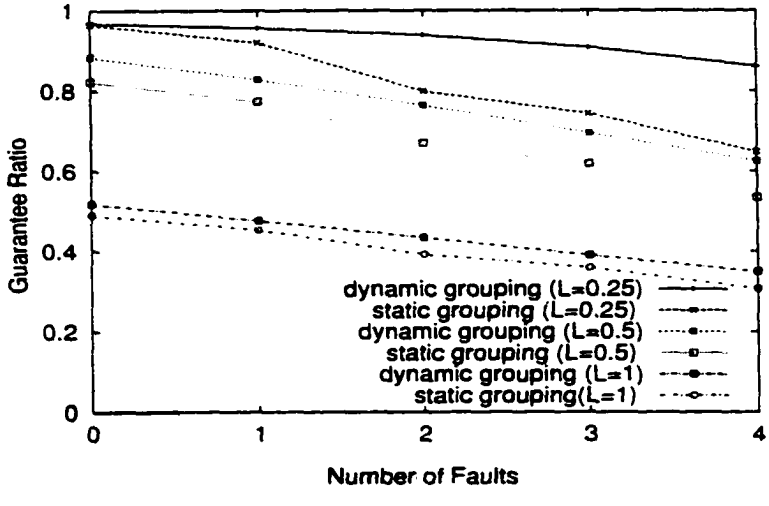


Figure 4.11 Effects of number of faults on GR for the grouping techniques ($N = 12$, $R = 6$)

4.5.1.2 Comparison of overloading techniques

In the previous subsection, we have shown that the proposed dynamic grouping indeed performs better than the static grouping under BB-overloading for all the conditions simulated. In this section, we compare our second proposal, PB-overloading, with BB-overloading and no-overloading under static grouping.

4.5.1.2.1 *Effects of task load*

The task load (L) is varied in Figure 4.12. As expected, increasing L decreases the guarantee ratio for all the techniques and the difference in performance between the techniques is maximum when the task load is greater than 0.5. This is because, for low loads, the task load is less than the system capacity and hence the techniques tend to behave similarly (i.e., there are no/less overloading taking place). The figure also shows that the guarantee ratio

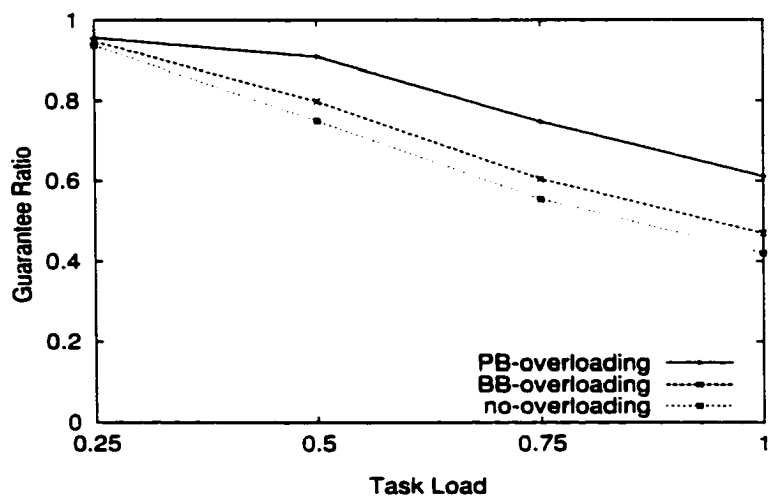


Figure 4.12 Effects of task load on GR for the overloading techniques ($N = 3$, $R = 6$)

offered by PB-overloading is better than BB-overloading and no-overloading for all task loads. The reason is that the PB-overloading can assign a start time for a primary earlier than that of BB-overloading because the primary can be overloaded on an already scheduled backup, which, in turn, helps its backup to find an earlier start time, thus resulting in a better chance

of meeting the deadline.

Figure 4.13 shows the effects of varying the task load on the $TTSF$ for the overloading techniques. The figure shows that the difference in the $TTSF$ between the PB- and BB-overloading techniques is significant at high load ($L = 1$) compared to at low load ($L = 0.25$). This is because, for low loads ($L = 0.25$), most of the tasks are schedulable even without overloading, thus making the techniques behave similarly. The figure also shows that $TTSF_{PB}$

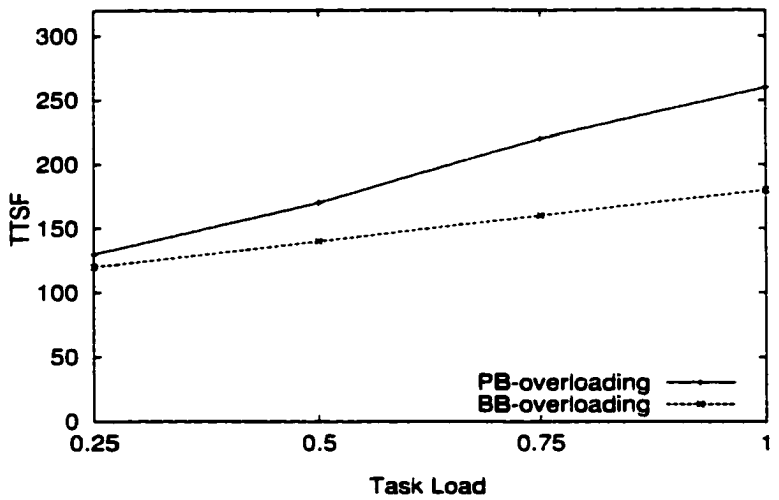


Figure 4.13 The effects of task load on $TTSF$ for the overloading techniques ($N = 3$, $R = 4$)

increases more rapidly than $TTSF_{BB}$ as the task load increases. This is because, $TTSF_{PB}$ is approximately twice that is offered by the BB-overloading technique and also due to better chances of overloading with increasing load. Whereas, the value of $TTSF_{BB}$ is approximately equal to the value that is offered by the no-overloading. The slight increase in $TTSF_{BB}$ is partly due to the increase in the number of tasks that are affected by the faults as the task load increases.

4.5.1.2.2 *Effects of task laxity*

The effects of task laxity (R) is studied in Figure 4.14. As expected, the guarantee ratio increases with increasing laxity. For lower laxities, the difference in guarantee ratio between

the PB- and BB- overloading techniques is higher and decreases with increasing laxity. From the figure, it can be seen that the PB-overloading performs better than BB-overloading for all laxity values for the same reasons.

4.5.1.2.3 *Effects of fault rate*

Figure 4.15 shows the effects of varying fault rate in the system for task loads of 0.25, 0.5 and 1. The faults are generated randomly using exponential distribution with minimum time between faults as 320, and the faults are randomly distributed in the system. The minimum time between faults is chosen to be greater than both $TTSF_{PB}$ and $TTSF_{BB}$. The figure shows that the difference in the guarantee ratio between the PB-overloading and BB-

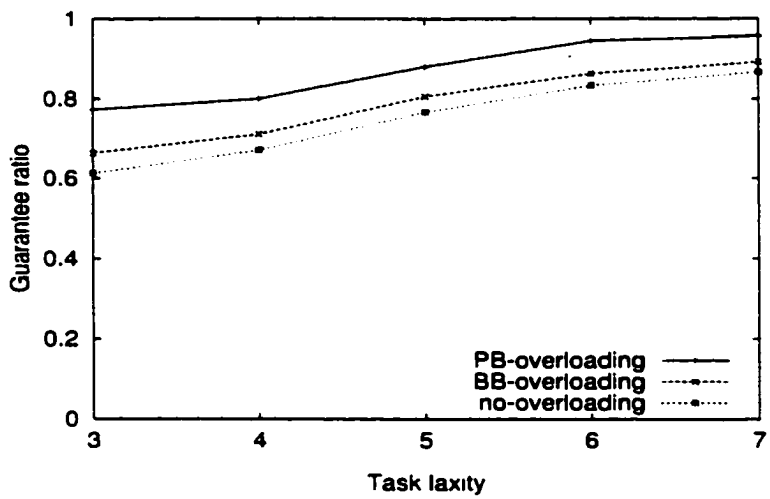


Figure 4.14 Effects of task laxity on GR for the overloading techniques ($N = 3$, $L = 0.5$)

overloading techniques is significant at high load ($L = 1$) compared to low load ($L = 0.25$). This is because, for low loads ($L = 0.25$), most of the tasks are schedulable even without overloading, thus making the techniques behave similarly. The figure also shows that the difference in guarantee ratio between the two techniques increases as the fault rate decreases. This is because, at high fault rates, processors fail more frequently which reduces the chances of overloading, thus making the techniques behave similarly.

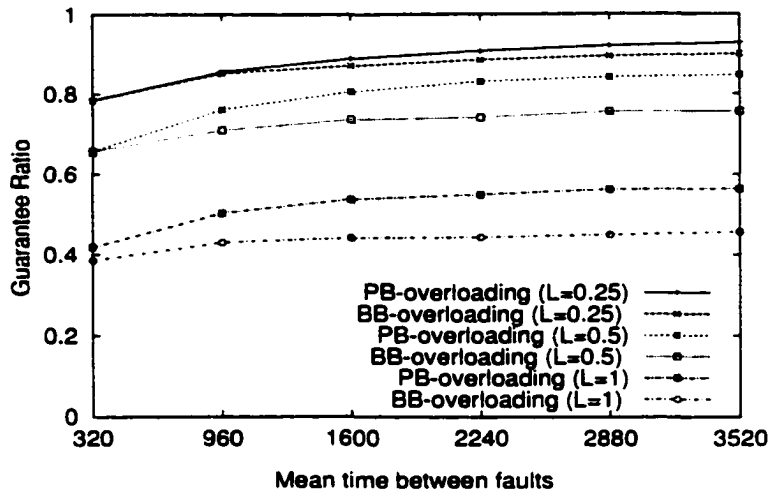


Figure 4.15 Effects of fault rate on GR for the overloading techniques ($N = 3$, $R = 4$)

4.5.2 Analytical Studies

In this section, we study analytically the effectiveness of the proposed grouping and overloading techniques in enhancing the system schedulability and/or reliability.

4.5.2.1 Schedulability analysis

The capability of a technique to schedule tasks can be measured in term of the probability of a task being rejected. In this section, We show that the probability of rejection in BB-overloading is higher than in PB-overloading, and the probability of rejection in static grouping is higher than in dynamic grouping. The analysis follows the methodology used in [23, 24, 57]. We use the following assumptions:

- All tasks have unit worst case computation time, i.e., $c_i = 1$.
- Backup slots are preallocated in the schedule.
- FIFO scheduling strategy is used.
- Task deadlines follow (discrete) uniform distribution in the interval $[W_{min}, W_{max}]$ relative to their ready times. We call this, *deadline window*. If $P_{win}(w)$ is the probability that

an arriving task has a relative deadline w, then $P_{win}(w) = \frac{1}{W_{max}-W_{min}+1}$, where $W_{min} \leq w \leq W_{max}$.

- Task arrivals follow (discrete) uniform distribution with mean $A_{av} = \frac{A_{max}}{2}$ in the interval $[0, A_{max}]$. If $P_{ar}(k)$ is the probability of k tasks arriving at a given time, then $P_{ar}(k) = \frac{1}{A_{max}+1}$, where $0 \leq k \leq A_{max}$.

4.5.2.1.1 *Pre-allocation strategies*

The pre-allocation strategies presented here are for a system with three processors (i.e., one group). When the system size is greater than three, one of the grouping techniques (static or dynamic) can be used to divide the system into groups and the pre-allocation strategies are applied on a group basis (see the next section).

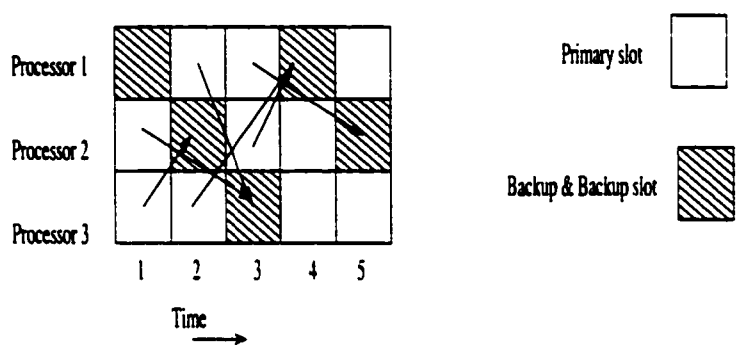


Figure 4.16 Pre-allocation strategy for BB-overloading

A simple pre-allocation policy for BB-overloading (as given in [23]) is to reserve a slot for backups every n (n is the number of processors, which is 3) time slots on each processor. Backup slots on the three processors can be staggered (Figure 4.16), that is, if a backup slot is pre-allocated at time t on processor P_i, then a backup slot is pre-allocated at time $t+1$ on processor $P_{(i+1) mod\ 3}$. Specifically, if a backup slot is pre-allocated at time t on processor P_i, then any task scheduled to run at time $t-1$ on P_j, $j \neq i$, can use this slot as a backup. Because, the task scheduled to run on P_i at time $t-1$ cannot have its backup slot on the same processor (space exclusion), then this task can use the backup slot at time $t+1$, which

is on $P_{(i+1)mod\ 3}$. In other words, for a task T_i, Bk_i is scheduled immediately after Pr_i with probability 0.5 and is scheduled two slots later than Pr_i with probability 0.5. Note that, in this scheme, two backups can potentially be overloaded on the same backup slot.

To define a pre-allocation strategy for PB-overloading we need to identify three different types of time (0, 1, and 2), wherein any time t has a type i if $(t-1)\ mod\ 3 = i$. In our pre-allocation strategy, at any time t, the number of primaries that can be scheduled to start at that time is s_0 if t is of type 0, s_1 if t is of type 1, and s_2 if t is of type 2. Figure 4.17 shows that $s_0 = 2$, i.e., two primaries can be scheduled at two empty slots (e.g. $t = 1$);

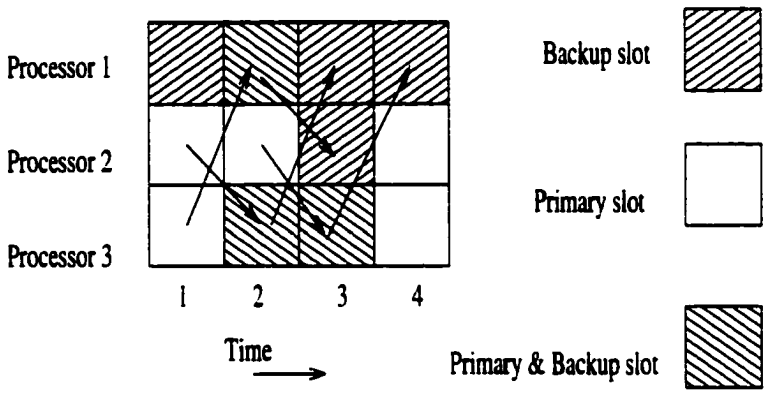


Figure 4.17 Pre-allocation strategy for PB-overloading

$s_1 = 3$, i.e., three primaries can be scheduled, where one of them at an empty slot and the other two are overlapped with backup slots (e.g. $t = 2$); and $s_2 = 1$, i.e., one primary can be scheduled to overlap with a backup slot (e.g. $t = 3$). In our pre-allocation strategy if a primary Pr_i is scheduled on processor P_i at time t its backup is scheduled at time $t+1$ in processor $P_{(i+1)\ mod\ 3}$. In other words, for a task T_i, Bk_i is scheduled immediately after Pr_i with probability 1. Note that, in this scheme, one backup and one primary can potentially be overloaded on the same slot.

4.5.2.1.2 *Analysis for the overloading algorithms*

Using FIFO scheduling is equivalent to maintaining a task queue, Q, to which arriving tasks are appended. Given that the number of task that can be scheduled on each time unit

is known, then the position of a task in the Q indicates its scheduled start time. In the pre-allocation strategy for the BB-overloading, two tasks can be scheduled on each time t (one slot is reserved for backups). If at the beginning of time slot t, a task T_i is the q^{th} task in Q, then T_i is scheduled to execute at time slot $t + g_q^{BB}$. Where g_q^{BB} is the time at which a task, whose position in the Q is q ($q = 1, 2, \ldots 2W_{max}$), will be executed and is defined as

$$g_q^{BB} = \lfloor \frac{q}{2} \rfloor. \tag{4.2}$$

In the pre-allocation strategy for the PB-overloading s_0, s_1, or s_2 tasks can be scheduled on a given time slot t depending on whether t is of type 0, 1, or 2, respectively. In this technique, the time g_q^{PB} is defined as

$$g_q^{PB} = (i+j+l) \text{ such that } \left[\sum_{c=1}^{i} s_0 + \sum_{c=1}^{j} s_1 + \sum_{c=1}^{l} s_2 \right] \leq q-1, \text{ and } |i-j| \leq 1, |j-l| \leq 1, |l-i| \leq 1. \tag{4.3}$$

where $i \geq j \geq l$ if t is of type 0, $j \geq l \geq i$ if t is of type 1, and $l \geq i \geq j$ if t is of type 2.

When a task T_i arrives at time t, its schedulability depends on the length of Q and on the relative deadline w_i of the task. In BB-overloading, if T_i is appended at position q of Q and $w_i \geq g_q^{BB}$, then the primary task, Pr_i, is guaranteed to execute before time $t + w_i$. Otherwise, the task is not schedulable since it will miss its deadline. Moreover, if $w_i \geq g_q^{BB} + 2$, then Bk_i is also guaranteed to execute before time $t + w_i$. In PB-overloading, if T_i is appended at position q of Q and $w_i \geq g_q^{PB}$, then the primary task, Pr_i, is guaranteed to execute before time $t + w_i$. Otherwise, the task is not schedulable since it will miss its deadline. Moreover, if $w_i \geq g_q^{BB} + 1$, then Bk_i is also guaranteed to execute before time $t + w_i$.

The dynamics of the above schemes can be approximately modeled using discreet time Markov process. For simplicity of presentation, a system without deadlines is modeled first, that is, a system in which no tasks are rejected. Such a system may be modeled by a linear discreet time Markov chain in which each state represents the number of tasks in Q and each transition represents the change in the length of the Q in one unit of time. The probabilities of different transitions may be calculated from the rate of task arrival, and the average number of tasks executed in a unit of time. The average number of task executed at a unit of time is two

for the BB-overloading and is also two ($= \frac{s_0+s_1+s_2}{3}$) for the PB-overloading. If S_u represents the state in which Q contains u tasks and $u \geq 2$, then the probability of a transition from S_u to S_{u-2+k} is $P_{ar}(k)$. This is because, at any time t, k tasks can arrive and two tasks get executed for the two schemes. If $u < 2$, then only u tasks are executed, then there is a state transition from S_u to S_k with probability $P_{ar}(k)$. For example, Figure 4.18 shows the transitions out of state S_u $(u > 2)$ in the Markov chain assuming that $A_{max} = 6$.

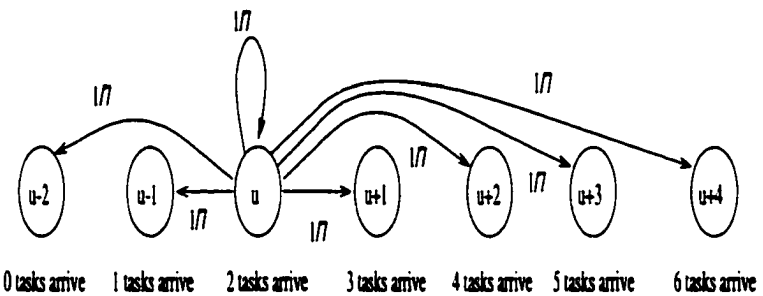


Figure 4.18 Transitions out of state S_u for a linear Markov chain

Now we consider the case of tasks with deadlines. When the k arriving tasks have finite deadlines, some of these tasks may be rejected. Let $P_{q,k}$ be the probability that one of the k tasks is rejected when the queue size is q. For the BB-overloading scheme the value of $P_{q,k}^{BB}$ is the probability that the relative deadline of the task is smaller than $g_b^{BB} + \sigma$, where $b = q + k/2$ (average case) and σ is the extra time needed to schedule the backup and is equal to 1 or 2 with a probability of 0.5. For the PB-overloading scheme the value of $P_{q,k}^{PB}$ is the probability that the relative deadline of the task is smaller than $g_b^{PB} + 1$, where $b = q + k/2$ (average case) and the extra one time unit is needed to schedule the backup. Then,

$$P_{q,k}^{*} = P_{win}(w < \bar{t}) = \begin{cases} 0 & \text{for } \bar{t} < W_{min} \\ 1 - \sum_{w=\bar{t}}^{W_{max}} P_{win}(w) & \text{for } W_{min} \leq \bar{t} \leq W_{max} \\ 1 & \text{for } \bar{t} > W_{max} \end{cases} \tag{4.4}$$

where $* = BB$, and $\bar{t}$=$g_b^{BB} + \sigma$ for the BB-overloading scheme, or $* = PB$ and $\bar{t}$=$g_b^{PB} + 1$ for the PB-overloading scheme. Hence, when the queue size is q, the probability, $P_{rej}^{*}(r, k, q)$, that r out of k tasks are rejected is

$$P_{rej}^{*}(r, k, q) = C_r^k \left(P_{q,k}^{*}\right)^r \left(1 - P_{q,k}^{*}\right)^{k-r} \tag{4.5}$$

where C_r^k is the number of possible ways to select r out of k elements, and $* = BB$ for the BB-overloading scheme, or $* = PB$ for the PB- overloading scheme.

In order to keep track of the number of rejected tasks, each state S_u is divided into $A_{max}+1$ states, $S_{u,r}$, $r = 0, \ldots, A_{max}$, where A_{max} is the maximum number of tasks arriving, and possibly rejected, in each time unit. In the two-dimensional discreet time Markov chain, $S_{u,r}$ represents the queue size as u and r tasks were rejected when the transition was made into $S_{u,r}$. The two-dimensional Markov chain contains $2W_{max} + 1$ rows (maximum Q length $+1$) and $A_{max} + 1$ columns (number of arrivals in unit time $+1$), and the transition probabilities become:

- If $u \geq 2$, then $P\{S_{u,i} \rightarrow S_{u-2+k-r,r}\}$=$P_{ar}(k)P_{rej}(r, k, u-2)$,
- If $u < 2$, then $P\{S_{u,i} \rightarrow S_{k-r,r}\}$=$P_{ar}(k)P_{rej}(r, k, 0)$,

where $k = 0, \ldots, A_{max}$ and $r = 0, \ldots, k$.

By computing the steady state probabilities of being in the rejection states, it is possible to compute the expected value of the number of rejected tasks (Rej) per time unit. If $P_{ss}(u, v)$ is the steady state probability of being in state $S_{u,v}$, then

$$Rej = \sum_{u=0}^{2W_{max}} \sum_{v=1}^{A_{max}} (vP_{ss}(u, v)). \tag{4.6}$$

Then, the rate of task rejection is given by Rej/A_{av}. Note that $P_{ss}(u, 0)$ is not included in (4.6) since these are the states corresponding to no rejection.

Up until now, we have not considered backup de-allocation in the model. Backup de-allocation means that if at time t no fault has occurred, then the backups pre-allocated at time slot $t + 1$ may be used to schedule a new task. In other words, if k tasks arrive during slot t, and $k > 0$, then one of these tasks can be scheduled in the de-allocated backup slot, and the remaining $k - 1$ tasks can be treated as above. The effect of backup de-allocation may be analyzed by changing k by $k - 1$ in the previous analysis. More specifically for g_b^*, $b = q + (k - 1)/2$.

4.5.2.1.3 *Analysis for the grouping algorithms*

To compare dynamic grouping and static grouping we use BB-overloading pre-allocation strategy. Figure 4.19 shows pre-allocation strategy for BB-overloading in a six processors system.

If static grouping is used in this system, then processors 1, 2, and 3 are in one group (G_1) and processors 4, 5, and 6 are in another group (G_2). In order to estimate the loss of schedulability caused by static grouping, we use the same FIFO scheduling algorithm as in the previous section. We assume that the number of arrival tasks (k) at any time t are divided equally between the two groups (G_1 and G_2). In other words, if k tasks arrive during slot t, then $\lfloor k/2 \rfloor$ of these tasks are scheduled in group G_1, and the remaining are scheduled in group G_2. The effect of static grouping may be analyzed by changing k by k/g in the previous analysis for BB-overloading algorithm, where $g = n/3$ is the number of group in the system, and n is the number of processors. More specifically for g_b^{BB}, b is equal to $q + k/2g$. Then the probability that one of the k tasks is rejected when the queue size is q for static grouping is given also by Equation (4.4), where $\bar{t}=g_b^{static}+\sigma$, σ is equal to 1 or 2 with probability 0.5, and

$$g_b^{static} = \lfloor \frac{q + k/2g}{2} \rfloor. \tag{4.7}$$

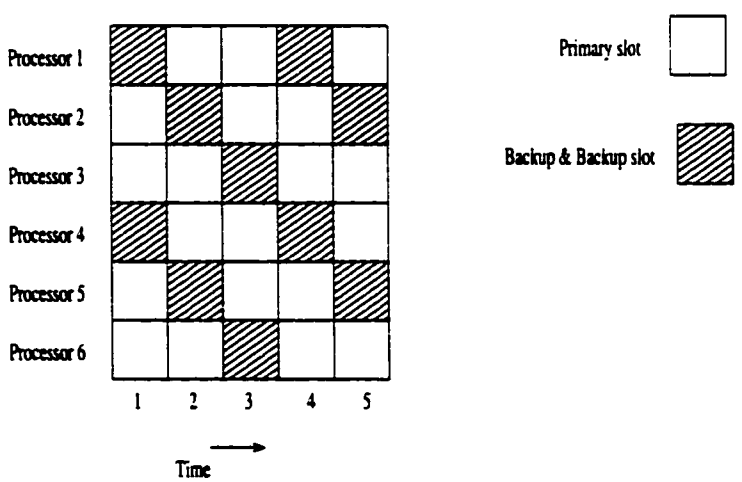


Figure 4.19 Pre-allocation strategy for BB-overloading

If dynamic grouping is used in Figure 4.19, then the groups are formed after scheduling the arrival tasks. If the FIFO scheduling algorithm is used in the dynamic grouping, then four tasks $(n - n/3)$ can be scheduled on each time t in Figure 4.19 (two slots, $n/3$, are reserved for backups). If, at the beginning of time slot t, a task T_i is the q^{th} task in Q, then T_i is scheduled to execute at time slot $t + g_q^{dynamic}$, where $g_q^{dynamic}$ is the time, from now, at which a task will execute whose position in the Q is q $(q = 1, 2, \ldots, 4W_{max})$ and is defined as

$$g_q^{dynamic} = \lfloor \frac{q}{(n - \frac{n}{3})} \rfloor. \tag{4.8}$$

When k tasks arrive at any time unit t, then the probability that one of the k tasks is rejected when the queue size is q for dynamic grouping is given also by Equation (4.4), where $\bar{t}=g_b^{dynamic} + \sigma$, σ is equal to 1 or 2 with probability 0.5, and $b = q + k/2$.

4.5.2.1.4 *Results*

From the analysis, we notice that the performance of a technique depends on the probability of rejecting a task $(P^*_{q,k})$ for that technique. Figure 4.20 shows the probability of rejecting a task, for the overloading techniques, as the number of arrived tasks (k) vary for both faulty

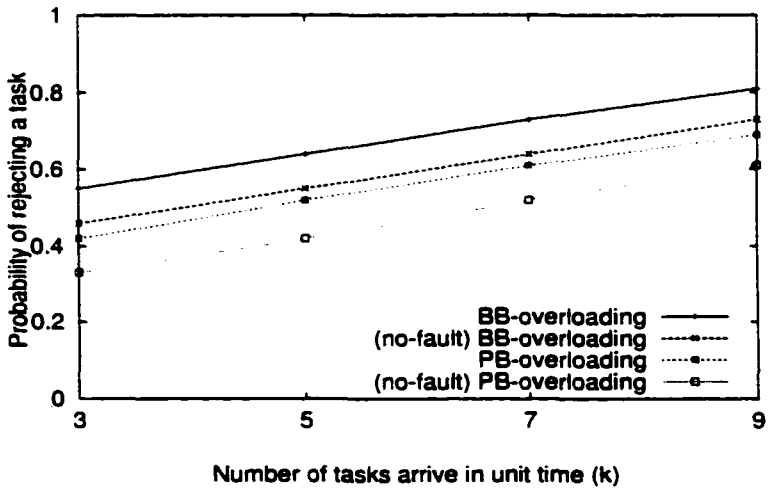


Figure 4.20 Effects of task load on $P^*_{q,k}$ for the overloading techniques $(W_{max} = 5,\ W_{min} = 3, A_{max} = 9)$

and fault-free cases. The figure shows that PB-overloading technique has a lesser probability to reject a task for all values of k for both faulty and fault-free cases. The figure also shows that the probability of rejecting a task increases as task load increases for both techniques. Figure 4.21 shows the probability of rejecting a task as W_{min} varies for the BB- and PB-overloading techniques. In the figure, for both techniques, the task rejection probability decreases as the deadline window (task laxity) increases.

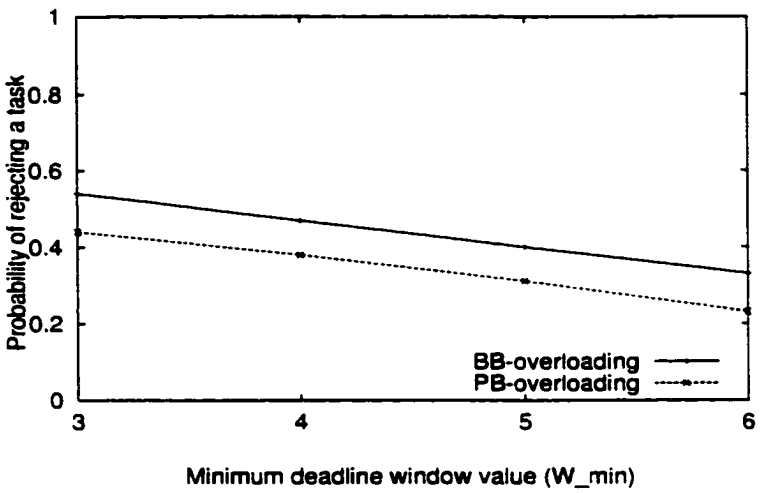


Figure 4.21 Effects of task laxity on $P^*_{q,k}$ for the overloading techniques ($W_{max} = 7, A_{max} = 6$)

Figure 4.22 shows the probability of rejecting a task,for the grouping techniques, as the number of task arrival (k) varies for both faulty and fault-free cases. The figure shows that dynamic grouping technique has less probability to reject a task for all values of k for both faulty and fault-free cases. Also the figure shows that the probability of rejecting a task increases as the task load increases for the two schemes. Figure 4.23 shows the probability of rejecting a task as W_{min} varies for the dynamic grouping and static grouping techniques. In the figure, for both techniques, the task rejection probability decreases as the deadline window (task laxity) increases.

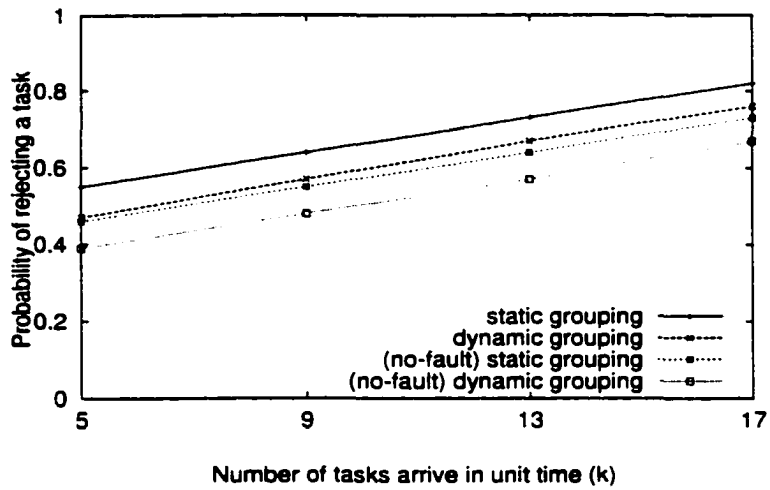


Figure 4.22 Effects of task load on $P^*_{q,k}$ for the grouping techniques ($W_{max} = 5$, $W_{min} = 3$,$A_{max} = 17$)

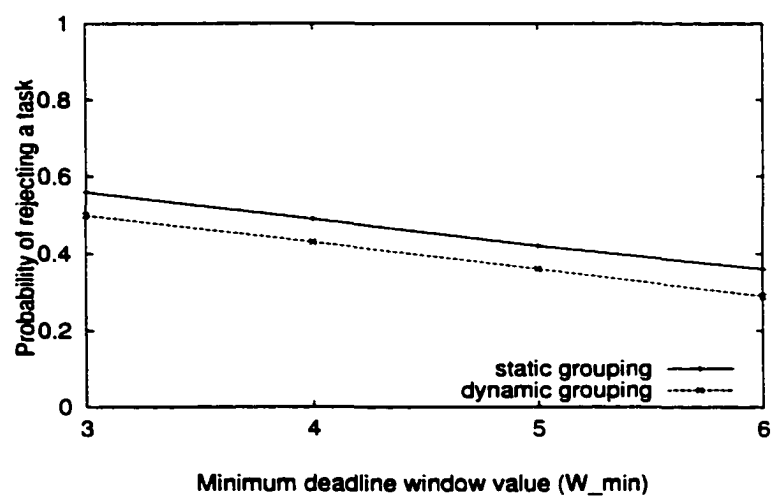


Figure 4.23 Effects of task laxity on $P^*_{q,k}$ for the the grouping techniques ($W_{max} = 7$,$A_{max} = 12$)

4.5.2.2 Reliability analysis

The capability of a fault-tolerant technique is assessed based on the number of and the frequency of faults that it can tolerate. The resilience of the system can be measured in terms of the time it takes for the system to be able to tolerate a second fault after the first fault using a given fault-tolerance technique. This latency is called the time to second fault ($TTSF$) [24]. The higher the $TTSF$, the poorer the performance of the fault-tolerant technique.

In the previous section, we have shown that PB-overloading technique offers better schedulability than BB-overloading. Here, we show that $TTSF_{PB}$ offered by the PB-overloading is bounded by twice that of the BB-overloading ($TTSF_{BB}$). Note that, a typical value for $TTSF$ is much smaller than the $MTTF$ (Assumption 3 of the fault model) and hence both the overloading techniques offer a similar reliability from the practical point of view. Theorems 1 and 2 below quantify the $TTSF$ offered by BB-overloading and PB-overloading respectively.

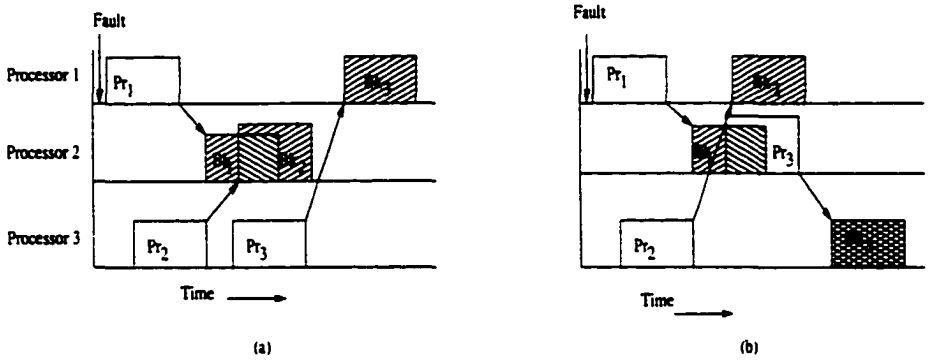


Figure 4.24 Tolerating a second fault

Theorem 1: In BB-overloading technique if a permanent fault occurs at time t in processor P_i, the technique will continue to operate if another fault does not occur in the group before time $\bar{t}$, where

$$\bar{t} > \max\{max_j\{ft(Bk_j) : Proc(Pr_j) = P_i\},\ max_j\{ft(Pr_j) : Proc(Bk_j) = P_i\}\} \qquad (4.9)$$

Proof: If a permanent fault occurs at time t in P_i, any task arriving later than t will be scheduled (both primary and backup) on the $n-1$ non-faulty processors (assumption 4 of the

fault model). Thus, such tasks are guaranteed to complete even if a second fault occurs. If a task T_j is already scheduled when the first fault occurs, then the following two cases arise:

case 1: $Proc(Pr_j) = P_i$ or $Proc(Bk_j) = P_i$: In this case, the restriction on $\bar{t}$ guarantees that Bk_j or Pr_j will execute successfully before the second fault occurs. The first part of the restriction on $\bar{t}$ (in Equation 4.9) guarantees that the second fault can occur only after all backups, whose primaries on the faulty processor, have been executed. The second part in Equation 4.9 ensures that the second fault occurs only after all primaries, whose backups on the faulty processor, have been executed. For example, in Figure 4.24a, primary Pr_1 is scheduled on P_1 and its backup Bk_1 on P_2. If a fault occurs on P_1 before Pr_1 executes, then another fault can be tolerated on P_2 only after Bk_1 completes. If a second fault occurs on P_2 before $ft(Bk_1)$, both copies of T_1 would be faulty. Using a similar logic, a second fault can be tolerated on P_3 only after Pr_3 completes. The maximum of all such combinations gives the minimum time at which the second fault can occur.

case 2: $Proc(Pr_j) \neq P_i$ and $Proc(Bk_j) \neq P_i$: In this case, T_j is guaranteed to complete even if a second fault occurs unless Bk_j overlaps with a backup Bk_k whose primary Pr_k is scheduled on P_i (for example, in Figure 4.24a, if $i = 1$, $j = 2$, and $k = 1$). Due to the first fault, Bk_k is activated and hence Bk_j cannot be used (since it overlaps Bk_k). Therefore, a second fault cannot be tolerated on $Proc(Pr_j)$ (P_3 in Figure 4.24a) until Pr_j has executed. However, this case is covered by the first part of the restriction on $\bar{t}$ (in Equation 4.9), according to which the second fault cannot be tolerated before Bk_k has executed. Since Bk_j and Bk_k overlap, Pr_j is scheduled earlier than $ft(Bk_j)$ in the system. This means that the second fault can be tolerated only after Pr_j finish its execution successfully.

Theorem 2: In PB-overloading if a permanent fault occurs at time t in processor P_i, the technique will continue to operate if another fault does not occur in the group before time $\bar{t}$, where

$$\bar{t} > \max\{max_j\{ft(Bk_j) : Proc(Pr_j) = P_i\},\ max_j\{ft(Pr_j) : Proc(Bk_j) = P_i\},$$
$$max_j\{ft(Bk_k) : Proc(Pr_k) = Proc(Bk_j),\ S(Pr_k) \cap S(Bk_j) \neq \emptyset,\ and\ Proc(Pr_j) = P_i\}\} \tag{4.10}$$

Proof: The first two parts in Equation 4.10 have the same proof as the BB-overloading. For the third part consider the following case

case 3: $Proc(Pr_j) \neq P_i$ and $Proc(Bk_j) \neq P_i$: In this case, T_j is guaranteed to complete even if a second fault occurs unless Pr_j overlaps with a backup Bk_k whose primary Pr_k is scheduled on P_i (for example, in Figure 4.24b, if $i = 1$, $j = 3$, and $k = 1$). Due to the first fault, Bk_k is activated and hence Pr_j cannot be used (since it overlaps Bk_k). Therefore, a second fault cannot be tolerated on $Proc(Bk_j)$ (P_3 in Figure 4.24b) until Bk_j has executed. This case is covered by the third part of the restriction on $\bar{t}$ (in Equation 4.10), according to which a second fault can occur only after all backups, whose primaries were overlapped with backups of primaries on the faulty processor, have executed. The maximum of all such combinations gives the minimum time at which the second fault can occur.

$TTSF$ can be calculated as $\bar{t} - t$. In the worst case, $TTSF_{PB} \leq 2 \times TTSF_{BB}$. For example in Figure 4.24a, if the fault was transient and it affect Pr_1 only then, the BB-overloading technique will continue to operate if another fault does not occur before $ft(Bk_1)$. In the worst case this interval is $[r_1, d_1]$. On the other hand in Figure 4.24b, the PB-overloading technique will continue to operate if another fault does not occur before $ft(Bk_3)$. In the worst case, this interval is $[r_1, d_3]$ which is approximately $2 \times [r_1, d_1]$.

4.6 Summary

In this chapter, we have proposed two new techniques to be used in a primary-backup based fault-tolerant dynamic scheduling algorithm in multiprocessor real-time systems. The first technique is called *dynamic grouping,* in which the processors are dynamically grouped into logical groups in order to achieve efficient overloading of tasks, thereby improving the

schedulability and the reliability of the system. The second technique is called *primary-backup-overloading*, in which the primary of a task can be scheduled into the same or overlapping time interval with the backup of another task on a processor in order to increase system utilization, thereby improving the schedulability. The proposed techniques can be incorporated easily into any dynamic scheduling algorithm. For our simulation studies, we have incorporated the proposed techniques into the Spring scheduling algorithm, a well-known dynamic scheduling algorithm.

Our simulation and analytical studies show that the proposed *dynamic grouping* technique offers significantly better guarantee ratio (15% gain) than the static grouping, and offers a graceful degradation in the performance (guarantee ratio) as the number of faults increase under all the interesting conditions that we have simulated in the system. The proposed *primary-backup overloading* offers better schedulability (25% gain) than BB-overloading also under all the interesting conditions that we have simulated in the system. We have also shown that the $TTSF_{PB}$ (a reliability metric) of PB-overloading is upper bounded by twice that of BB-overloading ($TTSF_{PB}$), which is a much smaller value than the $MTTF$ of the system. Hence, PB-overloading is a more effective technique than BB-overloading for many practical reliability requirements.

CHAPTER 5. A CASE OF SCHEDULABILITY-RELIABILITY TRADE-OFFS IN SOFT REAL-TIME SYSTEMS

In this chapter, we propose an adaptive PB scheduling scheme that makes use of an estimate of the primary fault probability and task's laxity to control the degree of overlap between its (task) versions. Two variations of the adaptive scheme are proposed by varying the adaptation mechanism. The adaptation can be done in a continuous manner which leads to an approach called PB-OVER continuous (PB-OVER-CONT), or it can be in a discrete manner which leads to an approach called PB-OVER switch (PB-OVER-SWITCH). In PB-OVER-CONT, the overlap interval varies from no overlap to full overlap in a continuous manner as the fault probability varies from 0 to 1. In PB-OVER-SWITCH, the scheduler uses a threshold value of fault probability to switch from PB-CONCUR to PB-EXCL, i.e., if the probability is less than the threshold, the adaptive scheduler behaves like PB-EXCL, otherwise it behaves like PB-CONCUR. In PB-OVER-SWITCH, the threshold value is adapted with task's laxity.

In section 5.1, the fault model is stated. In section 5.2, the performance index is proposed. In section 5.3, we analytically analyze and compare the performance of the existing PB-based fault-tolerant approaches. In section 5.4, we propose the adaptive fault-tolerant approaches and analyze their performance by analytical and simulation models. In section 5.5, we present the general adaptive scheduling algorithm. In section 5.6, we discuss the implementation issues related to the adaptive fault-tolerant scheduling scheme. In section 5.7, we present the simulation studies of the PB fault-tolerant approaches. Finally, in section 5.8, we summarize the results.

5.1 Fault Model

We assume that each processor, except the scheduler, may fail due to a hardware fault which results in tasks failure. We further assume that the scheduler is made fault-tolerant by executing it on more than one processor or by using any other fault tolerance technique. The faults can be transient or permanent and are independent of each other. Each independent fault results in the failing of only one processor. The following assumptions form the fault model.

Assumption 1: The fault rate in the system changes with time. For real-time systems operating in unpredictable environments, the fault rate is not known a priori. However, system performance can be specified under a set of representative fault rate profiles borrowed from control theory [52]; namely, the *step fault rate* and the *ramp fault rate.* In the context of real-time systems, the step fault rate represents the worst-case fault rate variation, and the ramp fault rate represents a nominal form of fault rate variation. A fault rate profile $FR(t)$ is the system fault rate as a function of time. The fault rate profiles are defined as follows.

- Step fault rate $SFR(t)$: a fault rate profile that instantaneously jumps from a nominal fault rate FR_{nom} to fault rate FR_{max} and stays constant after the jump. The step fault rate is represented with a tuple $SFR(FR_{nom}, FR_{max})$.

- Ramp fault rate $RFR(t)$: a fault rate profile that increases linearly from the nominal fault rate to a specific level of fault rate during a time interval. Compared with the step fault rate, the ramp signal represents a less severe and more realistic fault rate variation scenario. The ramp fault rate $RFR(t)$ is described with a tuple $RFR(FR_{nom}, FR_{max}, T)$, where FR_{nom} is the original fault rate, FR_{max} is the new fault rate, and T is the time it takes the fault rate to increase from FR_{nom} to FR_{max}.

Assumption 2: We assume that the $(d_i^f - r_i)\ \forall\ T_i$ are much smaller than the typical mean time to failure ($MTTF$) value of the system. $MTTF$ of the system is defined as the expected time for which the system operates before the first failure occurs. This assumption is used to enhance the probability of successfully executing the backup of a task, if its primary fails. Note

that, there is always the chance that both the primary and backup of a task can fail especially when the fault rate is high; this potentially leads to system failure. In our work we assume that there will be another mechanism that is activated if such a case occurs. Nevertheless, the functionality of the proposed techniques do not change if the backup fails but these techniques do not tolerate such a case.

Assumption 3: There exists fault-detection mechanisms such as a fail-signal and an acceptance test that detect processor and task failures, respectively. The scheduler will not schedule tasks to a known faulty processor.

5.2 Performance Index

The performance index that we use to compare the PB-based fault-tolerant approaches capture the trade-off between the processor utilization and the execution interval for each task. This performance index is called the *schedulability-reliability* index (SR).

For a task T_i, we define the processor utilization of the task as the ratio of worst case execution time (c_i) to the expected processor time used by the task for its execution $\overline{PT_i}$ (consumption time).

$$UTIL_i = \frac{c_i}{\overline{PT_i}} \tag{5.1}$$

We also define a value function that is used to credit the successful output from a task depending on its expected execution interval ($\overline{ET_i}$). Figure 5.1 shows the shape of the value function (VAL_i) wherein the task contributes a value of one if it finishes before its soft deadline, a monotonically decreasing value if it finishes between its soft and firm deadlines, a value of zero if it misses its firm deadline. The monotonically decreasing task value is inversely proportional to its expected execution interval.

$$VAL_i = \begin{cases} 1 & \text{for } \overline{ET_i} \leq d_i^s \\ \frac{d_i^s}{\overline{ET_i}} & \text{for } d_i^s \leq \overline{ET_i} \leq d_i^f \\ 0 & \text{for } \overline{ET_i} > d_i^f \end{cases} \tag{5.2}$$

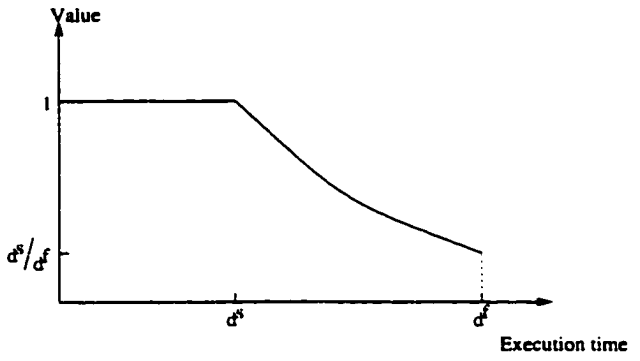


Figure 5.1 The value function used to credit the logical result

Therefore, the schedulability-reliability index (SR_i) for a task (T_i) is:

$$SR_i = UTIL_i \times VAL_i. \tag{5.3}$$

The schedulability-reliability index SR for a set of n tasks that are admitted into the system is computed as

$$SR = \frac{\sum_{i=1}^{n} SR_i}{n}. \tag{5.4}$$

As mentioned earlier, the performance index of a soft real-time system is usually measured as the sum of the values contributed by the admitted tasks. For a given system capacity, there are two options: (i) admit a few tasks with a higher value for each task or (ii) admit a large number of tasks with a lower value for each task. This trade-off is effectively captured by our schedulability-reliability index given in Equation (5.3). The effect on the number of admitted tasks is inherently captured by the processor utilization of each task. The higher the processor utilization of a task, the higher the number of admitted tasks and hence the better performance.

5.3 Analysis of the PB-Based Fault-Tolerant Approaches

In this section, we use simple assumptions about the system's parameters in order to mathematically analyze and compare the effect of the primary fault probability (f) on the performance of the three PB fault tolerance approaches (PB-EXCL, PB-CONCUR, PB-OVER).

5.3.1 Assumptions

1. The primary fault probability (f) is estimated using the fault monitoring mechanisms (see Section 5.6.1).
2. The worst case execution time for the tasks is fixed and is equal to c.
3. The relative soft deadline (d^s) for a task is equal to c, and its relative firm deadline (d^f) is equal to $2c$.

5.3.2 Primary-Backup EXCLusive (PB-EXCL)

In this approach, the primary and the backup versions of a task are excluded in space as well as in time in the schedule (as shown in Figure 5.2). From the figure, the expected execution interval $\overline{ET}$ for the task is:

$$\overline{ET} = 2cf + (1 - f)c = c(1 + f). \tag{5.5}$$

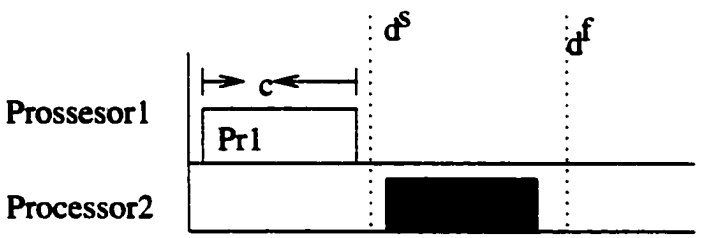


Figure 5.2 Primary-backup exclusive (PB-EXCL)

$\overline{ET}$ can take either of the following two values depending on whether the primary version of a task fails or not: (i) $\overline{ET} = 2c$ when the primary version of a task fails with probability f and the backup version succeeds; (ii) $\overline{ET} = c$ when the primary version succeeds with probability $1 - f$. In the latter case, the backup is deallocated.

The expected consumption time ($\overline{PT}$) for the task is given by

$$\overline{PT} = 2cf + (1 - f)c = c(1 + f). \tag{5.6}$$

Similarly, $\overline{PT}$ can take either of the following two values depending on whether the primary version of the task fails or not: (i) $\overline{PT} = 2c$ when the primary version of a task fails with probability f and the backup version succeeds; (ii) $\overline{PT} = c$ when the primary version succeeds with probability $1 - f$. In the latter case, the backup version is deallocated.

From these two equations, it can be seen that both $\overline{PT}$ and $\overline{ET}$ increase linearly from c to $2c$ as f varies from 0 to 1. Also, it can be observed that the PB-EXCL approach performs well when $f = 0$ as $\overline{ET} = \overline{PT} = c$, which is the best achievable. However, this approach performs very poorly when $f = 1$ as it doubles the execution interval for the task, i.e., $\overline{ET} = 2c$.

The processor utilization in this approach is given by:

$$UTIL_i = \frac{c}{(1+f)c} = \frac{1}{(1+f)}. \tag{5.7}$$

Since $d_i^s = c$ and $d_i^f = 2c$, the value function for this approach is $VAL_i = \frac{c}{(1+f)c} = \frac{1}{1+f}$. The schedulability-reliability index for this approach is then

$$SR_i = \frac{1}{(1+f)^2}. \tag{5.8}$$

5.3.3 Primary-Backup CONCURrent (PB-CONCUR)

In this approach, the primary and the backup versions of tasks are executed concurrently as shown in Figure 5.3. From the figure, the expected execution interval $\overline{ET}$ for the task is given by:

$$\overline{ET} = c. \tag{5.9}$$

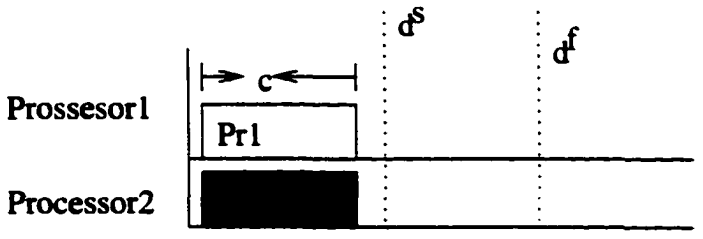


Figure 5.3 Primary-backup concurrent (PB-CONCUR)

This means that the execution interval for the task is constant and does not depend on f. Also, $\overline{PT} = 2c$ and does not depend on f. From these two equations, the approach is favorable when $f = 1$ as $\overline{ET} = c$ and $\overline{PT} = 2c$ for this case, which is the best that can be achieved. However, this approach is not favorable when $f = 0$ as $\overline{PT} = 2c$ which is twice that of PB-EXCL for the same case.

The processor utilization for this approach is given by

$$UTIL_i = \frac{c}{2c} = 0.5. \tag{5.10}$$

Since $d_i^s = c$ and $d_i^f = 2c$, the value function for this approach is $VAL_i = \frac{c}{c} = 1$. The schedulability-reliability index for this approach is then

$$SR_i = 0.5 \tag{5.11}$$

5.3.4 Primary-Backup OVERlap (PB-OVER)

In this approach. the primary and the backup versions of a task can overlap in execution by an amount equal to γc as shown in Figure 5.4.

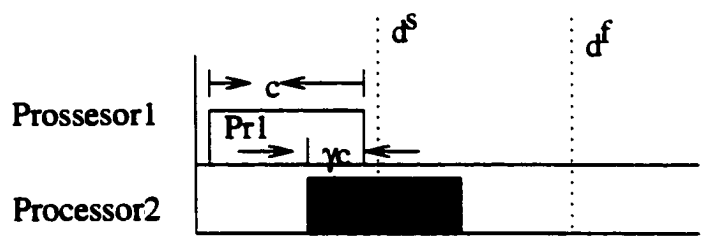


Figure 5.4 Primary-backup overlap (PB-OVER)

From the figure, the expected execution interval $\overline{ET}$ for the task is given by

$$\overline{ET} = f(c + (1-\gamma)c) + (1-f)c = cf(1-\gamma) + c. \tag{5.12}$$

$\overline{ET}$ can take either of the following two values depending on whether the primary version of the task fails or not: (i) $\overline{ET} = (c + (1-\gamma)c)$ when the primary version fails with probability

f and the backup version succeeds; (ii) $\overline{ET} = c$ when the primary version succeeds with probability $1 - f$.

The expected consumption time $\overline{PT}$ for the task is

$$\overline{PT} = 2cf + (1 - f)(c + \gamma c) = (1 - \gamma)cf + (1 + \gamma)c. \tag{5.13}$$

In this approach, $\overline{PT}$ can take either of the following two values depending on whether the primary version of the task fails or not: (i) $\overline{PT} = 2c$ when the primary version of a task fails with probability f and the backup version succeeds; (ii) $\overline{PT} = (1+\gamma)c$ when the primary succeeds with probability $1-f$ since a processor time of $(1-\gamma)c$ is reclaimed due to deallocation of part of the backup version.

The processor utilization for this approach is given by

$$UTIL_i = \frac{c}{(1 + f + \gamma(1 - f))c} = \frac{1}{1 + f + \gamma(1 - f)}. \tag{5.14}$$

Since $d_i^s = c$ and $d_i^f = 2c$, the value function for this approach is

$$VAL_i = \frac{c}{(1 + f(1 - \gamma)c} = \frac{1}{1 + f(1 - \gamma)}. \tag{5.15}$$

The schedulability-reliability index for this approach is then

$$SR_i = \frac{1}{(1 + f + \gamma(1 - f))(1 + f(1 - \gamma))}. \tag{5.16}$$

The most important property that is expected from PB-OVER is to combine the advantages of the PB-EXCL and PB-CONCUR approaches. That is, when $f = 0$, the desirable values of $\overline{ET}$ and $\overline{PT}$ is c, and when $f = 1$, the desirable values of $\overline{ET}$ and $\overline{PT}$ are c and $2c$, respectively.

5.4 Adaptive PB-OVER Fault-Tolerant Approaches

In this section, two adaptive PB-OVER approaches are introduced by varying the adaptation mechanism. The adaptation can be done in a continuous manner which leads to an approach called PB-OVER continuous (PB-OVER-CONT), or it can be in a discrete manner which leads to an approach called PB-OVER switch (PB-OVER-SWITCH).

5.4.1 Primary-Backup OVERlap CONTinuous (PB-OVER-CONT)

In PB-OVER-CONT, the overlap interval varies from no overlap to full overlap in a continuous manner as the fault probability varies from 0 to 1. From Equations (5.12) and (5.13), we found that this can be achieved by substituting $\gamma = f$ which results in

$$\overline{ET} = (-f^2 + f + 1)c \quad and \quad \overline{PT} = (-f^2 + 2f + 1)c. \tag{5.17}$$

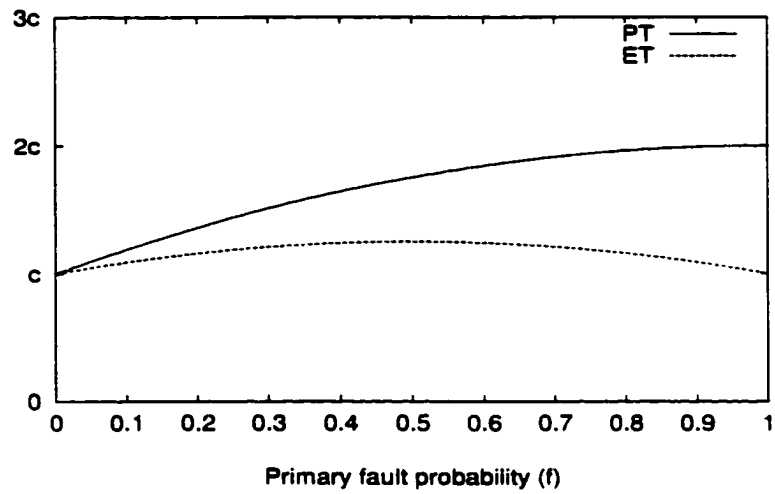


Figure 5.5 $\overline{PT}$, and $\overline{ET}$ for the PB-OVER-CONT approach

Figure 5.5 shows the values of $\overline{PT}$, and $\overline{ET}$ for the PB-OVER-CONT approach with varying f. From the figure, it can be seen that $\overline{PT}$ increases quadratically from c to $2c$ as f varies from 0 to 1. $\overline{ET}$ increases quadratically from c to $1.25c$ as f varies from 0 to 0.5 and then decreases quadratically from $1.25c$ to c as f varies from 0.5 to 1.

Using the assumption in Section 5.3 the processor utilization for this approach is given by

$$UTIL_i = \frac{c}{(-f^2 + 2f + 1)c} = \frac{1}{-f^2 + 2f + 1}. \tag{5.18}$$

Since $d_i^s = c$ and $d_i^f = 2c$, the value function for this approach is

$$VAL_i = \frac{c}{(-f^2 + f + 1)c} = \frac{1}{(-f^2 + f + 1)}. \tag{5.19}$$

The schedulability-reliability index for this approach is then

$$SR_i = \frac{1}{(-f^2 + 2f + 1)(-f^2 + f + 1)}. \tag{5.20}$$

5.4.2 Primary-Backup OVERlap SWITCH (PB-OVER-SWITCH)

In PB-OVER-SWITCH, the scheduler switches from PB-EXCL to PB-CONCUR depending on the value of f. If f is less than a threshold f_o, then the PB-OVER-SWITCH approach behaves like PB-EXCL, else it behaves like PB-CONCUR. The threshold f_o is the value of f at which the schedulability-reliability index of PB-EXCL is equal to that of PB-CONCUR. Thus, $\overline{ET}$ and $\overline{PT}$ of the task become

$$\overline{ET} = \begin{cases} c \times (1+f) & \text{for } 0 \leq f \leq f_o \\ c & \text{for } f_o < f \leq 1 \end{cases} \tag{5.21}$$

$$\overline{PT} = \begin{cases} c \times (1+f) & \text{for } 0 \leq f \leq f_o \\ 2c & \text{for } f_o < f \leq 1 \end{cases} \tag{5.22}$$

For the given assumption the value of f_o is the value of f that satisfies $\frac{1}{(1+f)^2} = 0.5$, which is $f = \sqrt{2} - 1$.

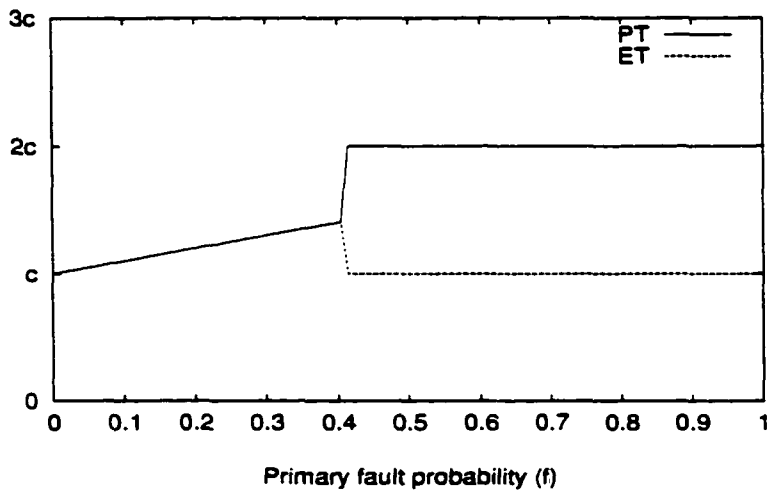


Figure 5.6 $\overline{PT}$, and $\overline{ET}$ for the PB-OVER-SWITCH approach

Figure 5.6 shows the values of $\overline{PT}$ and $\overline{ET}$ of the PB-OVER-SWITCH approach for varying f. From the figure, it can be noted that $\overline{PT}$ and $\overline{ET}$ increase linearly from c to $\sqrt{2}c$ as f varies from 0 to $\sqrt{2} - 1$. When $f > \sqrt{2} - 1$, the scheduler will switch to the PB-CONCUR approach causing $\overline{PT}$ to jump to $2c$ and $\overline{ET}$ to jump to c.

Using the assumption in Section 5.3 the schedulability-reliability index for this approach is given by

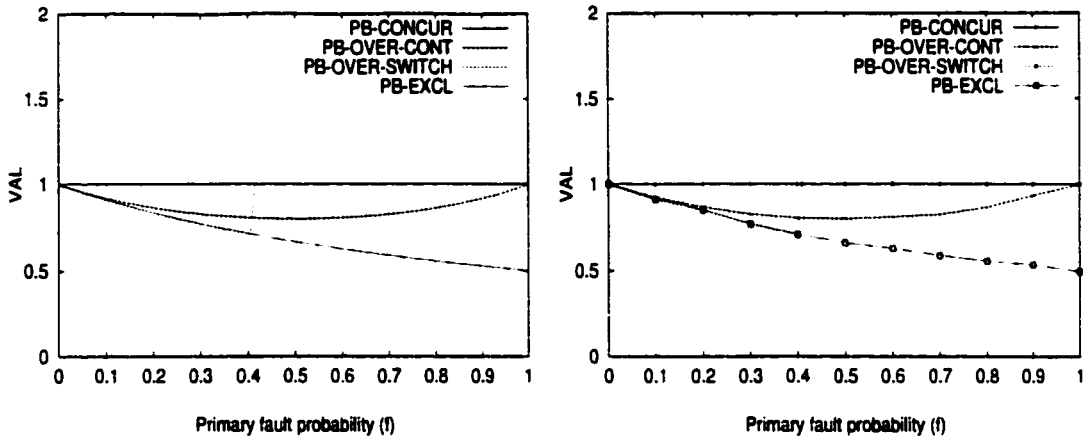
$$SR_i = \begin{cases} \frac{1}{(1+f)^2} & \text{for } 0 \leq f \leq \sqrt{2}-1 \\ 0.5 & \text{for } \sqrt{2}-1 \leq f \leq 1 \end{cases} \tag{5.23}$$

5.4.3 Analytical Results

In this section, we compare the performance of the proposed adaptive fault-tolerant approaches with that of the existing non-adaptive PB-based fault-tolerant approaches. The comparison is done using the analytical model described in Section 5.3. Also simulation studies (see Section 5.7.1), using the analytical assumption, have been carried out to confirm the analytical results. The task output value (VAL), processor utilization ($UTIL$), and schedulability-reliability index (SR) have been used as the performance metrics. For each point in the performance plots (Figures 5.7-5.9), the system was simulated with 20,000 tasks. This number of tasks has been chosen to cancel the effect of the warm-up period of the simulation and also to have a 97% confidence interval within ±0.0015, ±0.0019, ±0.0030 around each value of $UTIL$, VAL, and SR respectively.

5.4.3.1 Effects of the primary fault probability (f) on task value

Figures 5.7(a) and 5.7(b) show the effects of primary fault probability on the output value contributed by the tasks for all PB approaches. From the figures, it can be seen that the PB-CONCUR offers the maximum output value which is constant. This is because of its nature of scheduling both versions of a task within the soft deadline. Therefore, the value contributed by each of the admitted tasks is always one. The value offered by the PB-OVER-CONT lies between the PB-CONCUR and PB-EXCL for all fault probability. It offers the minimum value ($VAL = 0.8$) at $f = 0.5$. This is because the PB-OVER-CONT is designed in such a way that it takes an expected execution interval ($VAL_i = c_i/\overline{ET}$) of c when $f = 0$ and $f = 1$. For all other values of f, the expected execution interval is between c and 1.25c.



(a) Analytical (b) Simulation

Figure 5.7 Effects of primary fault probability (f) on task value

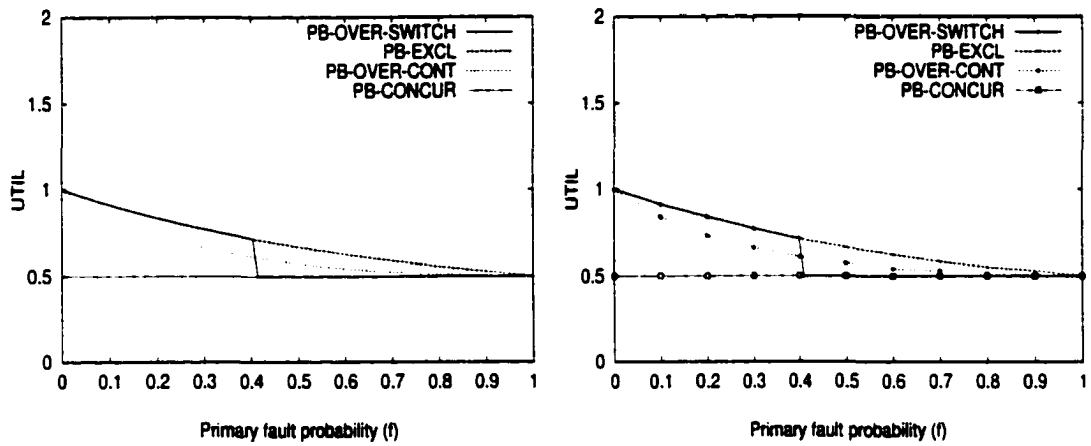
The PB-EXCL offers the minimum output value which is 1 when $f = 0$ and decreases to 0.5 as f increases to 1. Since there are no faults when $f = 0$, the primary always succeeds and the backup is deallocated, thus causing the primary to finish before the soft deadline and contributing one to the output value. On the other hand, when $f = 1$, the primary always fails, the backup must be executed, resulting in the doubling of the execution interval, thus contributing a 0.5 to the output value.

The PB-OVER-SWITCH behaves like PB-EXCL up to $f = \sqrt{2} - 1$ (the threshold value) after which it behaves like PB-CONCUR. Though the output value offered by the PB-OVER-SWITCH is lower than the PB-CONCUR for $f \leq \sqrt{2} - 1$, the SR offered by this approach is always better than the PB-CONCUR (which is discussed in Section 5.4.3.3). Figures 5.7(b) shows the output from the simulation. In the simulation the tasks are generated using the same analytical assumptions. From the figures, it can be seen that simulation results confirm the analytical results.

5.4.3.2 Effects of the primary fault probability (f) on processor utilization

Figures 5.8(a) and 5.8(b) show the effects of primary fault probability on the processor utilization for all PB approaches. From the figures, it can be noted that the PB-CONCUR

offers the minimum processor utilization ($UTIL = 0.5$) which is constant. This is because of its nature of scheduling both versions of a task concurrently. Therefore, the two versions of each task will always be executed resulting in a 0.5 processor utilization. The processor utilization offered by PB-OVER-CONT lies between PB-CONCUR and PB-EXCL for all fault probability. This is because the portion of the processor time ($UTIL_i = c_i/\overline{PT}$) that is deallocated from the backup goes from c (worst case execution time) to 0 when f goes from 0 to 1.



(a) Analytical (b) Simulation

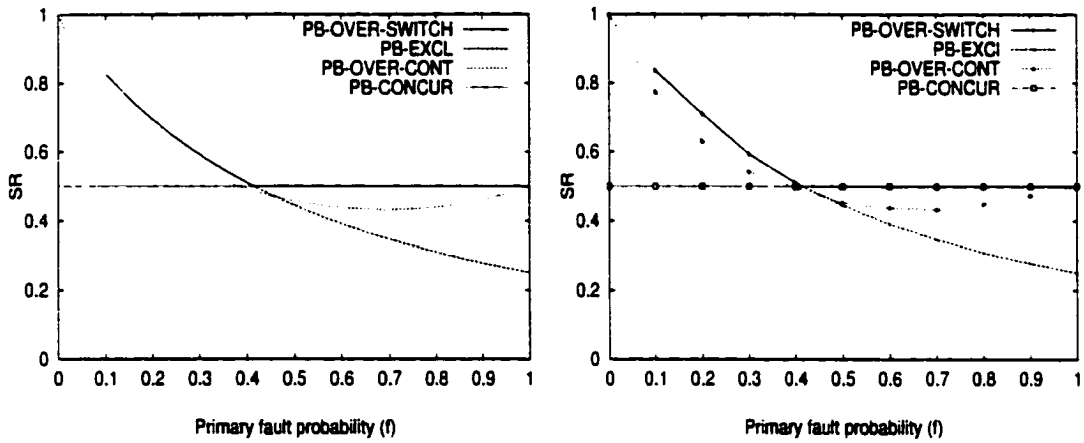
Figure 5.8 Effects of primary fault probability (f) on processor utilization

The PB-EXCL offers the maximum processor utilization which is 1 when $f = 0$ and decreases to 0.5 as f increases to 1. Since there are no faults when $f = 0$, the primary always succeeds and the backup is deallocated, thus executing only one version of a task, which results in full processor utilization. On the other hand, when $f = 1$, the primary always fails, the backup must be executed, resulting in doubling the consumption time for a task, thus contributing a 0.5 processor utilization.

The PB-OVER-SWITCH behaves like the PB-EXCL up to $f = \sqrt{2} - 1$ (the threshold value) after which it behaves like the PB-CONCUR. Though the processor utilization offered by the PB-OVER-SWITCH is lower than the PB-EXCL for $f > \sqrt{2} - 1$, the SR offered by this approach is always better than the PB-EXCL (which is discussed in Section 5.4.3.3). Figures 5.8(a) and 5.8(b) illustrate that the simulation results confirm the analytical results.

5.4.3.3 Effects of the primary fault probability (f) on SR index

Figures 5.9(a) and 5.9(b) show the effects of the primary fault probability on the SR index for all PB approaches. From the figures, it can be seen that the PB-CONCUR offers the best schedulability-reliability index when $f > \sqrt{2} - 1$. This is because in this region ($f > \sqrt{2} - 1$) the probability of fault is high. Therefore, scheduling both versions of a task concurrently within the soft deadline will give the best SR index, which is 0.5.



(a) Analytical (b) Simulation

Figure 5.9 Effects of primary fault probability (f) on SR index

The schedulability-reliability index offered by the PB-OVER-CONT lies between the PB-CONCUR and PB-EXCL for all fault probability. Except for the interval ($f \in [0.38, 0.46]$) where PB-OVER-CONT has the minimum schedulability-reliability index. The PB-EXCL offers the best schedulability-reliability index when $f \leq \sqrt{2} - 1$ because the probability of backup deallocation is high in this interval. Since the PB-OVER-SWITCH behaves as PB-EXCL when $f \leq \sqrt{2}-1$ and as PB-CONCUR when $f > \sqrt{2}-1$, it offers the best schedulability-reliability index for all values of f.Figures 5.9(a) and 5.9(b) illustrate that the simulation results confirm the analytical results.

5.4.4 Effects of Task's Soft Laxity on the Performance of the PB Approaches

In this section, we first study the effect of task's soft laxity on the performance of the PB-based fault-tolerant approaches. Next, we propose a mechanism that allows the fault probability threshold (f_o) to be adapted with task's soft laxity.

In Section 5.3, we assumed that the task's soft deadline (d^s) is equal to c. This assumption allows the PB-EXCL approach to schedule only one version of a task (Pr) within its soft deadline and the other version (Bk) must be scheduled after the soft deadline. This degrades the performance of the PB-EXCL approach as f increases since the correct output is always produced by the backup which has less value. When tasks have large soft deadline, then both the primary and the backup versions of tasks can be scheduled in an exclusive manner within their soft deadline. Therefore, all PB-based fault-tolerant approaches offer the same output value for the finished tasks which is equal to one and does not depend on f. Hence, the SR index will be 0.5, $\frac{1}{1+f}$, and $\frac{1}{-f^2+2f+1}$, respectively for the PB-CONCUR, PB-EXCL, and the PB-OVER-CONT approaches.

Figures 5.10 shows the effects of primary fault probability on the SR index for all PB approaches when tasks have large soft laxity. From the figures, it can be seen that the PB-EXCL offers the best SR index for all value of f. The PB-SWITCH offers the best SR index only when $f < \sqrt{2} - 1$ and it offers the lowest SR index when $f > \sqrt{2} - 1$. This is because the threshold value (f_o) that is used by the PB-SWITCH approach to switch between the PB-EXCL and PB-CONCUR approaches is constant ($f_0 = \sqrt{2} - 1$) and does not change with changing task's soft deadline. However, in Section 5.4.2, the threshold f_0 is defined as the value of f at which the SR index of PB-EXCL is equal to that of PB-CONCUR. Therefore, for the given task's deadline the value of f_0 is the value of f that satisfies $\frac{1}{(1+f)} = 0.5$, which is $f = 1$. By using this new threshold value ($f_0 = 1$), the PB-OVER-SWITCH always behaves like the PB-EXCL approach which offers the best performance in this case.

It is evident from the above discussion that the threshold value (f_0) has to be adapted with the task's soft laxity to be able to determine the correct overlap interval between the primary and the backup versions of each task that enhances the performance of the system. To do so,

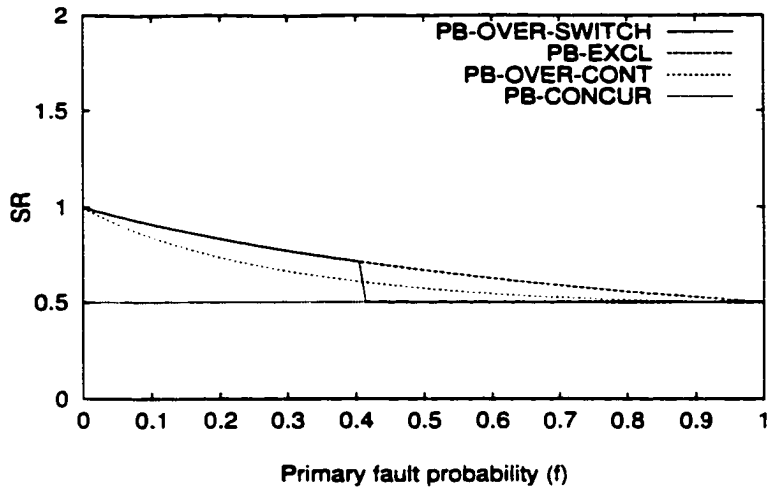


Figure 5.10 Effects of primary fault probability (f) on schedulability-reliability (SR) index

the scheduler behaves as follows for each task (T_i) that arrives in the system:

- Schedules the primary (Pr_i) version from the task (using the algorithm discussed in Section 5.5).
- If the primary was scheduled within the soft deadline, then
  - Determine the overlap interval ($\sigma_i c_i$) between the primary and the backup versions of the task so that both versions will be scheduled within the soft deadline.

$$\sigma_i = \frac{c_i - (d_i^s - ft(Pr_i))}{c_i} \tag{5.24}$$

    where $ft(Pr_i)$ is the relative finish time of the primary version.

  - If $\sigma_i < 0$ then, $\sigma_i = 0$.

Else if the primary was scheduled to finish after the soft deadline then, $\sigma_i = 1$.

Using this value (σ_i) the scheduler calculates the threshold value (f_0^i) that is used by the PB-OVER-SWITCH approach to select the correct overlap interval (i.e., if($f < f_0^i$) : $\gamma = 0$? $\gamma = 1$) between the primary and the backup versions for this task (T_i). To calculate f_0^i for a given

task T_i the scheduler uses the following equation:

$$f_0^i = (\sqrt{2} - 2)\sigma_i + 1 \tag{5.25}$$

Equation (5.25) is derived from the fact that f_0 changes from $\sqrt{2} - 1$ to 1 when the task's soft laxity change from the case where the scheduler is only able to schedule one version of task within its soft deadline to the case where it able to schedule both version in an exclusive manner within the soft deadline. In the next section, we will propose the general adaptive scheduling scheme that incorporates the two adaptive approaches (based on the fault rate and the soft laxity) to maximize the SR index.

5.5 The Proposed Adaptive Dynamic Scheduling Algorithm

Unlike static scheduling of periodic tasks, dynamic scheduling of aperiodic tasks must be very simple because its overhead has serious effects on real-time processing. In this section, we present a heuristic algorithm that incorporates the two adaptive approaches for scheduling two versions of each task in such a way that the SR index of the system is maximized. The scheduler works as follows for each task $T_i = (r_i,\ c_i,\ d_i^s,\ d_i^f)$ that arrives in the system:

1. Pr_i is scheduled first as follows:

   (a) Tries to find a free slot of length c_i between time r_i to time d_i^f. Since the scheduler tries to maximize the output value, it is appropriate to schedule Pr_i *as early as* possible so it will finish within the soft deadline (d_i^s). Note that a heuristic search algorithm such as the Spring scheduling [82] can be used to find the best fit free slot.

   (b) If Pr_i cannot be scheduled without overlapping any previously scheduled slot, task T_i is rejected.

2. If Pr_i is schedulable, then

   (a) Determines σ_i and calculates f_0^i as discussed in Section 5.4.4.

(b) Uses the estimated fault probability (f), to determine the overlap interval between the primary and backup versions of the task (γc_i) as discussed in Section 5.4.

(c) Sets ready time (r_i^b) for the backup version, $r_i^b = ft(Pr_i)$ - γc_i.

(d) Sets processor(Bk_i) $\neq$ processor(Pr_i).

3. Bk_i is scheduled as follows:

(a) Tries to find a free slot of length c_i between time r_i^b to time d_i^f. Since the scheduler tries to maximize the SR index, it is appropriate to schedule Bk_i *as early as possible.*

(b) If Bk_i cannot be scheduled without overlapping any previously scheduled slot, Pr_i is unscheduled and task T_i is rejected.

5.6 Implementation Issues

In this section, we introduce the fault monitoring system that is used to observe the fault rate and estimate the primary fault probability in the system. Next, we discuss the issue of the primary backup Synchronization.

5.6.1 Fault Monitoring System

A fault monitoring system periodically (with period p) monitors the completion of tasks in the system. For each period the monitoring system counts the number of faulty primaries ($n^f(t)$) and the total number of primaries completed ($n(t)$) during that period.

According to the *frequency interpretation* probability concepts, the probability of an event (primary fail) is the proportion of the time that events of the same kind will occur in the long run. Hence, we define the primary fault probability (f) as the ratio of the finished faulty primaries to the total finished primaries in an interval:

$$f(t) = \frac{n^f(t)}{n(t)} \tag{5.26}$$

This probability is calculated every p time units, where p is the sample period for the monitor. This fault probability is used to control the degree of overlap between the versions of the tasks.

5.6.2 Primary Backup Synchronization

The primary and backup versions of a task need to synchronize each other to produce the correct result. A *flag* bit, which is initialized to 0, is used to indicate the success of writing a valid result by the primary or backup version. The structures of the primary and backup versions are shown in Figure 5.11.

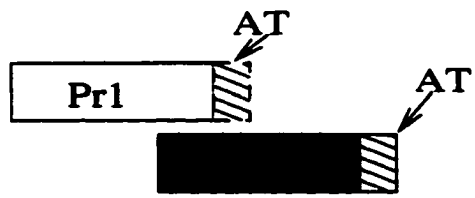


Figure 5.11 Structure of primary and backup versions

From the figure, the primary version works as follows. It does the intended computation, then performs the acceptance test (AT). If AT succeeds and $flag = 0$, it writes the result, sets the $flag = 1$, and initiates a signal to the kernel to de-allocate the backup. The operation of the backup is similar to that of the primary except that it does not de-allocate the primary as shown in Figure 5.12. The *lock* bit is used to ensure mutually exclusive access to the *flag* between primary and backup in the case of concurrent execution.

5.7 Simulation Studies

In this section, we first introduce the simulation model that is used to study the performance of the PB-based fault-tolerant approaches. Next, we compare the performance of the proposed new adaptive fault-tolerant approaches (PB-OVER-CONT, and PB-OVER-SWITCH) with the existing non adaptive approaches (PB-EXCL, and PB-CONCUR).

5.7.1 Simulation Model

A multiprocessor simulation of a soft real-time system was used to study the performance of the adaptive scheduling scheme. The parameters used in the simulation studies are given in Table 5.1. The tasks for the simulation are generated as follows:

```
Synchronize(Task version)
{
switch (Task version)
    case: Primary
        Acquire the lock.
        If (flag=0) then
            {Write the result.
            flag=1.
            de-allocate the backup. }
        Release the lock.
    case: Backup
        Acquire the lock.
        If (flag = 0) then
            {Write the result.
            flag=1. }
        Release the lock.
}
```

Figure 5.12 Primary backup synchronization

1. The worst case computation times of primary versions are chosen uniformly between Min_c and Max_c.
2. The soft deadline (d_i^s) of a task T_i is equal to $k_1 \times c_i$. Also, the firm deadline (d_i^f) of a task T_i is equal to $k_2 \times c_i$.
3. The inter-arrival time between tasks follows exponential distribution with mean θ.
4. The task load L is defined as the expected number of task arrivals per mean service time and its value is approximately equal to $\frac{C}{\theta}$, where C is the mean computation time of the system. The mean computation time C has been calculated based on fault free system.
5. The backup versions are assumed to have identical characteristics of their primary versions.

The simulator has five components: a source which generates tasks; a scheduler that makes admission/rejection decisions and determines the overlap interval on submitted tasks; a multiprocessor system that models the execution of the tasks; a monitor that periodically counts the

Table 5.1 Simulation parameters

Parameter	Explanation	Value used
Min_c	minimum computation time of tasks	2 sec
Max_c	maximum computation time of tasks	20 sec
k_1	the factor that relate d_i^s to c_i	1...5
k_2	the factor that relate d_i^f to c_i	2...5
f	the primary fault probability	0...1
L	the system offered task load	1
m	number of processors	6
p	monitoring period	40 sec.

number of finished faulty primaries; and an estimator that periodically estimates the primary fault probability.

5.7.2 Simulation Results

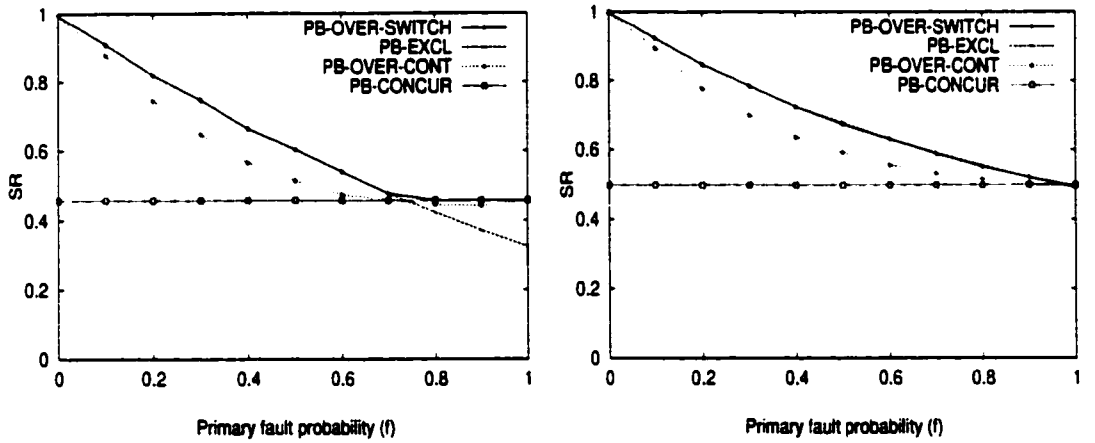
In this section, we compare the performance of the proposed new adaptive fault-tolerant approaches (PB-OVER-CONT, and PB-OVER-SWITCH) with the existing non adaptive approaches (PB-EXCL, and PB-CONCUR). Two experiments have been used to study the PB-based fault-tolerant approaches. In the first experiment (Experiment A) the fault rate for each simulation run is constant. In the second experiment (Experiment B) the fault rate for each simulation run is dynamically changed using the step and the ramp fault rate profiles (see Section 5.1).

5.7.2.1 Experiments A: Steady fault rate

Experiments A compares the performance of the proposed new adaptive fault-tolerant approaches with the existing non adaptive approaches. The comparison is done using the simulation model. The SR index has been used as the performance metric. For each point in the performance plots (Figures 5.13-5.15), the system was simulated with 20,000 tasks. The number of tasks has been chosen in order to have a 99% confidence interval within ± 0.0035 around each value of SR.

5.7.2.1.1 *Effects of the primary fault probability (f) on SR index*

Figures 5.13(a) and 5.13(b) show the effects of primary fault probability on the SR index for all PB approaches. Figure 5.13(a) shows the behavior of the PB approaches when f varies for tasks that have relative soft deadline equals to $2c_i$ and relative firm deadline equals to $5c_i$. Figure 5.13(b) shows the case for tasks that have relative soft deadline equals to $4c_i$ and relative firm deadline equals to $5c_i$.



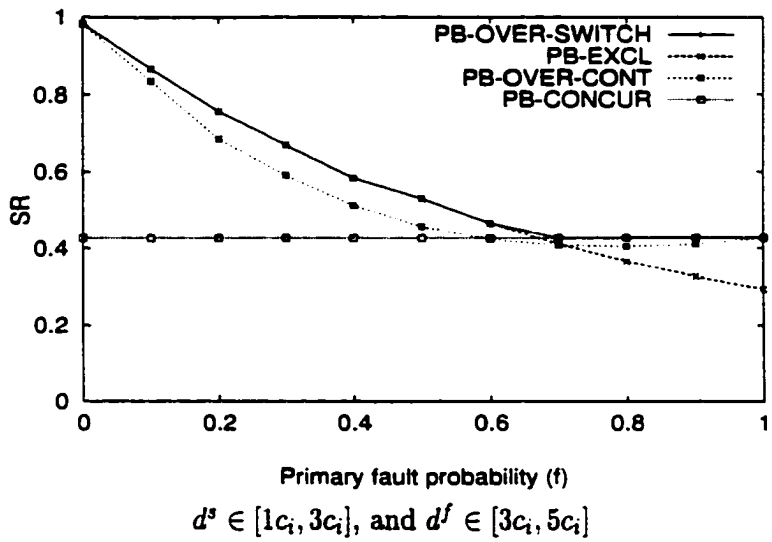
(a) $d^s = 2c_i$, and $d^f = 5c_i$ (b) $d^s = 4c_i$, and $d^f = 5c_i$

Figure 5.13 Effects of primary fault probability (f) on SR index

From the figures, it can be seen that when the relative soft deadline is large (Figure 5.13(b)) the PB-OVER-SWITCH and the PB-EXCL approaches behave similarly and offer the best SR index. This is because when d^s is large, the tasks have enough soft laxity to be scheduled in an exclusive manner within the soft deadline. The threshold value f_0 used by the PB-OVER-SWITCH approach is approximately equal to 1. From Figure 5.13(a), we can notice that the PB-EXCL approach has the highest SR index when $f < 0.75$, and the PB-CONCUR approach has the highest SR index when $f > 0.75$. The SR index offered by the PB-OVER-CONT lies between the PB-CONCUR and PB-EXCL for all values of primary fault probability. Since the PB-OVER-SWITCH behaves like PB-EXCL when $f \leq 0.75$ and like PB-CONCUR when $f > 0.75$, it offers the best SR index for all values of f.

Figure 5.14 shows the effects of primary fault probability on the SR index for all PB

approaches. For Figure 5.13, the task load was homogeneous (i.e., all the tasks have the same relative soft and firm laxities). For Figure 5.14, the tasks soft deadline is chosen uniformly in the interval $[c_i, 3c_i]$ and the task firm deadline is chosen uniformly in the interval $[3c_i, 5c_i]$ in order to generate a non-homogeneous task load. From Figure 5.14, we notice that the behavior of the PB-based fault-tolerant approaches stay the same as in the case of homogeneous task load.



$d^s \in [1c_i, 3c_i]$, and $d^f \in [3c_i, 5c_i]$

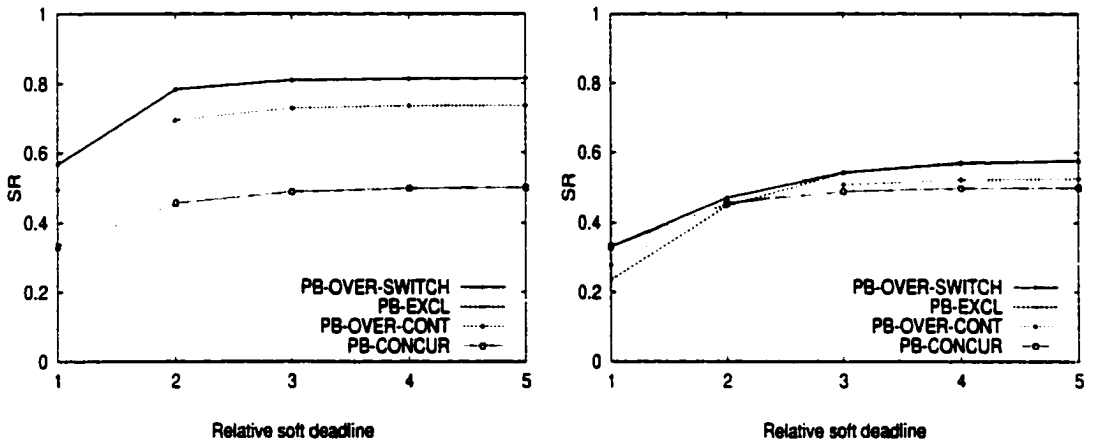
Figure 5.14 Effects of primary fault probability (f) on SR index

5.7.2.1.2 *Effects of task's soft laxity on SR index*

Figures 5.15(a) and 5.15(b) show the effects of tasks soft laxity on the SR index for all PB approaches. Figure 5.15(a) shows the behavior of the PB approaches when d^s varies for the system in which the primary fault probability is 0.25 and tasks' relative firm deadline is $5c_i$. Figure 5.15(b) shows the case for system in which the primary fault probability is 0.75 and tasks' relative firm deadline is also $5c_i$.

From the figures, it can be seen that when the primary fault probability (f) is low (Figure 5.15(a)) the PB-OVER-SWITCH and the PB-EXCL approaches behave similarly and offer the best SR index for all values of d^s. This is because when f is small, the PB-OVER-SWITCH

switch to an overlap interval equal to zero. The SR index offered by the PB-OVER-CONT lies between the PB-CONCUR and PB-EXCL for all value of d^s. From Figure 5.15(b), we can notice that the PB-CONCUR and PB-OVER-SWITCH approaches have the highest SR index when $d^s \leq 2$. This is because, when$f$ is high and tasks have small soft laxity then the threshold value that is used by the PB-OVER-SWITCH for each task is smaller than the fault



(a) $f = 0.25$, and $d^f = 5c_i$ (b) $f = 0.75$, and $d^f = 5c_i$

Figure 5.15 Effects of task's soft laxity on SR index

probability in the system ($f = 0.75$). Thus, PB-OVER-SWITCH behaves like PB-CONCUR for this region. We can notice that when $2 \leq d^s \leq 3$, the PB-OVER-SWITCH approach has the highest SR index. This is because, in this interval, the tasks have medium values of soft deadline. Thus PB-OVER-SWITCH enhances the performance of the system by adapting task's threshold based on its soft deadline. Finally, we can notice, from Figure 5.15(b), that the PB-EXCL and PB-OVER-SWITCH approaches have the highest SR index when $d^s > 3$.

5.7.2.2 Experiments B: Dynamic fault rate

To capture the transient behavior of the PB fault tolerant approaches in response to fault rate variations, we use the instantaneous value for the schedulability-reliability ($SR(t)$) index. The instantaneous $SR(t)$ index is defined as the product of the instant processor utilization ($UTIL(t)$) and the instant task value ($VAL(t)$) at time t. In contrast, the average SR index

is defined as the time average of the instantaneous $SR(t)$ index for the entire run-time. Each point in the performance plots (Figure 5.16) is the average value (with 95% confidence interval within ± 0.019 around the results) of 20 runs. In each run the system was simulated with 20,000 tasks.

5.7.2.2.1 *Step fault rate profile*

Figure 5.16(a) shows the effects on $SR(t)$ index for the four PB fault-tolerant approaches in response to the step fault rate $SFR(FR_{0\%}, FR_{100\%})$. $FR_{0\%}$ is the fault rate that makes the primary fault probability equal to zero ($f = 0$), and $FR_{100\%}$ is the fault rate that makes the primary fault probability equal to one ($f = 1$). Each time unit in the X-axes correspond to 40 seconds in the simulation time. The step fault rate was changed from $FR_{0\%}$ to $FR_{100\%}$ at $t = 500$ time units.

From the figure, we see that the PB-CONCUR offers a constant $SR(t)$ which does not change with t. This is because of its nature of always scheduling both the versions of a task concurrently within the soft deadline. The PB-EXCL offers $SR(t)$ equal to one when $t \leq 500$, otherwise a $SR(t)$ equals 0.25 when $t > 500$. This is because for $t < 500$ the probability of the primary to fail is equal to zero ($f = 0$) which will result in deallocating the backup versions. For $t > 500$ the probability of the primary to fail is equal to one which will result in executing the backups. The PB-OVER-CONT and PB-OVER-SWITCH behave similarly in response to step fault rate. They offer $SR(t)$ equal to one for $t \leq 500$, otherwise a $SR(t)$ equals 0.5 for $t > 500$. Since the fault rate switch from $FR_{0\%}$ to $FR_{100\%}$ the two adaptive approaches will behave similar to the PB-EXCL approach for the interval [0, 500], and they will adapt after a short period ($3p$) to behave similar to PB-CONCUR for the interval [500, 1000].

From the figure, we see that the smallest value of $SR(t)$ in the transient state ($t \simeq 500$) for the adaptive approaches is within 25% from its steady state value. This value is called the overshoot, which represents the worst-case transient performance of the system in response to the fault rate profile. Also, we can notice that the time taken for $SR(t)$ to enter the steady state after the step fault rate profile is equal to $3p$. This time is called the settling time, which

represents how fast the system can recover from a transient state. Since from the control theory point of view our system is an open-loop system, the overshoot and the settling time are only vary with the monitoring period p. Reducing the monitoring period p will enhance both the overshoot value and the settling time. In contrast, reducing the monitoring period p will reduce the accuracy of estimating the fault probability which will effect the stability of the system. According to the frequency interpretation probability concepts, the probability of an event should be calculated in a long run.

In summary, the average performance metrics of the PB fault-tolerant approaches in response to the step fault rate profile are listed in Table 5.2. From the table, we can see that adaptive PB fault-tolerant approaches offer the best average performance metrics ($VAL \simeq 1$, $UTIL \simeq 0.75$, $SR \simeq 0.75$) in response to the step fault rate profile. The PB-EXCL offers an average processor utilization ($UTIL \simeq 0.75$), and an average schedulability-reliability index ($SR \simeq 0.62$) higher than that of PB-CONCUR in response to the step fault rate profile.

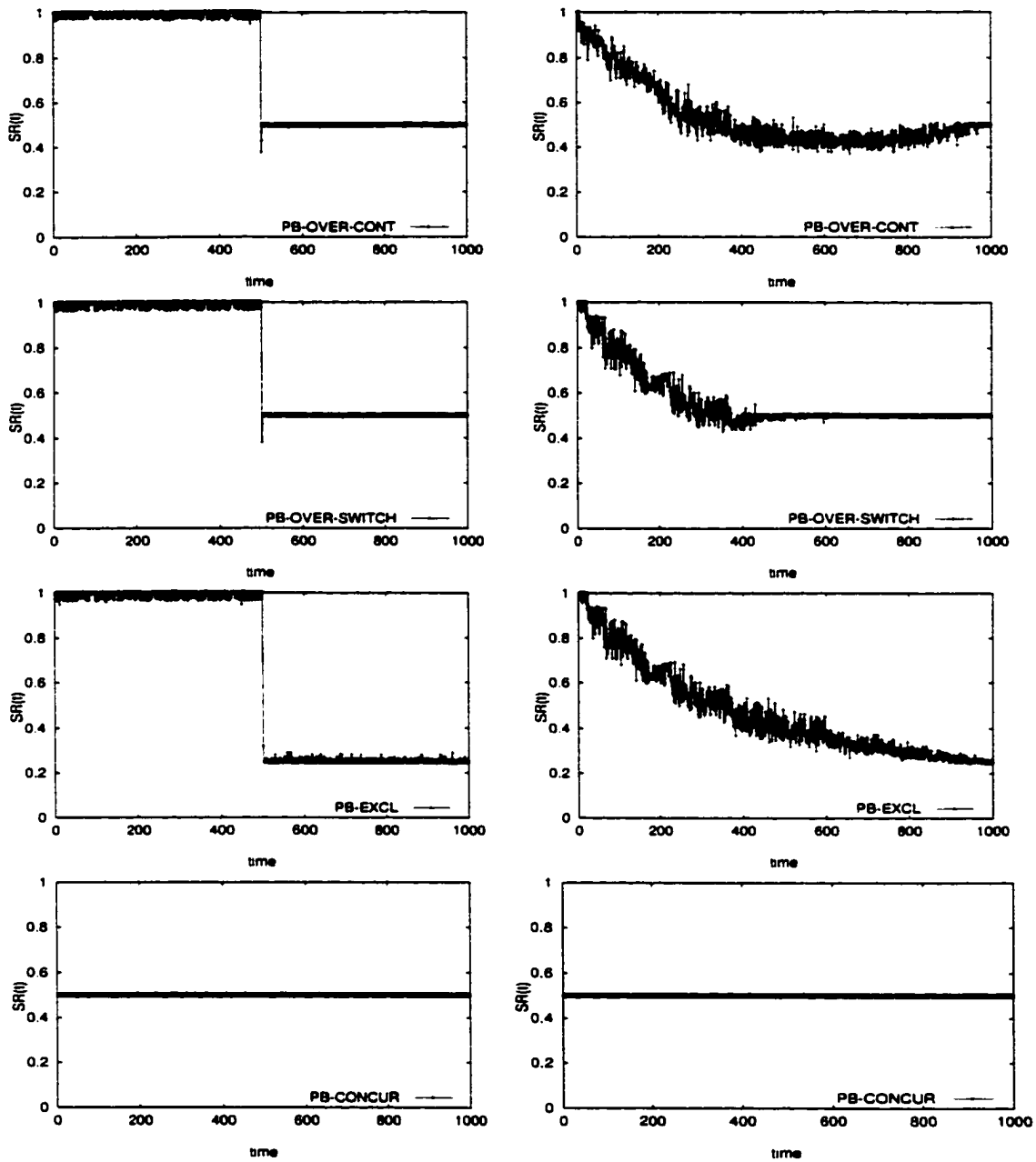
Table 5.2 The average performance metrics of the PB approaches in response to dynamic fault rate

approach	*Step Fault Rate Profile*			*Ramp Fault Rate Profile*		
	VAL	$UTIL$	SR	VAL	$UTIL$	SR
PB-EXCL	0.747	0.747	0.620	0.694	0.694	0.496
PB-CONCUR	1.000	0.500	0.500	1.000	0.500	0.500
PB-OVER-CONT	0.994	0.745	0.744	0.867	0.633	0.551
PB-OVER-SWITCH	0.995	0.745	0.7461	0.931	0.638	0.583

5.7.2.2.2 *Ramp fault rate profile*

Figure 5.16(b) shows the effects on $SR(t)$ for the four PB fault-tolerant approaches in response to the ramp fault rate $RFR(FR_{0\%}, FR_{100\%}, 1000)$. The ramp fault rate was increased from $FR_{0\%}$ to $FR_{100\%}$ during 1000 time units.

From the figure, it can be seen that the PB-CONCUR offers a constant $SR(t)$ index which does not change with t. The PB-EXCL offers $SR(t)$ that decreases quadratically from one to



(a) Step fault rate profile

(b) Ramp fault rate profile

Figure 5.16 $SR(t)$ of the PB approaches in response to dynamic fault rate ($d^s = c_i$, and $d^f = 2c_i$).

0.25 as t varies from 0 to 1000. This is because the primary fault probability increases linearly from zero to one as t varies from 0 to 1000. The PB-OVER-CONT offers an $SR(t)$ that decreases quadratically from one to approximately 0.433 as t varies from 0 to approximately 694, and increases quadratically from 0.433 to 0.5 as t varies from 694 to 1000. This is because the overlap interval between the primary and the backup versions of tasks will vary from no overlap to full overlap as t varies from 0 to 1000. The PB-OVER-SWITCH behaves like PB-EXCL when $t \leq 414$. and like PB-CONCUR when $t > 414$. This occurs because the primary fault probability f is greater than the threshold value ($f_0 = \sqrt{2} - 1$) when $t > 414$.

In summary, the average performance metrics of the PB fault-tolerant approaches in response to the ramp fault rate profile are listed in Table 5.2. From the table, we can see that the PB-OVER-SWITCH approach offers the best average schedulability-reliability index ($SR \simeq 0.58$) in response to the ramp fault rate profile. The PB-EXCL offers the best average processor utilization ($UTIL \simeq 0.69$) in response to the ramp fault rate profile. The PB-CONCUR offers the best average output value ($VAL \simeq 1$) in response to the ramp fault rate profile.

5.8 Summary

In this chapter, we have considered the problem of scheduling soft real-time tasks with PB-based fault-tolerant requirements in multiprocessor systems. We have proposed an adaptive fault-tolerant scheduling scheme for the problem. The scheme has a mechanism to control the overlap interval between the primary and backup versions of tasks in the schedule. The overlap interval is computed based on the estimated value of primary fault probability in the system and task's soft laxity. Two variants (PB-OVER-CONT and PB-OVER-SWITCH) of the adaptive scheme have been proposed and studied.

In PB-OVER-CONT, the overlap interval varies from no overlap to full overlap in a continuous manner as the fault probability varies from 0 to 1. In PB-OVER-SWITCH, the scheduler uses a threshold value of fault probability to switch from PB-CONCUR to PB-EXCL. This threshold value is adapted with task's soft laxity. We have also proposed a new metric, called

schedulability-reliability (SR) index and conducted the following analytical and simulation studies:

- Analytical studies to quantify the effects of primary fault probability on three performance metrics, *viz*, processor utilization, task value (utility), and SR index.
- Simulation studies to validate the above analytical results using the analytical assumptions.
- Simulation studies to quantify the effects of task's soft laxity in the performance of the PB-based fault-tolerant approaches.
- Simulation experiments to study the instantaneous behavior of $SR(t)$ in response to dynamically changing fault rates. In particular, we have studied this behavior for step and ramp fault rate profiles. Our studies show that both the variants of the adaptive scheme exhibit a similar behavior for step fault rate profile, while for the ramp fault rate profile, the PB-OVER-SWITCH performs better than the PB-OVER-CONT.

In summary, our studies show that the proposed PB-OVER-SWITCH adaptive scheme always performs better than its adaptive and non-adaptive counterpart for SR index.

CHAPTER 6. A CASE OF SCHEDULABILITY-RELIABILITY TRADE-OFFS IN FIRM REAL-TIME SYSTEMS.

The key issue addressed in this chapter is the relaxation of the requirement on a known *worst-case* workload parameters. Our main approach to this is to design a closed-loop scheduling algorithm, in which, the tasks are scheduled based on an estimation for their *actual execution time (AET)*. A feedback of the system performance is used to generate an error term. This error is the input to a control unit that adjusts the estimated value. We will model and analyze the closed-loop scheduling algorithm using existing control theory. The result is that this scheduling paradigm has a low miss ratio while maintaining a high guarantee ratio thereby improving the productivity of the firm/soft real-time systems.

Specifically, in this chapter, we first design an open-loop dynamic scheduling algorithm that employs a notion of task overlap in the schedule in order to provide flexibility in task execution times. This algorithm, dynamically guarantees incoming firm tasks via on-line admission control and planning. Secondly, we use feedback control theory to design three closed-loop scheduling algorithms derived from the open-loop algorithm. The loop is closed by feeding back *(i)* the deadline miss ratio in the first approach; *(ii)* the task rejection ratio in the second approach; *(iii)* both the miss ratio and rejection ratio in the final approach.

In Section 6.1, we define the terminology. In section 6.2, we propose the performance metrics. In Section 6.3, we propose a new open-loop firm scheduling algorithm, and study its performance. In Section 6.4, closed-loop scheduling, feedback control theory, and issues related to dynamic planning closed-loop scheduling are discussed. In Section 6.5, we propose the closed-loop approaches, present their analytical modeling, tune their parameters, analyze their performance, and verify their analytical models. In Section 6.6, we introduce the simulation

model and its parameters, and we compare the performance of the closed-loop scheduling algorithms. Finally in Section 6.7 we summarize the results.

6.1 Terminology

Definition 1: $st(T_i)$ is the scheduled start time of task T_i, which satisfies $r_i \leq st(T_i) \leq d_i - EET_i$. $ft(T_i)$ is the scheduled finish time of task T_i, which satisfies $r_i + EET_i \leq ft(T_i) \leq d_i$, where EET_i is the estimated execution time for task T_i.

Definition 2: $Proc(T_i)$ is the processor on which task T_i is scheduled.

Definition 3: Let P be the set of processors, and R_i be the set of resources requested by task T_i. The earliest start time of a task T_i is denoted by $EST(T_i)$, which is the earliest time when its execution can be started, and it is defined as :

$$EST(T_i) = \max(r_i, \min_{j \in P}(avail\ time(j)), \max_{l \in R_i}(EAT_l^u)) \tag{6.1}$$

where *avail time(j)* denotes the time at which the processor P_j is available for executing a task, and the third term denotes maximum among the available time of the resources requested by task T_i, in which u=s for shared mode and u=e for exclusive mode.

Definition 4: A task is said to be feasible in schedule if it satisfies the following constraints: $r_i \leq st(T_i) \leq ft(T_i) \leq d_i$.

6.2 Performance Metrics

- *Task guarantee ratio* (GR): This is the ratio of the number of tasks admitted into the system to the total number of tasks that arrived at the system. The rejection of tasks happens at the scheduler and depends on factors such as the schedulability check algorithm, estimated execution time, and the time at which the schedulability check is performed for a task. Task rejection ratio (RR) is equal to one minus the guarantee ratio.

- *Deadline hit ratio* (HR): This is the ratio of the number of admitted tasks that meet their deadlines to the total number of tasks admitted into the system. Though the schedulability check of tasks are performed while admitting them, tasks can still miss their deadline when their actual execution time (AET) is greater than their estimated execution time (EET) or due to the occurrence of unanticipated faults in the system. Deadline miss ratio (MR) is equal to one minus the hit ratio.

- *Task effective ratio* (ER): This is an integrated metric and is defined as the ratio of the number of tasks that meet their deadlines to the total number of tasks arrived at the system which is equal to the product of the deadline hit ratio (HR) and the task guarantee ratio (GR).

6.3 Open-Loop Dynamic Planning Scheduling

Dynamic planning based schedulers can dynamically guarantee incoming tasks via on-line admission control and planning. On-line admission control has been used to guarantee predictability of services where request patterns are not known in advance. In dynamic planning scheduling, when a new set of tasks arrive, the scheduler determines the feasibility of scheduling these new tasks without jeopardizing the guarantees that have been provided for the previously scheduled tasks. For predictable executions, schedulability analysis must be done before tasks' execution begins. For feasibility analysis, the tasks' worst case execution times are taken into account. A feasible schedule is generated if timing and other resource requirements of the tasks can be satisfied, i.e., if the schedulability analysis is successful. Tasks are dispatched according to this feasible schedule. Hence, in dynamic planning scheduling, there are three main activities: schedulability checking, schedule construction, and dispatching (task execution). In a multiprocessor system, schedulability checking and schedule construction are done by the scheduler and are independent of dispatching, thus allowing them to run in parallel. An example for a dynamic planning based scheduling is the Spring kernel [82]. In this section, we use the Spring scheduling approach for scheduling tasks with firm deadlines. We propose new approach from the Spring scheduling that differ in the schedulability checking, and scheduler

construction. Notice that Spring is only one example algorithm we use but our results are more general.

6.3.1 New Open-Loop Firm Scheduling Algorithm

In order to guarantee that hard real-time tasks meet their deadlines once they are scheduled, scheduling algorithms schedule tasks with respect to their worst-case execution time (*WCET*) [56, 68, 71]. In this approach (called *OL-NO-OVER-WCET* (Open-Loop NO OVERlap with *WCET* estimation) scheduling algorithm), *WCET* is used by the scheduler to perform the schedulability check for tasks (i.e. it checks whether $\text{EST}(T_i) + WCET_i \leq d_i, \forall\, T_i$). Moreover, each task is assigned a time slot equals to its *WCET* (the tasks are not allowed to overlap) when the schedule is constructed as shown in Figure 3.1a. Therefore, in the *OL-NO-OVER-WCET* approach, the tasks are allowed to execute to their *WCET* if needed. The actual execution time of tasks varies between their *BCET* and *WCET* due to non-deterministic behavior of several low-level processor mechanisms (e.g. caching, prefetching, and DMA data transfer), and also due to the fact that the actual execution time for these tasks are a function of the system state, and the amount, nature, and the value of input data [7, 65]. A resource reclaiming algorithm can be used to compensate for the performance loss due to the inaccuracy of the estimation of the worst case execution times of real-time tasks [58, 76]. Resource reclaiming on multiprocessor systems with resource constraints is expensive and complex. This is due to the potential parallelism provided by a multiprocessor and potential resource constraints among tasks, and also due to the fact that the resource reclaiming algorithm may have to be invoked very frequently. This reduces the effectiveness of the reclaimed time in enhancing the performance of the system. Therefore, in a non-hard real-time system it will be more effective to schedule tasks with respect to their average-case execution time ($AvCET = \lfloor \frac{WCET+BCET}{2} \rfloor$) rather than their *WCET*.

In firm real-time systems, the consequences of not meeting the deadline are not as severe as for hard real-time systems. Hence, tasks can be scheduled based on their *AvCET*. This reduces the amount of resources that become unused due to tasks being executed less than their

WCET. In the second approach (called *OL-NO-OVER-AvCET* (Open-Loop NO OVERlap with *AvCET* estimation) scheduling algorithm), *AvCET* is used by the scheduler to perform the schedulability check for tasks (i.e. it checks whether EST(T_i) + $AvCET_i \leq d_i$, $\forall\ T_i$). Moreover, each task is assigned a time slot equal to its *AvCET* (the tasks are not allowed to overlap) when the schedule is constructed as shown in Figure 3.1b. In the *OL-NO-OVER-AvCET* approach, the tasks are allowed only to execute to their *AvCET*. Therefore, the system is able to guarantee more tasks which enhance the guarantee ratio. Scheduling tasks with their *AvCET*, however, increases the chances of them missing their deadlines. Indeed, all tasks that have an actual execution time (*AET*) greater than their *AvCET* would miss their deadlines.

To achieve a guarantee ratio comparable to the *OL-NO-OVER-AvCET* scheduling algorithm and a miss ratio comparable to the *OL-NO-OVER-WCET* scheduling algorithm, we propose a new scheduling algorithm, called *OL-OVER-AvCET*. In this approach, *WCET* is used by the scheduler to perform the schedulability check for tasks (i.e. it checks whether EST(T_i) + $WCET_i \leq d_i$, $\forall\ T_i$). However, each task is assigned a time slot equal to its *AvCET* when the schedule is constructed and it is overlapped with both its neighbors by a time slot equal to $\frac{WCET-AvCET}{2}$ as shown in Figure 3.1c. In *OL-OVER-AvCET* approach, the time at which the processor P_j is available for executing a task (*avail time(j)* in Equation (6.1)) is the scheduled finish time of the last task (T_i) in its dispatch queue minus the overlap time. The overlapped time equals $\frac{WCET_i-AvCETT_i}{2}$. Therefore, in this approach, a task T_i can start and finish any time within the interval $[EST(T_i),\ EST(T_i) + WCET_i]$. Being that each task is overlapped with its both neighbors by time slots equal to $\frac{WCET-AvCET}{2}$, the task is allowed to execute to its *WCET* if its previous neighbor has completed its execution before the overlapped time.

6.3.1.1 Example of open-loop scheduling algorithms

Table 6.1 gives the *WCET*, the *BCET*, the *AvCET*, the *AET*, and the deadlines (*d*) for five tasks. The ready time for all these tasks is equal to zero, and the tasks do not have any resource requirement. Figure 6.2 shows the feasible schedule on two processors and the

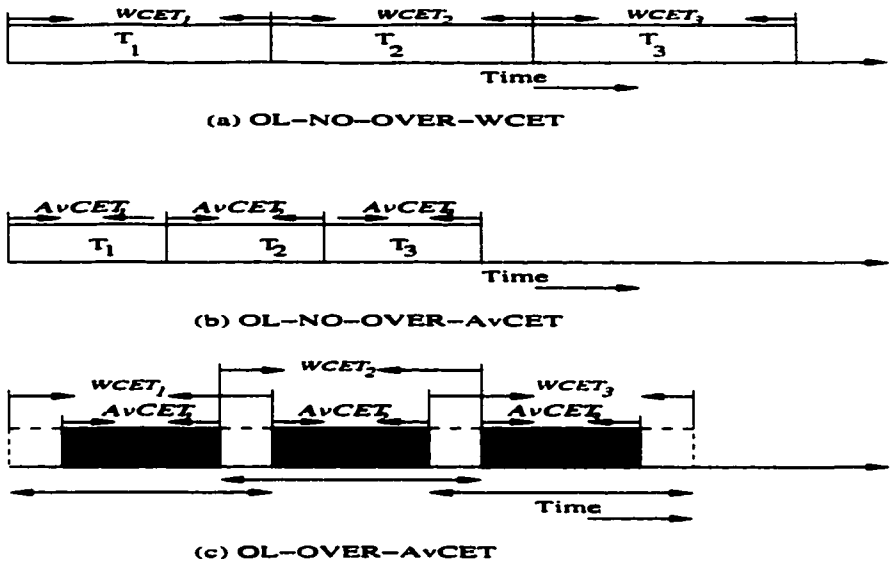


Figure 6.1 Open-loop scheduling algorithms

post-run schedule for these tasks. From Figure 6.2a, we notice that only three tasks have been accepted (T_1, T_2, and T_4) using the *OL-NO-OVER-WCET* scheduling algorithm. Since the tasks are scheduled using their *WCET*, all accepted tasks have met their deadlines. The figure also shows that the processors are idle for a total of 5 time units. From Figure 6.2b, we notice that all the tasks have been accepted using the *OL-NO-OVER-AvCET* scheduling algorithm. Since the tasks are scheduled using their *AvCET*, only 2 tasks have met their deadlines (T_1, and T_4). The figure also shows that the processors are idle only for 1 time unit. From Figure 6.2c, we notice that four tasks have been accepted (T_1, T_2, T_3, and T_4) using *OL-OVER-AvCET* scheduling algorithm. Since the tasks are scheduled using the overlap approach, all accepted tasks have met their deadlines. The figure also shows that the processors are idle only for 2 time units. This example clearly shows the superiority of the *OL-OVER-AvCET* over the other two algorithms.

6.3.2 Performance Studies of the Open-Loop Scheduling Algorithms

In this section, we compare the performance of the open-loop scheduling algorithms using the simulation model (see Section 6.6). The task guarantee ratio (*GR*), the deadline hit ratio

Table 6.1 Parameters for the tasks

Task	*BCET*	*WCET*	*AvCET*	*AET*	*d*
T_1	4	8	6	5	9
T_2	6	10	8	9	10
T_3	1	4	2	3	11
T_4	2	4	3	3	12
T_5	2	4	3	4	13

(HR), and the task effective ratio (ER) have been used as the performance metrics. For each point in the performance plots (Figures 6.3-6.5), the system was simulated for 10,000 tasks. This number of tasks has been chosen to obtain a 97% confidence interval within ±0.0017, ±0.0022, and ±0.0030 around each value of HR, GR, and ER, respectively.

6.3.2.1 Effects of task load (L) on the guarantee ratio (GR)

Figure 6.3 shows the impact of task load (L) on GR. As expected, increasing L decreases the guarantee ratio for all the algorithms and the performance difference between the techniques widens when the task load is greater than 0.5. This is because, for low loads, the task load is less than the system capacity and hence the techniques tend to behave similarly (i.e., all the arrived tasks are accepted in the system). The figure also shows that the guarantee ratio offered by *OL-NO-OVER-AvCET* scheduling algorithm is better than that of the OL-NO-OVER-WCET and OL-OVER-AvCET scheduling algorithms for all task loads. The reason is that the *OL-NO-OVER-AvCET* scheduling algorithm uses the tasks *AvCET* to perform the schedulability checking and to construct the schedule. This increases the system utilization since the amount of idle time due to overestimation is smaller. This results in accepting more tasks and hence enhancing the guarantee ratio. Also, note that the difference in GR between *OL-NO-OVER-AvCET* and *OL-OVER-AvCET* is small which means that the *OL-OVER-AvCET* approach increases the system's utilization to a point equal to the *OL-NO-OVER-AvCET* approach and the difference in the performance is due to the fact that the overlap approach uses the *WCET* to perform the schedulability checking. This difference in GR increases as the tasks laxity

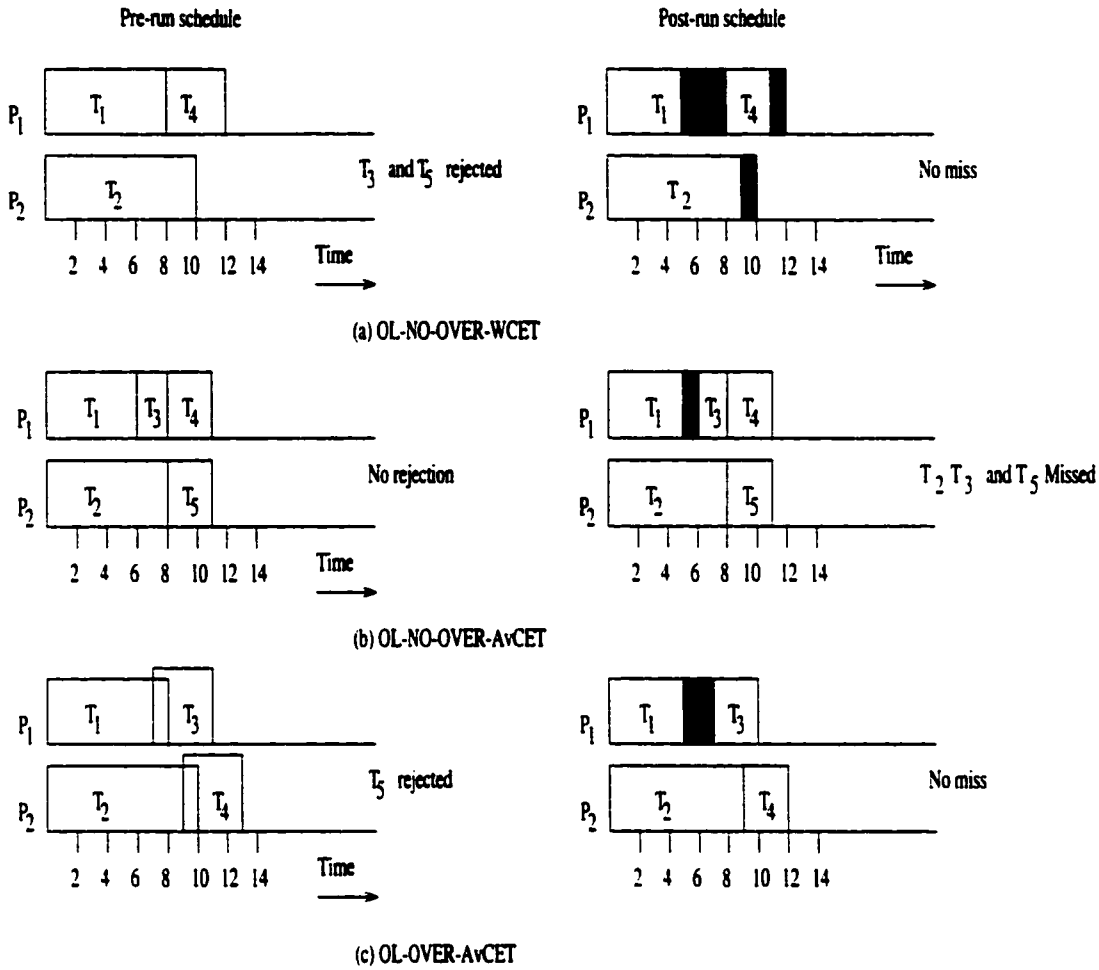


Figure 6.2 Pre-run and post-run schedules for the tasks

decreases.

6.3.2.2 Effects of task load (L) on the hit ratio (HR)

Figure 6.4 shows the effects of task load (L) on HR. As expected, varying L does not effect the hit ratio for the *OL-NO-OVER* approaches. This is due to the fact that in the *OL-NO-OVER-WCET* approach all the tasks that are guaranteed in the system meet their deadlines since their *WCET* have been used to construct the schedule. For the *OL-NO-OVER-AvCET* approach, approximately half of the admitted tasks meet their deadlines because their *AvCET* have been used to construct the schedule. The actual execution time (*AET*) for tasks are chosen uniformly between their *BCET* and *WCET*, in the long run approximately 50% of the

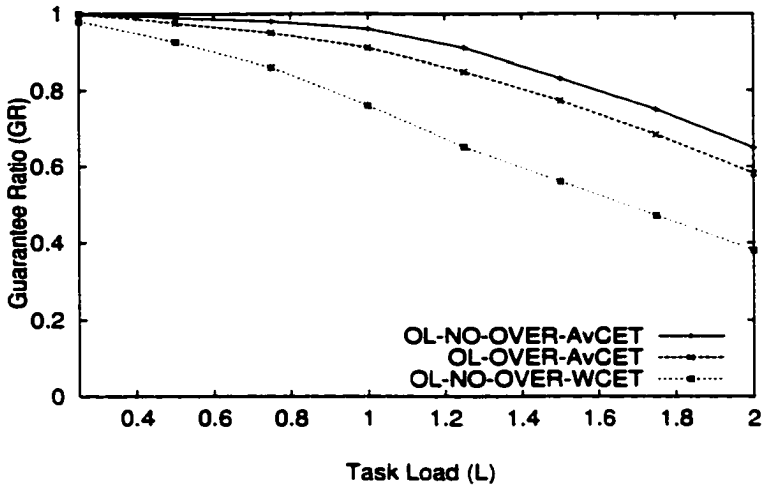


Figure 6.3 Guarantee ratio for the three open-loop algorithms

tasks have *AET* less than their *AvCET* and the other 50% of the tasks have *AET* greater than their *AvCET*. Hence, approximately 50% of the accepted tasks (i.e., tasks that have their *AET* less than or equal to their *AvCET*) meet their deadlines. The figure also shows that increasing L decreases the hit ratio for the *OL-OVER-AvCET* approach. The reason is that for low values of L the system is lightly loaded so most of the tasks are able to execute to their *WCET* if needed. As the task load (L) starts to increase, the system utilization increases and the tasks are overlapped in execution. This decreases the maximum time a task can be executed, therefore the system hit ratio decreases. From the figure we notice that for the *OL-OVER-AvCET* approach, the slope of the hit ratio carve decreases for high task load ($L > 1.4$). This is because for high task loads the system is fully utilized. Thus increasing the task load does not significantly change the maximum time the tasks can be executed.

6.3.2.3 Effects of task load (L) on the effective ratio (ER)

Figure 6.5 depicts ER changes with respect to task load (L). The figure reveals that increasing L decreases the effective ratio for *OL-NO-OVER-WCET and OL-OVER-AvCET* scheduling algorithms and the performance difference between the two algorithms widens as task load increases. The reason is that under high loads (overloaded system), the guarantee

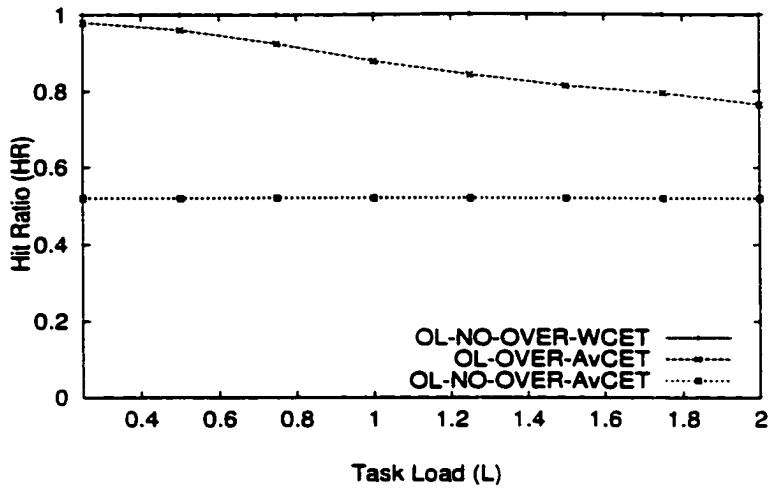


Figure 6.4 Hit ratio for the three open-loop algorithms

ratio offered by the *OL-NO-OVER-WCET* algorithm decreases significantly compared to *OL-OVER-AvCET* algorithm because of its *WCET* based schedulability test and time assignment. The figure also shows that the *OL-NO-OVER-AvCET* algorithm offers the smallest effective ratio which is approximately constant for $L \leq 1$ and decreases for $L > 1$. This is because for $L \leq 1$ the guarantee ratio offered by this algorithm is approximately equal to 1 and starts decreasing as L increases. The hit ratio offered by this approach is constant for all task loads. Hence, the effective ratio decreases for $L > 1$.

6.4 Closed-Loop Scheduling

The disadvantage of basing the schedulability test on a priori estimation is that an under-estimation of execution times may jeopardize the correct behavior of the system, whereas an overestimation will under-utilize system resources and cause performance degradation [14]. In this section, we present a novel approach in which the actual execution time of tasks can be dynamically estimated based on the current deadline miss ratio (MR) and task rejection ratio (RR) in the system. Our main idea, is to design a closed-loop scheduling algorithm, in which the tasks are scheduled based on an estimation of their actual execution time. A feedback from the performance of the system (miss ratio and rejection ratio) is used to generate an

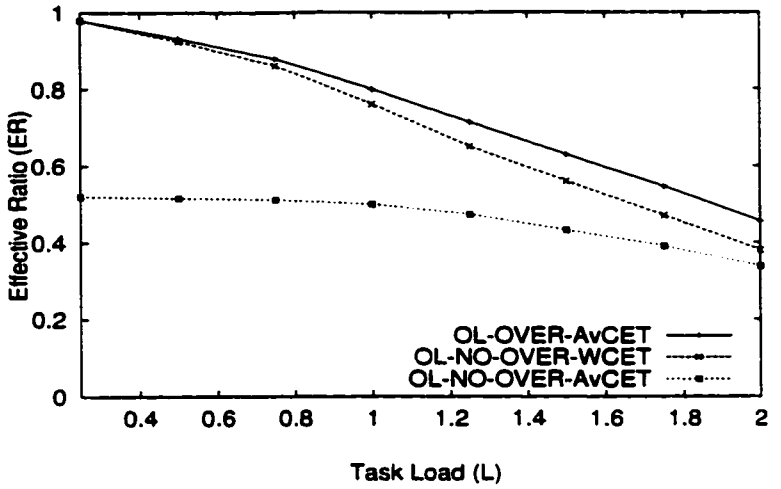


Figure 6.5 Effective ratio for the three open-loop algorithms

error term. This error is the input to a control unit that adjusts the estimated value. We model and analyze the closed-loop scheduling algorithm using control theory. The main result is that this scheduling algorithm accepts significantly more tasks and meets more deadlines than open-loop scheduling algorithms, thereby improving the productivity (effective ratio) of the firm/soft real-time systems.

6.4.1 Feedback Control Theory

Viewing a computing system as a dynamic system or as a controller is an approach that has proved to be fruitful in many cases. For example, the step-length adjustment mechanism in numerical integrating algorithms can be viewed as a PI-controller [27]. This approach can also be adopted for real-time scheduling, i.e., it is possible to view the on-line scheduler as a controller. The first step in abstracting the closed-loop scheduling into a control system is the identification of the inputs and the outputs.

Figure 6.6 shows the feedback control paradigm that will be used to model the closed-loop scheduling algorithms. It defines the following variables and blocks:

1. *Exogenous inputs* (w), are variables that effect the system and cannot be changed by the designer. Typically, any reference set point or disturbances form the exogenous signal w.

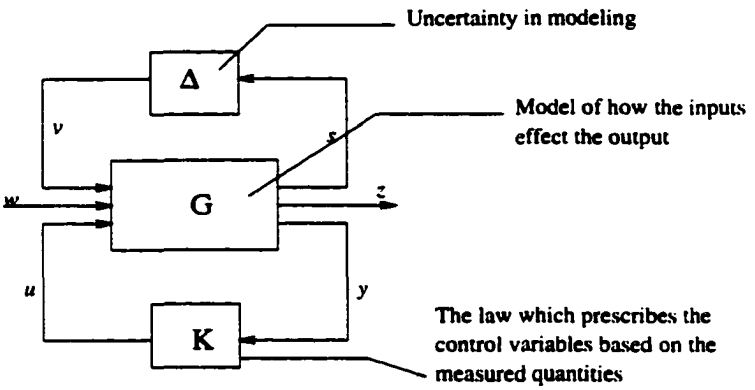


Figure 6.6 Feedback control paradigm to model a real-time system

2. *Control inputs* (u), are variables that effect the system and are available to the designer for manipulated.

3. *Regulated variables* (z), are quantities that are used to depict the performance of the system.

4. *Measured variables* (y), are variables that can be measured and used as feedback information. These variables are either available known quantities during the execution of the system or can be manipulated using the system state.

5. *Feedback control system blocks*, are composed of: *(i) system block* (G), which is the model which maps w, u, and v into z, y, and s; *(ii) uncertainty block* (Δ), which is the block that describes the uncertainty in the system model (G); *(iii) control block* (K), which is the control law that dictates how the control inputs u change in response to information gathered about the system through measured variables y in the presence of uncertainty Δ in the design.

The system is composed of a feedback loop as follows: (1) The system periodically measures and compares the measured variables to the set points to determine the error, (2) The controller computes the required control with the control function of the system based on the error, (3) The actuators change the value of the control variable to control the system.

6.4.2 Dynamic Planning Based Closed-Loop Scheduling Framework

To apply feedback control techniques in scheduling, we need to restructure schedulers based on the feedback control framework. We need to identify the set points, control variables, regulated variables, and measured variables. Once these variables are identified, a mathematical abstraction that provides the effect of the inputs on the outputs can be obtained. This is at the heart of the modeling of closed-loop scheduling into a control framework. Moreover, the most important principle of feedback is to employ the information gathered about the system through the measured variables that affect the controlled variable to produce the desired effect on the regulated variables. Therefore, the control law K to be designed must dictate how to alter the control inputs u in response to information gathered about the system through measured variables y.

6.4.2.1 Regulated variables

The choice of the regulated variables depends on the system goal. The performance of a firm real-time system usually depends on: (1) how many tasks it admits (rejects); (2) how many tasks among the accepted tasks meet (miss) their deadlines. Therefore, the deadline miss ratio (MR) and the task rejection ratio (RR) are natural choices of the regulated variables.

6.4.2.2 Control input

The control variable must be able to affect the value of the regulated variables. In the non-preemptive multiprocessor firm real-time system, it is a widely known fact that the deadline miss ratio and the task rejection ratio highly depend on the estimated execution time of tasks (see the example of open-loop scheduling algorithms in Section 6.3.1). Thus the estimated execution time (EET) of tasks is an appropriate choice for the control variable. The estimated execution time (EET_i) of a task (T_i) is calculated using the following equation:

$$EET_i = AvCET_i + etf_{kT}\,[AvCET_i - BCET_i] \tag{6.2}$$

where etf_{kT} is the estimation factor at instant time kT which can have a value in the interval $[-1,\ 1]$. For $etf_{kT} = -1$, the tasks' estimated execution times are equal to their $BCET$, for

$etf_{kT} = 0$ the tasks estimated execution time are equal to their $AvCET$, for $etf_{kT} = 1$ the tasks estimated execution time are equal to their $WCET$.

In the case of open-loop scheduling algorithms, etf is fixed which is equal to 0 for the algorithms that use the $AvCET$ of tasks as an estimation for their actual execution time, and it is equal to 1 for the algorithms that use the $WCET$ of tasks as an estimation for their actual execution time. In the case of closed-loop scheduling algorithms the estimated factor (etf) is used as the control variable.

6.4.2.3 Exogenous inputs

The set points MR_s and RR_s, which are the desired values for the measured variables, are chosen to have a small, but a non-zero value (e.g. $MR_s = RR_s = 1\%$) for both of them. Note that a value of 0 is not chosen to be the set points for the following reasons: A system with a set point of $MR_s = 0$ can achieve a 0% deadline miss ratio but causes extremely low utilization by rejecting too many tasks (i.e., overestimate the execution time of tasks). In contrast, a set point of $MR_s \neq 0$ will always try to (lightly) overload the system to achieve high utilization. Similarly, a system with a set point of $RR_s = 0$ can achieve a 0% tasks rejection ratio but causes extremely low throughput by missing too many deadlines (i.e., underestimate the execution time of tasks). In contrast, a set point of $RR_s \neq 0$ will always try to (lightly) underload the system to achieve high throughput (low miss ratio).

6.4.2.4 Measured variables

In closed-loop approach, it is important to be able to measure the appropriate signal on-line. The instantaneous miss ratio (MR_{kT}) is calculated every sample period (T), which is defined as the ratio of the number of tasks that missed their deadlines during the time interval $[(k-1)T,\ kT]$ to the total number of tasks that finished execution during the same interval, where k is the current time instant.

$$MR_{kT} = \frac{\#\ of\ missed\ tasks_{[(k-1)T,\ kT]}}{\#\ of\ finished\ tasks_{[(k-1)T,\ kT]}} \tag{6.3}$$

In contrast, the average miss ratio (MR) is defined as the time average of the instantaneous miss ratio (MR_{kT}) for the entire run-time.

The instantaneous rejection ratio (RR_{kT}) is calculated every sample period (T), which is define as the ratio of the number of tasks that are rejected during the time interval $[(k-1)T,\ kT]$ to the total number of tasks that arrived to the system during the same interval.

$$RR_{kT} = \frac{\#\ of\ rejected\ tasks_{[(k-1)T,\ kT]}}{\#\ of\ arrived\ tasks_{[(k-1)T,\ kT]}} \tag{6.4}$$

In contrast, the average rejection ratio (RR) is defined as the time average of the instantaneous rejection ratio (RR_{kT}) for the entire run-time.

6.4.2.5 Actuator model

An execution time estimator is used to manipulate the requested change in the estimated execution time. The controller computes the amount of change (Δetf_{kT}) that needs to be added or reduced from the estimated factor (etf_{kT}). Then the estimator updates the estimated factor using the following equation:

$$etf_{kT} = etf_{(k-1)T} - \Delta etf_{kT} \tag{6.5}$$

6.4.2.6 Control law

As a starting point, we will apply a proportional control in our work. A basic form for proportional control formula is:

$$\Delta etf_{kT} = K \times error_{kT} \tag{6.6}$$

where K and T are a tunable parameters. K is the coefficient of the controller and T is the sample period. The tuning of these parameter will be discussed later. Different controller types can be used (e.g. PID-controller as in [51]). The controller is the core of the closed-loop scheduling. It maps the performance of the accepted tasks (i.e., error) to the change in the estimated execution time of tasks (i.e., control signal) so as to drive the system performance back to the set point. The system performance is periodically (every T second) fed back to the

controller. Using the control formula the controller computes the required control action, i.e., the amount of change that needs to be added or reduced from the estimated execution time of tasks.

6.4.2.7 Uncertainty model

Since there is no mathematical system that can precisely model a scheduling algorithm, uncertainty is inevitable. Uncertainty results because we cannot predict the output even when we know the inputs. Uncertainty can also result when the inputs are unpredictable. We denote the uncertainty due to error in modeling by Δ.

In Section 6.5.2, we approximate the relation between the estimation factor (etf) and the rejection ratio and the miss ratio as a linear line. In this paper, we assume the models are accurate and use a $\Delta = 0$. This is not accurate since scheduling system typically contains non-linear factors which is not presented in the current model.

6.5 Closed-Loop Scheduling Algorithms

In this section, we present an algorithm called closed-loop overlap dynamic scheduling algorithm (CL-OVER), which integrates feedback controller with the open-loop overlap scheduling algorithm presented in Section 6.3.1. The loop is closed by feeding back *(i)* the deadline miss ratio in the first approach which is called *CL-OVER-MISS*; *(ii)* the task rejection ratio in the second approach which is called *CL-OVER-REJ*; *(iii)* both the miss ratio and rejection ratio in the final approach which is called *CL-OVER-MISSREJ*.

6.5.1 Architecture of CL-OVER Approaches

The *CL-OVER* schedulers, as shown in Figure 6.7, are composed of a controller, an execution time estimator and an overlap scheduler. In these approaches the deadline miss ratio (MR_{kT}) and/or the task rejection ratio (RR_{kT}) are periodically fed back to the controller. The controller computes the required control action Δetf, i.e., the amount of change that needs to be added or reduced from the estimate factor etf. Then the controller calls the execution time

estimator to change the estimated execution time of tasks. The overlap scheduler schedules the arrived tasks according to their estimated execution time.

The control formula that is used by the *CL-OVER-MISS* (*CL-OVER-REJ*) controller, as shown in Figure 6.7a(b), is as follows:

$$\Delta etf_{kT}^{m(r)} = K_{m(r)} \times error_{kT}^{m(r)} \tag{6.7}$$

where $K_{m(r)}$ is the coefficient of the controller, $\Delta etf_{kT}^{m(r)}$ is the amount of change that needs to be added or reduced from the estimate factor (etf) due to the error in the miss ratio (rejection ratio), and $error_{kT}^{m(r)}$ is the difference between the miss ratio (rejection ratio) at time instant k (MR_{kT} (RR_{kT})) and the set point (MR_s (RR_s)) i.e., $error_{kT}^{m} = MR_s - MR_{kT}$, and $error_{kT}^{r} = RR_s - RR_{kT}$. This approach can only control the deadline miss ratio (task rejection ratio) of the system. Hence, this approach may result in rejecting too many tasks (missing too many deadlines) in order to keep the deadline miss ratio (task rejection ratio) equal to the set point (MR_s (RR_s)).

In order to be able to control both the miss ratio and the rejection ratio of the system (enhance the effective ratio), we propose the *CL-OVER-MISSREJ* approach in which both the miss ratio and the rejection ratio are measured and fed back to the controller. The control formula that is used by the *CL-OVER-MISSREJ* controller, as shown in Figure 6.7c, is as follows:

$$\Delta etf_{kT}^{mr} = nf_{m/r} \times K_{mr} \times error_{kT}^{m} - nf_{r/m} \times K_{mr} \times error_{kT}^{r} \tag{6.8}$$

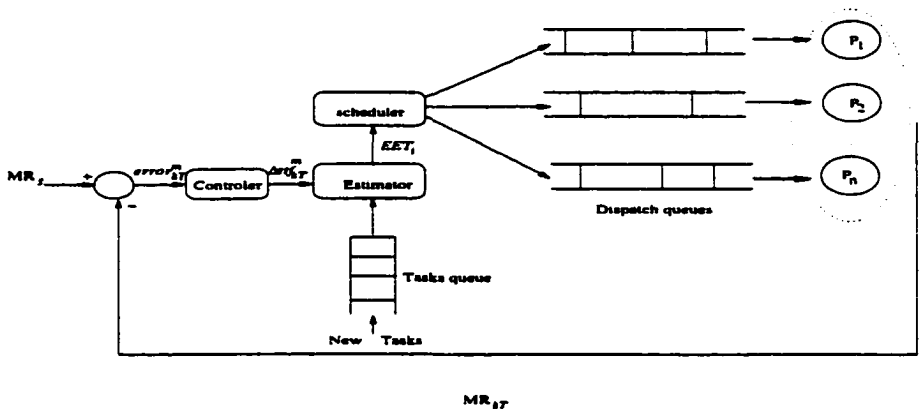
where K_{mr} is the coefficient of the controller, Δetf_{kT}^{mr} is the amount of change due to the error in both the miss ratio and rejection ratio, $nf_{m/r}$ ($nf_{r/m}$) is normalization factor that is used to normalize the amount of change Δetf_{kT}^{m} (Δetf_{kT}^{r}). The idea behind using these normalization factors is that the sensitivity of miss ratio (MR_{kT}) to a variation in the estimated factor (Δetf_{kT}) is different from the sensitivity of rejection ratio (RR_{kT}) to the same variation. Therefore, the normalization factors are used to normalize the amount of variation that is asked by each feedback loop so that the sum of these normalized variations results in the net change that needs to be applied to the estimated factor. The value of these normalization factors ($nf_{m/r}$ and $nf_{r/m}$) will be discussed in the next section.

This approach tries to maximize the effective ratio (ER). Note that we did not feed back the ER or $1 - ER$ to design the feedback system that controls the effective ratio directly. This is because there is no defined actuators that can be used to change the control variable (etf) to control the regulated variable (ER) to the desired value.

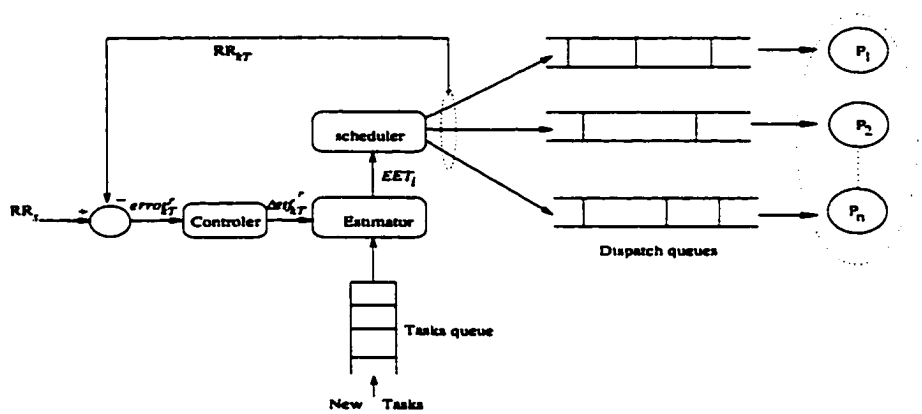
6.5.2 Modeling of CL-OVER Approaches

An important task in designing a closed-loop scheduler is to establish a model of the scheduling system so that we can apply control analysis to the scheduler. Before we present the models of the closed-loop scheduling algorithms, we define the following notions in the Z-transform:

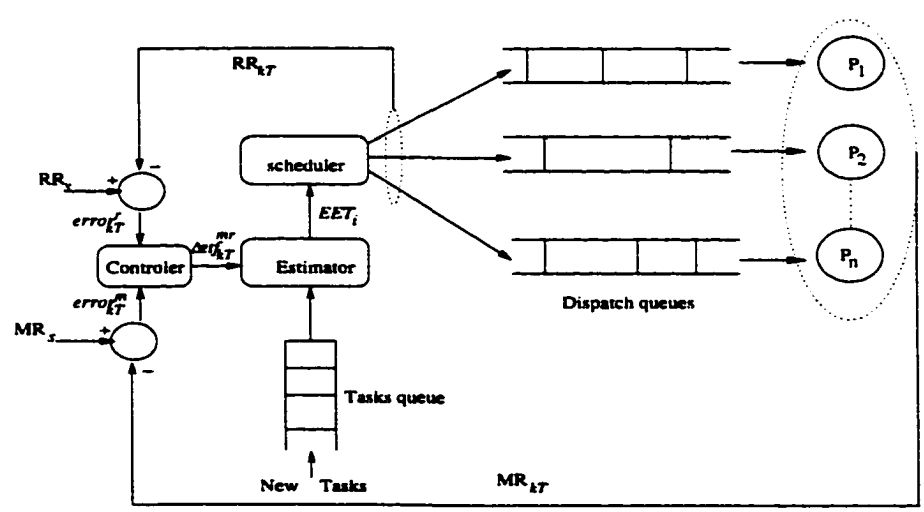
1. T: is a constant sampling period, which is the time elapsed in interval $[kT,\ (k+1)T]$, k being time instants.
2. $MR(z)$ and $RR(z)$: the miss ratio and the rejection ratio respectively, which are the system outputs and the controlled variables.
3. MR_s and RR_s: the set points in terms of miss ratio and rejection ratio.
4. $etf(z)$: the estimated factor, which is the control variable.
5. $\Delta etf(z)$: the change in the estimated factor ($etf(z)$). This is the system's input.
6. mgf and rgf: the gain that maps the estimated factor to the miss ratio and the rejection ratio respectively.
7. mdf and rdf: the disturbance associated with the miss ratio and the rejection ratio respectively which are resulting from the system bounds (e.g., $MR(z) \in [0,1]$, $RR(z) \in [0,1]$, $etf \in [-1,\ 1]$, and processor utilization bound).
8. $nf_{m/r}$ and $nf_{r/m}$: normalization factors that are used in the CL-OVER-MISSREJ approach to normalize the errors generated from the miss ratio and the rejection ratio respectively. The values for these normalization factors are the ratio between the miss ratio gain factor and the rejection ratio gain factor i.e., $nf_{m/r} = \frac{mgf}{rgf}$ and $nf_{r/m} = \frac{rgf}{mgf}$.



(a) CL-OVER-MISS



(b) CL-OVER-REJ



(c) CL-OVER-MISSREJ

Figure 6.7 Architecture of CL-OVER approaches

Using the above notations and Equation (6.5), the Z-transform for the estimated factor follows the following equation

$$etf(z) = \frac{z\Delta etf}{1-z} \tag{6.9}$$

We can derive the deadline miss ratio (task rejection ratio) based on the correlation between the $MR(z)$ ($RR(z)$) and $etf(z)$. They are approximately modeled as

$$MR(z) = -mgf \times etf(z) + mdf \tag{6.10}$$

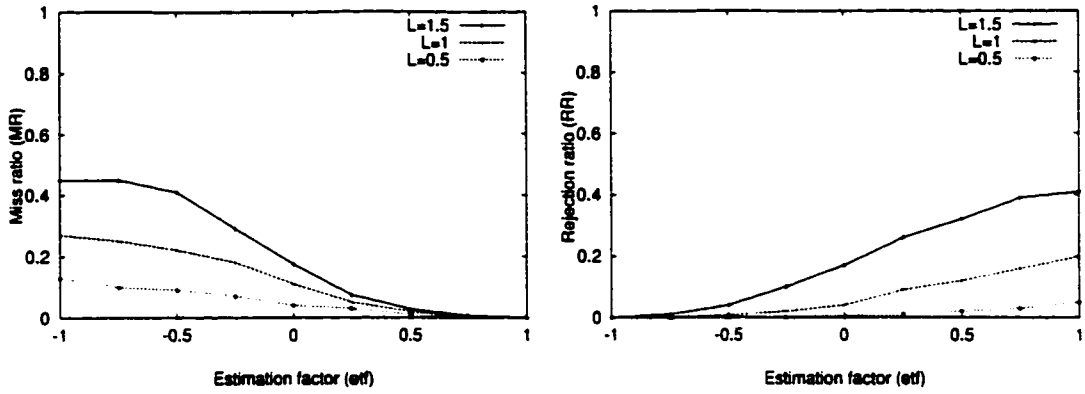
$$RR(z) = rgf \times etf(z) + rdf \tag{6.11}$$

The values for the mgf (rgf) and mdf (rdf) vary with the scheduling algorithm, the system load, and the system parameter (e.g. number of processors). The values of these factors can be found by studying the relation between the etf and the miss ratio (rejection ratio) using the simulation model.

In Equations (6.10) and (6.11), we approximate the relation between the estimation factor and the rejection ratio and the miss ratio as a linear line. This approximation is not accurate since scheduling system typically contains non-linear factors which are not presented in the current model. Figure 6.8 shows how the MR and the RR vary as the etf changes for different task loads (L) in the overlap scheduling algorithm.

These figures have been generated using the simulation model (see Section 6.6). For each point in Figures 6.8, the system was simulated for 10,000 tasks. This number of tasks has been chosen to have a 98% confidence interval within ± 0.0017, and ± 0.0022 around each value of MR, and RR respectively.

From the figure, we can notice that the relation between the etf and both the rejection ratio and the miss ratio are not linear. Figure 6.8a shows that the MR starts from a maximum value when $etf = -1$ and reduces quadratically as etf increases. For all task loads that are plotted the MR reaches zero when ($etf > 0.75$). Figure 6.8b shows that the RR starts from approximately zero when $etf = -1$ and increases quadratically to a maximum value as etf increases to 1. From the figure, we can also notice that the relation between the the etf and both the rejection ratio and the miss ratio varies as the system task load (L) varies.



(a) MR as etf varies (b) RR as etf varies

Figure 6.8 The MR and the RR as etf varies for different task loads

6.5.3 Tuning of CL-OVER Approaches

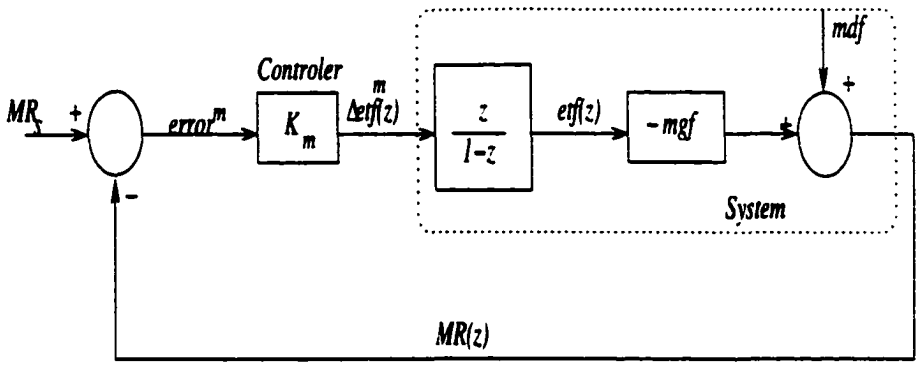
In this section, we study the effect of the Control gain (K) and the sample period (T) on the *percent overshoots* and *settling time* (T_s) for step task load. The *percent overshoots (PO)* is defined as

$$PO = \frac{M_{pt} - f_v}{f_v} \times 100\% \tag{6.12}$$

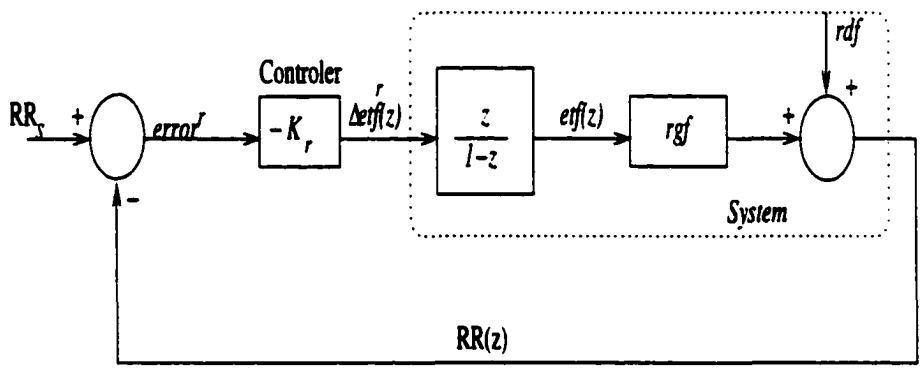
where M_{pt} is the peak value of the time response and f_v is the final value of the response. The *settling time* (T_s) is defined as the time required for the system to settle within a certain percentage δ of the final value.

6.5.3.1 Control gain

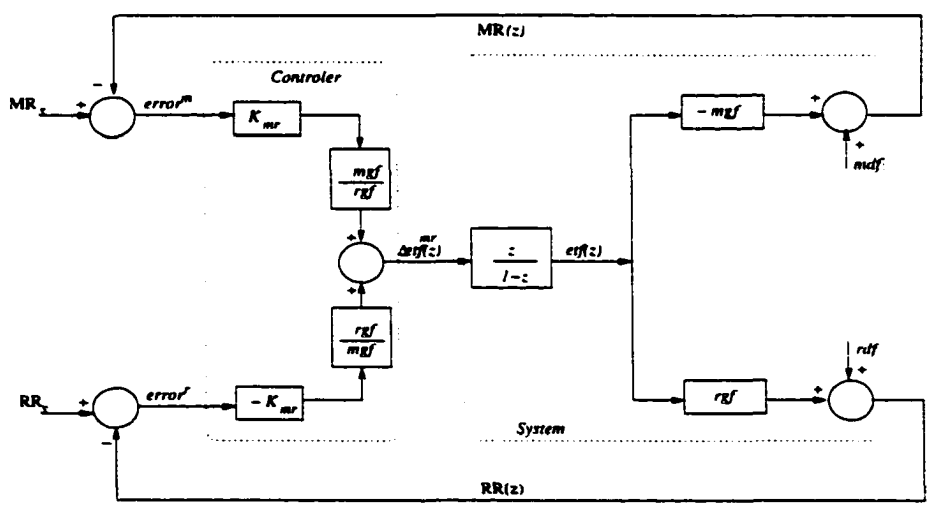
To find the value of the controller parameter for each approach (K_m, K_r, and K_{mr}) a stability analysis must be carried out. A *stable system* is defined as a system which results in bounded output when subjected to a bounded input. For linear systems, the stability requirement may be defined in terms of the location of the poles of the closed-loop transfer function. A discrete time system is stable if all the poles of the closed-loop transfer function lie within the unit circle (assuming no unstable pole-zero cancellation occur).



(a) CL-OVER-MISS



(b) CL-OVER-REJ



(c) CL-OVER-MISSREJ

Figure 6.9 Block diagram of the CL-OVER approaches

From Figure 6.9, the characteristic equation for the *CL-OVER-MISS*, *CL-OVER-REJ*, and *CL-OVER-MISSREJ* approaches, respectively are:

$$1 + \frac{z \times G_m}{z-1} = 0, \quad 1 + \frac{z \times G_r}{z-1} = 0, \quad \text{and } 1 + \frac{z \times G_{mr}}{z-1} = 0 \tag{6.13}$$

where G_m, G_r, and G_{mr} is the closed-loop gains, for the *CL-OVER-MISS*, *CL-OVER-REJ*, and *CL-OVER-MISSREJ* approaches, respectively. G_m is equal to $mgf \times K_m$, G_r is equal to $rgf \times K_r$, and G_{mr} is equal to $K_{mr} \times (mgf \times nf_{m/r} + rgf \times nf_{r/m})$. From Equation (6.13) the closed-loop approaches are stable for $G_m > 0$, $G_r > 0$, and $G_{mr} > 0$. Since rgf, mgf, $nf_{m/r}$, and $nf_{r/m}$ are all greater than 0, then the closed-loop approaches are stable for any value of K_m, K_r, and K_{mr} greater than 0. However, the transient response for these approaches is affected by the values that are used for these parameters (i.e., the percent overshoots and the settling time are functions of these parameter values).

Figure 6.10 shows the impact of the control gain (K) on percent overshoots (PO), for the three closed-loop scheduling algorithms, in response to a step task load. A step task load SL is a task load that instantaneously jumps from a nominal task load L_{nom} to a task load L_{max} and stay constant after the jump. The step task load is represented with a tuple $SL(L_{nom}, L_{max})$. In the context of real-time systems, the step task load represents the worst-

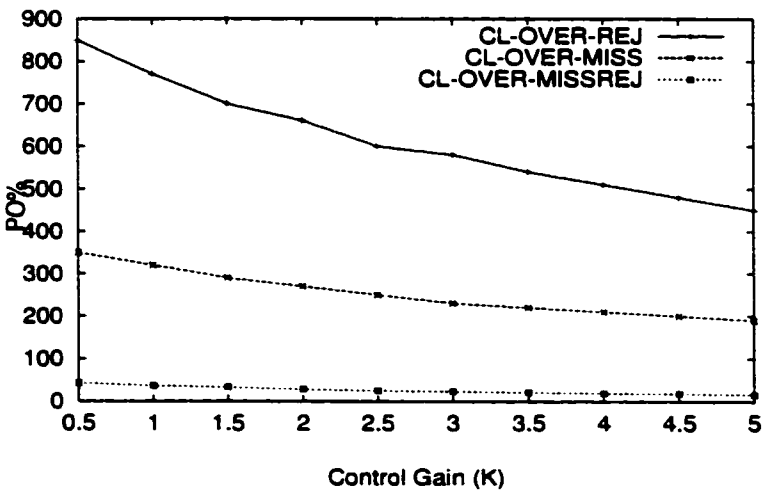


Figure 6.10 Percent overshoot (PO) vs control gains for SL(0.5,1.5)

case task load variation. where $MR_s = 1\%$, $RR_s = 1\%$, $T = 80sec$, and SL(0.5,1.5). We notice that the system overshoot decreased as the controller gains increased for the three scheduling algorithms. This is because the scheduler with small control gains adapted to the system load slowly and consequently allowed the miss ratio or rejection ratio to increase significantly. We also notice that the *CL-OVER-REJ* algorithm has the highest overshoot for all control gains. This is due to the fact that the sensitivity of the rejection ratio to a change in the system load is very high. On the other hand, the *CL-OVER-MISSREJ* algorithm has the lowest overshoot for all control gains which is always less than 50%. This is because the *CL-OVER-MISSREJ* uses the normalization factors to normalize the amount of variation that is asked by each feedback loop so that the sum of these normalized variations result in the net change that needs to be applied to the estimated factor. This net change minimizes the overshoot value for this algorithm.

Figure 6.11 shows the effects of the control gain (K) on the settling time (T_s) in response to SL(0.5,1.5) for the three closed-loop scheduling algorithms, where $MR_s = 1\%$, $RR_s = 1\%$, and $T = 80sec$. Here we note that the system settling time decreased as the controller gain increased for the three scheduling algorithms. This is because the controller can change the estimation factor by a higher magnitude for each sampling period if their control gains are higher. From the figure we can also notice that the *CL-OVER-MISS* algorithm has the highest settling time for all control gains. The reason is that this algorithm has the smallest loop gain (G_m), which results in increasing the time it takes to settle down to its final value. However, the *CL-OVER-MISSREJ* algorithm has the lowest settling time for all control gains which is always less than 40 T. This is because the *CL-OVER-MISSREJ* has the highest loop gain ($G_{mr} = G_m + G_r$), which decreases the time it takes to settle down to its final value.

6.5.3.2 Sampling period

In our simulation we found that the system was unstable when sample period (T) is smaller than a certain value T_0. The value of T_0 is equal to $\max(\theta, C)$, where C is the mean computation time of the system and θ is the mean arrival time of tasks. That is because, in the case

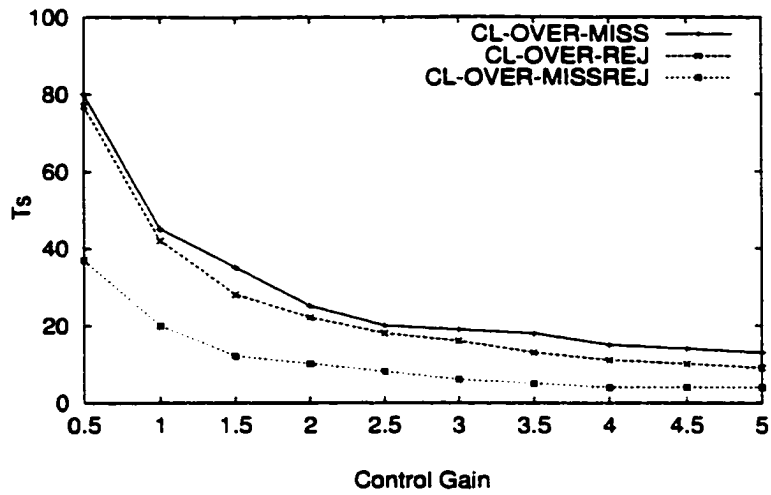


Figure 6.11 Settling time (T_s) vs control gains for SL(0.5,1.5)

of feeding back the rejection ratio, if the sample period is less than the mean arrival time of tasks, there is a chance that the number of tasks arrived in the sample interval $[kT,\ (k+1)T]$ is zero or a very small number. This results in feeding back an inaccurate estimation for the rejection ratio. In the case of feeding back the miss ratio, if the sample period is less than the mean computation time of the system , there is a chance that the number of tasks finished in the sample interval $[kT,\ (k+1)T]$ is zero or a very small number. This results in feeding back an inaccurate estimation for the miss ratio.

Among the stable sampling periods, the settling time (T_s) in response to the step load increased significantly as T increased. This is because the controllers with a smaller sampling period monitored and responded to load variations at a higher rate and thus settled down faster. Also, the closed-loop approaches achieved significantly lower overshoot in response to the step load as T decreased. This is also because the controllers adapted faster with a smaller sampling period.

6.5.4 Analysis of Closed-Loop Scheduling Algorithms

In this section, we compare the performance of the closed-loop scheduling algorithms (*CL-OVER-MISS*, *CL-OVER-REJ*, and *CL-OVER-MISSREJ*) using the transfer function for the

block diagrams in Figure 6.9. The steady state effective ratio (ER_f) has been used as the performance metrics.

From Figure 6.9a, the transfer function for the deadline miss ratio ($MR(z)$) and the task rejection ratio ($RR(z)$) in the Z domain are:

$$MR(z) = \frac{z \frac{mdf - MR_s}{1+G_m}}{z - \frac{1}{1+G_m}} + \frac{z\, MR_s}{z-1} \tag{6.14}$$

$$RR(z) = \frac{z \frac{(MR_s - mdf)\ nf_{r/m}}{1+G_m}}{z - \frac{1}{1+G_m}} + \frac{z\left((mdf - MR_s)\ nf_{r/m} + rdf\right)}{z-1} \tag{6.15}$$

By using the inverse Z-transform the miss ratio (MR_{kT}) and the rejection ratio (RR_{kT}) in the discrete time domain are:

$$MR_{kT} = \begin{cases} 0 & \text{for } kT < 0 \\ \frac{mdf - MR_s}{1+G_m}\left(\frac{1}{1+G_m}\right)^{kT} + MR_s & \text{for } kT \geq 0 \end{cases} \tag{6.16}$$

$$RR_{kT} = \begin{cases} 0 & \text{for } kT < 0 \\ \frac{(MR_s - mdf)\ nf_{r/m}}{1+G_m}\left(\frac{1}{1+G_m}\right)^{kT} + (mdf - MR_s)\ nf_{r/m} + rdf & \text{for } kT \geq 0 \end{cases} \tag{6.17}$$

From Equations (6.16) and (6.17) we notice that the steady state values for miss ratio and rejection ratio (MR_f and RR_f) in the *CL-OVER-MISS* are MR_s and $(mdf - MR_s)\ nf_{r/m} + rdf$ respectively. Therefore, the steady state effective ratio (ER_f) is equal to:

$$ER_f = (1 - MR_s) \times (1 - rdf - (mdf - MR_s)\ nf_{r/m}) \tag{6.18}$$

From Figure 6.9b, the transfer function for the task rejection ratio ($RR(z)$) and deadline miss ratio ($MR(z)$) in the Z domain are:

$$RR(z) = \frac{z \times \frac{rdf - RR_s}{1+G_r}}{z - \frac{1}{1+G_r}} + \frac{z \times RR_s}{z-1} \tag{6.19}$$

$$MR(z) = \frac{z \frac{(RR_s - rdf)\ nf_{m/r}}{1+G_r}}{z - \frac{1}{1+G_r}} + \frac{z\left((rdf - RR_s)\ nf_{m/r} + mdf\right)}{z-1} \tag{6.20}$$

By using the inverse Z-transform the rejection ratio (RR_{kT}) and the miss ratio (MR_{kT}) in the discrete time domain are:

$$RR_{kT} = \begin{cases} 0 & \text{for } kT < 0 \\ \frac{rdf - RR_s}{1+G_r}\left(\frac{1}{1+G_r}\right)^{kT} + RR_s & \text{for } kT \geq 0 \end{cases} \tag{6.21}$$

$$MR_{kT} = \begin{cases} 0 & \text{for } kT < 0 \\ \frac{(RR_s - rdf)\ nf_{m/r}}{1+G_r}\left(\frac{1}{1+G_r}\right)^{kT} + (rdf - RR_s)\ nf_{m/r} + mdf & \text{for } kT \geq 0 \end{cases} \tag{6.22}$$

From Equations (6.22) and (6.21) we notice that the steady state values for miss ratio and rejection ratio (MR_f and RR_f) in the *CL-OVER-REJ* are $(rdf - RR_s)\ nf_{m/r} + mdf$ and RR_s respectively. Therefore, the steady state effective ratio (ER_f) is equal to:

$$ER_f = (1 - RR_s) \times (1 - mdf - (rdf - RR_s)\ nf_{m/r}) \tag{6.23}$$

From Figure 6.9c, the transfer function for the deadline miss ratio ($MR(z)$) and the task rejection ratio ($RR(z)$)in the Z domain are:

$$MR(z) = \frac{z\ \frac{(RR_s - rdf)\ nf_{m/r}\ G_r + (mdf - MR_s)\ G_m}{G_{mr}\ (1+G_{mr})}}{z - \frac{1}{1+G_{mr}}} + \frac{z\ \frac{MR_s\ G_m + \left((rdf - RR_s)\ nf_{m/r} + mdf\right)\ G_r}{G_{mr}}}{z - 1} \tag{6.24}$$

$$RR(z) = \frac{z\ \frac{(MR_s - mdf)\ nf_{r/m}\ G_m + (rdf - RR_s)\ G_r}{G_{mr}\ (1+G_{mr})}}{z - \frac{1}{1+G_{mr}}} + \frac{z\ \frac{RR_s\ G_r + \left((mdf - MR_s)\ nf_{r/m} + rdf\right)\ G_m}{G_{mr}}}{z - 1} \tag{6.25}$$

the rejection ratio (RR_{kT}) and the miss ratio (MR_{kT}) in the discrete time domain are:

$$MR_{kT} = \begin{cases} 0 & \text{for } kT < 0 \\ \left(\frac{(RR_s - rdf)\ nf_{m/r}\ G_r + (mdf - MR_s)\ G_m}{G_{mr}\ (1+G_{mr})}\right)\left(\frac{1}{1+G_{mr}}\right)^{kT} & \\ \quad + \frac{MR_s\ G_m + \left((rdf - RR_s)\ nf_{m/r} + mdf\right)\ G_r}{G_{mr}} & \text{for } kT \geq 0 \end{cases} \tag{6.26}$$

$$RR_{kT} = \begin{cases} 0 & \text{for } kT < 0 \\ \left(\frac{(MR_s - mdf)\ nf_{r/m}\ G_m + (rdf - RR_s)\ G_r}{G_{mr}\ (1+G_{mr})}\right)\left(\frac{1}{1+G_{mr}}\right)^{kT} & \\ \quad + \frac{RR_s\ G_r + \left((mdf - MR_s)\ nf_{r/m} + rdf\right)\ G_m}{G_{mr}} & \text{for } kT \geq 0 \end{cases} \tag{6.27}$$

From Equations (6.26) and (6.27) we notice that the steady state values for miss ratio and rejection ratio (MR_f and RR_f) in the *CL-OVER-MISSREJ* are $\frac{MR_s\ G_m+\left((rdf-RR_s)\ nf_{m/r}+mdf\right)\ G_r}{G_{mr}}$ and $\frac{RR_s\ G_r+\left((mdf-MR_s)\ nf_{r/m}+rdf\right)\ G_m}{G_{mr}}$, respectively. Therefore, the steady state effective ratio (ER_f) is equal to:

$$ER_f = \begin{array}{l}(1-\frac{MR_s\ G_m+\left((rdf-RR_s)\ nf_{m/r}+mdf\right)\ G_r}{G_{mr}}) \\ \times(1-\frac{RR_s\ G_r+\left((mdf-MR_s)\ nf_{r/m}+rdf\right)\ G_m}{G_{mr}})\end{array} \tag{6.28}$$

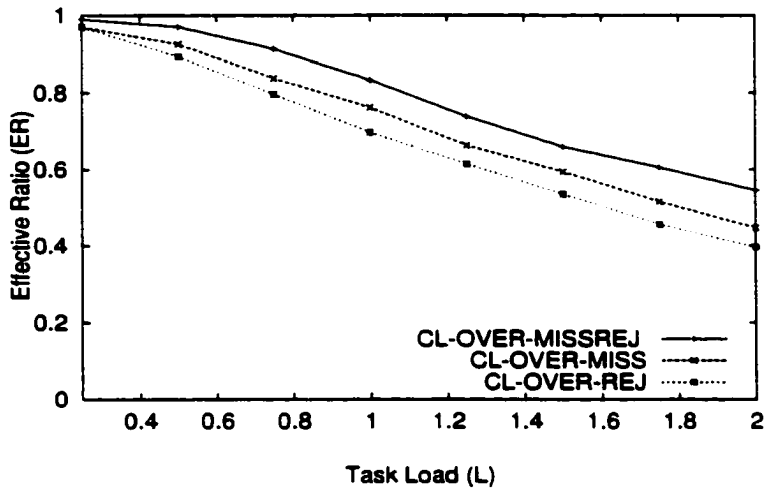


Figure 6.12 Steady state effective ratio for the three closed-loop algorithms

Equations (6.18), (6.23), and (6.28) have been plotted in Figure 6.12 for different task loads. With $MR_s = 0.01$ and $RR_s = 0.01$. The figures show that the *CL-OVER-MISSREJ* approach offers the highest effective ratio for all task loads. This is because, in this approach the feedback mechanism and the controller try to adapt, as the task load varies, the estimated execution time for the tasks to a value that maximizes the product of deadline hit ratio by the task guarantee ratio (i.e., maximize the effective ratio). The figure also shows that the *CL-OVER-REJ* approach offers the lowest effective ratio for all task loads. This is due to the fact that this approach tries to achieve a rejection ratio equal to 1%. This results in significantly decreasing the hit ratio as the task load increases, which degrades the effective ratio.

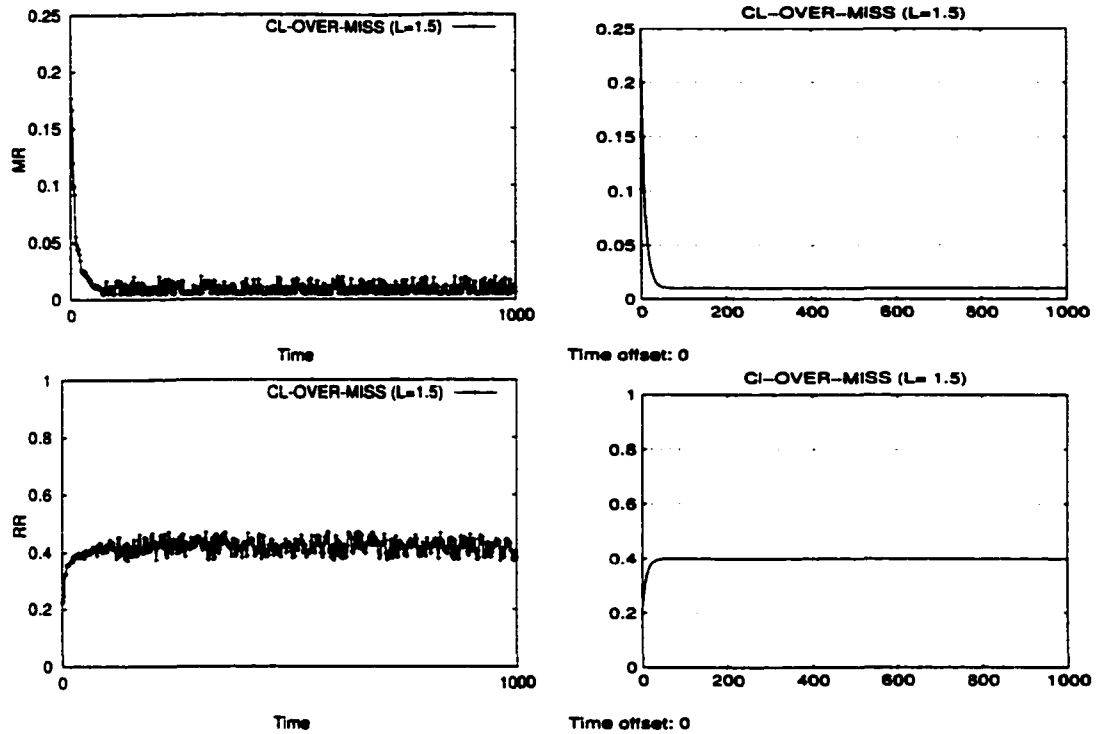
6.5.5 Verification of CL-OVER Models

In this section, we compare the behavior of the simulation models (see Section 6.6) with that of the block digram models for the closed-loop scheduling algorithms. The instantaneous values of miss ratio (MR_{kT}) and the instantaneous value of the rejection ratio (RR_{kT}) have been used as the performance metrics. Two experiments have been used to study the behavior of the closed-loop scheduling algorithms. In the first experiment (Experiment A), the task load (L) for each simulation run is statistically steady (i.e. the mean arrival time between tasks (θ) is constant for the whole simulation run). In the second experiment (Experiment B), the task load (L) for each simulation run is dynamically changed using the step task load profile. Each point in the performance plots (Figures 6.13a-6.18a) is the average of 20 runs. In each run the system was simulated with 10,000 tasks. This number of runs has been chosen to have a 98% confidence interval within ± 0.013, and ± 0.015, around each value of MR, and RR respectively.

6.5.5.1 Experiments A: Steady task load

Experiment A studies the behavior of the closed-loop scheduling algorithms when the task load (L) offered to the system is statistically steady and equal to 1.5. Figures 6.13, 6.14, and 6.15 show the MR_{kT} and the RR_{kT} for the *CL-OVER-MISS*, *CL-OVER-REJ*, and *CL-OVER-MISSREJ* approaches respectively. $K_m = K_r = K_{mr} = 0.5$, $MR_s = RR_s = 0.01$, and $T = 40$ *sec.* Figures 6.13a, 6.14a, and 6.15a show the results from the simulation model and Figures 6.13b, 6.14b, and 6.15b show the corresponding results from the Matlab simulation.

The figures show that the simulation and the Matlab models behave similarly in the steady state region. From the figure we, notice that there is a slight difference between the simulation and the Matlab models in how fast the algorithms reach the steady state values and in the initial values of the MR_{kT} and the RR_{kT}. This difference is due to the fact that in our modeling we assume that the relation between the miss ratio (rejection ratio) and the estimation factor is linear which is an approximation since the scheduling system typically contains non-linear factors which are not presented in the current model.

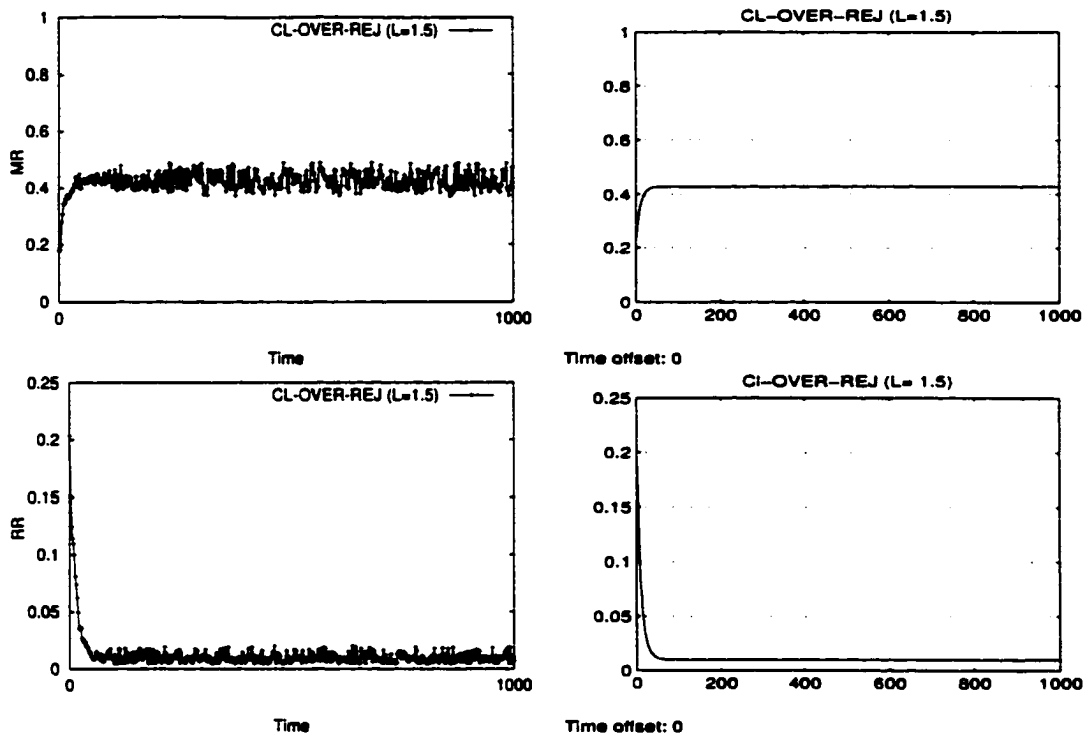


(a) Simulation (b) Analytical

Figure 6.13 MR_{kT} and RR_{kT} for the *CL-OVER-MISS* algorithm

6.5.5.2 Experiments B: Dynamic task load

Experiment B studies the behavior of the closed-loop scheduling algorithms (*CL-OVER-MISS*, *CL-OVER-REJ*, and *CL-OVER-MISSREJ*) when the task load (L) for each simulation run is dynamically changed using the step task load profile. Figure 6.16 shows the MR_{kT} and the RR_{kT} for the *CL-OVER-MISS* algorithm when $k_m = 0.5$, $MR_s = 0.01\%$, $T = 80$ *sec*, and step load profile that jumps form $L = 0.25$ to $L = 2$ at $k = 500$ ($SL(0.25, 2)$). Figure 6.16a shows the results from the simulation model and Figure 6.13b shows the corresponding results from the Matlab simulation. The figure reveals that for the time interval [0, 500] the system is lightly loaded ($L = 0.25$) which results in approximately both a zero miss ratio and a zero rejection ratio. At time $k = 500$ the system task load jumps to 2 which results in a heavily loaded system. In response to this jump in the task load, the instantaneous miss ratio in the system (MR_{kT}) overshoots to a maximum value approximately equal to 0.2. The *CL-OVER-*

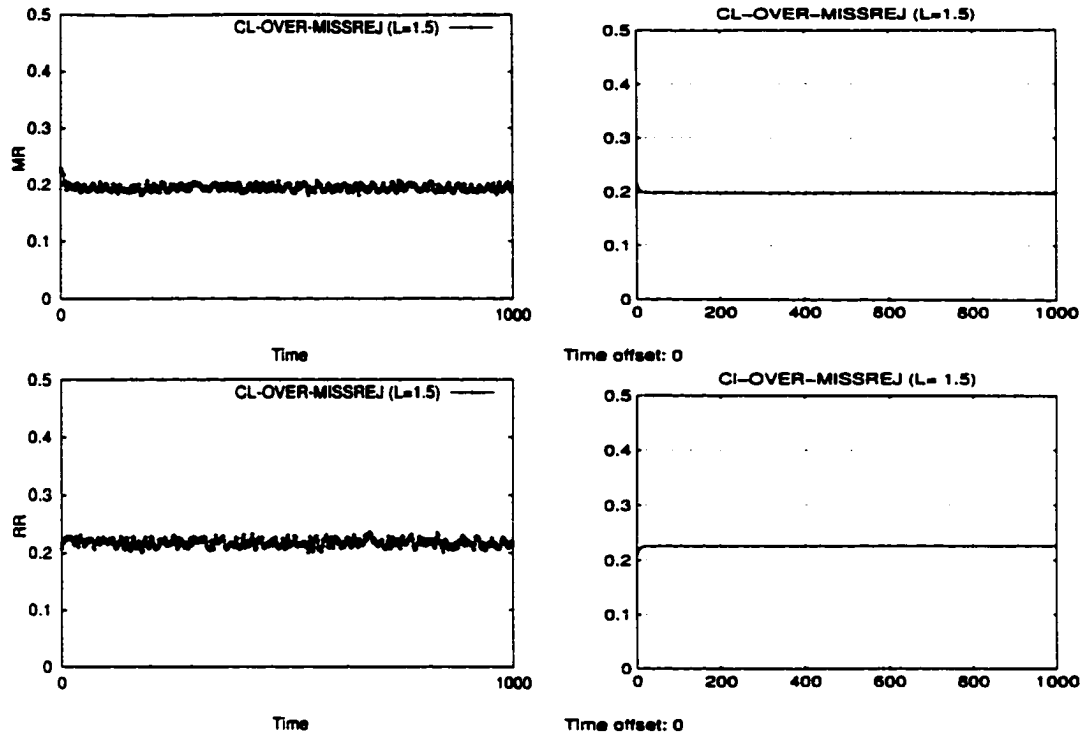


(a) Simulation (b) Analytical

Figure 6.14 MR_{kT} and RR_{kT} for the *CL-OVER-REJ* algorithm

MISS responds immediately by increasing the estimated factor within $80T$. The system comes back to a low miss ratio and stays this way until the end of the run. On the other hand, the instantaneous rejection ratio in the system (RR_{kT}) jumps to a value approximately equal to 0.6 and stays this way until the end of the run. The figures indicate that the simulation and the Matlab models agree with each other. However, there is a slight difference between the simulation and the Matlab models in the overshoot value of the MR_{kT} and how fast the algorithm return back to the steady state value.

Figure 6.17 shows the MR_{kT} and the RR_{kT} for the *CL-OVER-REJ* algorithm when k_m = 0.5, $RR_s = 0.01\%$, $T = 80$ *sec*, and step load profile that jumps form $L = 0.5$ to $L = 1.5$ at $k = 500$ ($SL(0.5, 1.5)$). The figure shows that for the time interval [0, 500] the system load is medium ($L = 0.5$) which results in approximately zero rejection ratio and 0.05 miss ratio. At time $k = 500$ the system task load jumps to 1.5 which results in a heavily loaded system. In response to this jump in the task load the instantaneous rejection ratio in the system (RR_{kT})

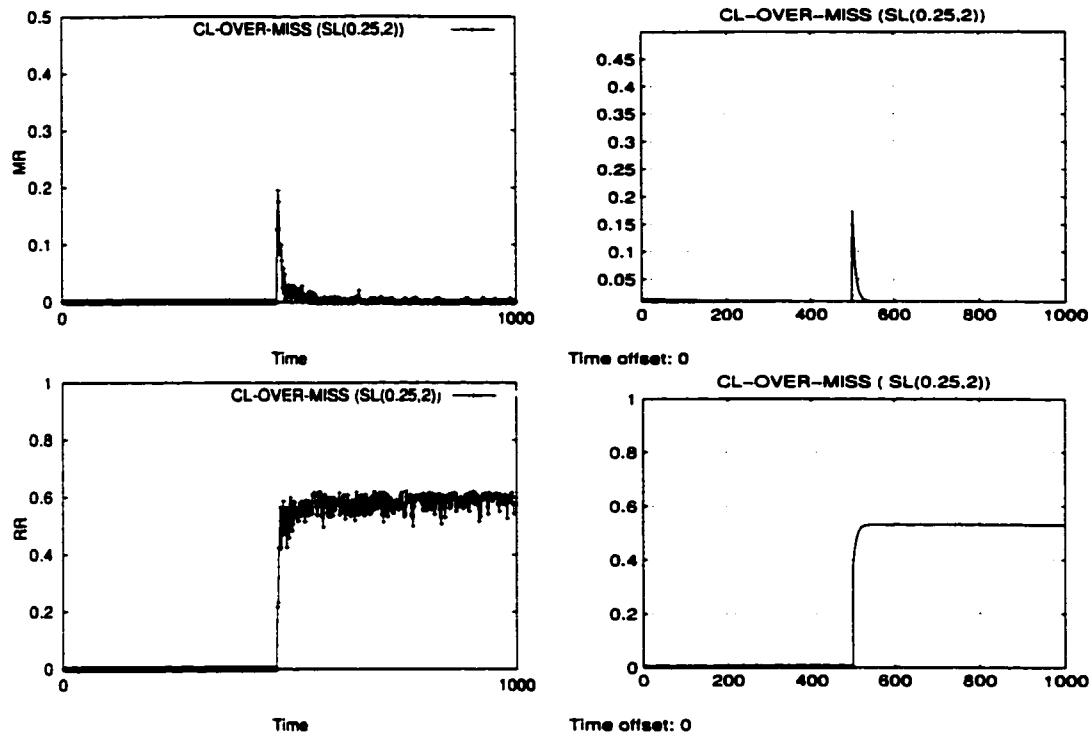


(a) Simulation (b) Analytical

Figure 6.15 MR_{kT} and RR_{kT} for the *CL-OVER-MISSREJ* algorithm

overshoots to a maximum value approximately equal to 0.12. The *CL-OVER-REJ* responds immediately by decreasing the estimated factor within $75T$. The system comes back to low rejection ratio and stays this way until the end of the run. On the other hand, the instantaneous miss ratio in the system (MR_{kT}) jumps to a value approximately equal to 0.4 and stays this way until the end of the run.

Figure 6.18 shows the MR_{kT} and the RR_{kT} for the *CL-OVER-MISSREJ* algorithm when $k_m = 0.5$, $RR_s = 0.01\%$, $T = 80$ *sec*, and the step load profile that jumps form $L = 0.5$ to $L = 1.5$ at $k = 500$ ($SL(0.5, 1.5)$). The figure indicates that for the time interval [0, 500] the system load is medium ($L = 0.5$) which results in approximately 0.035 rejection ratio and 0.025 miss ratio. At time $k = 500$ the system task load jumps to 1.5 which results in a heavily loaded system. In response to this jump in the task load the instantaneous rejection ratio in the system (RR_{kT}) overshoots to a maximum value approximately equal to 0.35. The *CL-OVER-MISSREJ* responds immediately by adjusting the estimated factor within 38 T. The



(a) Simulation (b) Analytical

Figure 6.16 MR_{kT} and RR_{kT} for the *CL-OVER-MISS* algorithm

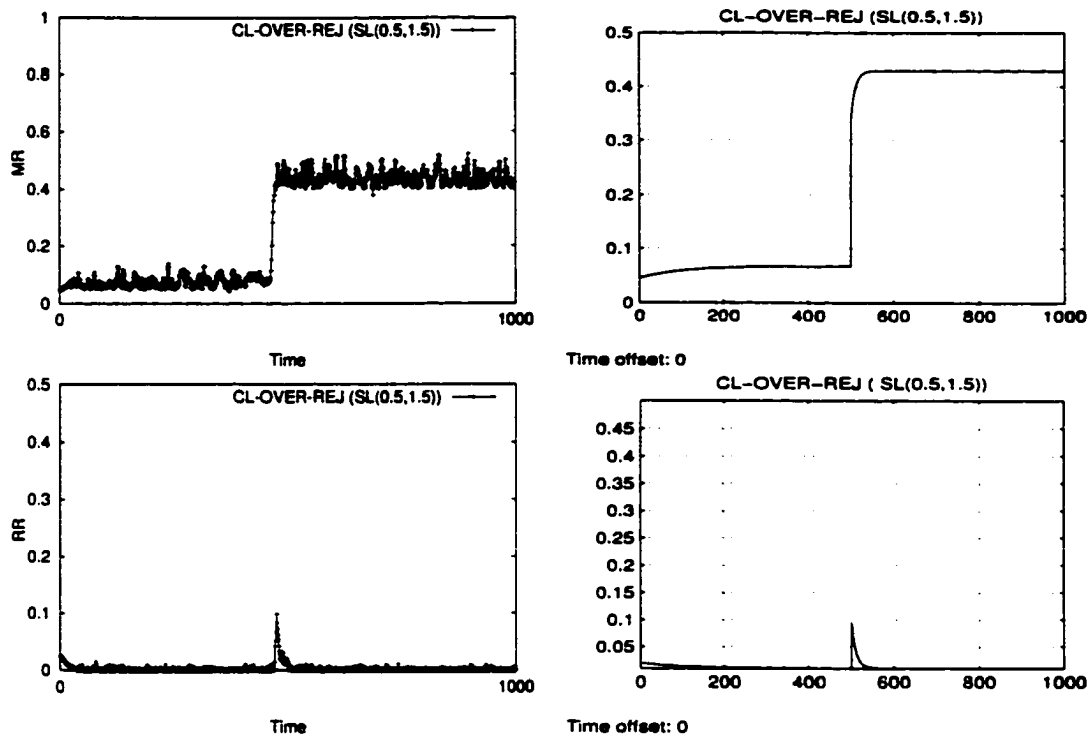
system comes back to approximately 0.25 rejection ratio and stays this way until the end of the run. On the other hand the instantaneous miss ratio in the system (MR_{kT}) jumps to a value approximately equal to 0.2 and stays this way until the end of the run.

6.6 Simulation Studies

A multiprocessor firm real-time system simulator was developed and used to study the performance of the open-loop and the closed-loop scheduling algorithms. The parameter used in the simulation studies are given in Table 6.2. The tasks for the simulation are generated as follows:

1. The best-case execution times ($BCET$) of tasks are chosen uniformly between Min_{BCET} and Max_{BCET}.

2. The worst-case execution time ($WCET_i$) of a task (T_i) is equal to f_{WCET} times its



(a) Simulation (b) Analytical

Figure 6.17 MR_{kT} and RR_{kT} for the *CL-OVER-REJ* algorithm

best-case execution time ($BCET_i$).

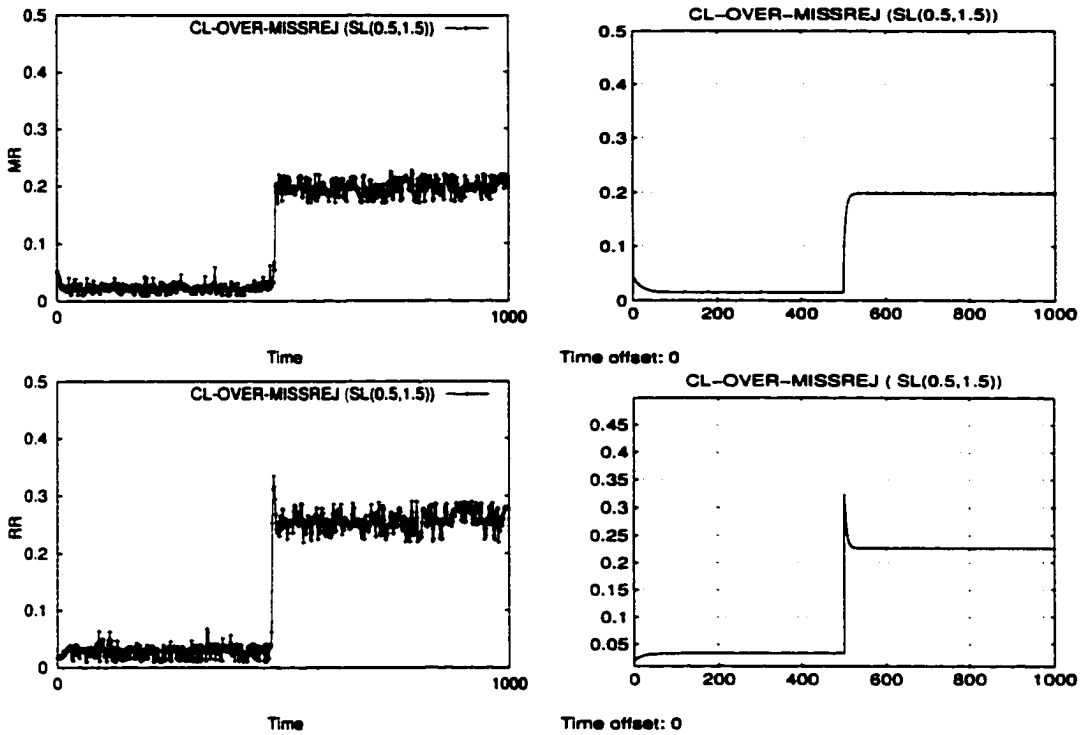
3. The average-case execution time ($AvCET_i$) of a task (T_i) is equal to $\frac{WCET_i+BCET_i}{2}$.

4. The actual execution time (AET_i) of a task (T_i) is computed as a uniform random variable in the interval $[BCET_i\ WCET_i]$.

5. The firm deadline (d_i) of a task T_i is uniformly chosen between $r_i + 2 \times WCET_i$ and $r_i + R \times WCET_i$, where R $\geq$ 2.

6. The arrival time of tasks follows exponential distribution with mean θ.

7. The task load L is defined as the expected number of task arrivals per mean service time and its value is approximately equal to $\frac{C}{\theta}$, where C is the mean computation time of the system. The mean computation time C has been calculated based on the $AvCET$ of tasks (i.e., $C = \frac{\frac{1}{n}\sum_{i=1}^{n} AVCET_i}{N}$, where n is the total arriving tasks).



(a) Simulation (b) Analytical

Figure 6.18 MR_{kT} and RR_{kT} for the *CL-OVER-MISSREJ* algorithm

The closed-loop simulator, shown in Figure 6.19a, has six components: a source which generates tasks; an estimator that estimates tasks execution time; a scheduler (overlap scheduling algorithm) that makes admission/rejection decisions on submitted tasks; a executor that models the execution of the tasks on multiprocessor system; monitors that periodically collect the performance statistics of the system; and a controller that takes the feedback miss ratio and/or rejection ratio, and the set points to calculate the amount of change that needs to be applied to the estimated execution time of tasks. The open-loop simulation, as shown in Figure 6.19b, has five components only. Wherein the controller and the feedback loops are deleted.

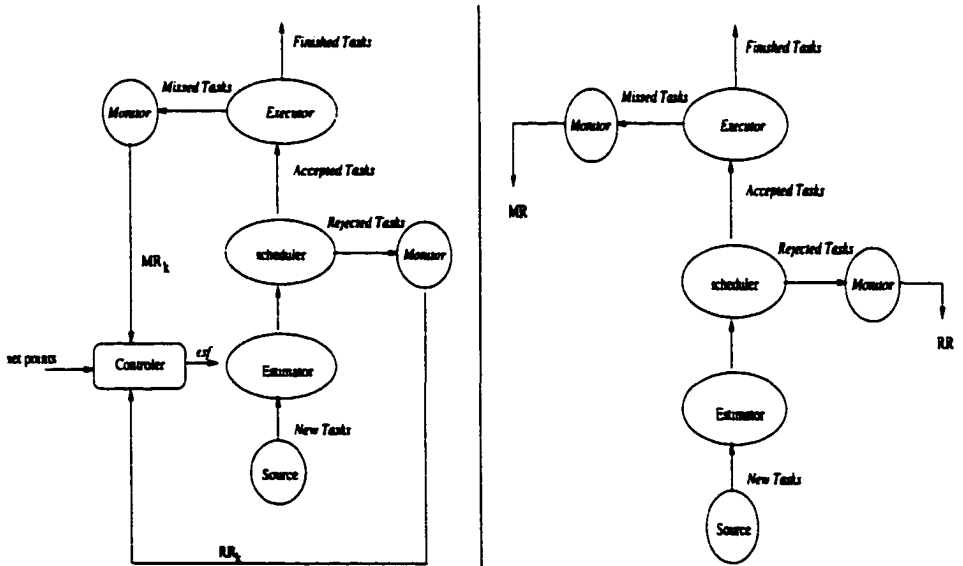
6.6.1 Performance Studies of CL-OVER Approaches

In this section, we compare the performance of the three closed-loop scheduling algorithms using the simulation model with the best open-loop scheduling algorithm (*OL-OVER-AvCET*) studied in Section 6.3.1. The average values of the task guarantee ratio (GR), the deadline

Table 6.2 Simulation parameters.

Parameter	Explanation	Values
Min_{BCET}	minimum BCET of tasks	10 sec.
Max_{BCET}	maximum BCET of tasks	20 sec.
f_{WCET}	the factor between the BCET and WCET of tasks	4
L	the offered task load to the system	0.25... 2
T	sample period	40,80,120,160 sec.
K_m, K_r, K_{mr}	The coefficients of the controllers	0.5 ... 5
R	laxity parameter	4
N	number of processors	5

hit ratio (HR), and the task effective ratio (ER) have been used as the performance metrics. For each point in the performance plots (Figures 6.20a-6.22a), the system was simulated for 10,000 tasks. This number of tasks has been chosen to have a 97% confidence interval within ± 0.0013, ± 0.0020, and ± 0.0028 around each value of HR, GR, and ER, respectively. In the analytical studies (Figures 6.20b-6.22b), we compare the performance of the closed-loop scheduling algorithms using the Matlab Simulink tools to verify the simulation results. For



(a) Closed-loop (b) Open-loop

Figure 6.19 Simulated models

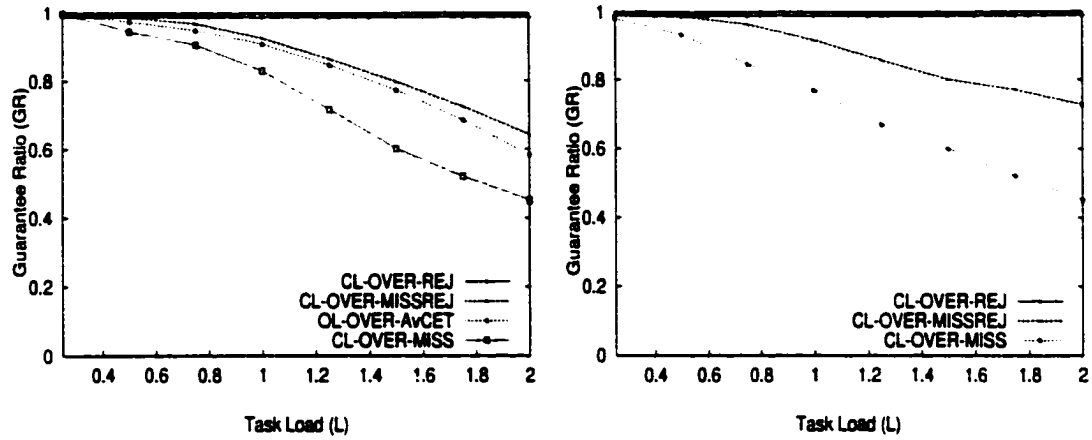
these studies the values of the gain factors (mgf and rgf) and the disturbance factors (mdf and rdf) have been estimate from the simulation studies for each task load.

6.6.1.1 Effects of task load (L) on the guarantee ratio (GR)

Figure 6.20 depicts GR changes with respect to task load (L). Figure 6.20a is the result from the simulation model and Figure 6.20b is the result from the Matlab simulation. As expected, an increasing L decreases the guarantee ratio for all the algorithms except for the *CL-OVER-REJ* where the guarantee ratio is constant and equals to 99%. This is due to the fact that in the *CL-OVER-REJ* approach the feedback mechanism always drives the rejection ratio to the set point ($RR_s = 1\%$) and since the GR is equal to $1 - RR$, the *CL-OVER-REJ* offers a GR equals to $1 - RR_s$. The figures also show that the *CL-OVER-MISS* approach offers the minimum guarantee ratio for all task loads and the difference between this approach and the other approaches increases as the task load increases. This is because, the *CL-OVER-MISS* approach tries to achieve a miss ratio equal to 1% for all task loads. Since the miss ratio increases as the task load increases, for the *OL-OVER* approaches, the feedback mechanism increases the estimated execution time of tasks as the task load increases. This results in significantly decreasing the guarantee ratio for this approach. From Figure 6.20a, we notice that the *CL-OVER-MISSREJ* approach offers a guarantee ratio greater than the ones offered by *CL-OVER-MISS* and *OL-OVER-AvCET* approaches. Figure 6.20b shows that the results obtained by simulating the block diagram using the Matlab tools confirms the simulation studies in their behavior. Since the average guarantee ratio (GR) has been calculated for a long run, its value is approximately equal to the steady state guarantee ratio from the Matlab simulation.

6.6.1.2 Effects of task load (L) on the hit ratio (HR)

Figure 6.21 shows the impact of task load (L) on HR. The figure shows that the *CL-OVER-MISS* approach offers a constant hit ratio which is equals to 99%. This is due to the fact that in the *CL-OVER-MISS* the feedback mechanism always drives the miss ratio to the set point



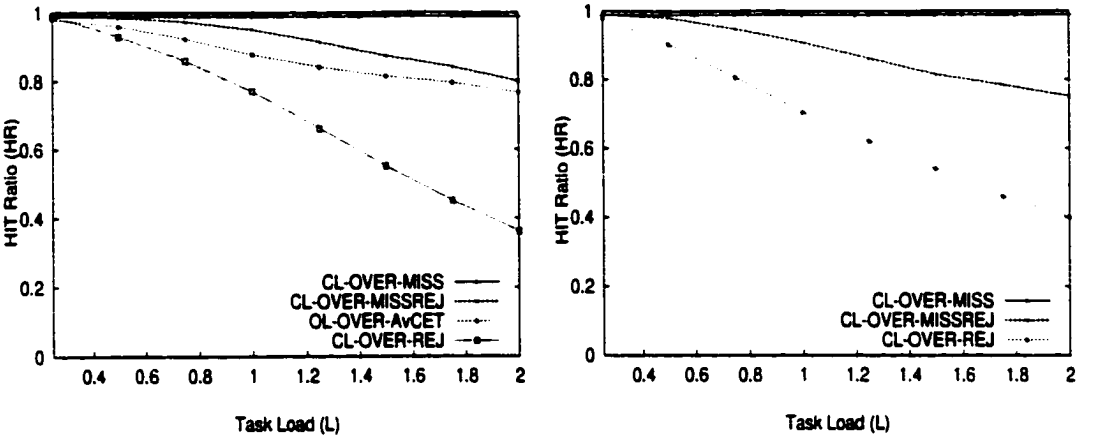
(a) Simulation (b) Analytical

Figure 6.20 Guarantee ratio for the three closed-loop and an open-loop algorithms

($MR_s = 1\%$) and since the HR is equal to $1 - MR$, the *CL-OVER-MISS* offers a HR equals to $1 - MR_s$. We also notice that the *CL-OVER-REJ* approach offers the minimum hit ratio for all task loads and the difference between this approach and the other approaches increases significantly as the task load increases. This is because, the *CL-OVER-REJ* approach tries to achieve a rejection ratio equal to 1% for all task loads. Since the rejection ratio increases as the task load increases, for all approaches, the feedback mechanism reduces the estimated execution time of tasks as the task load increases. Therefore, it significantly decreases the hit ratio for this approach. Figure 6.21a, indicates that the *CL-OVER-MISSREJ* approach offers a hit ratio greater than the ones offered by the *CL-OVER-REJ* and *OL-OVER-AvCET* approaches. Whereas, Figure 6.21b marks that the results obtained by simulating the block diagram using the Matlab tools indeed confirms the simulation studies in their behavior.

6.6.1.3 Effects of task load (L) on effective ratio (ER)

Figure 6.22 shows the effects of task load (L) on ER. The figure indicates that the *CL-OVER-MISSREJ* approach offers the highest effective ratio for all task loads. This is because, in this approach the feedback mechanism and the controller try to adapt, as the task load varies, the estimated execution time for the tasks to a value that maximizes the product of the



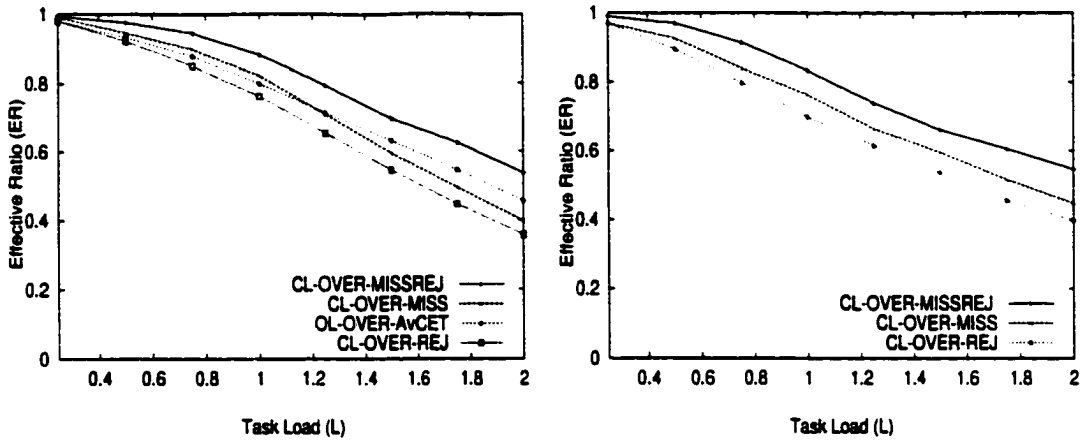
(a) Simulation (b) Analytical

Figure 6.21 Hit ratio for the three closed-loop and an open-loop algorithms

deadline hit ratio and task guarantee ratio (i.e., maximize the effective ratio). The figures also show that the *CL-OVER-REJ* approach offers the lowest effective ratio for all task loads. This is due to the fact that this approach tries to achieve a rejection ratio equal to 1%. This results in significantly decreasing the hit ratio as the task load increases, which degrades the effective ratio. From Figure 6.22a, we notice that the *CL-OVER-MISS* approach offers an effective ratio greater than the one offered by *OL-OVER-AvCET* approach for $L \leq 1.2$, where as it offers an effective ratio lower than the one offered by *OL-OVER-AvCET* approach for $L > 1.2$. This is because, for $L > 1.2$ the guarantee ratio offered by *CL-OVER-MISS* decreases significantly as compared to *OL-OVER-AvCET* when the tasks load increases.

6.7 Summary

In this chapter, we have considered the problem of dynamic scheduling of firm real-time tasks in multiprocessor systems where the task execution times vary dynamically. We have proposed an open-loop scheduling algorithm, which employs a notion of task overlap, that dynamically guarantees incoming firm tasks via on-line admission control and planning based on their average execution time. Secondly, we have used the feedback control theory to design a closed-loop approach. The loop is closed by feeding back *(i)* the deadline miss ratio in the



(a) Simulation (b) Analytical

Figure 6.22 Effective ratio for the three closed-loop and an open-loop algorithms

first approach; *(ii)* the task rejection ratio in the second approach; and *(iii)* both the miss ratio and rejection ratio in the final approach. The contributions and the results of this work are:

- Proposed a new performance metric, called effective ratio (ER), that is used to evaluate the proposed open-loop and closed-loop scheduling algorithms.
- Developed and analyzed an open-loop scheduling algorithm *(OL-OVER-AvCET)*, based on a concept of task overlap, for firm real-time tasks. Our studies show that the overlap based algorithm improves the effective ratio of the system by 15% over non-overlap algorithms.
- Developed and analyzed closed-loop scheduling algorithms using the feedback control theory.
- Developed and analyzed three closed-loop scheduling algorithms *(CL-OVER-MISS, CL-OVER-REJ,* and *CL-OVER-MISSREJ)* for firm real-time tasks. Our studies show that the proposed *CL-OVER-MISSREJ* algorithm performs better, in terms of the effective ratio, than *CL-OVER-MISS* or *CL-OVER-REJ.* Also, it performs better (20% gain) than the open-loop scheduling algorithm *(OL-OVER-AvCET).* Our studies Also show

that the *(OL-OVER-AvCET)* performs better, in terms of the effective ratio, than both *(CL-OVER-MISS*, and *CL-OVER-REJ* when the system load is high.

- Simulation and analytical experiments to study the instantaneous behavior of the miss ratio (MR_{kT}) and rejection ratio (RR_{kT}) in response to dynamically changing task load. In particular, we have studied this behavior for step task load. The purpose of these studies are to verify the analytical model against the simulation model.

CHAPTER 7. CONCLUSIONS

In this thesis, techniques have been introduced to offer trade-offs between schedulability and reliability in dynamic multiprocessor real-time systems through the use of scheduling. Different techniques have been introduced for different system models. The general approach to fault tolerance is the use of time redundancy. Time redundancy can be added to a system with very little changes, and thus can be used to upgrade the fault tolerance capabilities of existing real-time systems. The proposed techniques managed the processing capability of the system such that it maximizes the number of tasks that satisfied their timing constraints in the presents of faults and uncertainties. The mechanisms that were used to offer trade-offs in these techniques differ based on the deadline strictness for each model.

For the hard real-time systems, the techniques use certain schedulability tests, which provide guarantees regarding the number and frequency of faults that can be tolerated. These tests use a set of conditions. The efficiency of these techniques have been measured in terms of the percentage of incoming tasks they can schedule and the frequency of faults they can tolerate. For soft real-time systems, the techniques use an estimate of the primary fault probability and the incoming task's soft laxity to control the overlap interval between its (task) versions in order to enhance the system's performance. The efficiency of these techniques have been measured in terms of the system's utilization and the output value of tasks. For firm real-time systems, the techniques use a feedback from the system performance to estimate the execution time for incoming tasks. The efficiency of these techniques have been measured in terms of the percentage of incoming tasks that they can schedule and the percentage of scheduling tasks that meet their deadlines.

In Chapter 4, we proposed two new techniques to be used in a PB-based fault-tolerant

dynamic scheduling algorithm in hard real-time systems. The first technique is called *dynamic grouping,* in which the processors are dynamically grouped into logical groups in order to achieve efficient overloading of tasks, thereby improving the schedulability and the reliability of the system. The second technique is called *primary-backup-overloading,* in which the primary of a task can be scheduled onto the same or overlapping time interval with the backup of another task on a processor. This is done in order to increase system utilization, thereby improving the schedulability. The proposed techniques can easily be incorporated into any dynamic scheduling algorithm. The contributions of this work are:

- Development and analysis of a fault-tolerant technique (dynamic grouping) that offers significantly better guarantee ratio (15% gain) than static grouping, and offers a graceful degradation in the performance (guarantee ratio) as the number of faults increases under all the interesting conditions that we have simulated in the system.

- Development and analysis of a fault-tolerant technique (primary-backup overloading) that offers better schedulability (25% gain) than BB-overloading under all the interesting conditions that we have simulated in the system. We have also shown that the $TTSF_{PB}$ (a reliability metric) of PB-overloading is upper bounded by twice that of BB-overloading ($TTSF_{PB}$), which is a much smaller value than the $MTTF$ of the system. Hence, PB-overloading is a more effective technique than BB-overloading for many practical reliability requirements.

- Statement of conditions and rules which must be satisfied for a processor fault to be tolerated under the two techniques.

- Analysis of the time at which the multiprocessor system can tolerate a second fault.

- Comparison of the proposed techniques with existing techniques to determine the efficiency of their performance.

- Markov analysis of a uniform task model for the two techniques.

In Chapter 5, we have considered the problem of scheduling real-time tasks with PB-based fault-tolerant requirements in multiprocessor systems. We have proposed an adaptive fault-tolerant scheduling scheme for the problem. The scheme has a mechanism to control the overlap interval between the primary and backup versions of tasks in the schedule. The overlap interval is computed based on the estimated value of primary fault probability in the system and task's soft laxity. Two variants (PB-OVER-CONT and PB-OVER-SWITCH) of the adaptive scheme have been proposed and studied.

In PB-OVER-CONT, the overlap interval varies from no overlap to full overlap in a continuous manner as the fault probability varies from 0 to 1. In PB-OVER-SWITCH, the scheduler uses a threshold value of fault probability to switch from PB-CONCUR to PB-EXCL. This threshold value is adapted with the task's soft laxity. The contributions of this work are:

- A new performance metric, called schedulability-reliability (SR) index, that is used to evaluate the proposed adaptive fault-tolerant scheme.

- Development and analysis of an adaptive fault-tolerant scheme that improves the SR index of soft real-time system. Two approaches from this scheme have been studied (PB-OVER-CONT and PB-OVER-SWITCH).

- Mathematical probability studies to quantify the effect of primary fault probability on three performance metrics, *viz*, processor utilization, task value (utility), and SR index.

- Comparison of the proposed adaptive scheme with the existing non-adaptive schemes to determine the efficiency of their performance. Our studies show that the proposed PB-OVER-SWITCH adaptive scheme always performs better than its adaptive and non-adaptive counterpart in terms of SR index under all the interesting conditions that we have simulated in the system.

- Simulation experiments to study the instantaneous behavior of $SR(t)$ in response to dynamically changing fault rates. In particular, we have studied this behavior for step and ramp fault rate profiles. Our studies show that both the variants of the adaptive

scheme exhibit a similar behavior for step fault rate profile while for the ramp fault rate profile the PB-OVER-SWITCH performs better than the PB-OVER-CONT.

In Chapter 6, we have considered the problem of dynamic scheduling of firm real-time tasks in multiprocessor systems where the task execution times vary dynamically. We have proposed an open-loop scheduling algorithm, which employs a notion of task overlap that dynamically guarantees incoming firm tasks via on-line admission control and planning. Secondly, we have used the feedback control theory to design three closed-loop scheduling approaches. The loop is closed by feeding back *(i)* the deadline miss ratio in the first approach; *(ii)* the task rejection ratio in the second approach; and *(iii)* both the miss ratio and rejection ratio in the final approach. The contributions of this work are:

- A new performance metric, called effective ratio (ER), that is used to evaluate the proposed open-loop and closed-loop scheduling algorithms.
- Development and analysis an open-loop scheduling algorithm *(OL-OVER-AvCET)*, based on a concept of task overlap, for firm real-time tasks. Our studies show that the overlap based algorithm improves the effective ratio of the system by 15% over non-overlap algorithms.
- Developed and analyzed closed-loop framework algorithms using the feedback control theory.
- Development and analysis three closed-loop scheduling algorithms *(CL-OVER-MISS, CL-OVER-REJ,* and *CL-OVER-MISSREJ)* for firm real-time tasks. Our studies show that the proposed *CL-OVER-MISSREJ* algorithm performs better, in terms of the effective ratio, than *CL-OVER-MISS* or *CL-OVER-REJ*. Also, it performs better (20% gain) than the open-loop scheduling algorithm *(OL-OVER-AvCET)*. Our studies Also show that the *(OL-OVER-AvCET)* performs better, in terms of the effective ratio, than both *(CL-OVER-MISS,* and *CL-OVER-REJ* when the system load is high.

- Simulation and analytical experiments to study the instantaneous behavior of the miss ratio (MR_{kT}) and rejection ratio (RR_{kT}) in response to dynamically changing task load. In particular, we have studied this behavior for step task load. The purpose of these studies are to verify the analytical model against the simulation model.

In conclusion, the performability of existing real-time systems can be improved with minimal changes using the efficient techniques described in this thesis. The amount of redundancy can be varied based on the number and type of faults that are needed to be tolerated by changing the overloading or/and grouping techniques in hard real-time systems. The type of redundancy can be varied dynamically by changing the amount of overlapped time between the primary and the backup of tasks based on the estimated fault probability and tasks deadline in soft real-time systems. Moreover, the amount of time that is assigned for a task to be executed can also be dynamically estimated based on feedback from the deadline miss ratio and tasks rejection ratio in firm real-time systems.

BIBLIOGRAPHY

[1] T. F. Abdelzaher, and C. Lu, "Modeling and performance control of internet servers," Invited Paper,*39th IEEE Conference on Decision and Control*, Sydney, Australia, Dec. 2000.

[2] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin, "QoS negotiation in real-time systems and its application to automated flight control," *IEEE Real-Time Technology and Applications Symp.*, 9–11, June 1997.

[3] T. F. Abdelzaher, and K. G. Shin, "End-host architecture for QoS-adaptive communication," *IEEE Real-Time Technology and Applications Symp.*, Denver, CO., 121–130, June 1998.

[4] K. D. Ahn, J. Kim, and S. J. Hong, "Fault-tolerant real-time scheduling using passive replicas," *Proc of the Pacific Rim Int'l. Symp. on Fault Tolerant Systems (PRFTS)*, Taipei, Taiwan, 98–103, Dec. 1997.

[5] R. Al-Omari, G. Manimaran, and A. K. Somani, "An efficient backup-overloading for fault-tolerant scheduling of real-time tasks," *Proc. IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, 1291-1295, 2000.

[6] R. Al-Omari, A. K. Somani, and G. Manimaran, "A new fault-tolerant technique for improving schedulability in multiprocessor real-time systems," *15th Annual Int'l. Parallel and Distributed Processing Symp. (IPDPS 2001)*, Apr. 2001

[7] K.-E. *Aºrzén*, B. Bernhardsson, J. Eker, A. Cervin, K. Nilsson, P. Persson, and L. Sha, "Integrated control and scheduling", *Internal report TFRT-7582*, Department of Automatic Control, Lund University, Aug. 1999.

[8] H. Aydin, P. Mejia-Alvarez, R. Melhem, and D. Mosse, "Optimal reward-based scheduling of periodic real-time tasks," *Proc. of The Real-time System Symp. (RTSS)*, 79–89, Dec. 1999.

[9] P. R. Blevins, and C. V. Ramamoorthy, "Aspects of a dynamically adaptive operating Systems," *IEEE Trans. on Computers*, *25*(7), 713–725, July 1976.

[10] A. Burns, "Scheduling hard real-time systems: A review," *Software Engineering Journal*, *6*(3), 116–128 May 1991.

[11] G. Buttazzo, and F. Sensini, "Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments," *Proc. of the 3rd IEEE Int'l. Conference on Engineering of Complex Computer Systems (ICECCS'97)*, 39–48, Como, Italy, Sep. 1997.

[12] G. Buttazzo, G. Lipari, and L. Abeni, "Elastic task model for adaptive rate control," *Proc. of the IEEE Real-Time Systems Symp.*, Madrid, Spain, 286–295, Dec. 1998.

[13] Giorgio C. Buttazzo, " Hard real-time computing systems: Predictable scheduling algorithms and applications," *Kluwer Academic Pub.*, 1997.

[14] Giorgio Buttazzo, and Luca Abeni, "Adaptive rate control through elastic scheduling," *Proc. of the 39th IEEE Conference on Decision and Control*, Sydney, Australia, Dec. 2000.

[15] M. Caccamo, and G. Buttazzo, "Exploiting skips in periodic tasks for enhancing aperiodic responsiveness," *In Proc. of the 18th IEEE Real-Time System Symp.*, 330–339, Dec. 1997.

[16] M. Caccamo, and G. Buttazzo, "Optimal scheduling for fault- tolerant and firm real-time systems," *Proc. of the IEEE Int'l. Conference on Real-Time Computing Systems and Applications*, Hiroshima, Japan, 223–231, Oct. 1998.

[17] M. Caccamo, G. Buttazzo, and L. Sha, "Elastic feedback control," *IEEE Proc. of the 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, 121–128, June 2000.

[18] L. Chen, and A. Avizienis, "N-version programming: A fault- tolerant approach to reliability of software operation," *Symp. on Fault Tolerant Computing (FTCS8)*, 3–9, June 1978.

[19] J. Eker, and A. Cervin. "A matlab toolbox for real-time and control systems co-design," *In Proc. of the 6th Int'l. Conference on Real-Time Computing Systems and Applications*, Hong Kong, China, Dec. 1999.

[20] D. S. Fussell, and M. Malek, "Responsive computer systems: Steps toward fault-tolerant real-time systems," *Kluwer Academic Pub.*, 1995.

[21] M. R. Gareyand, and D. S. Johnson, "Computers and intractability, a guide to the theory of NP-completeness," San Francisco, W. H. Freeman Company 1979.

[22] Sunondo Ghosh, "Guaranteeing fault-tolerance through scheduling in real-time systems," Ph.D. Thesis, University of Pittsburgh, 1996.

[23] S. Ghosh, R. Melhem, and D. Mosse, "Fault-tolerance scheduling on a hard real-time multiprocessor systems," *Int'l. Parallel Processing Symp.* Apr. 1994.

[24] S. Ghosh, R. Melhem., and D. Mosse, "Fault-tolerance through scheduling of aperiodic tasks in hard real-time multiprocessor systems," *IEEE Trans. Parallel and Distributed Systems, 8*(3), 272–284, Mar. 1997.

[25] O. González, H. Shrikumar, J.A. Stankovic, K. Ramamritham, "Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling," *Proc. of the IEEE Real Time System Symp.*, 79–89, Dec. 1997.

[26] Indranil Gupta, G. Manimaran, and C. Siva Ram Murthy, "Primary-backup based fault-tolerant dynamic scheduling of object-based tasks in multiprocessor real-time systems,"

Proc. IEEE Workshop on Fault Tolerant Parallel and Distributed Systems, Puerto Rico, 83-107, Apr. 1999.

[27] K. Gustafsson , "Control theoretic techniques for stepsize selection in explicit Runge-Kutta methods," *ACM Trans. on Mathematical Software*, *17*(4), 533–554, Dec. 1991.

[28] M. Hamdaoni, and P. Ramanathan, "A dynamic priority assignment technique for streams with (m k)-firm deadlines," *IEEE Trans. Computers*, *44*(12), 1443–1451, Dec.1995.

[29] J. R. Haritsa, M. Livny, and M. J. Carey, "Earliest deadline scheduling for real-time database systems," *Proc. of the IEEE Real Time System Symp. (RTTS)*, 232–242, Dec. 1991.

[30] B. W. Johnson, "Design and analysis of fault tolerant digital systems," *Addison Wesley Pub. Co., Inc*, 1989.

[31] R. M. Kieckhafer, "Fault-tolerant real-time task scheduling in the MAFT distributed system," *22nd Hawaii Int'l. Conference on System Sciences*, 145–151, Jan. 1989.

[32] K. Kim, and J. Yoon, "Approaches to implementation of reparable distributed recovery block scheme," *Proc. IEEE Fault-tolerant Computing Symp.*, 50–55, 1988.

[33] M. H. Klein, and et.al., "A practitioner's handbook for real-time analysis: Guide to rate-monotonic analysis for real- time systems," Boston, Kluwer Academic Pub. July 1993.

[34] M. H. Klein, J. P. Lehoczky, and R. Rajkumar, "Rate-monotonic analysis for real-time industrial computing," *IEEE Computer*, *27*(1), 24–33 Jan. 1994.

[35] H. Kopetz, A. Damm, C. Koza, and Mulozzani, "Distributed fault-tolerant real-time systems: The MARS approach," *IEEE Micro*, *9*(1), 25–40, Feb. 1989.

[36] G. Koren, and D. Shasha, "Skip-over: Algorithms and complexity for overloaded systems that allow skips," *In Proc. of the IEEE Real-Time Systems Symp.*, 110–117, Dec. 1995.

[37] C. M. Krishna, and E.G. Shin, "On scheduling tasks with quick recovery from failure," *IEEE Trans. Computers*, *35*(5), 448–455, May 1986.

[38] C. M. Krishna, and E.G. Shin, "Real-time systems," *McGraw-Hill Int'l.*, 1997.

[39] C. M. Krishna, and Y.H. Lee, "Guest-editors' introduction: Real-time systems," *IEEE Computer*, *24*(5) 10–11, May 1991.

[40] T.-W. Kuo, and A. Mok, "Load adjustment in adaptive real-time systems," *IEEE Trans. on Computers*, *46*(12), 1313-1324, 1997.

[41] S. Lauzac, R. Melhem, and D. Mosse, "A comparison of global and partitioning schemes for scheduling rate monotonic tasks on a multiprocessor," *10th IEEE Euromicro Real-Time Workshop*, Berlin, Germany, 188–195, June 1998.

[42] S. Lauzac, R. Melhem, and D. Mosse, "An efficient RMS admission control and its application to multiprocessor scheduling," *Int'l. Parallel Processing Symposium - IPPS*, Orlando, FL., 511–518, Mar. 1998.

[43] E. L. Lawler, and C.U. Martel, "Scheduling periodically occurring tasks on multiprocessors," *Information Processing Letters*, *12*(1), 9–12, Feb. 1981.

[44] C. Lee, R. Rajkumar, and C. Mercer, "Experiences with processor reservation and dynamic QoS in real-time Mach," *In Proc. of Multimedia*, Japan, Apr. 1996.

[45] Baochun Li, and Klara Nahrstedt, "A control theoretical model for quality of service adaptations," *IEEE Int'l. Workshop on Quality of Service (IWQoS 98)*, 145–153, May 1998.

[46] F. Liberato, S. Lauzac, R. Melhem, and D. Mosse, "Global fault tolerant real-time scheduling on multiprocessors," *Proc. of The 10th IEEE Euromicro Real-Time Workshop*, York, UK, June 1999.

[47] A. L. Liestman, and R. H. Campbell, "A fault tolerant scheduling problem," *IEEE Trans. Software Engineering*, *12*(11), 1089–1095, Nov. 1986.

[48] C. Liu, and J. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of ACM*, *20*(1), 45–61, Jan. 1973.

[49] J. W. S. Liu, K. J. Lin, W. E. Shih, A. C. Yu, J. Y. Chung, and W. Zhao, "Algorithms for scheduling imprecise computations," *IEEE Computer*,*24*(5), 58–68, May 1991.

[50] X. Liu, L. Sha, M. Caccamo, and G. Buttazzo "Online control optimization using load driven scheduling," *Proc. of the 39th IEEE Conference on Decision and Control*, Sydney, Australia, Dec. 2000.

[51] C. Lu, J. A. Stankovic, G. Tao, and S. H. Son, "Design and evaluation of feedback control EDF scheduling algorithm," *Real-Time Systems Symp. (RTSS)*, 56–67 Dec. 1999.

[52] C. Lu, J. A. Stankovic, T. Abdelzaher, G. Tao, S. Son, and M. Marley, "Performance specifications and metrics for adaptive real-time systems," *Real-Time Systems Symp. (RTSS)*, Dec. 2000.

[53] "Lynx programmer's reference manual, version 2.4," *Lynx Real-time Systems*, San Jose, CA, 1996.

[54] L. V. Mancini, "Modular redundancy in a message passing system," *IEEE Trans. on Software Engineering*, 12(1), 79–86, Jan. 1986.

[55] G. Manimaran, "Resource management with dynamic scheduling in parallel and distributed real-time systems," Ph.D Thesis, Indian Institute of Technology, Madras, India, July 1998.

[56] G. Manimaran, and C. Siva Ram Murthy, "An efficient dynamic scheduling algorithm for multiprocessor real-time systems," *IEEE Trans. on Parallel and Distributed Systems*, *9*(3), 312–319, Mar. 1998.

[57] G. Manimaran, and C. Siva Ram Murthy, "A fault-tolerant dynamic scheduling algorithm for multiprocessor real-time systems and its analysis," *IEEE Trans. on Parallel and Distributed Systems*, *9*(11), 1137–1152, Nov. 1998.

[58] G. Manimaran, C. Siva Ram Murthy, Machiraju Vijay, and K. Ramamritham, "New algorithms for resource reclaiming from precedence constrained tasks in multiprocessor

real-time systems," *Journal of Parallel and Distributed Computing, 44*(2), 123–132, Aug. 1997.

[59] Pedro Mejea-Alvarez, and Daniel Mosse, "A responsiveness approach for scheduling fault recovery in real-time systems," *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium (RTAS)*, Vancouver, Canada, 1–10, June 1999.

[60] G. Miremadi, and J. Torin, "Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection," *IEEE Trans. on Reliability, 44*(3), 441–453, Sep. 1995.

[61] T. Nakajima, and H.Tezuka, "A continuous media application supporting dynamic QoS control on real-time Mach," *In Proc. of ACM Multimedia*, 289–297, 1994.

[62] T. Nakajima, "Resource reservation for adaptive QoS mapping in real-time Mach," *In Proc. of the 6th Int'l. Workshop on Parallel and Distributed Real-Time Systems*, 1047–1056, Apr. 1998.

[63] Y. Oh, and S.H. Son, "Multiprocessor support for real-time fault-tolerant scheduling," *IEEE Workshop on Architectural Aspects of Real-time Systems*, 76–80, Dec. 1991.

[64] L. Palopoli, L. Abeni, G. Buttazzo, F. Conticelli, and M. Di Natale, "Real-time control system analysis: An integrated approach," *Proc. of the IEEE Real-Time Systems Symp.*, Orlando, Florida, Dec. 2000.

[65] Patrik Persson, and Görel Hedin, "Int'l. active execution time predictions using reference attributed grammars," *Proceedings of 2nd Workshop on Attribute Grammars and their Applications (WAGA'99)*, Amsterdam, Netherlands, 173–184, Mar. 1999.

[66] J. H. Purtilo, and Pankaj Jalote, "An environment for developing fault-tolerant software," *IEEE Trans. on Software Engineering, 17*(2), 153–159, Feb. 1995.

[67] P. Ramanathan, "Graceful degradation in real-time control applications using (m,k)-firm guarantee," *Fault-tolerant Computing Symp.*, 132–141, June 1997.

[68] K. Ramamritham, "Allocation and scheduling of precedence- related periodic tasks," *IEEE Trans. on Parallel and Distributed Systems*, *6*(4), 412–420, Apr. 1995.

[69] K. Ramamritham, "Dynamic priority scheduling in real-time systems," *Real-time Systems - Specification, Verification, and Analysis (Mathai Joseph, ed.)*, Prentice Hall, 66–96, 1996.

[70] K. Ramamritham, and J. A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proc. of the IEEE*, *82*(1), 55–67 Jan. 1994.

[71] K. Ramamritham, J.A. Stankovic, and P.F. Shiah, "Efficient scheduling algorithms for real-time multiprocessor systems," *IEEE Trans. on Parallel and Distributed Systems*, *1*(2), 184–194, Apr. 1990.

[72] B. Randell, "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, *SE-1*(2), 220–232, June 1975.

[73] M. Ryu, and S. Hong, "Toward automatic synthesis of schedulable real-time controllers," *Integrated Computer-Aided Engineering*, *5*(3) 261–277, 1998.

[74] M. Saksena, J. da Silva, and A. K. Agrawala, "Design and implementation of Maruti-II," sang son ed., *Principles of Real-Time Systems*, Prentice Hall, 1994.

[75] D. Seto, et al., "On task schedulability in real-time control systems," *IEEE Real-Time Systems Symp.*, 13–21, Dec. 1996.

[76] C. Shen, K. Ramamritham and J.A. Stankovic, "Resource Reclaiming in Multiprocessor Real-Time Systems," *IEEE Trans. on Parallel and Distributed Systems*, *4*(4), 382–397, Apr. 1993.

[77] K. G. Shin, "HARTS: A distributed real-time architecture," *IEEE Computer*, *24*(5), 25–35, May 1991.

[78] K. G. Shin, and C. L. Meissner, "Adaptation and graceful degradation of control system performance by task reallocation and period adjustment," *11th EuroMicro Conference on Real-Time Systems*, 29–36, June 1999.

[79] K. G. Shin, and P. Ramanathan, "Real-time computing: A new discipline of computer science and engineering," *Proc. of IEEE Computer*, *82*(1), 6–24, Jan. 1994.

[80] A. K. Somani, and N. H. Vaidya, "Understanding fault-tolerance and reliability," *IEEE Computer*, *30*(4), 45–50, Apr. 1997.

[81] J. A. Stankovic, and et al., "Implications of classical scheduling results for real-time systems," *IEEE Computer*, 16–25, June 1995.

[82] J. A. Stankovic, and K. Ramamritham, "The Spring kernel: A new paradigm for real-time systems," *IEEE Software*, *8*(3), 62–72, May 1991.

[83] J. A. Stankovic, C. Lu, S. Son, and G. Tao, "The case for feedback control real-time scheduling," *EuroMicro Conference on Real-Time Systems*, 11–20, June 1999.

[84] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *IEEE Computer*, *21*(10), 10–19, Oct. 1988.

[85] David C. Steere, et al., "A feedback-driven proportion allocator for real-rate scheduling," *Operating Systems Design and Implementation (OSDI)*, 145–158, Feb. 1999.

[86] N. Suri, C.J. Walter, and M.M. Hugue, "Advances in ultra-dependable distributed systems," *IEEE Computer Society Press*, 1995.

[87] H. Tokuda, T. Nakajima, and P. Rao, "Real-time Mach: Toward a predictable real-time system," *Proc. of USENIX Mach Workshop*, 73–82, Oct. 1990.

[88] S. Tridandapani, A. K. Somani, and U. Reddy, "Low overhead multiprocessor allocation strategies exploiting system spare capacity for fault detection and location, " *IEEE Trans. on Computers*, *44*(7), 865–877, July 1995.

[89] T. Tsuchiya, Y. Kakauda, and T. Kikuno, "A new fault-tolerant scheduling technique for real-time multiprocessor system," *2nd Int'l. Conference on Real-Time Computing Systems and Applications*, 197–202, 1995.

[90] T. Tsuchiya, Y. Kakuda, and T. Kikuno, "Fault-tolerant scheduling algorithm for distributed real-time systems," *Proc. IEEE Workshop on Parallel and Distributed Real-time Systems*, 99–103, Apr. 1995.

[91] F. Wang, K. Ramamritham, and J. A. Stankovic, "Determining redundancy levels for fault tolerant real-time systems," *Special Issue of IEEE Trans. on Computers on Fault Tolerant Computing*, *44*(2), 292–301, Feb. 1995.

[92] C. M. Woodside, and D. W. Craig, "Local non-preemptive scheduling policies for hard real-time distributed systems," *Real-Time Systems Symp.*, 12–16, Dec. 1987.

[93] K. Yu, and I. Koren, "Reliability enhancement of real-time multiprocessor systems through dynamic reconfiguration," *Annual IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems*, IEEE Computer Society Press, Los Alamitos, CA, 161–168, 1995.

[94] W. Zhao, K. Ramamritham, and J. A. Stankovic, "Scheduling tasks with resource requirements in a hard real-time system," *IEEE Trans. on Software Engineering*, *SE-13*(5), 564–577, May 1987.

[95] Jianwei Zhou, G. Manimaran, and Arun K. Somani, "A dynamic scheduling algorithm for improving performance index in multiprocessor real-time systems," *In Proc. Int'l. Conference on Advanced Computing*, Roorkee, India, Dec. 1999.

[96] Hengming Zou, and Farnam Jahanian, "A Real-time primary-backup replication service," *IEEE Trans. on Parallel and Distributed Systems*, *10*(6), June 1999.

ACKNOWLEDGEMENTS

First and foremost I would like to express my gratitude to my supervisor, Prof. Arun K. Somani, whose expertise, understanding, and patience, added considerably to this research work. I appreciate his gentle encouragement, constructive criticism, and for training me to be a successful researcher. I really appreciate the long hours he spent reading and correcting the papers and the reports that I gave to him even when he was traveling or at home. It was a pleasure to work with him from both the intellectual and personal points of view. His confidence in me has increased my confidence in myself.

I wish to express my sincere gratitude to Dr. Govindarasu Manimaran for his help which also enabled me to pursue my research in the field of real-time systems. I have deep regards for the advice and direction that he gave me. I could walk into his office at any time with a question or a new idea. I really appreciate the long hours that he spent with me working on papers. His encouragement and respect to my ideas were the reasons for starting this research work.

I would like to sincerely thanks Dr. Murti V. Salapaka for his interest in the subject of the research work. His interest, guidance, and knowledge in the area of control theory have qualified part of this research work. Thanks are also extended to other members of the committee, Dr. Doug Jacobson, Dr. Sigurdur Olafsson, and Prof. Suraj C. Kothari for the time and effort that they provided.

The people most responsible for my success at higher studies are my parents. They made me understand the importance of education, and guided me through the initial years of schooling. I also take this opportunity to thanks my parents in law for making my life in the U.S., so far away from home, an enjoyable experience. I would like to thank my wife, Maisaa Hawana, for

the support, help, and encouragement that she gave to me, even through sick days.

I have gained many friends during my life at ISU. I sincerely acknowledge all of them for their encouragement and help. It was fun to interact with all my office mates and several other students in the department. Finally, I thank God (SWT) the Almighty for his mercy, without which I would never have been able to reach this stage.