Summer 1996

Scalability in Real-Time Systems

Ramesh Yerraballi
Old Dominion University

SCALABILITY IN REAL-TIME SYSTEMS

by

RAMESH YERRABALLI

B.E June 1991, Osmania University

Computer Science Department, College of Engineering

A Thesis (or Dissertation) submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

COMPUTER SCIENCE

OLD DOMINION UNIVERSITY

August 1996

Approved by:

Supervisor: ______________________
(Dr. Ravi Mukkamala)

____
(Dr. Kurt J. Maly)

____
(Dr. Hussein Abdel-Wahab)

____
(Dr. Larry W. Wilson)

____
(Dr. John W. Stoughton)

SCALABILITY IN REAL-TIME SYSTEMS

Ramesh Yerraballi, Ph.D.
The Old Dominion University, 1996

Supervisor: Ravi Mukkamala

The number and complexity of applications that run in real-time environments have posed demanding requirements on the part of the real-time system designer. It has now become important to accommodate the application complexity at early stages of the design cycle. Further, the stringent demands to guarantee task deadlines (particularly in a hard real-time environment, which is the assumed environment in this thesis) have motivated both practioners and researchers to look at ways to analyze systems prior to run-time. This thesis reports a new perspective to analyzing real-time systems that in addition to ascertaining the ability of a system to meet task deadlines also qualifies these guarantees. The guarantees are qualified by a measure (called the scaling factor) of the systems ability to continue to provide these guarantees under possible changes to the tasks. This measure is shown to have many applications in the design (task execution time estimation), development (portability and fault tolerance) and maintenance (scalability) of real-time systems. The measure is shown to bear relevance in both uniprocessor and distributed (more generally referred to as end-to-end) real-time systems.

However, the derivation of this measure in end-to-end systems requires that we solve a fundamental (very important, yet unsolved) problem—the end-to-end schedulability problem. The thesis reports a solution to the end-to-end schedulability problem which is based on a solution to another fundamental problem relevant to single-component real-time systems (a uniprocessor system is a special instance of such a system). The problem of interest here is the schedulability of a set of tasks with arbitrary arrival times, that run on a single component. The thesis presents an optimal solution to this problem. One important consequence of this result (besides serving as a basis for the end-to-end schedulability problem) is its applicability to the classical approach to real-time scheduling, viz., static scheduling. The final contribution of the thesis comes as an application of the results to the area of real-time communication. More specifically, we report a heuristic approach to the problem of admission control in real-time traffic networks. The heuristic is based on the scaling factor measure.

To my Parents

Acknowledgements

First and foremost I owe this thesis to the part of me that persisted inspite of the frustrations of pursuing a seemingly never ending goal, that is a PhD thesis. I'd like to thank my advisor Dr. Ravi Mukkamala for believing in my abilities and constantly reminding me of what little was left for me to finish my thesis. Though it was never "little", I am glad I took his advice. I would like to acknowledge the financial support I received from NASA Langley Research Center for pursuing my thesis. I thank Mr. Wayne H. Bryant, Assitant Division Chief, Flight Electronics Technology Division, NASA LaRC, for approving and funding my thesis proposal under the grant NAG-1-1114.

I would like to thank my committee – Dr. Kurt Maly, Dr. Hussein Abdel-Wahab, Dr. Larry Wilson and Dr. John Stoughton for their support and approval of my work. Both Dr. Maly and Dr. Wahab have tolerantly guided me through the preliminary stages of my PhD. I'd like to acknowledge Dr. Stoughton's valuable comments on the final thesis. Among other faculty, Dr. Stephan Olariu and Dr. Chester Grosch have contributed significantly in making my stay at ODU academically worthwhile. I'd also like to acknowledge the arrival of Sameera (who has since become my wife) into my life in August of 1994 which also overlapped with my finding most of the results reported in this thesis. In a sense, this presents a case against the popular french saying "The first sigh of love is the last sign of wisdom". I would like to thank all my colleagues in the Computer Science department for giving me company through the travails of graduate life. In particular I'd like to thank Dharmavani for her prodding me not to quit my PhD. Lastly, I'd like to thank my high school tutor, Mr. Gopalan, to whom I owe dearly for my academic achievements. He built in me a fascination for logical reasoning and thought.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The scope of real-time systems has expanded over the last two decades to encompass a wide array of applications such as industrial process control systems, nuclear power plants, air traffic control systems, aircraft navigation, robot navigation and automobile control. While, in the past these systems were predominantly centralized, most current approaches tend to be distributed in nature. Further, the complexity of these systems (in addition to that added by its distributed nature) has grown rapidly to a point where the dependability (or determinism) of the system as a whole has become an important issue. Real-time systems are primarily categorized into two types, hard real-time systems and soft real-time systems. In *hard real-time* systems, the missing of task deadlines can lead to severe consequences and hence there is a strict need to meet these deadlines. In contrast, *soft real-time* systems are characterized by the fact that they can tolerate temporary deadline misses. Soft real-time systems continue to operate even after missing deadlines, and the only consequence being a temporary decline in performance and an increase in response time. For example, a robot operating in a hazardous terrain would be a hard real-time system and a system that periodically generates a weather report can be considered a soft real-time system.

The stringent need to meet deadlines in hard real-time systems implies that there is a need to analyze the system pre-runtime, to ascertain its ability to guarantee performance (that is meeting deadlines). Therefore, this has motivated enormous efforts from practioners to investigate the system behavior prior to its actual installation. In other words, though the system is said to function in real-time, the guarantees it provides in meeting the timing requirements of the various tasks have to be ascertained a priori. This thesis presents issues and finds solutions that we believe will aid practioners in guaranteeing system behavior prior to run-time in hard real-time systems. The issues in soft real-time systems overlap significantly with those in conventional systems where the primary performance metrics are throughput and response time (an average measure unlike deadline that is an absolute measure). These systems have been well-studied and the results (pertinent to these systems) are directly applicable to soft real-time systems.

A real-time system can be characterized by two important components: the environment in which the system is operating and the computer system that controls/monitors the environment. The main issues in the design of the first component concern interfacing with the environment [41]. Solutions in this area are primarily dictated by the technology. There are many issues of concern in the design of the second component, the computer system. The computer system involves both the hardware and the software that runs on them. The choice of hardware is dictated primarily by such parameters as cost, availability and the application at hand. The primary issue in software design is not so much the particular choice of language or programming paradigm as it is the mechanism by which the various tasks are scheduled.

1.1 Issues in Real-Time Systems

We observe that only the last issue (mentioned above) can be speculated over because the others are more or less dictated by the environment and the application at hand. This is the reason why we have efforts from numerous researchers [40, 48] on the problem of scheduling in real-time systems. There have been two important fronts of research: On one front there have been efforts [2, 20, 22, 24] to find scheduling mechanisms that could guarantee performance under different assumptions about the system. On a second front, researchers [46, 19, 3] have tried to answer questions posed by schedulability analysis. Both these are inter-related in the sense that schedulability analysis is a mechanism to evaluate the effectiveness of a scheduler. To this end, in the following discussion when we refer to schedulability we implicitly assume that the tasks are being scheduled by an arbitrary scheduler (where appropriate, we give a more detailed description of the scheduler assumed). If a scheduler is built on a strong theoretical basis then its schedulability analysis can be a trivial comparison. For example, a dynamic scheduling mechanism, the earliest deadline first (EDF) scheduler, has the theoretical property that, provided the sum of the utilizations[1] of the tasks in a task-set is less than 1, the EDF scheduler guarantees to meet their deadlines (schedulable). Clearly, in this case the schedulability test is a simple one. There are other cases where the schedulability test is non-trivial [19, 50].

A common assumption that distinguishes one scheduling mechanism (and thus the corresponding schedulability analysis) from another is the oper-

[1]The utilization of a periodic task is given by the ratio of its execution time requirement to its periodicity

ational environment of the real-time system. If the environment is completely known a priori, then we can use a static approach to design a scheduler. The schedule can be practically pre-computed (as a list), with the scheduler being a simple mechanism to pick the next task in the list. On the other hand, if the environment is dynamic by nature and no a priori knowledge can be assumed about the environment then the scheduling mechanism must be dynamic, adapting to the changing needs of the system. Clearly, a dynamic scheduler is more expensive (in terms of overhead) to implement compared to a static scheduler.

The use of dynamic approaches are perfectly justified in systems where the various internal (system) and external (environment) tasks characteristics are not known a priori [40]. However, we observed that such systems are far outnumbered by those where the environment is well understood, deterministic (in the sense that the worst possible scenarios can be identified), and with tasks whose timing, resource, communication and other requirements are known a priori [49]. This thesis addresses a host of related problems that concentrate on such static environments.

1.2 Issues Addressed in this Thesis

The problems of interest to us in this study are motivated by the evolutionary nature of real-time system software. As real-time systems continue to grow in size and scope there is a need to build portable standard software that would be guaranteed to operate correctly both in the logical and the temporal sense. By correctness in the logical sense, we are referring to the domain of proving the correctness of a piece of software with regards to generating a correct output for any given input (The traditional program correctness issues). The emphasis of research in real-time systems has been not so much to prove logical correctness as it has been to show that the output is produced in a *timely* manner. Therefore, a logically valid output generated beyond a specified time limit is deemed incorrect. In this thesis, we concentrate primarily on the temporal correctness.

This notion of correctness (temporal that is) of a task in real-time systems has been captured by the concept of schedulability [46] of tasks. A task is primarily characterized by the following parameters: the arrival time, the execution time, the periodicity and the deadline. Schedulability analysis therefore attempts to ascertain whether or not each task will be able to complete its required execution before its deadline for all its instances when scheduled by an assumed scheduler. Tasks being periodic, they occur repeatedly at an interval given by their period. Various such occurrences of tasks are referred to as instances. The basic approach taken in schedulability analysis is to use the information about tasks' arrival times, execution times and periodicities and compute their worst-case completion times assuming that they are scheduled by a given scheduler. The worst-case task completion times so computed are compared against their deadlines to determine if the tasks will be schedulable. Therefore, the worst-case completion time computation is the essence of schedulability analysis.

We are not interested in deriving new schedulability tests but rather in extending the guarantees made by schedulability analysis as a system undergoes changes. The types of changes we are mainly interested in, manifest themselves as changes in execution times of tasks. In Chapter 3 we discuss sources of such changes pertinent to the design, development and maintenance of real-time systems. More specifically, we are interested in the effects on the guarantees made by schedulability analysis when some or all of the tasks' execution times are scaled (up or down)[2]. We refer to this problem as scalability of real-time systems. There are two important scenarios in which the factor has relevance: (i) Uniprocessor systems and (ii) end-to-end systems.

The problem of scalability in uniprocessor systems can be informally defined as follows:

> Given a task-set T, determine the maximum scaling factor with which a subset (S) of these task-set's execution times can be scaled without affecting the schedulability of the task-set.

If a task-set is not-schedulable[3] to start with then scaling a subset of the task will in no way improve the situation and this case is of no interest to us. On the other hand, if a task-set is schedulable to start with, then we are guaranteed the existence of a scaling factor (possibly 1, in the case that the task-set requirements are tight) that does not affect the task-set's schedulability. The first step therefore is to find whether the given task-set is schedulable. In the context of uniprocessor systems, Lehoczky's [19] schedulability test can be used for this purpose. Finding the scaling factor now can be viewed as extending this schedulability test to accommodate for changes in task execution times. There are two possible approaches here: (i) using an approximation

[2]Scaling down of task execution times can be trivially handled, therefore from here onwards when we refer to scaling we mean scaling up

[3]That is, at least one of the tasks misses its deadline when scheduled by the assumed scheduler.

technique by making small increments to the scaling factor (starting from 1) and repeatedly performing the schedulability test, or (ii) embedding the scaling factor computation into the schedulability test. We have taken the second approach for performance reasons that will be described in detail in chapter 4.

As opposed to uniprocessor systems where we have a single schedulable resource, end-to-end systems (e.g., a distributed system) have more than one schedulable resource. Therefore an end-to-end system can be characterized by tasks that do not necessarily execute on a single component[4]. Typically, a task would comprise of a sequence[5] of sub-tasks that each execute on a different component (e.g., processors, network) in the system. The requirements of period, deadline and arrival time are specified for the task as a whole with the execution times being specified at the sub-task level. The problem of finding the schedulability (worst-case completion time computation) of a task (T_i) in such a scenario can be reduced to solving the schedulability of the m (number of sub-tasks in task T_i) sub-tasks in turn, provided we are able to compute the characteristics (period and arrival time) of the sub-tasks (T_{ik} $1 \leq i \leq n$; $1 \leq k \leq m$). For reasons that will become clear in chapter 3, we cannot use Lehoczky's schedulability test for the sub-tasks running on these individual components.

The scalability problem in the context of end-to-end systems takes two forms depending on whether we view the scaling to occur as a result of a change in one or more of the components or a change to a subset of the sub-tasks. Solving either of these two forms requires that we first find whether the

[4]We use the term component to indicate any schedulable entity in the system.

[5]The treatment in this study is restricted to sequential tasks, however, it can be extended to more complex tasks.

given task-set (of end-to-end tasks) is schedulable to start with (we call this end-to-end schedulability). Secondly, we have to extend this schedulability test to accommodate component and/or task changes.

We have investigated the applicability of the scalability problem in other areas of real-time systems. Particularly, in the area of real-time communication. The application of interest to us is admission control in real-time (RT) channels [9, 8]. The role of real-time channels in communication is analogous to end-to-end tasks in distributed systems. Admission control poses the question: "Having guaranteed the performance requirements of $n-1$ real-time channels, is it possible to admit a new real-time channel, while continuing to honor the guarantees already made?" The problem of admission control is analogous to: "Given a schedulable task-set of $n-1$ end-to-end tasks, is it possible to accommodate a new task without violating the schedulability of the $n-1$ prior tasks?"

1.3 Summary of Results

The primary contribution of this thesis to the area of real-time systems is in presenting solutions to the following two fundamental problems related to schedulability analysis. The first of these problems involves schedulability analysis of task-sets where tasks have non-zero arbitrary arrival times. The second involves extending schedulability analysis to accommodate scaling up of task execution times. The impact these problems (and their solutions) have on the current state-of-the-art of real-time system research can be summarized as follows:

- Helps real-time system designers in doing a precise analysis of task-sets. Such a precise analysis, as opposed to the pessimistic analysis approach that was popularized by the RMA [6] (Rate Monotonic Approach) group at SEI helps prevent under-utilization of system resources.

- The thesis identifies many important issues in real-time systems that motivate the need for using the arrival time information of tasks in schedulability analysis. Prominently, the issues of data and resource sharing among tasks, precedence constraints between tasks, controlling task jitter can be addressed naturally by the use of task arrival times.

- The use of static schedules was popular in practice in real-time systems till the late 70s. The approach however, suffered from the inability to guarantee task schedulability a priori as opposed to RMA, which was based on the critical instant argument. As a by-product of doing a schedulability analysis of task-sets with arrival times (reported here), we are able to build static schedules whose ability to guarantee task schedulability can be ascertained a priori.

- There is no known schedulability analysis approach in the context of distributed real-time systems (or more generally end-to-end real-time systems). Using the single-component schedulability analysis of tasks with arbitrary arrivals, we are able to perform an end-to-end schedulability analysis.

- The thesis reports the first effort in addressing the issues of scalability and portability in real-time systems.

- The scaling problem is shown to help address issues of concern to designers in the design, development and maintenance real-time systems. In the design phase it allows us in analyzing the task-set by assuming an arbitrary target environment which can be later adapted to a specific target environment. In the development phase it allows us to add new tasks or enhance the existing task's functionality. In the maintenance phase it helps address the ability of the system to tolerate faults.

- The scalability problem is also solved in the context of distributed systems.

- Lastly, we report a heuristic approach to the problem of admission control in real-time traffic networks. The heuristic used is based on the study of the scaling factor problem.

1.4 Organization of the Thesis

The rest of the chapters of the thesis are organized as follows. Chapter 2 lays down the framework and terminology used through the rest of the paper. We describe the uniprocessor system model and task characteristics of interest to us. The special sense attributed to the arrival time parameter leads to the consideration of dependent and independent task-sets. The end-to-end system model is defined both in a restricted flow-shop sense and also a more generalized sense. Finally, the real-time channel model used in the study of admission control in real-time traffic networks is described.

In Chapter 3, we give a brief discussion on some theoretical background in scheduling that is pertinent to this thesis. In particular we discuss the work of Lehoczky in the context of schedulability analysis of fixed priority schedulers. The use of the critical instant argument and its consequences in both uniprocessor and end-to-end systems is critiqued. We also discuss the limited work reported in the areas of end-to-end scheduling and admission control.

In Chapter 4, the problems of interest in this thesis are formally stated and their solutions are shown to reduce to solving three fundamental problems that are the subject of the next four chapters. Chapter 5 presents the problem of uniprocessor scalability. A pre-requisite to solving the end-to-end scalability problem is the end-to-end schedulability problem which is the subject of Chapter 6. Chapter 7 considers the end-to-end scalability problem from two different perspectives viz., component change and task change.

The problem of admission control of real-time channels is the subject of Chapter 8. Here, we discuss a simulation study to compare two heuristics to solve the admission control problem.

Finally in Chapter 9, we describe a detailed example that puts the reported results in perspective and also concludes this thesis. The chosen example is derived from the case study of the "Olympus Attitude and Orbital Control System"(AOCS). This case study was performed by Alan Burns and his colleagues at University of York in association with British Aerospace Space Systems Ltd. for ESTEC.

Chapter 2

System Model

In this chapter, we introduce the modeling assumptions and establish the notation and terminology used in the rest of the thesis. We identify three models relevant to the thesis viz., uniprocessor system model, end-to-end system model and real-time channel model.

2.1 Uniprocessor System Model

The uniprocessor system model is characterized by the fact that there is only one allocatable component in the system, viz., the processor. More generally, this model can be referred to as "single component model."[1] The role of the processor is to monitor/control the target environment. For example, if the environment is that of a chemical experiment, then the processor interacts with the environment through sensors and actuators. The sensors serve to convey the current information about the experiment as inputs to the processor. These inputs together with locally (local to the processor) maintained state information capture the state of the experiment. The processor performs predetermined

[1]The term component is used to refer to any independently schedulable resource. Examples include, processors, communication medium, input/output processors,disk storage etc.

operations on these inputs (along with the information) and generates outputs that are then conveyed to the experiment through the actuators. Therefore, the interaction of the processor with the environment in which it operates can be captured by the inputs and outputs.

The operations which process the inputs to compute the outputs are contained in the *tasks*. In addition, to tasks that operate on the external inputs, we can also have tasks that are triggered solely by internal events or timed events. The operation of the complete system can be captured by specifying the characteristics of its tasks. There is one distinguishing characteristic about tasks that affect the complexity of the system, viz., task dependence. We therefore identify the following two cases separately. The following description applies for both these scenarios:

Here, n independent tasks, $\{T_1, T_2, \ldots, T_n\}$, capture the activity performed on a processor. Each task T_i (i is called the identifier of the task T_i) is characterized by the following parameters:

- e_i: The execution time requirement of a task. Note that if we look at the model as a "single component model" then this parameter could mean the service time requirement of the task from the component in question.

- a_i: The arrival time of the first instance of a task. This parameter is also referred to as the offset of the task. Given a task-set T we can assume that the task that is the earliest to arrive (say a_{min}) does so at time $t = 0$ ($a_{min} = 0$). Therefore all other task arrival times are relative to this reference.

- p_i: The periodicity of a task. Consistent with the assumptions of researchers in real-time systems, we assume that tasks are of a periodic nature. This parameter implies that a task would be ready for execution every p_i units of time. We refer to successive occurrences of a task as its instances or jobs. Therefore the j^{th} instance of task T_i will be referred to as T_i^j. As opposed to periodic tasks, aperiodic tasks are characterized by the fact that they are not strictly periodic. However, the minimum inter-arrival time between successive occurrences of an aperiodic task is assumed to be known. Note that in case the task is an aperiodic task we treat this parameter (p_i) to be the minimum inter-arrival time between the task's successive instances.

- d_i: The deadline of a task. Every instance of a task is required to complete its execution before the task deadline. Therefore, if the first instance of a task T_i arrives at time $t = 0$ then its deadline is at time $t = d_i$. Subsequently, the j^{th} instance will arrive at time $t = a_i + (j - 1) \times p_i$ and will have its deadline at time $t = a_i + (j - 1) \times p_i + d_i$. Throughout the study, we assume this parameter of a task to be less than or equal to its period. In other words, the completion of a task's instance can be delayed at most till its next instance arrival. In this study we assume this to be a hard deadline. This assumption can be justified as follows: The problems we are interested in, involve schedulability analysis which is typically done offline and before the actual system is built. If the offline analysis would show that a task's deadline cannot be met, then the factors that the analysis failed to account for (compared to the real system) would make the task's chances of meeting its deadline only worse.

Therefore it would seem only logical to assume the deadline to be a hard deadline.

- Pr_i: The relative priority of the task in the system. We assume that every task has a priority assigned to it. The priority could be dictated either by the scheduler (e.g., the rate monotonic scheduler assigns priorities to tasks based on their periods) or by the inherent importance of the task relative to other tasks in the system. Unless specified otherwise, we assume that the tasks are ordered in the non-increasing order of their priorities. A simple transformation can convert this non-increasing order to a strictly decreasing order. For example consider a task-set, T containing 5 tasks with priorities, $Pr_1 = 9, Pr_2 = 8, Pr_3 = 8, Pr_4 = 4, Pr_5 = 2$. Tasks T_2 and T_3 have the same priority. Since equal priorities are arbitrarily broken, we can reassign T_3's priority, (say to 6) to be smaller than T_2's (we use task identifiers to break conflicts between tasks). Note that if Pr_5 was equal to 7 and the priorities had to be integers then we cannot assign a new priority to T_3. In such a case we can reassign new priorities to T_4 and T_5 in order to make room for T_3. In other words, the transformation guarantees that the first task T_1 is the highest priority task and the priority of task T_j is greater than T_i if and only if $j < i$.

- W_i: The worst-case response time. This is also referred to as the worst-case completion time of task T_i. This term gives the worst possible time elapsed between an instance of the task T_i's arrival and its corresponding completion. Clearly, if the response time of the j^{th} instance of the task T_i was W_i^j then, W_i is given by the maximum $W_i^j \ \forall j$.

The characteristic that distinguishes the two scenarios of independent and dependent tasks arise from assumptions about the arrival time parameter.

2.1.1 Systems with Independent Tasks

The arrival time a_i is the arrival of the first instance of a task. Task independence is primarily captured by assuming that the arrival times of tasks do not have any interdependence. Therefore leading to the assumption that the arrival times of all tasks are equal to zero. This assumption has a significant impact on the study of task schedulability. It allows us to use the critical instant argument. The critical instant argument is used in finding the schedulability of the i'th task among n tasks scheduled by a fixed priority scheduler. It can be briefly summarized as follows:

> A task T_i suffers its worst-case completion time (or response time) when its arrival coincides with the arrival of every other higher priority task T_j $(i < j \leq 1)$. Such an arrival is called a critical instant for the task T_i.

It is important to understand that the occurrence of the critical instant for a task T_i is not mandatory, in the sense that given a task-set (of tasks with arbitrary arrivals) a task is not guaranteed to encounter its critical instant. To this end, we assume that the arrival times of tasks are given to be zero, thus forcing the occurrence of the critical instant. Therefore, the critical instant argument is sometimes referred to as the critical instant assumption.

2.1.2 Systems with Dependent Tasks

The case for considering task dependence has been addressed by many researchers in different contexts [49]. Krithi Ramamrithm, in his discussion [41] on the complex nature of real-time environments states that, task interdependence contributes significantly to the complexity. Alan Burns makes similar observations in the context of the case study on the Orbital Control System [5]. Here, we briefly list some situations that impose task dependence. We also identify how these different situations can be addressed by incorporating the offset (arrival time) parameter defined in the previous subsection.

- Data and Resource Sharing: It is important to regulate the accesses of multiple tasks to a shared data item or resource. A costly solution to this problem is to implement a concurrency control mechanism (such as the priority ceiling protocol [33]). As an alternative to using a concurrency control mechanism, we observe that by inhibiting two or more tasks from accessing a resource simultaneously we can regulate their access [45]. Such an inhibition can be achieved by deriving suitable arrival times (offsets) for tasks. For example, if two tasks, T_i and T_j, access a common resource (or data item) then with the knowledge about their expected duration of use of this shared resource one can arrive at their relative arrival times. These arrival times can be computed such that the request by T_j always follows the release by T_i. In other words, we can impose constraints on the tasks to the effect that their accesses to the shared entity are ordered. This situation can be described as an exclusion constraint that was solved by imposing a precedence order on the tasks.

- Precedence Constraint: If the tasks inherently possess a precedence constraint, then it would directly manifest itself as an offset in each task. For example, if the partial results (outputs) generated by a task T_i are used (as inputs) by a second task T_j, then we are forced to impose the condition that the task T_j will be ready to execute only after T_i completes. Therefore, there is an inherent precedence constraint on T_j. The conveyance of these partial results can be done either through shared memory or through communication. Thus, inter-task communication can also impose precedence constraints.

- Controlling Task Jitter: The irregularity in the response times (different instances) of a task T_i can hurt the schedulability of tasks that depend upon its output [27]. This entails an output jitter bounded (from above) by the difference of the worst-case response time and the task's execution time. The output jitter of a given task T_i can be reduced by dividing it into two tasks T_j and T_k. T_j performs the bulk of the execution and writes the results to a buffer shared by T_j and T_k; T_k is released at an offset from task T_j that is large enough to ensure that the data is always available. This approach can also be used to bound jitter on input [45].

From the above discussions it is clear that. task dependence can be captured by the notion of timing offsets for tasks. Further, given a task-set and the details of inter-task dependencies, we can arrive at individual task arrival times.

2.2 End-to-End System Model

This model differs from the uniprocessor system model (single component model) in that it considers more than one independently allocatable component in the system. A task in such a system can require execution on multiple components. Hence, a task is no longer viewed as an indivisible entity but as a sequence of sub-tasks. We assume that each sub-task of a task is associated with a component. Therefore a task that uses r components is decomposed into r sub-tasks, one corresponding to each component. A discussion of reasons and guidelines for task-decomposition can be found in [49].

We assume that the components in the system are ordered. The traditional flow-shop model [4] is based on the assumption that all tasks in the task-set access *all* resources and that they do so in the same order. A more general view to flow shops would be to relax the requirement about tasks having to access all resources but still maintaining the order constraint. This model will be referred to as the ordered flow shop model. If there are m components in the system, $R_1, R_2, \ldots, R_m$, then a task T_i can be considered to be a sequence of sub-tasks $T_{i1} \rightarrow T_{i2} \rightarrow \ldots \rightarrow T_{im}$. In the case of traditional flow-shop model, each sub-task T_{ik} is required to have a non-zero execution time requirement on the component it runs. Ordered flow shop model relaxes this constraint.

A sub-task T_{ik} of task T_i is characterized primarily by its execution time requirement on the component (R_k) it runs. In the case of the ordered flow shop model, if a component k is not used by a task T_i then the execution time requirement of the task T_{ik} is assumed to be zero. The parameters of periodicity and deadline are characteristics of a task and not that of the sub-tasks. Since these parameters apply to the task as a whole (from the start of the first sub-task to the end-of the last sub-task) we refer to these as the end-to-end parameters of the task. The last parameter associated with the task is its priority Pr_i which may be inherited by its sub-tasks. Alternatively, we can allow individual sub-tasks of a task to be assigned priorities independently. Unless otherwise specified, throughout this study, we assume that sub-tasks of a task inherit its priority.

2.3 Real-Time Channel Model

The two models described above are computational models. The real-time (RT) channel model however is a communication model that abstracts the communication activity in real-time packet switched networks [42, 38]. A real-time channel is uni-directional[2]. An entity (say a process) wishing to communicate with another entity on a remote machine does so by establishing a real-time channel that has certain characteristic timing and buffer space requirements.

A real-time (RT) channel timing requirement can be defined by the following parameters:

- The minimum message inter-generation time
- A *maximum* message size
- An end-to-end deadline for the RT channel

It is reasonable to assume prior knowledge of these parameters for many applications such as real-time timing control and monitoring, interac-

[2]A bi-directional R-T channel can be created by combining two uni-directional RT-channels [54]

tive voice/video transmission and many other multimedia applications. In applications where these parameters are less predictable, estimates can be used. Note that any guarantees that the underlying communication subsystem provides to the application is sensitive to the ability of the application to correctly specify its requirements. In this thesis, we are not interested in how such a correct specification is achieved, but given such a specification, how does the underlying system guarantee its being met.

Formally, an RT channel can be defined as follows [53]:

Definition 2.3.1 *A real-time channel C_i described by a tuple (g, m, d) is a connection between two nodes and require that every message at the source be delivered to the destination in duration of time no longer than d, under the conditions that the message inter-generation time is g, and the message size is m.*

This definition of an RT channel helps in network management and also provides a convenient means of charging users for their connection requests. For example, a user will pay lower connection fee for a voice channel than a video channel since the former uses less bandwidth. A connection that demands a low end-to-end delay (or deadline) is likely to cost more than one that tolerates a higher end-to-end delay (or deadline).

2.4 Glossary of Notation

The following table summarizes the notation used throughout the thesis.

Table 2.1: Glossary of Notation

Notation	Description
t	Time
T	A task-set
T_i	The i^{th} task in a task-set T
a_i	The arrival time of the first instance of task T_i
e_i	Execution time of task T_i
p_i	Period of task T_i
d_i	Deadline of task T_i
Pr_i	Priority of task T_i
W_i	Worst-case response time of task T_i
T_i^j	The j^{th} instance of task T_i
a_i^j	Arrival time of the j^{th} instance of task T_i
d_i^j	Deadline of the j^{th} instance of task T_i
W_i^j	The response time of the j^{th} instance of task T_i
T_{ik}	The k^{th} sub-task of task T_i
a_{ik}	Arrival time of the first instance of task T_{ik}
e_{ik}	Execution time of the sub-task T_{ik}
p_{ik}	Period of sub-task T_{ik}, if known
d_{ik}	Deadline of sub-task T_{ik}, if known
Pr_{ik}	Priority of sub-task T_{ik}
W_{ik}	Worst-case response time of sub-task T_{ik}
R_r	The component with an assigned index r in the system
C_i	Real-time channel i
g_i	The inter-message generation time of RT channel C_i
m_i	The maximum message size of RT channel C_i
d_i	The end-to-end deadline of RT channel C_i

Chapter 3

Motivation and Relevant Background

We are interested in extending the current schedulability analysis to accommodate changes in task execution time. It is only befitting to spend some time in describing the principles and assumptions that underlie this analysis. Most schedulability results [24, 19, 44, 46] are based on the critical instant argument, which defines a worst-case condition for a task. Clearly, a task suffers its worst completion time when it has to compete for the processor (or component in question) with every higher priority task in the system. That is, when it arrives at a time when all other higher priority tasks also arrive. This instant is called the critical instant. Therefore, it is sufficient to look at the completion time of this one instant in order to ascertain the task schedulability. But does this computation really give us the worst-case completion time of a task? In other words, given a task's characteristics, will it ever suffer this completion in reality?

Notice that the critical instant argument clearly ignores the arrival information of tasks and makes the assumption that, sooner or later at least one of the instances of a task will face a critical instant. It can be seen, however, that this is not necessarily true and therefore, the actual worst-case completion time of a task can be less than or equal to the completion time computed by the critical instant assumption. A simple example will clarify this point: Consider a task-set with two tasks, T_1 and T_2 whose characteristics are, $a_1 = 0, e_1 = 2, p_1 = 12, d_1 = 10$ and $a_2 = 3, e_2 = 1, p_2 = 12, d_2 = 9$ respectively. Further assume that T_1 is the task with the higher priority. Clearly, task T_2 will never encounter a critical instant because, its every instance will be ready only 3 units of time after the arrival of T_1. Further, T_1 needing only 2 units of execution time will complete before T_2's instance is ready. In this scenario, the worst-case response time of task T_1 will be 2 and that of T_2 will be 1. Ignoring the arrivals and using the critical instant argument will result in T_2's worst-case completion time being computed as 3 and not 1. Therefore, ignoring the arrival times of tasks and using the critical instant argument leads to a pessimistic computation.

Can we tolerate the pessimism inherent to this computation? The answer to this question depends on the environment under consideration, viz., a uniprocessor or a distributed (more generally end-to-end) system. In uniprocessor systems, depending on the assumptions (task independence for example) made, practioners [6] have argued that the cost of finding a more precise measure of the task completion time far outweighs the benefit gained (say, in terms of saved resource utilization). However, there are convincing arguments to the contrary by Tindell in [45]. He discusses scenarios that show the importance of considering the task arrival information in schedulability analysis[1]. We believe that the importance can be really felt in end-to-end systems and in uniprocessor systems with dependent tasks and not so much in uniprocessor systems with independent tasks.

[1]Look at the discussion in Chapter 2 about dependent and independent tasks.

Now, let us look at the problem of schedulability analysis in end-to-end systems. The schedulability of a task in an end-to-end system can be reduced to a sequence of uniprocessor schedulability problems provided we are able to compute the characteristics (period and arrival time) of the sub-tasks. Let us assume for now that we have a mechanism to compute the sub-task periodicities (the mechanism will be described in detail later). We don't require the arrival time information if we follow the critical instant argument, since we are going to ignore it anyway. We can use the critical instant argument (ignoring the arrival time a_{ik}) to find the worst-case completion times of all sub-tasks T_{ik} ($1 \leq k \leq m$). Clearly, the worst-case completion time of the task T_i is given by the sum of the worst-case completion times computed above. Observe that we have a cumulative measure of pessimistic computations that is bound to be more pessimistic. Therefore, we can see that even if one can tolerate the pessimism inherent in the critical instant argument, in the context of uniprocessor systems, we cannot do so in the context of end-to-end systems.

Before we give a description of the problem we are interested in addressing in this study, we would like to motivate the reader by briefly discussing the source of the problem. In the chapter 1, we mentioned that the kinds of changes (that interest us) that systems undergo, manifest themselves as task execution time changes. A brief discussion of these changes follows.

Note that, the task parameters, deadline and periodicity are dictated primarily by the environment. The arrival time of a task is governed by the environment and the inter-dependence between the tasks. The execution time of a task on the other hand is governed among other things by: (i) the programming language chosen, (ii) the compiler, (iii) the operating system, and

(iv) the processor architecture (e.g., pipeline, cache). Therefore, finding the execution times of tasks is complex and involved [31, 23, 1]. In most cases it is almost impossible to compute a deterministic measure of the execution time of a task. Most research efforts use the worst-case task execution time and not the mean execution time. While this choice can be justified by the fact that the analysis is based on the worst-case scenario, it nevertheless results in an over-design of the system. Also, this assumption can result in poor resource utilization.

Using mean task execution times in the computation does reduce the pessimism but unfortunately we could have cases where the guarantees provided by the schedulability analysis could be invalid (The number of such cases being determined directly by the variance in the computed mean execution time). Therefore, it is necessary to accommodate the variance information along with the mean (for task execution times). For example, if the mean execution time of a task is $\overline{e}$ and the variance of this mean is σ then it implies that the actual execution time is most likely to lie in the interval $(\overline{e}-\sigma,\ \overline{e}+\sigma)$. Schedulability analysis done using the mean execution time will remain valid even when the actual execution time falls between $(\overline{e} - \sigma,\ \overline{e})$. However, the same does not hold for the interval $(\overline{e},\ \overline{e} + \sigma)$. Assuming, the variance is expressed in terms of the mean (which is quite a common practice), we can represent σ as $fac \times \overline{e}$, where fac is a constant. If we can extend the analysis done by using the mean execution time to accommodate the possibility of the execution time being scaled by a factor sf then, it can be seen that this is equivalent to: allowing a variance of $fac \times \overline{e}$. Where, $sf = 1 + fac$.

As a system evolves the functionalities of tasks expand, reflecting in terms of improvement in the data handling of tasks. For example, as an air traffic control system adapts to new traffic (say from monitoring 8 flights to 12 flights) though the tasks themselves (their code that is) might not change the data handled by the tasks can change, resulting in an increase in the execution times of the tasks. This increase does affect the schedulability guarantees made using the previous execution times. Therefore, what we are interested in is, finding a factor sf by which the execution times can be scaled (capturing the data handling change) without invalidating the schedulability guarantees.

A more direct scenario that affects the completion time computation occurs when the target platform changes. Any analysis performed (to guarantee performance) assuming particular values of task execution times becomes invalid once the target platform changes. For example, a faster processor could result in a lower execution time (not invalidating the analysis), but a slower processor would surely have an adverse affect on the schedulability analysis. As a system evolves, though in general the overall system is likely to improve, the performance of individual components (some processors for example) might not always improve. Another instance where a target platform is in general slower, arises in the case of prototype building and testing [51].

A last case where we observe the need to do schedulability analysis for at least two target platforms arises in the area of fault tolerance. It is common practice to provide fault-tolerant operation by the use of redundant components (often at least one secondary component). In general, secondary components provide only a minimal functionality (sufficient to keep the system operational till the primary is fixed) and therefore tend to be slower. Any schedulability analysis guarantees provided with the primary component as the target will be invalid once the system falls back onto the secondary.

From the above discussion we note that, what we need is a measure (will be referred to as the scaling factor for obvious reasons) that in some sense qualifies the schedulability analysis. Provided the task execution times (as a result of the changes described above) satisfy a bound dictated by this measure the schedulability analysis remains valid.

We now discuss the underlying theory derived from past results in the area of real-time systems that is used in this study.

3.1 Scheduling Theory

Research in schedulability analysis has been focused mainly on uniprocessor systems. In recent years the original fixed priority analysis [24] has been considerably extended, relaxing many of the assumptions of the original computation model. Lehoczky *et. al.*'s [20] efforts to find the worst-case timing behavior of rate-monotonic tasks was the first in this direction. They have subsequently extended this result further, to accommodate any fixed priority task assignment [19]. In this thesis we make extensive use of this result.

The following, is a brief discussion of scheduling under different assumptions about the environment and tasks. A good source of related discussion can be found in [48] and [40].

3.1.1 Static versus Dynamic Scheduling

Static scheduling mechanisms assume complete a priori knowledge about the task characteristics including inter-task dependencies. Such assumptions are valid in many of today's practical real-time systems [39]. For example, real-time control of a process control application might have a fixed set of sensors and actuators, and a well defined environment whose processing requirements are all known a priori. The operation of the static scheduling algorithm in such a system involves producing a fixed schedule for what is called a hyperperiod. The fixed schedule repeats every hyperperiod [48]. For example if the arrival times of all tasks in a task-set are 0 then the hyperperiod is given by the *least common multiple* (LCM) of the task periods. A static scheduling algorithm assigns a fixed priority to each task that remains unchanged for the lifetime of the task.

It has been shown by Liu and Layland in their very well known paper [24] that the rate monotonic priority assignment (RMS) guarantees the schedulability of a task-set (of n tasks), if the utilization of the task-set is less than or equal to $n(2^{1/n} - 1)$. For large n this bound tends to 0.693. Further, the RMS was shown to be an optimal static fixed priority assignment when the deadlines of tasks coincide with their periods. Other significant results in this direction were, Leung's [21, 22] formulation of an alternative (static fixed) priority assignment to accommodate tasks whose deadlines are less than or equal to their periodicities. Audsley *et. al.* [2] allowed the addition of guaranteed sporadic tasks and Tindell *et. al.* and Locke [26, 27] considered the possibility of tasks having a release jitter.

If a real-time system operates in a dynamic environment where it is impractical to assume complete knowledge of the processing requirements of tasks (and their interactions) we use a dynamic scheduling mechanism. In such a case the chosen dynamic scheduling algorithm is analyzed off-line using the expected requirements of the dynamic environment. The same algorithm is then used at run-time with the assumption that the run-time behavior of the system does not depart markedly from the expected behavior for which the scheduling mechanism was tested. A static or dynamic scheduling algorithm can be applied in either of the cases, viz., the environment is known or changes dynamically. However, what distinguishes the two is the performance guarantees that can be made about the scheduling mechanism. For example, if the assumption of complete a priori knowledge about the system does not hold then, while a static scheduling algorithm can be used but it will not be able to make any schedulability guarantees.

The earliest deadline first (EDF) scheduling mechanism [24] is the most widely used dynamic scheduling mechanism. EDF runs that task among the task-set that is ready to run and is closest to its deadline. Therefore, as a task nears its deadline its priority relative to other tasks increases. The EDF scheduler was shown to be an optimal dynamic scheduler in the sense that, if there exists a scheduler that can guarantee that all the tasks would meet their deadlines then, so will EDF. A drawback of the EDF scheduler is that in its comparison of tasks, T_i, T_j, with deadlines, d_i, d_j, there is no regard for their execution times, e_i, e_j. Therefore, even if the two tasks' deadlines differ by a small amount $(d_i - d_j = +\epsilon)$, d_i will be chosen to run instead of d_j even if their execution times differ by a large amount $(e_i << e_j)$. The least laxity first (LLF) scheduler [29] uses a different basis for priority assignment that partly answers the need to accommodate the execution times of tasks. The laxity of a task, T_i is the difference $(d_i - e_i)$, between the deadline and the execution time of a task. It essentially captures the room for meeting the deadline of a task. LLF scheduler has also been shown to be an optimal dynamic scheduler.

In summary, the choice of a particular scheduling mechanism is governed by such considerations as: (i) The assumptions that can be made about the environment (static vs. dynamic), (ii) the guarantees provided by the scheduler being considered, (iii) the cost in terms of computational overhead of the scheduler and (iv) the constraints on the task characteristics (e.g., deadline $\leq$ period of tasks).

3.1.2 Relationship between deadline and period

The classical scheduling result by Liu and Layland [24] is built on the assumption that the deadlines of tasks are equal to the periods of tasks. In other words, an instance of a task is required to be completed before its next instance is ready. As already mentioned, the rate-monotonic priority assignment (RMS) gives an optimal fixed priority scheduling mechanism for this scenario.

However, if the deadlines of tasks are allowed to be less than or equal to their periods (i.e., $d_i \leq p_i \; \forall T_i$) then the optimality of RMS does not hold. As shown by Leung and Whitehead in [22], the deadline monotonic scheduling (DMS) mechanism is an optimal for this scenario. The DMS assigns the highest priority to the task with the shortest deadline. This DMS scheme is optimal in the sense that if any fixed priority scheme can schedule a task-set then so can the DMS scheme. One should not confuse the deadline monotonic scheduler with the EDF which is a dynamic scheduling mechanism where a task's assigned priority can change dynamically. A special case of this scenario occurs when the deadlines of tasks are a constant factor of their periods. In other words, $\forall T_i \, , d_i = \kappa \times p_i$, where $\kappa \leq 1$. Note that both RMS and DMS would end up being the same in this case.

The third scenario occurs less commonly in real-time applications (more common in imprecise computation [25, 36, 37]), where the deadlines of tasks can be beyond the end of their periods. This scenario was first studied by Lehoczky [20], where he considered the possibility of κ (in the formulation of the previous paragraph) being greater than 1. He showed that for a value of $\kappa = 2$ the utilization bound of RMS increases from 0.693 to 0.811. He reported simulation studies that show a more promising (close to 1.000) increase in the achievable utilization.

3.1.3 Precedence Constraints and Resource Sharing

An inherent characteristic that governs current complex real-time systems is the cooperation of tasks to achieve the goal of an application. Such cooperation can be captured by various types of communication semantics. Depending upon the chosen semantics, tasks experience precedence constraints or blocking or both. Blocking occurs due to the use of a synchronization mechanism (like priority inheritance protocol [33]) to regulated resource sharing. Similarly the use of critical sections to achieve concurrency control (Sha *et. al.* [34]) can result in blocking. An alternative to using a concurrency control mechanism for regulating resource accesses is to impose strict order on these accesses. Such an order can be captured by imposing precedence constraints on tasks that share the same resource. As was shown by Tindell *et. al.* in [45] and will be explained in more detail in chapter 5 of this thesis, these two scenarios can be captured by considering tasks to have arrival time characteristic in addition to execution time, period and deadline.

3.2 Uniprocessor Schedulability

Most schedulability results [24, 19, 46] are based on the critical instant argument, which defines a worst-case condition for a task. As noted before, worst-case completion time computation is the crux of schedulability analysis. The critical instant argument gives us a situation under which a task will undergo its worst possible completion:

Lemma 3.2.1 *The worst-case completion time for task T_i occurs when it arrives at a critical instant, $a_1 = \ldots = a_i = 0$.*

This lemma tells us that any instance of a task that arrives at a point in time when all higher priority tasks also arrive suffers the worst completion time. We still have to compute this completion time. The following equation gives a mechanism for this computation:

$$W_i is = the\ smallest\ X\ for\ which(\sum_{j=1 to i-1} e_j \lceil \frac{X}{p_j} \rceil + e_i) \leq X$$

The above equation can be viewed in terms of *demand* and *supply*. The term $\sum_{j=1 to i-1} e_j \lceil \frac{X}{p_j} \rceil$ captures the demand for processor time from all instances of tasks with priority higher than i over X units of time. Therefore, the fraction in the above formula gives the ratio of the *demand* to the *supply*. The shortest supply X for which the demand is met, i.e., *supply* $\geq$ *demand*, gives the completion time of the task $= W_i$. Further, if this value W_i is less than or equal to the deadline of the task (D_i), then the task meets its deadline.

3.3 Other Relevant Work

The area of end-to-end scheduling is a relatively new are in real-time systems. Prominent work in this area has been reported by Bettati in his thesis [4]. As he showed in [4], the problem of finding an optimal scheduler for scheduling end-to-end tasks is NP-complete [13]. To this end, he proposed and analyzed heuristic approaches to solving this problem. The schedulability test he uses to test his heuristic schedulers is based on the critical instant argument. As was discussed before, this results in a pessimistic evaluation of the scheduler. It is therefore possible that he rejected heuristics that did not perform well under the pessimistic test but would in fact have been able to guarantee schedulability.

Other ongoing research on this problem was reported by Etamadi in [7]. He proposes to enhance the analyzability of end-to-end systems without making constraining assumptions that restrict resource utilization. Further, he proposes building robust application models that would allow enhancements like synchronization, communication. Related work can also be found in [14, 30].

Finally, on the problem of admission control of RT channels [28, 9]. The Tenet group's Ferrari et. al. were the first to deal with this problem extensively. The principle they followed [8, 9] in the design of an admission control scheme is based on verifying, whether the resources available on the path of the newly requested RT channel are sufficient even in the worst possible case, to

1. provide the new RT channel with the QoS it needs and,
2. allow the guarantees offered to all the existing RT channels to continue being satisfied.

The above verification depends upon the kinds of QoS parameters allowed. The most important QoS parameter of concern to real-time system designers is meeting a latency bound (deadline). We restrict our interest to this parameter. There are two tests that are relevant in this context:

- Schedulability Test: Does the addition of the new channel to the already established channels using this link cause either the new channel or one of the already established channels to miss their deadline?
- Buffer Space Test: Is the available buffer space at the link sufficient to allow the messages of the new channel to be stored for a length of time equal to the delay faced by the channel at this link?

Different approaches to the admission control problem (in real-time systems) will differ in the way the above two questions are answered. Therefore, a study in admission control reduces to the study of these tests. The buffer space test has been successfully addressed by the Tenet group [9]. We concentrate mainly on the schedulability test because it is our belief that there is room for improvement here. In particular, there are many situations that have not been considered in this context.

Chapter 4

Problem Statement and Description

4.1 Scalability of Uniprocessor Systems

The uniprocessor scalability problem can be formally defined as follows:

Problem Definition 4.1.1 *Given a task-set T consisting of n tasks, and a subset S of T. Find the maximum common scaling factor by which the execution times of each of the tasks in the subset S can be scaled, without affecting the schedulability of the task-set T.*

As described in the previous chapter, the schedulability of tasks running on a uniprocessor can be determined by lehoczky's [19] schedulability test. The scalability problem now involves extending this test to compute the scaling factor.

4.2 Scalability of End-to-End Systems

The problem of interest here is the "Scalability of task-sets in end-to-end real-time systems". The problem can be looked at from two different viewpoints: (i) The first viewpoint stems from assuming the scaling to occur as a result of a change in one or some of the components in the system; (ii) The second viewpoint stems from assuming the scaling to occur as a result of a change in the functionality of some or all of the sub-tasks in the system.

4.2.1 Component Change

A change in a component R_r can result in a gain or a loss in the speed of processing for the sub-tasks running on it. Clearly, if there is a gain in speed of a component then this will not have any adverse affect on the completion times of sub-tasks running on it. However, if the component is replaced by a slower one then it will affect the completion times and hence the schedulability of the sub-tasks running on it. The problem of interest therefore is, to find the maximum factor by which all the sub-tasks on a particular component R_r can be scaled such that the schedulability of the task-set (comprising all n tasks that is) is unaffected.

In the following formulation we assume that a single component is undergoing a change. We can however, generalize it to a sub-set of components. The problem of scaling occurring as a result of a component change can now be formally posed as:

Problem Definition 4.2.1 *Given a task-set T of n end-to-end tasks executing in a system of m, $(m \geq 1)$ components, find the optimal scaling factor $1/sfc$ (corresponding to a maximum sfc) with which the processing speed of a given component r can be scaled (down), without affecting the schedulability of the task-set.*

In other words, we are interested in the maximal component change the task-set can survive. The reason for representing the scaling factor as a reciprocal will become obvious once we realize that a lowering in processing speed of a component will reflect as an increase in the execution times of sub-tasks running on the component. For example, if the speed of the component is S (instructions per unit time) then an execution time requirement of a sub-task T_{ik} being e_{ik} (time units) implies that the number of instructions that the sub-task requires to execute are $S \times e_{ik}$. If the processing speed is scaled down by $1/sfc$ (implying that $sfc \geq 1$) then, we have the new speed $S' = S \times 1/sfc$. Therefore, the amount of time it would take to execute $S \times e_{ik}$ instructions[1] is given by:

$$\begin{aligned} e'_{ik} &= \frac{S \times e_{ik}}{S'} \\ &= \frac{S \times e_{ik}}{S \times 1/sfc} \\ &= sfc \times e_{ik} \end{aligned}$$

In this formulation, we assume that all sub-tasks that execute on component r will be equally affected. That is for all sub-tasks T_{jr} $(1 \leq j \leq n)$ running on component r their execution times as a result of the change would become $sfc \times e_{jr}$. The next perspective to the scaling problem however, allows for the possibility that only a subset of the sub-tasks running on a component are affected as opposed to all sub-tasks being affected.

4.2.2 Task Changes

As opposed to a change in one or more components, we can envision one or more sub-tasks being affected by a change. For example as a system evolves,

[1]Assuming that a change in the component is such that the same code is able to run on the new component

to encompass more functionality, some of the sub-tasks (their code that is) need to be modified (enhanced), resulting in an increase in their execution times. Alternatively, enhancements could come in the form of increased data handling, manifesting as an increase in the execution times of tasks (as before we do not consider decreases because they do not violate prior schedulability guarantees). The problem of scaling occurring as a result of task changes can now be formally posed as:

Problem Definition 4.2.2 *Given a task-set T of n end-to-end tasks executing in a system of $m, (m \geq 1)$ components, find the maximum scaling factor, sft with which a subset of the sub-tasks (say S : $\{T_{ik}$, where $1 \leq i \leq n; 1 \leq k \leq m\}$) execution times can be scaled, so that the task-set T's schedulability remains unaffected.*

As it will be clear from the following discussion, solving the end-to-end schedulability problem can be reduced to solving m independent (deemed independent by an important transformation to be described later) single component schedulability problems. In other words, solving the above formulated scalability problem for a subset S will become equivalent to solving m single component scalability problems on each of the subsets $S_1, S_2, \ldots, S_m$. A subset S_r contains all sub-tasks T_{ir}, $(\forall i)$ belonging to S. If for a particular component r, there are no sub-tasks T_{ir} $(\forall i)$ in S then we set the corresponding set $S_r = \phi$ (null set).

We can observe one step that is common to the above two formulations, viz., determining the schedulability of the given task-set T. This is the initial step to be done in solving both these problems. Note that, if a task-set T is unschedulable to start with then, any adverse change either to a component or to a subset of the sub-tasks is only bound to make the situation worse. The problem of interest can therefore be posed as:

Problem Definition 4.2.3 *Given a task-set T of n end-to-end tasks executing in a system of m components, find if the task-set is schedulable.*

In order to find the schedulability of end-to-end task-set, we have to find if each end-to-end task in turn will be schedulable, i.e., meet its deadline when the individual sub-tasks compete for processing on their respective components. Therefore, for each task we have to find its worst-case completion time which can then be compared against its deadline. The worst-case completion time of a task T_i can be computed by assuming that all its sub-tasks simultaneously suffer their worst-case completions. The worst-case completion time of the task (T_i), is then given by the sum of the worst-case completion times of the individual sub-tasks (T_{ik}). For a given sub-task T_{ik}, executing on the component R_k, the information we need to find its worst-case completion time is:

- The arrival time of all sub-tasks T_{jk} $(j \leq i)$[2], which are of higher priority than T_{ik} and are running on the same component, R_k.

[2]Unless otherwise specified, the arrival time of a sub-task T_{ik} implies the arrival of its first instance

- The periodicity of all sub-tasks T_{jk} $(j \leq i)$, which are of higher priority than T_{ik} and are running on the same component R_k.

Notice that when, $i = n$ we have to find the arrival time and periodicities of all sub-tasks in the system to determine the schedulability of the task T_n. Therefore, we need a mechanism by which we can derive these two parameters (since these are not given a priori). Note that, only the first sub-task of any task is truly periodic. The arrivals of the consecutive instances of any sub-task T_{ik}, $(1 \leq i \leq n; 1 < k \leq m)$ are dictated by the completion times of the sub-task preceding it, i.e., $T_{i,k-1}$. These completions are obviously non-periodic and so are the arrivals of sub-task T_{ik}. We however can impose a periodicity on these sub-tasks by a proper justification. The **phase adjustment** mechanism [51], is one such mechanism that derives sub-task arrival times and also their periodicities. The term phase here is used to denote arrival time.

Imposing a period on the arrivals (of consecutive instances that is) of a sub-task T_{ik} $(1 < k \leq m)$, implies that, even if the preceding sub-task $T_{i,k-1}$ does finish at a particular time[3] (say $F_{i,k-1}$), the sub-task T_{ik} will not be ready immediately. A finite amount of time (say $W_{i,k-1} - F_{i,k-1}$)[4] has to elapse before the sub-task T_{ik} is ready to execute. It is necessary to limit this finite amount of wait time in the sense that, if it is too large then it could hurt the utilization of the component R_k. This is due to the fact that, while the sub-task is being intentionally delayed, the component R_k could be idle. On the other hand this delay must be large enough to be able to accommodate all possible finish

[3]All references to time are relative to $t = 0$, unless otherwise specified

[4]Here, $W_{i,k-1}$ is a constant for the task $T_{i,k-1}$; therefore, the delay is a variable for each instance of $T_{i,k-1}$

times (of its various instances) of task $T_{i,k-1}$. Clearly therefore, in the limiting condition (delay = 0) $W_{i,k-1}$ must be given by the worst-case completion time of the sub-task $T_{i,k-1}$.

An effect of this adjustment is that a sub-task T_{ik} will always be ready (or arrive) after a constant amount of time from the arrival of the preceding sub-task $T_{i,k-1}$. Therefore, knowing the arrival time of the sub-task T_{i1}, we can find the arrival of the sub-task T_{i2}, knowing which we can find the arrival of T_{i3} and so on. It should be clear to the reader that the above adjustment also allows all sub-tasks belonging to a task to inherit its period.

What the above adjustment has afforded us is, the ability to treat each of the components independently, provided we are able to find the terms W_{ik} $(\forall i, k)$. Observe that we have all the information about sub-tasks T_{i1} $(1 \leq i \leq n)$, running on the first component, R_1 (That is, we have their arrival times, periods and execution times). Now the problem we wish to solve is finding the worst-case completion times of these tasks. Once we find these worst-case completion times we have all the information about sub-tasks T_{i2} $(1 \leq i \leq n)$, running on the second component, R_2 and so on. The problem of interest can therefore be formally posed as:

Problem Definition 4.2.4 *Given a task-set T of n tasks executing on a single component, find the worst-case completion times of all tasks in the task-set.*

Observe that this problem is similar in sense to the schedulability problem solved by Lehoczky [19] (refer to Chapter 3). However, while his solution using the critical instant argument can be used in the context of uniprocessor systems, we cannot use it here (in the context of end-to-end systems that is). Finding a solution to this problem is one of the results of this thesis.

Now that we described a mechanism to test whether a given task-set is schedulable, we have answered the question of whether there exists a scaling factor as defined by the two problems, 4.2.1 and 4.2.2. Clearly, if the tasks are so stringent that any increase in the execution times of the sub-tasks cannot be tolerated, then the scaling factors sfc (as defined in problem 4.2.1) and sft (as defined in problem 4.2.2) will both be equal to 1.0.

The end-to-end schedulability problem has been reduced to m single component worst-case completion time computation problems and not m single component schedulability problems. Therefore, we cannot talk about extending a single component's schedulability, unless we derive the sub-task deadlines. A major research issue in end-to-end scheduling has been the derivation of sub-task deadlines. Given an end-to-end task's deadline the problem of finding an optimal[5] division of this deadline among the sub-tasks is intractable [15] (NP-complete [12]). This result has prompted a heuristic approach [4, 15, 30], two such heuristics being: (i) divide the task's slack[6] equally among the sub-tasks; (ii) divide the task's slack among its sub-tasks in a weighted proportion of their execution times.

The above two heuristics vary mainly in their sensitivity to the execution times of tasks. For example, the second heuristic is built on the assumption that the shorter a task's execution time requirement, the more likely it will have

[5]In the sense that, if there exists a division that would help the task meet its deadline then the mechanism should find it

[6]The slack of a task is given by the difference between its deadline and its execution time

its requirement met and therefore the lower is the slack assigned to it. The first heuristic is built on the assumption that the priority inherited by a subtask has a greater impact on its ability to meet its execution time requirement than its execution time itself. Thus the slack is divided equally among all subtasks. This allows us to reduce the end-to-end scalability problem to m single component scalability problems.

Now, finding the common scaling factor is a simple matter of finding the minimum of the m scaling factors (each corresponding to one component). The problem of interest therefore is the single component scalability problem, which can be formally defined as follows:

Problem Definition 4.2.5 *Given a schedulable task-set T of n tasks executing on a single component and a subset S of T, find the maximum scaling factor sf with which all tasks in S can be scaled without violating the schedulability of any of the tasks in T.*

Now, we can observe that solving the two problems 4.2.1 and 4.2.2 amount respectively to:

- Solving the single component scalability problem (4.2.5) with $S = T$.
- Solving the m single component scalability problems and taking the minimum among these scaling factors.

We can now summarize this discussion on end-to-end scalability by noting that, solving this problem entails finding solutions to the two problems, 4.2.4 and 4.2.5.

4.3 Admission Control of RT Channels

The problem of admission control of real-time (RT) channels was first investigated by Ferrari et. al. [9] at the Tenet group. Admission control is the mechanism by which multiple real-time connections can simultaneously share the resources of a packet switching network without resulting in congestion. Further, the connections are guaranteed a particular *quality of service* (QoS) that is initially (at connection set up) agreed upon. Admission control comes into play when a new RT channel is being requested. An RT channel (or a connection request) is accompanied with a QoS list that describes the requirements of this connection. Popular QoS requirements in the literature of distributed real-time systems are - throughput, latency (or deadline), packet loss tolerance [28, 10, 35] etc.

The mechanism used to determine the admissibility of a real-time channel involves verifying at each intermediate link (along the path) in turn whether the RT channel's QoS requirements can be guaranteed. If a channel's requirements can be met at each of the intermediate links then we can accept the channel. If however, the channel's requirements cannot be met at any of the intermediate link then we can reject the channel. In fact the first such link that deems the channel inadmissible is sufficient to confirm that the channel would not be admissible.

In order to test whether a channel's requirements will be met at an intermediate link we have to know its deadline and its period at that link. Finding the period is straightforward according to the phase adjustment mechanism. However we do have to derive the deadline of the RT channel at intermediate links. Since the service time of the channel on each of the links is the same one way to derive the deadlines would be to divide the slack of the RT channel equally among the intermediate links. However, if one wishes, one can use a more sophisticated heuristic [15, 4] to derive these deadlines. This reduces the problem of finding the admissibility of an RT channel to be equivalent to solving the admissibility at each of the intermediate link. From here onwards when we refer to the admissibility of an RT channel we mean its admissibility at an intermediate link.

Now, the question that admission control has to answer when accepting a new connection can be broadly phrased as:

Problem Definition 4.3.1 *Given the QoS requirements of a new RT channel is it possible to accept this channel without violating the QoS guarantees made to RT channels that have already been accepted?*

To summarize this chapter, we have defined four problems of interest:

- The uniprocessor scalability problem (4.1.1),
- The single component schedulability problem (4.2.4),
- The single component scalability problem (4.2.5), and
- The problem of admission control of RT channels (4.3.1).

The next chapter discusses the first of these problems. Note that, the third problem in the above list is different from the first in that, it involves tasks whose arrival times cannot be assumed to be zero (as in the critical instant assumption).

Chapter 5

Scalability in Uni-processor Environments

As discussed in Chapter 1, a host of schedulability related issues translate into a more general problem called the scaling problem. Observe that the scaling factor as defined in the problem statement attempts to capture a common factor by which a sub-set of tasks belonging to a task-set can be scaled together. In our first attempt at this problem we made an assumption that the sub-set S is the same as the task-set T. That is, we were interested in scaling the complete task-set as opposed to a sub-set of tasks. A solution to this problem can be found in [52]. The following discussion however considers the general scaling problem as stated in Problem 4.1. The model assumed is the uniprocessor model described in chapter 2. We repeat the problem statement here and give a discussion about the possible approaches to the solution followed by the details of the solution approach we have taken.

5.1 Problem Statement

- Given a task-set T consisting of n tasks, and a subset S of T. Find the maximum common scaling factor by which the execution times of each of the tasks in the subset S can be scaled, without affecting the schedulability of the task-set T.

The particulars about the scheduling algorithm used to schedule these tasks have not been specified in order to keep the problem general. The choice of scheduler can be either a dynamic scheduler (like earliest deadline first) or a fixed priority static scheduling algorithm. If the chosen scheduler is the latter then the tasks are assumed to be numbered (decreasing order) according to their priorities as dictated by the scheduler. The term, scaling factor is used to refer to a scale up in the execution times and not a scale down. It can be shown that if the execution time of a task is reduced then the schedulability of the task (and other lower tasks) will remain unaffected.

The use of the term, "maximum" needs some explanation here. The scaling factor we desire is one that cannot be improved upon. In other words, given that sf is the maximal scaling factor and ϵ is an infinitesimally small quantity. Using sf to scale the tasks in S would not affect the schedulability of the task set whereas using $sf + \epsilon$ as the scaling factor results in at least one of the tasks in T being unschedulable.

5.2 Discussion of Possible Solution Approaches

We concern ourselves mainly with a static fixed priority scheduling mechanisms because the above problem has a rather trivial solution when we assume a dynamic preemptive scheduling algorithm (say EDF). It is possible to find a feasible schedule using a dynamic scheduling mechanism provided the following condition holds for the utilization [24]:

$$U = \sum_{\forall j \in T}^{n} \frac{e_i}{p_i} \leq 1$$

If the utilization of the task-set is greater than 1, then clearly the task-set is not schedulable by any dynamic scheduling mechanism and further the question of scaling the tasks is not relevant anymore. The above condition is both a necessary and sufficient condition for EDF to be able to guarantee the schedulability of the task-set. Therefore, meeting the above condition ensures the existence of a scaling factor. Now, given such a task-set we can scale the tasks in the sub-set such that the new utilization $U' = 1$.

$$\begin{aligned} U' &= \left(\sum_{\forall j \in S} \frac{sf_{edf} \times e_j}{p_j}\right) + \left(\sum_{\forall j \in (T-S)} \frac{e_j}{p_j}\right) \\ &= sf_{edf} \times \left(\sum_{\forall j \in S} \frac{e_j}{p_j}\right) + U - \left(\sum_{\forall j \in S} \frac{e_j}{p_j}\right) \\ &= (sf_{edf} - 1) \times \left(\sum_{\forall j \in S} \frac{e_j}{p_j}\right) + U \end{aligned}$$

The scaling factor of interest therefore is when $U' = 1$, given by:

$$sf_{edf} = \frac{1 - U}{\sum\limits_{\forall j \in S} \frac{e_j}{p_j}} + 1$$

This factor is not valid in the case of static fixed priority preemptive scheduling algorithms because the above condition on utilization (i.e., $U \leq 1$) does not necessarily guarantee the existence of a fixed priority scheduling algorithm. A similar bound does exist for the rate monotonic scheduler (RMS: a fixed priority scheduling mechanism), under the assumption that the deadlines are equal to periods: $n(2^{1/n} - 1)$. The rate monotonic priority assignment is known to be optimal in this case [20]. Further, the total utilization of the task-set being less than or equal to $n(2^{1/n} - 1)$ is a sufficient (not necessary) condition for optimality. In other words the above condition guarantees that a rate monotonic priority assignment will result in the task-set being schedulable. Therefore, one can say that a scaling factor sf_{rms} (following the same derivation as above but replacing the utilization bound $n(2^{1/n} - 1)$ for 1) given by the following equation does not violate the schedulability of the task-set.

$$sf_{rms} = \frac{n(2^{1/n} - 1) - U}{\sum\limits_{\forall j \in S} \frac{e_j}{p_j}} + 1$$

The above computation of the scaling factor does give us a valid factor in the sense that using this factor to scale tasks does not violate the schedulability of the task-set. However, it is not necessarily optimal in the sense that the resulting utilization bound is not a tight bound. In order to understand why this bound is not tight one has to look more carefully at the meaning of the schedulability bound, $n(2^{1/n} - 1)$. This bound is only a *sufficient* and not a *necessary* condition for the task-set to be schedulable by the RMS mechanism [20]. Therefore it is possible that a task-set does not meet this schedulability bound and yet is schedulable by the RMS mechanism. Therefore we observe that a more precise analysis is necessary to get the *maximal* scaling factor.

A second observation one has to make about the above scaling factor computation is that, the computation derives its validity from the fact that the rate monotonic priority assignment is optimal when deadlines and periods of tasks coincide. If this condition (deadlines *equal* periods) does not hold then, we can no longer use the above result. If the deadlines of tasks are known to be less than their periods, then the deadline monotonic priority assignment (DMS) is known to be optimal [22]. However, there is no known sufficient condition on the total utilization. Therefore, in order to compute the scaling factor we have to do a more precise analysis of the task-set.

As a special case of the scaling problem, if the sub-set S is same as T then the scaling factor would be a simple reciprocal of the utilization in the case of EDF (i.e., $sf_{edf} = \frac{1}{U}$). Similarly, in the case of RMS, the scaling factor using the approach above would be, $sf_{rms} = \frac{n(2^{1/n}-1)}{U}$ (this is not optimal, as already discussed above).

In the following, we give the algorithm to find the maximal scaling factor when an arbitrary (RMS and DMS being two instances) fixed priority assignment is used. Before the details of the mechanism are presented we would like to intuitively motivate the idea behind it. We consider the case of scaling all tasks to present the motivation. Since we are interested in the common scaling factor, one approach would be to consider a successive approximation technique as taken by [6]. Incremental factors are used to scale tasks and perform a schedulability analysis to confirm if the increment is acceptable. Clearly, such a technique would be expensive.

An alternative approach would be to incorporate the scaling factor computation into the schedulability test. This is the approach we have taken. The schedulability test we use is the one proposed by Lehoczky in (refer to Chapter 2). The idea behind Lehoczky's schedulability test is to ascertain the schedulability of each task in turn starting from the highest priority task. The schedulability of each task involves considering all tasks that are of higher priority than itself. Therefore, the schedulability test of a task T_i can be interpreted as follows: To ascertain whether task T_i will meet its deadline while continuing to honor the timing requirements of all higher priority tasks. Note that the test does not consider whether the higher priority task meets its deadline. It only makes sure that any higher priority task will not wait for the processor while a lower priority task is using it. In other words, it ensures that in every p_j $(j < i)$ time units the task corresponding task T_j would get e_j units of the processor's time. It is possible for example that, a higher priority task T_j gets its last unit of required execution time between d_j and p_j (note $d_j \leq p_j$ $1 \leq j \leq n$), thus meeting its demand[1] but not its deadline.

On the same lines our approach to finding the scaling factor attempts to find the scaling factor for each task in turn starting from the highest priority task. The scaling factor (sf^i) obtained with respect to a task T_i therefore guarantees that the task T_i would meet its deadline continuing to honor the scaled (scaled by sf^i) requirements of all higher priority tasks. In other words, sf^i is the factor with which the execution times of all tasks with priority greater than T_i and including T_i can be scaled without T_i missing its deadline even after accommodating all the scaled higher priority tasks. The required scaling is then the minimum of all computed scaling factors sf_i. A more detailed treatment of the solution follows.

5.3 Details of the Approach Taken

An analogy can be drawn between the computation of the scaling factor sf^i and assessing the schedulability of the task T_i. In order to assess the schedulability of task T_i we compute the worst-case completion time of task T_i and compare it against its deadline. This computation takes into account the execution time

[1]It will not wait for the processor while a lower priority task is using it

demands of higher priority tasks (but is independent of the higher priority tasks' ability to meet their deadlines). Similarly in the computation of the scaling factor with respect to task T_i, we only account for the execution time (scaled) demands of the higher priority tasks and not the ability of these tasks to meet their deadlines.

We find such scaling factors for all tasks and the required scaling factor is the minimum among these, i.e., $sf = Minimum(sf^i)$. Note that each of the scaling factors sf^i only considers the schedulability of task T_i and any scaling factor that is less than sf^i will continue to guarantee its schedulability. Since we are interested in a common scaling factor, the lowest of the scaling factors $sf^i \quad h \leq i \leq n$ (The index h is defined below). In the following paragraphs, we present the details of the technique for the general scaling problem and a proof of its operation.

We make use of the schedulability test described in [19, 6] to find the worst-case response times of tasks.

Note that in the previous section we assumed that $T = S$[2] in order to simplify the explanation of the solution. In this context we gave a definition of sf^i that needs a slight refinement to adapt to the case that the set S is not necessarily equal to T. The scaling factor sf^i is the factor with which the tasks in the set S with priorities higher than T_i can be scaled without affecting T_i's schedulability, while continuing to honor the demands of *all* tasks with priority higher than T_i. The requirements of higher priority tasks include both: (i) the requirements of higher priority tasks that are not included in S and (ii)

[2] We assume that the tasks in S are sorted in a non-increasing order of their priority

the scaled (scaled by sf^i that is) requirements of higher priority tasks that are included in S. There are two important observations to be made:

- In the computation of the scaling factor sf^i the task T_i is not necessarily a member of S. This is because there are tasks in T that do not belong to S whose schedulability could be affected by the scaling of tasks in S. And we cannot ignore them in computing the desired scaling factor.
- The number of scaling factors to be computed is equal to $n - h$. The number of tasks in T of priority less than the highest priority task in S. Clearly from the definition of sf^i in the above paragraph, we see that for all tasks T_i whose priority is greater than the any in S, sf^i is undefined.

The given sub-set S is assumed to be sorted by the decreasing order of priorities. Let T_h be the highest priority task (first task) in S[3]. For each task T_i where $h \leq i \leq n$ (starting from $i = h$ and counting up), we find the scaling factor by which all tasks in S whose index is $\leq i$ can be scaled, while continuing to maintain T_i's schedulability (i.e., meeting its deadline and the demands of all tasks with priority higher than T_i are honored).

Since we make the critical instant assumption, only the first task instance of any task T_i needs to be considered for its schedulability [19]. Note that only higher priority tasks affect the schedulability of a task, because lower priority tasks will be preempted. We consider the execution profile of task $T_i (i \geq h)$ along with all higher priority tasks T_j where $j < i$. In Figure 1, the first continuous block (no idle time in between) of used time is represented as

[3]Task T_h is the highest priority task that needs to be scaled.

U_1. The notation U_1 is also used to represent the length of this block. The j^{th} such used time block is represented as U_j. Further, $U_{j,L}$ and $U_{j,R}$ represent the left and right boundaries of the block U_j (i.e., $U_j = U_{j,R} - U_{j,L}$) respectively, relative to the start time of consideration (i.e., $U_{1,L}$, which can be assumed without loss of generality to be zero). The first task instance of T_i (refer to Figure 5.1) completes at a point U_1 units of time after it has arrived, with all higher priority tasks also arriving at $U_{1,L}$, the same instant as T_i arrives - *critical instant.*

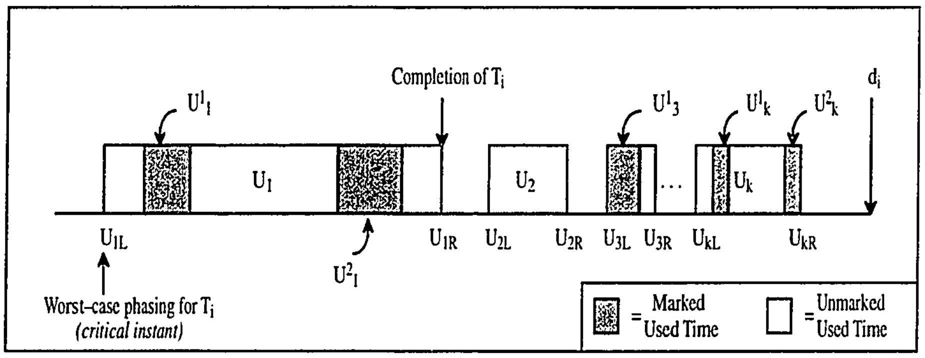

Figure 5.1: Task T_i's Execution Profile

The blocks of execution between the points $U_{1,R}$ (The earliest point in time after the completion of task T_i at which the processor becomes idle) and d_i (deadline of T_i) are : $U_2, U_3, \ldots, U_k$ (There are k used time blocks in all). These blocks represent the higher priority tasks that would have to be accommodated if we want to push the completion time of T_i towards d_i. Each block of used time is divided into marked and unmarked sub-blocks. A sub-block of block U_j is said to be marked if the execution that spans it belongs to a task (or tasks) that belong to the sub-set S. A marked sub-block, denoted as U_j^p, indicates the p'th marked sub-block in U_j. There are k such marked sub-blocks in all.The way the scaling factor, sf^i for task T_i is computed is as follows:

$$sf^i = \max_{1 \leq m \leq k-1} sf_m$$

where

$$sf_m = \frac{\sum\limits_{1 \leq j \leq m} \left((\sum\limits_{\forall l} U_j^l) + (U_{j+1,L} - U_{j,R}) \right)}{\sum\limits_{1 \leq j \leq m} \sum\limits_{\forall l} U_j^l}$$

sf_m in the equation above is the factor which when used to scale the execution times of all tasks in S of higher priority than T_i, will be able to stretch the completion time of task T_i at most till $U_{m+1,L}$. The first term in the numerator (same as the denominator) is the total of the execution times of tasks in S that are considered for scaling at this point (i.e., tasks in S whose priority is higher than T_i). The second term in the numerator is the total idle time that these tasks are being scaled to consume. Therefore the right hand side of this equation in a simplified sense can be viewed as $\frac{usedtime+idletime}{usedtime}$. Observe that, each sf_m is a valid scaling factor in the sense that it does not result in T_i missing its deadline. Since we are interested in the maximum scaling factor, the maximum among these valid factors is the required solution. The resultant scaling factor sf^i is therefore the maximum scaling factor w.r.t task T_i. However, from the definition of sf^i one can see that the possibility of a higher priority task missing its deadline is not accounted for in this factor (only its demand is accounted for). Therefore, this scaling factor is valid only in the context of task T_i. In order to find a common scaling factor for the sub-set S now, we have to find the minimum among all sf^i ($h \leq i \leq n$).

To understand why the minimum has to be taken, note that w.r.t task T_i $(h \leq i \leq n)$ any scaling factor value less than sf^i will still continue to be valid. However, w.r.t some other task T_j $(h \leq i \leq n)$, where $sf^j < sf^i$, sf^i will not serve as a valid scaling factor. Observe that sf^j will surely serve as a valid scaling factor w.r.t both T_i and T_j. If we generalize this observation, it is clear why the minimum is the required solution.

The complete algorithm to compute maximal scaling factor is given below.

```
1 Algorithm Scale_Factor(T, S)

Parameters: $T$ is the given task-set which is schedulable. $S$ is the
sub-set whose scaling factor is desired. $S$ is assumed to be sorted in
the increasing order of their priorities. Assume that $T_h$ is the highest
priority task in the sub-set $S$.
Step 1: For $(i = h; i \leq n; i++)$
    Step 1.1: Compute first approximation for the completion time
    of task $T_i$'s first job:
        $compl_0 = \sum_{j=1 to i} e_j$
    Step 1.2: Calculate the next approximation for completion time:
        $compl_{t+1} = e_i + \sum_{j=1 \; to \; i-1} \lceil \frac{compl_t}{p_j} \rceil e_j$
    Step 1.3: If $(compl_{t+1} > d_i)$ then The job missed its deadline:
    Exit(-1);
```

Step 1.4: **If** $(compl_{t+1} \neq compl_t)$ **then** we have not arrived at the completion time of the task, so, **goto Step 1.2**;

Step 1.5: The completion time for the job is $compl_t$;

Step 1.6: Fit higher priority task instances that would arrive between the points $compl_t$ and d_i. The scheduling points are $U_{2L}, U_{3L}, \ldots, U_{kL}$[4]. where, $U_m = U_{mR} - U_{mL}$ denotes the m^{th} used time block (refer to Figure 5.1). Further we identify each used block as a sequence of marked and unmarked sub-blocks where a sub-block of block U_m is marked (referred to as U_m^j, if it is the j'th marked sub_block of U_m) if it belongs to the sub-set S and if its priority is greater than that of task T_i. unmarked otherwise

Step 1.7: Compute the *maximum* possible scaling factor sf^i:

$$sf^i = \max_{1 \leq m \leq k-1} sf_m$$

where

$$sf_m = \frac{\sum\limits_{1 \leq j \leq m} \left((\sum\limits_{\forall l} U_j^l) + (U_{j+1,L} - U_{j,R}) \right)}{\sum\limits_{1 \leq j \leq m} \sum\limits_{\forall l} U_j^l}$$

Step 2: sf = Minimum (sf^i) $\forall i$

Step 3: sf is the required maximal factor.

[4] U_k is the used block of time that overlaps with the deadline d_i. However, if the last block of used time, U_k, does not overlap with the deadline d_i, i.e., $d_i < U_{k,L}$ then we set $U_{k,L} = d_i$

end

The fact that the above algorithm returns an maximal scaling factor is confirmed by the following proof:

5.4 Proof for the Presented Solution

Following are the observations about the Solution that would help us prove that the derived scaling factor is in fact maximal.

- There is no idle time in the interval $[U_{1,L}, U_{1,R}]$, because if there were any idle time then it would have been used by T_i resulting in T_i completing before the point $U_{1,R}$.
- Blocks of execution $U_i (i \geq 2)$ belong to only higher priority tasks. This is true because we have not taken any lower priority tasks into consideration here.
- The scaling factor we are trying to find for task T_i only guarantees that the task T_i will meet its deadline, by using the processor at times when its free (i.e., not executing any higher priority tasks). It is possible that the scaling factor derived can cause a higher priority task to miss its deadline. However, if a higher priority task does miss its deadline, it is not because of task T_i but due to its own execution time and the execution times of tasks higher than itself being scaled (this point is explained using an example later).

To see the effect of scaling the tasks by a factor, we look at the first scaling factor considered, namely $sf_1 = \frac{(\sum_{\forall l} U_1^l)+(U_{2,L}-U_{1,R})}{(\sum_{\forall l} U_1^l)}$ (refer to Figure 5.2).

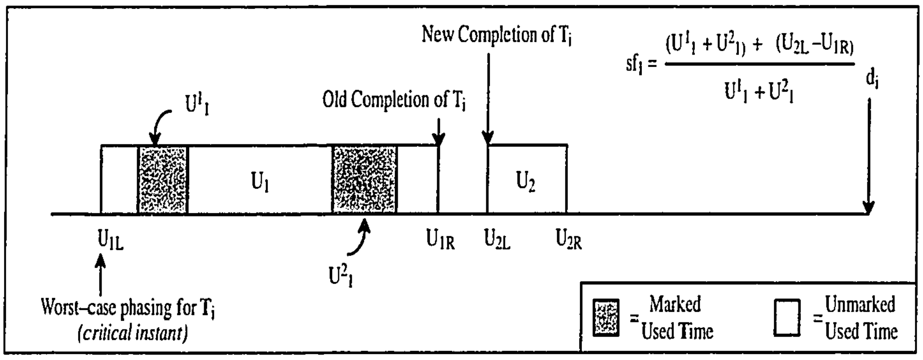

Figure 5.2: Effect of Scaling by sf_1

Since, this scaling does not affect the periods of tasks, if there were I_j instances of a task $T_j(j \leq i)$ in the interval $(U_{1,L}, U_{2,L})$ before the scaling, there will still be the same number of instances and further they will arrive at the same points as before.

$$U_1 = U_{1,R} - U_{1,L} = \sum_{1 \leq j \leq i} I_j \times e_j \;\; where \;\; I_i = 1$$

The processing requirement of task $T_j(\forall j \leq i)$, after scaling would become $e'_j = sf_1 \times e_j$, if $(j \in S)$, or $e'_j = e_j$, if $(j \notin S)$. However, So long as the following condition holds true, task T_i would complete by $U_{2,L}$:

$$\sum_{1 \leq j \leq i} (I_j \times e'_j) \;\; = U_{2,L}$$

We can confirm that this is in fact true:

$$\begin{aligned}\sum_{1 \leq j \leq i} (I_j \times e'_j) &= \sum_{(1 \leq j \leq i) \& (j \in S)} (I_j \times sf_1 \times e_j) \\ &\quad + \sum_{(1 \leq j \leq i) \& (j \notin S)} (I_j \times e_j) \\ &= sf_1 \times \sum_{(1 \leq j \leq i) \& (j \in S)} (I_j \times e_j) \\ &\quad + \sum_{(1 \leq j \leq i) \& (j \notin S)} (I_j \times e_j)\end{aligned}$$

$$
\begin{aligned}
&= \frac{(\sum_{\forall l} U_1^l) + U_{2,L} - U_{1,R}}{(\sum_{\forall l} U_1^l\ )} \times (\sum_{\forall l} U_1^l) \\
&\quad + \sum_{(1 \le j \le i) \& (j \notin S)} (I_j \times e_j) \\
&= (\sum_{\forall l} U_1^l\ ) + U_{2,L} - U_{1,R} \\
&\quad + \sum_{(1 \le j \le i) \& (j \notin S)} (I_j \times e_j) \\
&= U_{1,R} + U_{2,L} - U_{1,R} \\
&= U_{2,L}
\end{aligned}
$$

While the above shows that the scaling factor is valid in the sense that it moves (forwards) the completion time of T_i to the point $U_{2,L}$ (this argument can be extended to show that a factor sf_m will forward the completion time of task T_i at most till $U_{m+1,L}$), it does not necessarily guarantee to be the maximal scaling factor. The maximal scaling factor is the largest such scaling factor among all sf_m where $1 \le m \le k$. In order to see why this is so, observe that any factor sf_m would result in the task T_i finishing before its deadline, therefore all sf_m are valid, however, the one that is the largest (say $sf_{max} = sf^i$) is the desired result. To see why this is maximal, we note that any larger a factor would result in T_i finishing beyond $U_{max+1,L}$ and any smaller would leave more room for scaling the tasks in question.

Observe that computation of scaling factor w.r.t task T_i only guarantees that T_i will meet its deadline honoring the processing requirements of all higher priority tasks. The scaling factor thus obtained does not guarantee against higher priority tasks missing their deadlines. If any higher priority task misses its deadline as a result of this scaling, then obviously, it would miss its deadline in spite of T_i and therefore a prior scaling factor would prevail (an example below demonstrates this). In this way we compute the scaling factor sf^i. We perform this computation for all values of i from h to n and find the minimum of them, which is the desired final result.

5.5 Examples Demonstrating the Solution

The following examples demonstrate the various aspects of the technique. The first example involves three tasks, whose characteristics are given in Table 5.1 and the sub-set S has only one task $\{T_2\}$.

Table 5.1: Example Task Table

Task Id	Period	Exec Time	Deadline
1	100	40	100
2	150	40	150
3	350	35	280

Figure 5.3 gives a pictorial description of how the technique works on this example. The required maximal scaling factor is 1.5625. There are a few important points to note, that are not evident through this example. The next two examples are used to show these.

The second example demonstrates that, when computing the scaling factor w.r.t a given task T_i, it is not necessarily true that the last of the computed scaling factors, viz., sf_k is the maximum of all sf_m. To see an example of this case, consider the following task-set:

The task-set T has two tasks and the sub-set S contains both of them. The computation of sf^2 would be $Max(100/80 = 1.25, 145/120 = 1.20) = 1.25$.

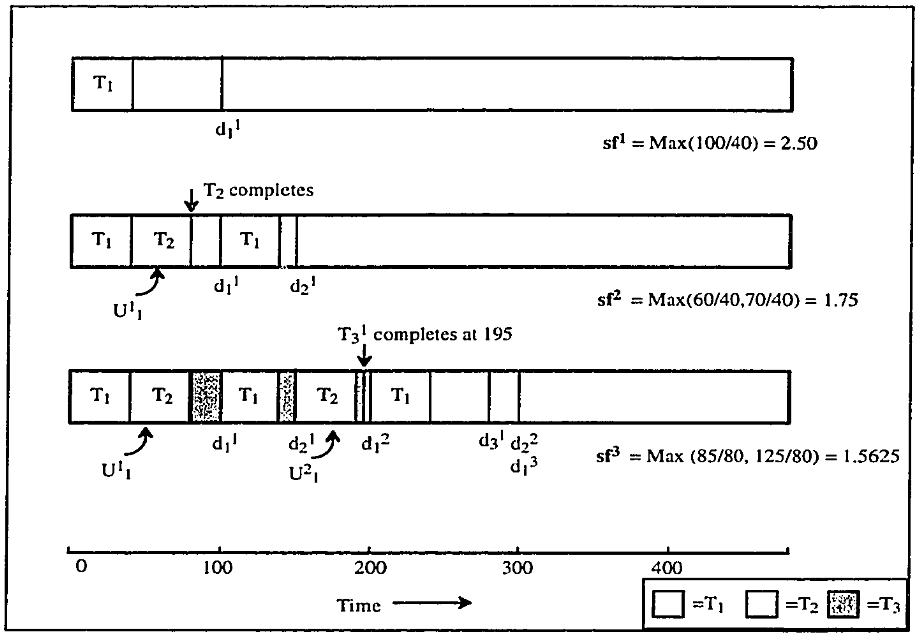

Figure 5.3: Operation of Task Set in Example 1

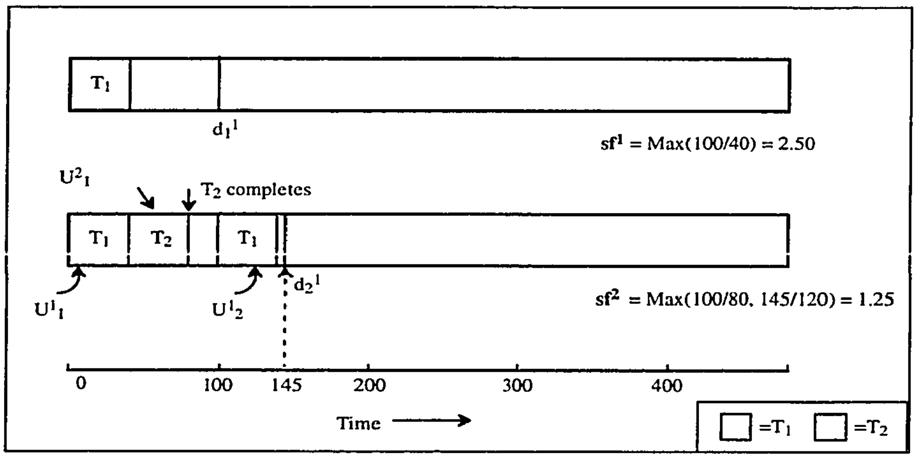

Figure 5.4: Operation of Task Set in Example 2

Table 5.2: Example 2

Task Id	Period	Exec Time	Deadline
1	100	40	100
2	150	40	145

Note that the scaling factor sf^2 is not determined by the second sf_2 (last) computed scaling factor but by that factor which is the maximum. In this case the factor sf_1. This same variation on the example also gives us a case for the point we made before, i.e., when computing the scaling factor w.r.t task T_i, the maximum of all the factors $sf_m, 1 \leq m \leq k$ is to be taken. Clearly, if we were to take sf^2 to be 1.20 (145/120) rather than 1.25 then there would still be some room for scaling.

In example 1, we see that the scaling factors are decreasing as we go form task $i = 2$ ($sf^2 = 1.75$) to task $i = 3$ ($sf^3 = 1.5625$). This however, is not true in general. A simple variation on the example will show us why. Consider the task-set in Table 5.3 with $S = \{T_2\}$.

Table 5.3: Example 3

Task Id	Period	Exec Time	Deadline
1	100	40	100
2	150	40	150
3	350	35	300

We now have $sf^3 = Max(85/80, 145/80) = 1.8125$. Thus illustrating that the scaling factors don't have to follow a decreasing trend as we add more tasks. This example also illustrates that the desired maximal scaling factor is the minimum sf^i, i.e., 1.75 and not 1.8125.

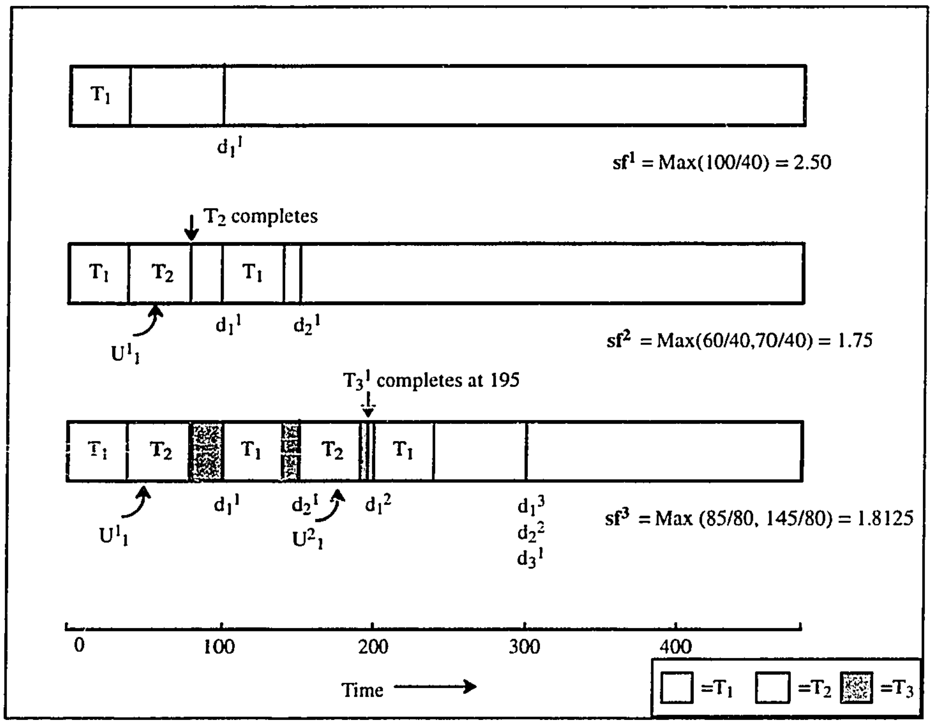

Figure 5.5: Operation of Task Set in Example 3

Chapter 6

Schedulability of Task-Sets with Arrivals

The source of this problem as discussed in Chapter 4 stems from the first stage of solving the end-to-end schedulability problem. To recall, the problem of interest here is the schedulability of tasks which have end-to-end schedulability constraints, i.e., a task is a sequence of sub-tasks that execute on independent components. However, the task as a whole has an arrival time, period specification and a deadline requirement.

We showed in Chapter 4 that a solution to the problem of end-to-end schedulability (and subsequently scalability) requires that we are able to solve the single component schedulability of a set of tasks whose arrival times are non-zero. The reduction was facilitated by an important transformation, viz., **phase adjustment.** Phase adjustment is a technique that allows us to derive the parameters of arrival and periodicity of sub-tasks of a task. The principle behind the technique was briefly described in Chapter 4, a more detailed description follows

6.1 Phase Adjustment

Clearly, the parameters of arrival and periodicity of the first subtask of any task are known a priori (inherited from the task). However, subsequent sub-tasks

T_{ij} ($j > 1$) of task T_i are not necessarily periodic in nature. Therefore, their arrivals and their periodicities do not correspond directly to that of the tasks. We have to account for this unpredictability in timing behavior of sub-tasks following the first sub-task. The first sub-task T_{i1} has the same periodicity as the original task T_i, therefore, it always arrives (or is ready to execute) at the start of the period p_i. However, the subsequent sub-task arrival times are dependent upon the completion time of the previous sub-task, i.e., if $T_{ij} \rightarrow T_{i,j+1}, j \geq 1$, then the arrival time of a particular instance l of $T_{i,j+1}$ is dependent upon the completion time of the l^{th} instance of sub-task T_{ij}. Further, the completion time of a sub-task instance is a function of its priority among the other ready tasks on the component. Therefore, we observe that there is a dependency between successive sub-tasks that has to be taken care of.

Phase adjustment is a mechanism that allows us to remove this dependence. Since a_i is the arrival time or phase of task T_i, sub-task T_{i1} inherits the phase of the task T_i, i.e, $a_{i1} = a_i$. The l^{th} job instance of sub-task T_{i1} arrives at $a_i + (l - 1)p_i$. Let the worst-case completion time (or response time) of sub-task T_{i1} be WC_{i1}, i.e., any instance of T_{i1} (call it the l^{th}) would complete no later than $a_{i1} + (l - 1)p_i + WC_{i1}$. We use the term WC_{i1} to adjust the phase of the next sub-task T_{i2}. Therefore, the phase of T_{i2}, a_{i2} is given by $a_{i1} + WC_{i1}$. This also guarantees that consecutive instances of subtask T_{i2} will repeat periodically at an interval of p_i time units.

This can be further generalized to find the phase $a_{i,j}$,of the sub-task T_{ij} as $a_{i,j-1} + WC_{i,j-1}$. Also all sub-tasks of task T_i are now guaranteed to directly inherit its period. We have the following a recurrence relation that captures the arrival time of a sub-task T_{ij}:

$$a_{ij} = a_{i,j-1} + WC_{i,j-1}$$

In order to find a closed-form solution to this recurrence relation we have to know the base values, a_{i1} and WC_{i1}. We already showed how to obtain a_{i1} from a_i. The worst-case completion time WC_{i1} of sub-task T_{i1} can be obtained if we solve the problem of schedulability of tasks running on the component that T_{i1} runs. A solution to this problem is the subject of this chapter. Assuming for now that we do have a solution to this problem and hence are able to find the worst-case completion time of T_{i1}, we complete the requirements to convert this to the following closed form:

$$a_{ij} = a_i + \sum_{l=1}^{j-1} WC_{il}$$

Having obtained the value of WC_{i1} we can now use it to find the arrival time and hence the worst-case completion time of task T_{i2}. Again, we are assuming that we know how to compute the worst-case completion times of subtasks running on a single component with non-zero arrivals. Note that the schedulability test for the end-to-end task T_i would now be a trivial comparison: *if* $\sum_{1\leq j\leq r} WC_{ij} \leq D_i$ *then the task* T_i *is schedulable.*

In the above discussion we have assumed that we have a mechanism that computes the worst-case completion times of subtasks given their arrival times, periodicities and priorities. This is the subject of the following discussion.

6.2 Problem Statement and Solution

We recall from Chapter 4, the formal statement of the problem (Problem 4.2.2) of interest to us here:

Given a task-set T of n tasks executing on a single component, find the worst-case completion times of all tasks in the task-set.

The solution to this problem is based on the following observations:

1. Is there a period L for the task-set such that, looking at the behavior of a task T_i during the interval a_i and L is sufficient to determine the worst-case response time of the task T_i? Note that, if $a_i = 0, \forall i$, then L is given by the LCM of the task periods. The worst-case response time of a task T_i is the maximum response time of all instances of T_i in this interval.

2. For arbitrary arrival phasings of tasks, the repetition of the initial phasing pattern[1] occurs at a point L units later (where L is given by the LCM of the periods). The state of the scheduler (defined later) is not the same at these two points. Therefore, repetition of phasing pattern does not necessarily guarantee that the task-set behavior will repeat itself.

3. If the task arrival times are *inverse monotonically increasing* with the priority, i.e., the highest priority task is the earliest to arrive ($a_i < a_j$ if $i < j$), then the repetition of the phasing pattern is an indication that the task-set would repeat its behavior.

[1]The phasing pattern is the relative arrivals of the various tasks under consideration

4. Given an arbitrary task phasing a, we can derive an alternate phasing a' which has the characteristic that the arrival times monotonically increase with the priority. Further, this phasing can be used to determine the worst-case response time of the tasks in task-set.

The following theorem is the basis for the approach.

Theorem 6.2.1 *: Given that the arrival times of tasks in a task-set are inverse monotonic with priority ($a_i \leq a_j$ if priority of T_i is greater than priority of T_j, i.e, $j > i$), the worst-case response time instance of a task T_i belongs to the interval $[a_i, a_i + LCM(T_1, \ldots, T_i)]$.*

Proof. For task, T_i, the only tasks that it would have to compete with, are the higher priority tasks $T_1, T_2, \ldots, T_i$. We are therefore interested in finding that point in time at which, the phasing of task T_i (given by $a_i + x_i \times p_i$, for the x_i^{th} instance) with respect to other higher priority tasks is same as that at time a_i. Further, this point must be such that the state of the scheduler must be same as it was at a_i.

The relative phasing of task T_i with respect to the task T_1 can be captured as: Task T_i comes $a_i - a_1$ units of time after task T_1. Assuming the existence of a point where this phasing repeats, and further that there are x_1 and x_i instances respectively of T_1 and T_i before this point, we have the following condition:

$$(a_i + x_i \times p_i) - (a_1 + x_1 \times p_1) = a_i - a_1$$

$$\Rightarrow x_1 \times p_1 = x_i \times p_i$$

We can derive similar conditions for task T_i w.r.t other tasks. The resultant condition is:

$$x_1 \times p_1 = x_2 \times p_2 = \ldots = x_i \times p_i = L$$

where $a_i + L$ is the desired point. Clearly the LCM of p_j^s is solution for the above equation if we assume integral values of p_i.

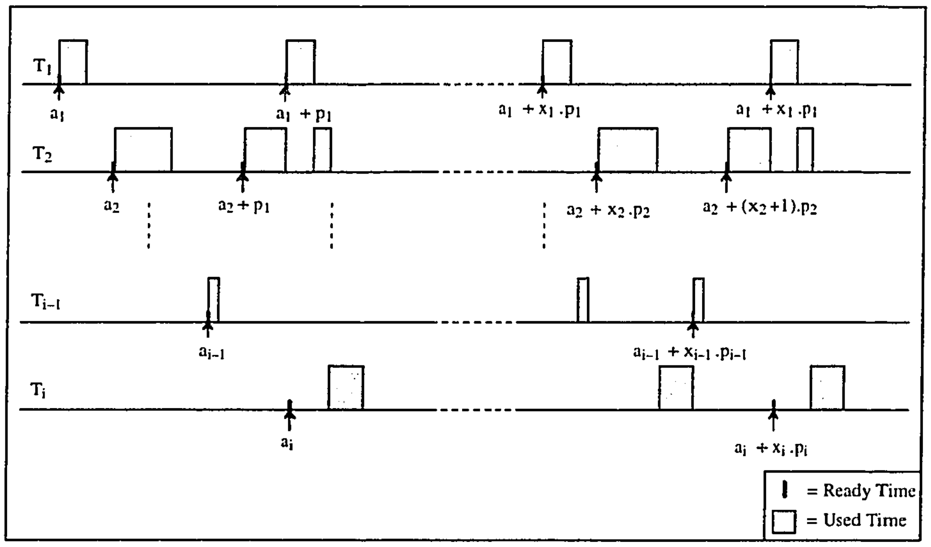

Figure 6.1: A task-set's execution between the start and L

Next, we have to show that the state of the scheduler with respect to the task T_i is the same at both points a_i and $a_i + L$. We use the method of mathematical induction to show this.

Definition 6.2.1 : *The state of the scheduler w.r.t task T_i at the time of arrival of the k'th instance of task T_i is given by $S_i^k = \{S_{1,i}^k, \ldots, S_{i-1,i}^k, S_{i,i}^k\}$. The term $S_{j,i}^k$, is the amount of time that the task T_j executed for, before the point $a_i + (k-1) \times p_i$ and since its first invocation (taken modulo its period).*

Note that, since the state of the lower priority tasks $T_j, j > i$ do not affect the schedulability of the task T_i, they don't figure in the system state with respect to task T_i. Also note that, we are concerned only with the state of scheduler at points that are arrival times of tasks because we seek to show that the state at these points repeats. Further, $S^k_{i,i} = 0$, because, the last invocation of task T_i, viz., $k-1$'th should have completed before the arrival of the k'th instance (otherwise we would have declared that the task missed its deadline and thats the end of it).

Basis: Consider the point $a_i + L$ where we have already shown that the phasing of the task T_i is the same as it was at a_i. The highest priority task, T_1 has an arrival at $a_1 + x_1 \times p_1$ and acquires the processor (being the highest priority ready task). The duration of time between this completion and the arrival of the task, T_2 at $a_2 + x_2 \times p_2$ (refer to Figure 6.1), can be used by any of the tasks $T_j (2 \leq j \leq i)$. Note that this same duration at the beginning, i.e., when the first instance of T_1 completes and the first instance of T_2 is ready was necessarily idle. However, the state of the processor *with respect to task* T_2 at the point $a_2 + x_2 \times p_2$ is exactly the same as at the point a_2, because, the lower priority tasks would not affect the completion times of the instances of task T_1 and further the latest instance of task T_2 would have completed. Therefore, the state of the scheduler w.r.t task T_2 at the point $a_2 + x_2 \times p_2$ is same as it was at a_2, viz., $\{S^1_{1,2}, S^1_{2,2}\}$.

Inductive Hypothesis: Let us assume that the result holds for the $i-1$'th task, i.e., the state of the scheduler w.r.t. task T_{i-1} at $a_{i-1} + x_{i-1} \times p_{i-1}$, viz.,$\{S^{x_{i-1}}_{1,i-1}, \ldots S^{x_{i-1}}_{i-2,i-1}, S^{x_{i-1}}_{i-1,i-1}\}$ as it was at a_{i-1}, viz., $\{S^1_{1,i-1}, \ldots S^1_{i-2,i-1}, S^1_{i-1,i-1}\}$. Note that between the points $a_{i-1} + x_{i-1} \times p_{i-1}$ and $a_i + x_i \times p_i$, the number of tasks of *priority higher than* task T_i that would arrive are the same as in the interval a_{i-1} and a_i. Further, since the task T_i has to complete before its deadline (which is less than its period; If it does not then we just report so and thats the end of it), each of the higher priority tasks would have gotten the same amount of execution time in these two intervals, implying that:

$$S^{x_i}_{j,i} - S^{x_{i-1}}_{j,i-1} = S^1_{j,i} - S^1_{j,i-1} \quad \forall j \leq i-1$$

Note that, when $j = i-1$, the terms $S^{x_{i-1}}_{i-1,i-1}$ and $S^1_{i-1,i-1}$ are both 0. Now, since the result we are trying to prove, holds for the task T_{i-1}, we have:

$$S^{x_{i-1}}_{j,i-1} = S^1_{j,i-1} \quad \forall j \leq i-1$$

Therefore,

$$S^{x_i}_{j,i} = S^1_{j,i} \quad \forall j \leq i \quad ^2$$

Which implies that the state of the scheduler at the point $a_i + x_i \times p_i$ is the same as that at a_i. Therefore, the result holds for the task T_i. □

Having shown that both, the phasing repeats after L units of time and also that the state of the scheduler is same when this repetition occurs, the result follows.□

In deriving the result in theorem 6.2.1 we have assumed that the arrival times of tasks are such that the highest priority task is the earliest to arrive and the arrival times increase with priority. However, in reality, this assumption restricts the practicality of the result. In the following, we describe a mechanism by which we can get rid of this assumption without hurting the result.

[2]The last term when $j = i$ has been conveniently added because $S^{x_{i-1}}_{i-1,i-1} = S^1_{i-1,i-1} = 0$.

Given an arbitrary arrival phasing of tasks the following algorithm converts it into an alternate phasing where the arrival times increase with the priorities.

2 *Algorithm* **Arb_to_Incr**

```
Parameters: $a_1, a_2, \ldots a_n$, and $p_1, p_2, \ldots p_n$ the arrival times and periodicities of tasks $T_1, T_2, \ldots T_n$ respectively.
Result: $a'_1, a'_2, \ldots, a'_n$,
Initialize: $a' = a$;
The first task arrival is unchanged.
for ($i = 2$ to $n$) do
    If ($a_i < a'_{i-1}$)
        y = 1; while ($a_i + y \times p_i < a'_{i-1}$)
            y ← y+1;
        end
        $a'_i \leftarrow a_i + y \times p_i$;
    end
end

end
```

We take an example to demonstrate the operation of the above algorithm. Consider a task-set with four tasks (T_1, T_2, T_3, T_4), with the following values for arrival times and periodicities: $(a_1 = 5, a_2 = 3, a_3 = 4, a_4 = 0)$, $(p_1 = 10, p_2 = 10, p_3 = 16, p_4 = 12)$. The first task's arrival time remains unchanged, however since the task T_2's arrival is before T_1's, its new arrival time,

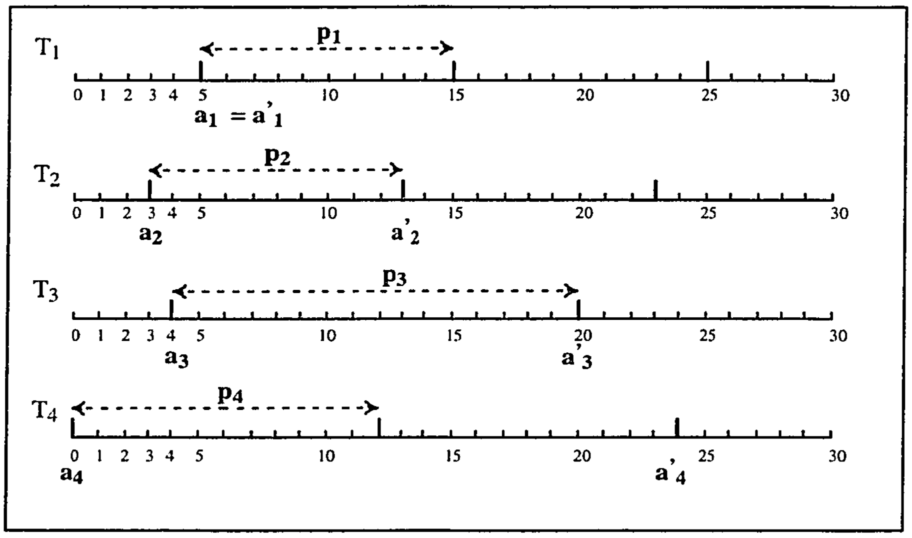

Figure 6.2: A task-set's execution between the start and L

a'_2, is computed to be $a_2 + p_2$ which is 13. Now task T_3's arrival time $a_3 = 4$ is less than $a'_2 = 13$, therefore its new arrival time a'_3 is $a_3 + p_3$ which is 20. Task T_4 arrives at $a_4 = 0$ which is less than $a'_3 = 20$, therefore its new arrival time a'_4 is $a_4 + 2 \times p_4$ which is 24. Now the new arrival times of the tasks in the task-set are $(a'_1 = 5, a'_2 = 13, a'_3 = 20, a'_4 = 24)$.

Before we discuss the mechanism in detail, it is important to ascertain the relationship between the original arrival phasing and the modified arrival phasing. Since, the modified arrival pattern guarantees the repetition of the task-set behavior, in order to find the worst-case response time of any task, we only have to look for its instances between its original arrival time and the point at which the new phasing repeats itself.

The algorithm for the complete mechanism follows:

3 *Algorithm*

```
Parameters: $a_1, a_2, \ldots a_n$, and $p_1, p_2, \ldots p_n$ the arrival times and periodicities of tasks $T_1, T_2, \ldots T_n$ respectively.
    –Find the modified arrival times, $a'$, for tasks by invoking the procedure **Arb_to_Incr**.
    –Repeat for each task $T_i$ in turn:
        ∗Determine if the task meets all its deadlines assuming a worst-case phasing (i.e., ignoring arrivals). If it does not then, report so and **QUIT**.
        ∗Find the completion time of all task instances of $T_i$ occurring during the interval $a_i$ and $a'_i + LCM\{T_j, j \leq i\}$.
        ∗Find the maximum and report it as the worst-case response time of the task $T_i$.
end
```

6.3 Example Demonstrating the Solution

Consider the task-set in Table 6.1 below.

Table 6.1: Example task-set

Task Prio	Arrival	Period	Exec Time	Deadline	WC^c	WC^r
1	0	2	12	12	2	2
2	2	4	24	24	6	4
3	3	3	16	16	9	6
4	5	4	24	24	15	10

The computation of the response times of tasks in this task-set using the mechanism described above is given in Figure 6.3. In the Table 6.1, the last two columns give respectively the worst-case response times of the tasks using the critical instant assumption and our approach. It is clear that the critical instant assumption has resulted in the computation of a higher response time in the case of the last three tasks. In order to best appreciate the merit of finding the worst-case completion time of a task using the precise test described in this chapter as opposed to using the critical instant assumption, we introduce a couple of new measures of comparison, viz., apparent slack and slack savings.

Note that though both these worst-case response times are still within the deadline bound, there is a difference in the apparent slack of tasks. We define the apparent slack of a task[3] to be the difference between the deadline of a task and its worst-case completion time. Note that a positive apparent slack for a task guarantees its meeting its deadline. However, a larger apparent slack signifies that a task is more capable with regards to, accommodating task interdependence (eg., precedence), withstanding temporary overloads, accommodating aperiodics in the system, restricting jitter in end-to-end systems.

We define a measure called the *slack savings*, ss_i for a task T_i as the ratio of the gain in the apparent slack per unit real slack:

$$ss_i = \frac{WC_i^c - WC_i^r}{d_i - e_i}$$

Note that in the above example we have chosen the deadlines of tasks to be equal to their periods. However, in general the deadlines can be less than or greater than the periods. Also note that the example shows that the arrival times are monotonic with priority, however, this need not be the case in general.

[3]as opposed to the original slack of the task which is independent of other tasks it has to compete with and is defined to be the difference between the deadline and the execution time of the task

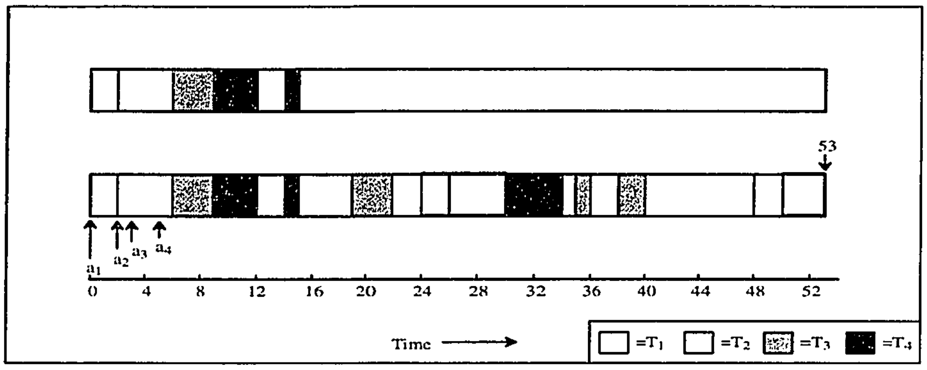

Figure 6.3: Operation of example task set

In the above example we achieve the following slack savings: $ss_1 = 0\%$, $ss_2 = 10\%$, $ss_3 = 23\%$, $ss_4 = 25\%$.

6.4 Discussion of the Result

The reader will observe that the above treatment of the end-to-end schedulability problem assumes that all tasks access the components in the same order. This scenario is similar to the classical periodic flow shop model [4]. However, as will become clear in the following discussion, this assumption can be relaxed.

We consider the following scenarios for the order in which the tasks use the various components:

- *Case 1: Periodic flow shops* [15, 4]: All tasks execute on the same components in the same order. Every task T_i has exactly r (the number of components) sub-tasks, $T_{i1} \rightarrow T_{i2} \rightarrow, \ldots, \rightarrow T_{ir}$, where each sub-task T_{ij} executes on component j. This case is a special instance of the next case, however, we treat it separately because of its practical significance.

- *Case 2:* The use of components by the tasks are ordered but the tasks

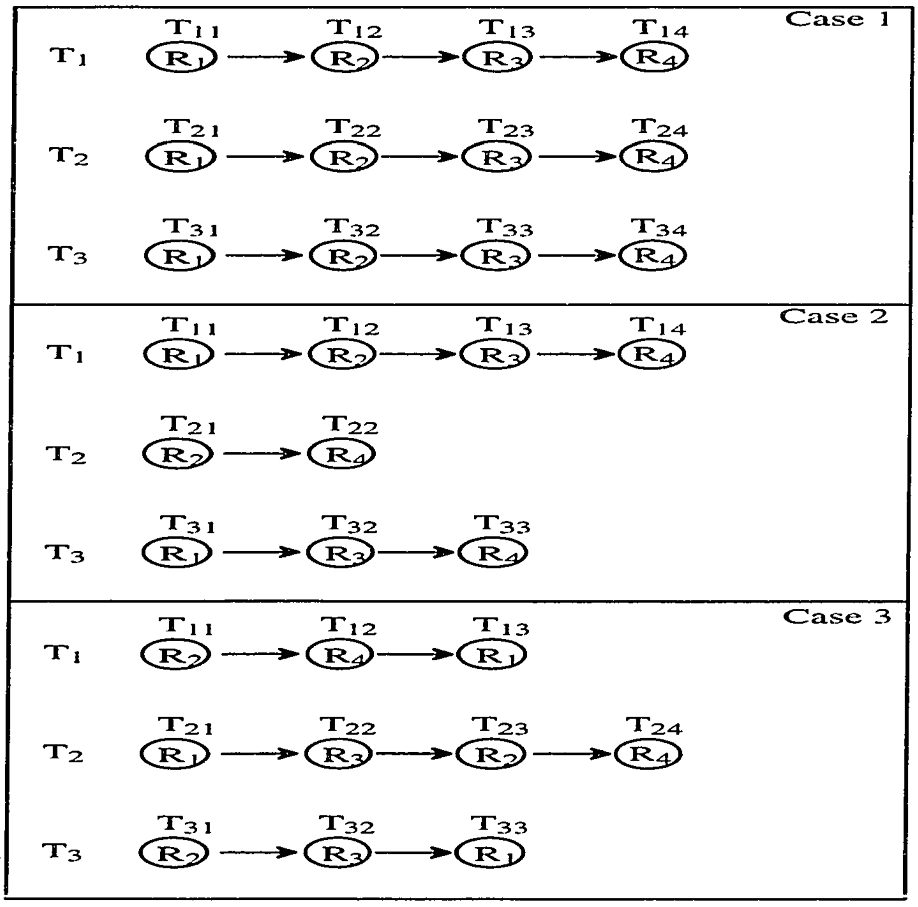
Case 1
T1
T11 R1 T12 R2 T13 R3 T14 R4
T2
T21 R1 T22 R2 T23 R3 T24 R4
T3
T31 R1 T32 R2 T33 R3 T34 R4
Case 2
T1
T11 R1 T12 R2 T13 R3 T14 R4
T2
T21 R2 T22 R4
T3
T31 R1 T32 R3 T33 R4
Case 3
T1
T11 R2 T12 R4 T13 R1
T2
T21 R1 T22 R3 T23 R2 T24 R4
T3
T31 R2 T32 R3 T33 R1

Figure 6.4: End-to-End scenarios

don't necessarily access the same resources: We assume that any two tasks T_i, T_j that have subtasks that run on two components R_k, R_l, do so in the same order in both tasks. Further, the component used by a sub-task T_{ij} is not used by any other sub-task of T_i.

- *Case 3: Arbitrary order flow shops*: The order in which the components are used by a task can be arbitrary (as opposed to ordered access in Case 2). There are two possibilities under this case, one which disallows components from being reused and the second where components are allowed to be reused.

6.4.1 Periodic flow shops

If we assume an ordering of the r components in use to be in numerical order then the function $Res(T_{ij})$ can be taken to be equal to j (i.e., the j^{th} component). Therefore we now have n tasks with each task consisting of r subtasks where the j^{th} subtask of every task T_i ($\forall i$) runs on component number j.

In order to determine the schedulability of the task-set we have to study the schedulability of each task in turn. Phase adjustment guarantees that subtasks of a task inherit its periodicity and further they are independent of each other. Therefore the schedulability test for a task T_i is given by:

$$a_i + \sum_{j=1}^{r} WC_{ij} \leq D_i$$

Where, WC_{ij} is the worst-case completion time of the subtask T_{ij}. In order to find the worst-case completion of the subtask T_{ij} which runs on the component j we have to consider all the subtasks that run on the component

j. Starting from $j = 1$, we see that for the first component we know the arrival times, periods and priorities (note that the subtask priority can be explicitly given or the subtask can inherit the original task's priority) of all subtasks that run on it. That is, for a given subtask T_{i1}, its arrival time is a_i and its period is p_i. We can find its worst-case completion time by the mechanism described in the previous section. Let WC_{i1} be the worst-case completion time thus determined (Note that this worst-case completion time is the time taken by the subtask to complete after its arrival).

We now fix the arrival time of the second subtask of T_i (i.e., T_{i2}) as $a_i + WC_{i1}$. This fixing of the arrival is a result of the phase adjustment mechanism described before. Further it ensures that the second subtask will be periodic with period, p_i. We now know all the parameters (viz., arrival phasing, period and priority) we need to determine the worst-case completion time of the second subtasks of all the tasks. Knowing the value of WC_{i2} ($\forall i$) we are able to find the timing parameters of the third subtasks and so on.

6.4.2 Ordered Access

This a more general case than the periodic flow shop case in that tasks don't necessarily access all the components. However, when they do access a particular component its relative order with respect to other components is honored in all tasks. We once again assume that the components are numbered in order. We do the following modification to the formal specification of this case:

We assume that each task T_i is a sequence of subtasks $T_{i1} \rightarrow T_{i2} \rightarrow, \ldots, \rightarrow T_{ir}$. However, if the task T_i does not have a subtask running on component j then the corresponding subtask T_{ij} has an execution time, $e_{ij} = 0$.

By specifying the model in this way we are able to treat this case similar to the previous case. However, note that for all subtasks that have a zero execution time their worst-case completion times are also zero.

6.4.3 Arbitrary order with no revisit

The major problem with this ordering scenario is that it is not always possible to find the timing parameters of all subtasks that run on a particular component. For example in Figure 6.4 we see that task T_1 uses the components in the order $R2 \rightarrow R4 \rightarrow R1$, task T_2 uses components $R1 \rightarrow R3 \rightarrow R2 \rightarrow R4$ and task T_3 uses components $R2 \rightarrow R3 \rightarrow R1$. Determining the parameters (mainly arrival times) of subtasks that run on component $R1$, $T_{1,3}, T_{2,1}, T_{33}$ involves finding the worst-case completion times of subtasks $T_{1,1}, T_{1,2}, T_{3,1} and T_{3,2}$. It can be seen that this is not possible without addressing the schedulability on the components $R2, R3 and R4$.

An alternative approach to this case would be to ignore the arrival information of tasks (and subtasks). Note however that the penalty of ignoring arrival information is that we end up doing a pessimistic schedulability analysis.

Chapter 7

Scalability in End-to-End Systems

As shown in Chapter 4, the problem of scalability of tasks in end-to-end systems manifest itself in two forms, viz., (i) Changes to components and, (ii) Changes to Tasks. We have also shown that solving this problem in either of these two flavors reduces to solving the following two problems 4.2.4 and 4.2.5.

4.2.4: Given a task-set T of n tasks (with non-zero arrival times) executing on a single component, find the worst-case completion times of all tasks in the task-set.

4.2.5: Given a schedulable task-set T of n tasks executing on a single component and a subset S of T, find the maximum scaling factor sf with which all tasks in S can be scaled without violating the schedulability of any of the tasks in T.

The first of the two problems was the subject of the previous chapter. This chapter is devoted to presenting a solution to the second. As shown in the previous Chapter 4, the problem of schedulability in end-to-end systems can be reduced to a series of single component schedulability problems. However, the single component schedulability problem has to accommodate task arrival times. Similarly, the problem of scalability in end-to-end systems can be reduced to a series of single component scalability problems provided we accommodate task arrival information into the computation. Further, to find the scalability of a sub-task we have to know its deadline and also the deadlines of all other sub-tasks involved in its analysis. There is no straightforward way to derive the sub-task deadlines.

A major research issue in end-to-end scheduling has been the derivation of sub-task deadlines. Given an end-to-end task's deadline the problem of finding an optimal[1] division of this deadline among the sub-tasks is intractable [15] (NP-complete [12]). This result has prompted a heuristic approach [4, 15], two such heuristics being: (i) divide the task's slack[2] equally among the sub-tasks; (ii) divide the task's slack among its sub-tasks in a weighted proportion of their execution times;

The above two heuristics vary mainly in their sensitivity to the execution times of tasks. For example, the second heuristic is built on the assumption that the shorter a task's execution time requirement, the more likely it will have its requirement met and therefore the lower is the slack assigned to it. The first heuristic is built on the assumption that the priority inherited by a sub-task has a greater impact on its ability to meet its execution time requirement than its execution time itself. Thus the slack is divided equally among all sub-tasks. This allows us to reduce the end-to-end scalability problem to m single component scalability problems.

[1]In the sense that, if there exists a division that would help the task meet its deadline then the mechanism should find it

[2]The slack of a task is given by the difference between its deadline and its execution time

In our research, we have chosen the second heuristic because it is more general of the two. The intuition behind the heuristic is to divide the slack of the task proportionally among its sub-tasks. We can find the *total slack* of the task T_i as $sl_i = d_i - \sum_{\forall j} e_{ij}$. We divide this among the sub-tasks in the ratio of their execution times, e_{ij}. Therefore,

$$d_{ij} = e_{ij} + \frac{e_{ij}}{\sum_{\forall j} e_{ij}} \times sl_i$$

The following section describes a mechanism for finding the scaling factor that incorporates the arrivals of tasks. We also give an informal proof for its correctness. In order to simplify the presentation we assume that the scaling factor we desire is a common scaling factor for all tasks in the task-set. Note that the case of general scaling (sub-set scaling) can be easily derived on the same lines.

7.1 Problem Statement and Solution

As we did when we dealt with the problem of schedulability using arbitrary task arrivals in the previous chapter, we assume that the arrival times of task are in increasing order of their priorities. Therefore, the highest priority task T_1 is the first to arrive (time $t = 0$) and T_i arrives prior to T_j if $i < j$.

The procedure for finding the common scaling factor of a task set, proceeds on the same lines as that for arrival times being all equal to 0. We find the scaling factor sf^i, with respect to each task i ($1 \leq i \leq N$) and take the minimum as the required result. Any scaling factor sf^i has the following sense: This is the maximum factor by which all task execution times can be scaled such that the task T_i will meet its deadline while continuing to honor all higher priority task requirements (not necessarily their deadlines). However, the difference comes in the fact that when we are finding the scaling factor with respect to a particular task T_i, we no longer can settle by considering only one instance (the worst-case instance, which is the first instance using the critical instant argument) but we have to consider all instances of this task between the points a_i and $a_i + L$ (refer Chapter 6).

Following are some of the distinguishing characteristics of the problem when compared to the treatment in Chapter 5.

- It would seem that it is sufficient to consider the worst-case execution instance (of a task T_i) and apply the same technique as before (as in Chapter 4) to find the scaling factor. However, this is not true for the following reason: the scaling factor is determined by both the completion and the idle time left before the deadline; the worst case-completion of a task instance does not necessarily guarantee that the idle time left between its completion and its deadline after accommodating higher priority tasks is a minimum.

- The critical instant assumption, in addition to restricting our consideration to a single instance, has also allowed us to conveniently ignore any higher priority tasks that would arrive prior to task T_i's arrival. The possibility of the following scenario (refer to Figure 7.1) has to be taken into account for arbitrary arrivals: In computing the scaling factor for the first instance of task T_i, we cannot ignore the blocks of execution that precede the point a_i (i.e., $U_1, U_2, \ldots, U_{q-1}$). This is so because, it is likely that a factor computed ignoring these could cause the used blocks of execution before a_i to be scaled in such a way that the begin time of tasks in the block that a_i belongs to, could be affected by tasks other than the ones within. This results in the computed factor being invalid.

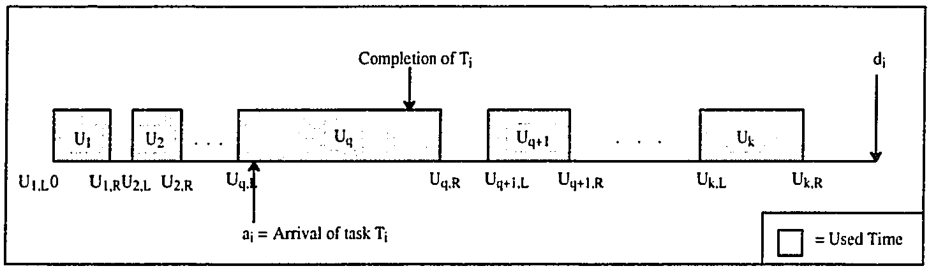

Figure 7.1: Execution Profile Task T_i's First Instance

We now discuss the mechanism along with an explanation of why the mechanism works. We are interested in computing the scaling factor sf^i, with respect to a particular task T_i. We once again note that this factor does not guarantee that all higher priority tasks would meet their deadlines, it only ensures that the task T_i will meet its deadline in spite of honoring the requirements of higher priority tasks.

Let us consider the first task instance of task T_i. Assume that there is only one block of execution before the arrival of task T_i at a_i and there are a number of blocks after the completion and before the deadline (refer to Figure 7.2). We are interested in stretching the deadline as far as possible while honoring the requirements of higher priority tasks. The only way this can be accomplished is by stretching the completion a step at a time with each step attempting to consume the next available idle time (Refer to Chapter 5 for reasoning). The required result (the scaling factor with respect to this instance of task T_i) would then be obtained by taking the maximum (because all these factors are valid and we are interested in finding the optimal one) among such computed factors.

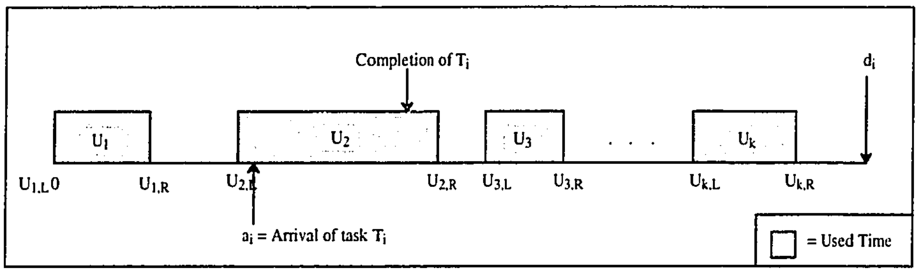

Figure 7.2: Figure 7.1 assuming $q = 2$

We now look at how we can stretch the completion time to achieve the motive described above. Since we assumed that there is only one block of execution (obviously comprises of at least one instance of every higher priority task), following are the points to note while trying to stretch the completion time of the first instance of the task T_i to consume the first slot of idle time:

- If we ignore the block of execution before the arrival of task T_i then the scaling factor would be $f = \frac{U_{3,L} - U_{2,L}}{U_2}$. However, it is possible that this factor could result in the ignored block (i.e., U_1) being scaled beyond the point $U_{2,L}$ (we call this the unfavorable event for this choice of scaling factor, **NFE1**), thus invalidating the factor. On the contrary, in the event that this scaling factor does not scale U_1 beyond the point $U_{2,L}$ (we call this the favorable event for this choice of scaling factor, **FE1**), this factor is clearly valid and effective in stretching the task completion time till $U_{3,L}$.

- If instead, we use the scaling factor to be $f' = \frac{U_{3,L} - U_{1,L}}{U_2 + U_1}$, it is possible that the resultant factor does not scale U_1 to occupy the whole of the idle time between $(U_{1,R}, U_{2,L})$, resulting in U_2 being stretched beyond $U_{3,L}$ and consequently the completion time being stretched beyond $U_{3,L}$ (we call this the unfavorable event for this choice of scaling factor **NFE2**). Note that this possibility has come up because the task T_i is not ready to use the idle time between $(U_{1,R}, U_{2,L})$. On the contrary, in the event that this factor causes U_1 to be scaled beyond the point $U_{2,L}$ (we call this the favorable event for this choice of scaling factor, **FE2**) then clearly the completion time of task T_i will be within $U_{3,L}$ (in fact it will be exactly $U_{3,L}$).

We note that there are two possibilities (or events) in favor of each of the above choices and two that are not in favor. However, we will show that the true answer lies in finding the minimum of these two possible factors. That is, picking the minimum of these two factors as the solution leads us to realize that the unfavorable possibility is actually not possible. An explanation follows:

We have two possibilities to consider:

- $f < f'$: The favorable event (**FE1**) corresponding to this choice of the factor is valid in giving us the desired result. However, we have to show that unfavorable event, **NFE1** will not occur. We show this by contradiction:

  Let us say U_1 gets scaled beyond the point $U_{2,L}$ (i.e, the event **NFE1** does occur). f', being the larger of the two, using it as the scaling factor would scale U_1 beyond $U_{2,L}$ too. But, since f' has been derived to stretch both

U_1 and U_2 over $(0, U_{3,L})$, if it does stretch U_1 into the start of U_2, then there would be no idle time between the points $(0, U_{3,L})$. This implies that $f' < f$ because, the bumped time[3], say δ $(= f' \times U_1 - U_{2,L})$ and the scaled U_2 $(= f' \times U_2 - U_2)$ together fitted within the interval between $(U_{2,R}, U_{3,L})$, whereas f scaled only U_2 to occupy the same interval. The conclusion that, $f' < f$ contradicts our assumption that f is the smaller of the two factors. Hence the result.

- $f > f'$: The favorable event (**FE2**) corresponding to this choice of the factor is valid in giving us the desired result. However, we have to show that unfavorable event, **NFE2** will not occur. We show this by contradiction:

  Lets say U_1 does not get scaled beyond the point $U_{2,L}$ when scaled by f' (i.e., the event **NFE2** does occur). Since, $f > f'$, U_2 does not go beyond $U_{3,L}$ when scaled by f'. However, the very definition of **NFE2** says that f stretches $U2$ beyond $U_{3,L}$. This is a contradiction. Hence the result.

Observe that the favorable events in both choices of scaling factors achieve the following: The completion time of the task T_i is stretched to the point $U_{3,L}$. We now extend this to the case that the number of blocks of execution prior to the arrival of the first instance task T_i is more than one. In fact, we wish to extend this argument to the case that there are $q - 1$ blocks of execution before the arrival of the first instance of T_i. The generalization is straightforward. If there is more than one block of execution then the scenario would be as in Figure 7.1. The scaling factor associated with stretching the

[3]the excess scaled time that was carried from scaling U_1 beyond the point $U_{2,L}$

completion time of the first instance of task T_i to consume the first idle interval beyond its completion would be given by:

$$F_q = Min\left(\frac{U_{q+1,L} - U_{1,L}}{\sum_{r=1\ to\ q} U_r}, \frac{U_{q+1,L} - U_{2,L}}{\sum_{r=2\ to\ q} U_r}, \ldots, \frac{U_{q+1,L} - U_{q,L}}{U_r}\right)$$

Where q is the index of the block that contains the arrival of the first instance of T_i (from the fact that there are $q - 1$ blocks of execution before its arrival). Note that this is also the index of the block that contains the completion of T_i, because, there cannot be any (processor) idle time between a task's arrival and its completion. We represent this factor by F_q to signify that this is the factor with which all T_j ($j <= i$) must be scaled to fill the first idle interval after the completion (known to overlap with the block U_q) of this instance (the first that is) of task T_i. The subscript q here is only to identify the block which overlaps with the completion of this instance of T_i. The representation will become clear when we proceed to the next stage of derivation, i.e., the scaling factor for an arbitrary instance of T_i (not just the first that is).

Now consider the point corresponding to the deadline of this instance of T_i, $a_i + d_i$. Our aim is to try to extend the completion of this instance at most till this point. Clearly, if this point overlaps with a used block (call it $U_{k+1,L}$), then we cannot possibly extend T_i^1's completion beyond the start of this interval. This is obvious from the fact that the overlapped block in question contains executions of higher priority tasks that cannot be preempted by T_i. On the other hand if the point in question does not overlap with any used block then we can consider filling only part of the idle interval that contains this point, viz., the idle interval between the right end of the used-block preceding the deadline point and the deadline point itself. In this second case, we set $U_{k+1,L} = a_i + d_i = U_{k+1,R}$, i.e., we create a zero sized used block that overlaps with the deadline. Here k is the index of the used-block that precedes the deadline.

Therefore, if we assume that there are $k - q$ such idle intervals beyond U_q and before the deadline of this instance at d_i^1 then we have to find $k - q$ such scaling factors F_m (that is $q \leq m \leq k$). Accordingly, k is the index of the used-block that precedes the deadline point $a_i + d_i$. Now, the general formula for F_m is given by:

$$F_m = Min\left(\frac{U_{m+1,L} - U_{1,L}}{\sum_{r=1\ to\ m} U_r}, \frac{U_{m+1,L} - U_{2,L}}{\sum_{r=2\ to\ m} U_r}, \ldots, \frac{U_{m+1,L} - U_{q,L}}{\sum_{r=q\ to\ m} U_r}\right)$$

The scaling factor for the first-instance of T_i is the maximum among all computed factors for accommodating the next idle interval. Clearly, each of these factors is a valid factor in the sense that it does not extend the completion time of the first instance beyond its deadline. Therefore, the required factor is the maximum among such valid factors given by:

$$sf^{i1} = Max_{q \leq m \leq k}\ \ F_m$$

We now have to generalize the above formula for any arbitrary instance of T_i (say the l'th). Clearly there are x_i (refer to Chapter 6 instances of T_i that have to be considered. Therefore, l ranges from 1 to x_i. If we find the scaling factors sf^{il} for each of the x_i instances of T_i then we can obtain the scaling factor sf^i as the minimum among all these. This is clear from the fact that picking a factor larger than the minimum results in at least one of the instances missing its deadline. So, we have:

$$sf^i = Min_{1 \leq l \leq x_i} \ sf^{il}$$

Before, we find the general scaling factor sf^{il} of an arbitrary instance of T_i, it is important to notice some important considerations in dealing with the second instance (which will easily extend to arbitrary instances). The second instance of T_i is ready at time a_i+p_i. Its ability to start (i.e., get the processor) is affected by higher priority tasks arriving beyond the point $a_i + p_i$ and, also those tasks executing between the deadline of its previous instance at a_i+d_i and the point a_i+p_i. Note that, we have already taken care of tasks arriving before the point a_i+d_i in finding the scaling factor of the first instance. Therefore, the point a_i+d_i for task T_i's second instance is equivalent to the point a_1 (assuming that the task arrivals are in increasing order; further this point can be taken to be $t = 0$). In finding the scaling factor for this instance, we have to consider used-blocks from that which overlaps $a_i + d_i$ (if this is a zero-sized block then consider the next block), to the block that contains the arrival $a_i + p_i$ on one side. On the other side, we have to consider used-blocks between the block that contains $a_i + p_i$ to the block that contains the deadline of this instance at $a_i + p_i + d_i$ (remember that if there is no such block that overlaps with the deadline then we create a zero-sized used-block to overlap it).

Now in the general case, that is, when we wish to find the scaling factor for an arbitrary instance l we define the following notation (refer to Figure 7.3):

- v: U_v is the used-block that contains the deadline of the $(l-1)$'th instance of T_i. If however, U_v is a zero-sized block then v is the index of the next block following the deadline of the $(l-1)$'th instance at $a_i+(l-1)\times p_i+d_i$. As a special case, for the first instance $v = 1$.

- q: Is the block that overlaps with the arrival of the l'th instance of task T_i. This is also the block that contains the completion of the l'th instance.
- k: U_{k+1} is the block that contains the deadline of the l'th instance at $a_i + (l-1) \times p_i + d_i$. Note that, if the deadline does not overlap with a used block then we create a zero-sized used-block at $a_i+(l-1)\times p_i+d_i$. k is then given by the used-block that precedes this newly created zero-sized block.

The formula for the scaling factor of an arbitrary instance (say l) of T_i (represented as sf^{il}) is now given by:

$$sf^{il} = Max_{q \le m \le k} \quad F_m$$

where F_m is given by:

$$F_m = Min \left(\frac{U_{m+1,L} - U_{v,L}}{\sum_{r=v \ to \ m} U_r}, \frac{U_{m+1,L} - U_{v+1,L}}{\sum_{r=v+1 \ to \ m} U_r}, \ldots, \frac{U_{m+1,L} - U_{q,L}}{\sum_{r=q \ to \ m} U_r} \right)$$

We now have the scaling factor (sf^i) with respect to a task T_i. In order to find the final common scaling factor sf we follow the same lines as in Chapter 5. Therefore, the required scaling factor sf is given by:

$$sf = Min_{1 \le i \le n} \quad sf^i$$

The complete algorithm to find the scaling factor for task T_i follows:

4 *Algorithm* **Scale_Factor**(T_i)

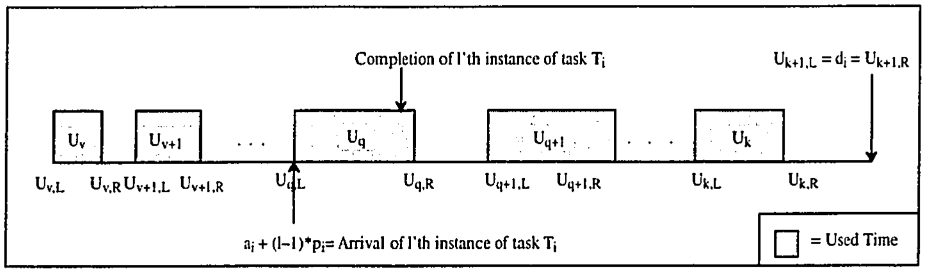

Figure 7.3: Execution Profile of the l'th instance of T_i

Variables:

$l = 0$: task instance

sf^{il}: the scaling factor for task i instance l.

Step 0: Initialization. $sf^i = \infty$

Step 1: For each task instance l of T_i between a_i and $a_i + L$ Repeat

Step 1.1: Find the completion time for the job $l = compl_t$;

Step 1.2:

Fit equal and higher priority task instances that would arrive between the points G and $a_i + (l + 1) \times d_i$. The point G is a_1 for the first instance, $l = 1$ and for subsequent instances, $l > 1$ it is given by the deadline of the previous instance, $a_i + l \times d_i$.

The scheduling points are, $U_{1,L}, U_{2,L}, \ldots, U_{k,L}$. where, $U_m = U_{m,R} - U_{m,L}$ denotes the m^{th} used time block (refer to Figure 7.1).

The used interval among these blocks which overlaps with $a_i + l \times P_i$ is $= q$ (note that this is also the block that contains $compl_t$, because there can be no idle time between the instances arrival and it completion).

Step 1.3:

Compute the scaling factor sf^{il} for job l:

$$Max_{q \le m \le k} \quad F_m$$

where:

$$F_m = Min\left(\frac{U_{m+1,L} - U_{1,L}}{\sum_{r=1\ to\ m} U_r}, \frac{U_{m+1,L} - U_{2,L}}{\sum_{r=2\ to\ m} U_r}, \ldots, \frac{U_{m+1,L} - U_{m,L}}{U_m}\right)$$

Step 1.4: if $(sf^{il} < sf^i)$ then $sf^i = sf^{il}$.

Step 2: sf^i is the desired scaling factor for task T_i.

end

Having obtained the scaling factor sf^i for each i in turn we can now determine the optimal scaling factor, sf for the task set, which is the minimum of $sf^i, \forall i$.

7.2 Example Demonstrating the Solution

To demonstrate the solution we take an example with three tasks whose characteristics are given in Table 7.1. The timing analysis is shown in Figure 7.4. The scaling factor derivation for the first task is straightforward. The derivation for the other two tasks is shown in the figure. The common scaling factor for this example task-set is 1.6363.

We compare the scaling factors obtained by taking the approach in this chapter as opposed to the critical instant approach followed in chapter 5 to appreciate the benefits. If this task-set was subjected to the approach in chapter 5 then the common scaling factor would be 1.3636. Using the approach

Table 7.1: Example Task Table

Task Id	Period	Arrival	Exec Time	Deadline
1	12	0	2	12
2	24	4	4	24
3	16	3	3	15

described in this chapter we get a scaling factor of 1.6363. This is a huge gain considering that it is a multiplicative factor and not additive. This will become more evident if we express the improvement in execution times as percentages.

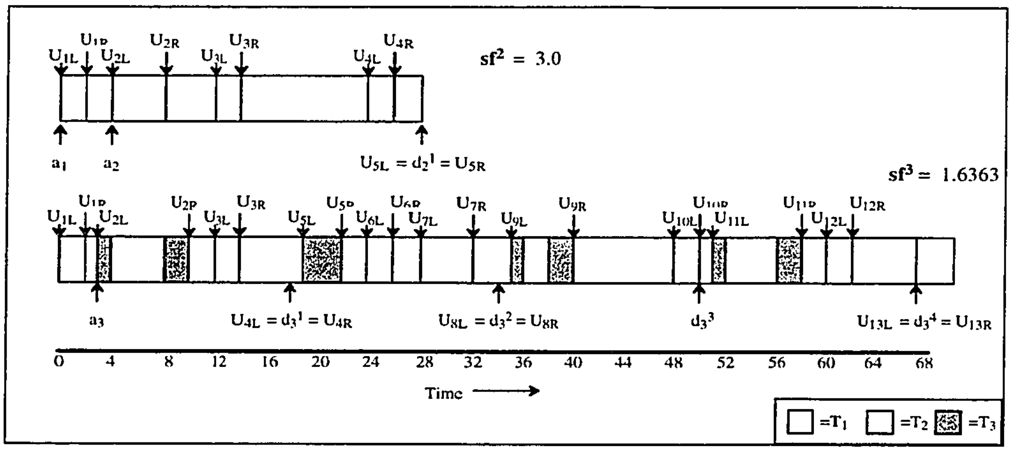

Figure 7.4: Operation of example task set

Chapter 8

Admission Control for Real-Time Communication

The model assumptions in this chapter are based on the Real Time Channel model described in Chapter 2. Admission control is the mechanism by which multiple real-time connections can simultaneously share the resources of a packet switching network without resulting in congestion. Further, the connections are guaranteed a particular *quality of service* (QoS) that is initially (at connection set up) agreed upon. Admission control comes into play when a new RT channel is being requested. An RT channel (or a connection request) is accompanied with a QoS list that describes the requirements of this connection. Popular QoS requirements in the literature of distributed real-time systems are - throughput, latency (or deadline), packet loss tolerance [17, 28, 10, 35, 32] etc.

The mechanism used to determine the admissibility of a real-time channel involves verifying at each intermediate link (along the path) in turn whether the RT channel's QoS requirements can be guaranteed. If a channel's requirements can be met at each of the intermediate links then we can accept the channel. If however, the channel's requirements cannot be met at any of the intermediate link then we can reject the channel. In fact the first such link that deems the channel inadmissible is sufficient to confirm that the channel would not be admissible.

In order to test whether a channel's requirements will be met at an intermediate link we have to know its deadline and its period at each of that link. Finding the period is straightforward according to the phase adjustment mechanism. However we do have to derive the deadline of the RT channel at intermediate links. Since the service time of the channel on each of the links is the same one way to derive the deadlines would be to divide the slack of the RT channel equally among the intermediate links. However, if one wishes, one can use a more sophisticated heuristic [15, 4, 47] to derive these deadlines. This reduces the problem of finding the admissibility of an RT channel to be equivalent to solving the admissibility at each of the intermediate link [11, 18]. From here onwards when we refer to the admissibility of an RT channel we mean its admissibility at an intermediate link.

Now, the question that admission control has to answer when accepting a new connection can be broadly phrased as:

- Given the QoS requirements of a new RT channel is it possible to accept this channel without violating the QoS guarantees made to RT channels that have already been accepted?

The principle followed by researchers (for example Tenet [8, 9]) in the design of an admission control scheme is based on verifying, whether the resources available on the path of the newly requested RT channel are sufficient even in the worst possible case, to

1. provide the new RT channel with the QoS it needs and,

2. allow the guarantees offered to all the existing RT channels to continue being satisfied.

The above verification depends upon the kinds of QoS parameters allowed. The most important QoS parameter of concern to real-time system designers is the meeting a latency bound (deadline). We restrict our interest to this parameter. There are two tests that are relevant in this context:

- Schedulability Test: Does the addition of the new channel to the already established channels using this link cause either the new channel or one of the already established channels to miss their deadline?

- Buffer Space Test: Is the available buffer space at the link sufficient to allow the messages of the new channel to be stored for a length of time equal to the delay faced by the channel at this link?

Different approaches to the admission control problem (in real-time systems) will differ in the way the above two questions are answered. Therefore, a study in admission control reduces to the study of these tests. The buffer space test has been successfully addressed by the Tenet group [9]. We concentrate mainly on the schedulability test because it is our belief that there is room for improvement here. In particular, there are many situations that have not been considered in this context. We broadly classify two situations which differ in terms of the assumptions made about the scheduling mechanism used to schedule channels on the intermediate links.

8.1 Dynamic Scheduling of RT Channels

The Tenet schedulability test involves a deterministic test at each intervening link along the path. An assumption is made that the scheduling mechanism used at an intermediate link is based on the EDD [9] (earliest due date or popularly referred to as the earliest deadline first). The test is based on extending the fundamental task scheduling result by Liu and Layland [24] to message communication. It can be summarized as follows: A given set of RT-channels (at a particular link) is schedulable[1] by the EDD policy if the sum of the utilizations of the RT channels is less than one. The utilization of the i^{th} RT channel whose characteristics are a message service time of m_i and a message inter-arrival time of g_i is given by, $u_i = m_i/g_i$. If the current total utilization at a link is U_{det} then the utilization as a result of accepting the new connection (i^{th}) would be $U_{det} = U_{det} + m_i/g_i$, and the schedulability test would be to check whether $U_{det} < 1$.

We have taken a different approach to the schedulability test that is based on the scaling problem defined in Chapter 4. The principle involved in the test can be described as follows. At each intermediate link an *admittance measure* is computed that essentially captures the tightness of the traffic already passing through the link. A new connection request is allowed or disallowed depending upon whether a specific *relationship* between this measure and the new connection's characteristics is satisfied. The computation of the admittance measure is dependent upon the choice of the scheduling mechanism and the characteristics of the connections already accepted. Further the tested

[1]all the RT channel deadlines will be guaranteed to be met.

relationship referred to above, is a heuristic comparison between the current admittance measure and the new connection's characteristics.

The admittance measure we use is the scaling factor (refer to Chapter 4) with which the message service times of channels already accepted can be multiplied by, so that the channels' requirements can still be guaranteed. The new connections characteristics are captured by its utilization demand. The heuristic used can be explained as follows. Intuitively, the greater the scaling factor greater is the potential to allow a new connection. Further, the room for accommodating new connections is intuitively captured by the term, $\frac{sf_{n-1}-1}{sf_{n-1}}$. This expression, can be viewed as the percentage improvement possible in the utilization of the existing channels. The expression can be simplified into the form, $1 - \frac{1}{sf_{n-1}}$. We show later, how this heuristic turns out to be equivalent to the deterministic test of Tenet (in the context of EDD that is).

The following table, shows a comparison of our approach (using the scaling factor) and Tenet's approach. The scheduling mechanism chosen at a link is assumed to be the EDD. We later show how the two approaches are equivalent.

Table 8.1: Admission Control Test

Approach	Computation	Test
Tenet	$U_n \leftarrow U_{n-1} + \frac{m_n}{g_n}$	$U_n < 1$
Scaling	sf_{n-1} (precomputed)	$\frac{m_n}{g_n} \leq 1 - \frac{1}{sf_{n-1}}$

The second column in the table gives the computation that has to be done in order to test for the admissibility of a new channel. This test can either be done at the time the new connection is made (Tenet's approach) or it can be precomputed (our approach). The advantage of completing this computation before the channel is requested is that it will cause minimal delay in ascertaining admissibility. Further, it affords the designer to attempt a more sophisticated computation because it is done prior to the actual channel admission test. The third column gives the test performed when a new connection is requested.

We now show how the two approaches given in the table are equivalent. In the case of Tenet, the admissibility test can be viewed as a simple comparison to check if the total utilization resulting from the addition of the new channel is above the allowed bound (1). Observe that the computation in the second column involves the characteristics of the new connection, thus making it a computation that has to be performed when the new connection is requested. We can however, modify Tenet's approach so that the computation (just compute U_{n-1}) is independent of the new channel characteristics and can thus be done before hand. Further, this modification would result in the test changing to: $\frac{m_n}{g_n} \leq 1 - U_{n-1}$.

The reader is referred to Chapter 5 for a discussion of the scaling factor problem. More specifically, in section 5.2, a special instance of this problem is identified when the subset to be scaled S is the same as the given task-set T. It was shown that the common scaling factor (in the case of EDF) is then given by the reciprocal of the total utilization of the RT channels.

$$\begin{aligned} sf_{n-1} &= \frac{1}{\sum_{1 \leq i \leq n-1} \frac{m_i}{g_i}} \\ &= \frac{1}{U_{n-1}} \end{aligned}$$

The test in third column can therefore be interpreted as the $\frac{m_n}{g_n} \leq$

$1 - U_{n-1}$. Therefore, we see that the two approaches reduce to be the same. Observe that, the computation of the scaling factor, sf_{n-1} is more involved if the scheduling mechanism is not EDF. This is the subject of the following section.

8.2 Fixed Priority Scheduling of RT Channels

Our next concern is to extend the approach described in the previous section to, general fixed priority preemptive scheduling mechanisms. Note that the Tenet approach is only valid for dynamic preemptive scheduling. We use the same approach to admissibility as described in the previous section, except that we have to pay special attention to the computation of the scaling factor. We concentrate our attention to extending our approach to incorporate the Rate Monotonic Scheduling (RMS) mechanism (a particular instance of the fixed priority preemptive scheduling mechanism). An extension of the approach to Deadline Monotonic Scheduling and more generally to any arbitrary fixed priority scheduling mechanism is straightforward.

As we already have seen in Chapter 4, there is no straightforward way to compute the scaling factor of a set of tasks (read as RT channels in the present context) scheduled by a general fixed priority scheduling mechanism. However, in the particular case of RMS, we can find a non-optimal scaling factor that is given by:

$$sf_{n-1}^{rms} = \frac{(n-1)(2^{1/(n-1)} - 1)}{U_{n-1}} \tag{8.1}$$

This factor is not optimal in the sense that it is possible to improve it further. Unlike task schedulability where we were interested in an optimal scaling factor, in the current context (admission control that is) the above computation does carry a certain merit as will be demonstrated shortly. Though the heuristic used in the admissibility test reduced to the deterministic test in the context of EDD, this is not necessarily true in the current context. In other words, failing to pass the heuristic test does not necessarily imply that the new channel will interfere with the schedulability of the already existing channels. This implies that, using the heuristic it is possible that a new channel request is rejected even though it could have been accommodated.

An alternative to the above computation is to use a more precise computation, one which would help us obtain an optimal scaling factor. We have shown in Chapter 4, how such a computation works. This alternative is appealing in its ability to reduce the number of rejections (as described in the previous paragraph). However, it does not necessarily guarantee 100% admissibility. 100% admissibility is said to be achieved if the test never rejects a new channel that would have not interfered with already accepted channels. The failure of this alternative to ensure 100% admissibility is due to the fact that though the scaling factor computation is precise, the comparison in which it is used is a heuristic.

It is important to observe that, the scaling factor computation is not performed at the time of a channel request and therefore we can afford the cost involved in finding an optimal scaling factor. However, if the benefit (reducing the number of rejections) obtained by using the optimal scaling factor is not large enough (compared to using the non-optimal computation), we cannot justify it. Since, the basis of the test is a heuristic, the only way one can confirm the benefits is to perform a simulation study.

Simulation Study

The goal of this study was to compare the two alternatives for admission control (described above) when the underlying mechanism used to schedule the RT channels is the Rate Monotonic Scheduling. An RT channel is characterized, among other parameters by the source and destination of the channel. This information is used to find the route of the RT channel. As already described the admissibility test of an RT channel that traces a route of, say k links, reduces to ascertaining its admissibility at each of the k links in turn. Therefore, we restrict our study to admissibility at a single link. From here onwards when we refer to the characteristics of an RT channel we don't mean its end-to-end characteristics but its characteristics at an intermediate link.

We use the following notation in the following discussion:

$x \sim U(a, b)$ to indicate that the random variable x is uniformly distributed over the interval from a to b.

$x \sim N(\mu; \sigma)$ to indicate that the random variable x has a normal distribution with mean μ and standard deviation σ.

There are two major steps to the simulation study:

1. The workload generation. The workload of interest to us is the generation of characteristics of n RT channels at a link. We would like to characterize the workload with a set of parameters that capture its essence. We use the following two parameters to characterize (and distinguish between) workloads:

(a) The utilization U, of the set of RT channels is used to identify the cumulative demand of the workload.

(b) The laxity factor κ, dictates in addition the closeness of the deadline to the end of the period of the RT channels.

2. The simulation of the alternatives and their comparison. The two alternatives of concern to us are, using the non-optimal scaling factor vs. using the optimal scaling factor in the admissibility test. The details of the comparison are explained later.

Before we explain the generation process, it is important to understand what we are attempting to generate. We are interested in generating a workload of n RT channels with a total utilization of U. For each RT channel C_i, we wish to know its service time m_i, its inter-message generation time g_i and its deadline d_i.

The following parameters were used in the generation process.

n: The number of RT channels in the link.

m: The mean service time of an RT channel.

U: The total utilization of the n RT channels. The utilization of an RT channel C_i with service time m_i and and inter-generation time of g_i is given by m_i/g_i.

$\kappa(0 < \kappa < 1)$: Is the laxity factor.

$\mu_l(0 < \mu_l < 1)$: This parameter controls the laxity of an RT channel. The deadline of an RT channel C_i with a laxity of l is given by $m_i + l \times (g_i - m_i)$.

Therefore, greater the value of l (directly controlled by μ_l), closer is the deadline to the period and more is the room for meeting its deadline.

σ_l: The standard deviation of the normal distribution of the laxities of the channels. We constrain this parameter so that following conditions hold:

$$mu_l - 3 \times \sigma_l > 0 \ \ and$$

$$mu_l + 3 \times \sigma_l < 1$$

The above two conditions guarantee [16] that the majority ($\approx$ 99.98%) of the samples derived from the distribution, $N(\mu_l, \sigma_l)$ are within the bounds(0 and 1).

The approach taken for workload (n RT channels) generation can be described as follows. We generate the characteristics of each RT channel C_i in turn.

1. The service time m_i of channel C_i is derived from a uniform distribution over the range $[1, 2 \times m]$:

$$m_i \sim U(1; 2 \times m)$$

2. The utilization of u_i of channel C_i is derived from a uniform distribution over the range $[0, 2 \times \frac{U}{n}]$:

$$u_i \sim U(0; 2 \times \frac{U}{n})$$

3. The inter-generation time (or period) g_i, of channel C_i is obtained by using its service time and utilization already generated above, as:

$$g_i = \frac{m_i}{u_i}$$

4. Channel C_i's deadline d_i is obtained as:

$$d_i = m_i + \kappa \times (g_i - m_i)$$

$$where\ \kappa \sim N(\mu l; \sigma_l)$$

A special case of interest in the simulation (discussed below) we need a workload where the laxity factor of the RT channels is a constant. We can generate a workload with such a characteristic by assigning the parameter σ_l to be equal to zero and the parameter μ_l to equal the constant desired.

Having generated the workload we are now in a position to compare the two heuristic alternatives against the generated workload. As explained before the test mechanism we use to determine whether a new RT channel $C_n(m_n, g_n, d_n)$ can be admitted at a link, having already accepted $n-1$ RT channels is given by:

$$\frac{m_n}{g_n} \leq 1 - \frac{1}{sf_{n-1}}$$

Where the term sf_{n-1} is the factor by which the $n-1$ (already accepted) channel service times can be scaled without violating their schedulability requirements. The two alternatives we are interested in comparing differ in the way this scaling factor is arrived at.

- $\mathcal{R}$: Uses the non-optimal computation of sf_{n-1} given by Equation 8.1.
- $\mathcal{S}$: Uses a precise (optimal) computation of the sf_{n-1} described in Chapter 4.

In order to explain the criteria that were chosen for the comparison it is important to understand that the workload generated (of n RT channels) is arbitrary in the sense that they can be either admissible (together) or not. For a given workload however, we can test whether it is schedulable or not. In other words, whether all the RT channels can be accommodated together or not. We refer to the outcome of this test as the admissibility (denoted by $\mathcal{A}$) of the workload.

Observe that the above test finds the admissibility of a workload whereas, the heuristics are designed to test whether a given RT channel can be admitted to an already existing list of RT channels at a link. In other words, the outcome $\mathcal{A}$ can be either, $\mathcal{A}_{yes}$: the workload can be admitted together, or $\mathcal{A}_{no}$: the workload is not admissible together. On the other hand, the outcome of the heuristic $\mathcal{H}$ ($\mathcal{R}$ or $\mathcal{S}$) test can be either, $\mathcal{H}_{yes}$: admit the new channel, or $\mathcal{H}_{no}$ do not admit the new channel. However, the heuristic $\mathcal{H}$'s decision can be compared against $\mathcal{A}$ by defining the following criteria:

1. If the heuristic $\mathcal{H}$ arrives at the decision $\mathcal{H}_{yes}$ when the workload is in fact admissible ($\mathcal{A}_{yes}$), then we say that the heuristic has succeeded on a **YES** match.

2. If the heuristic $\mathcal{H}$ arrives at the decision $\mathcal{H}_{no}$ when the workload is in fact inadmissible ($\mathcal{A}_{no}$), then we say that the heuristic has succeeded on a **NO** match.

3. If neither criterion 1 nor criterion 2 are met then we say that the heuristic has failed.

Note that the reason for having two criteria for a match is because the generated workload was arbitrary in the sense that it could either be feasible or not. While we are primarily interested in a heuristic's ability to admit (reach a **YES** match that is) an RT channel, we cannot ignore the impact of an incorrect decision. The ability of a heuristic to reject infeasible workloads (captured by criterion 2) is important in that it gives us an idea about the heuristic's sensitivity. For example, it is possible that the heuristic admits a new channel to only realize later that it would result in one or more of the channels' guarantees being violated.

For a given total utilization U and number of channels n (input parameters), the simulation involves generating workloads of n RT channels and testing the admissibility of each of them. Before we use one of the two heuristics ($\mathcal{R}$ or $\mathcal{S}$) to determine whether they admit a given channel, we first ascertain the admissibility of the workload ($\mathcal{A}$ described before). Next, for each RT channel (say C_i) in turn we test its admissibility (using a heuristic) assuming that the $n-1$ other channels have already been accepted. The test is repeated with the two heuristics we are attempting to compare. If the heuristic we are testing is say $\mathcal{R}$, then the outcome of the test can be one of $\mathcal{R}_{yes}$ (admit the channel C_i) or $\mathcal{R}_{no}$ (don't admit the channel C_i). We now compare this outcome against the outcome from the admissibility test for the workload $\mathcal{A}$ which was already computed. The comparison follows the criteria explained before. With respect to this channel we record whether the heuristic achieved a match (could be a **YES** or **NO**) or has failed. The simulation records the same for each channel in turn and obtains the heuristic's performance on this particular workload (This is repeated for the other heuristic also).

The performance of a heuristic for a given workload is characterized by three parameters:

1. The percentage of (the total n admissibility tests) tests that result in a **YES** match.

2. The percentage of (the total n admissibility tests) tests that result in a **NO** match.

3. The percentage of (the total n admissibility tests) tests that result in failure.

Observe that, the generated workload is only one of an almost infinite possible workloads with the same input parameters. Therefore we repeat the above experiment for a large number of workloads and take an average performance. Further we repeat this for different values of κ (or μ_l and $sigma_l$). The results of the simulation are presented in Appendix A.

Simulation Results

The performance measure of primary interest to us is the admissibility of a heuristic. And, we are interested in comparing the two heuristics to see which of the two is better at admitting channels. Therefore, the graphs we present here compare the performance using the percentage **YES** match (see above).

Recollect that, the heuristic $\mathcal{R}$ assumes that the underlying scheduling mechanism is the rate monotonic scheduling. It has been shown that the RMS is an optimal scheduling mechanism [20] if the deadlines of tasks are a constant factor of their periods. Therefore, we assume that the parameter κ is a constant and not derived from a distribution. This assumption was made in order to choose a scenario that is favorable to both heuristics (and not biased to either). This assumption however has no impact on the second heuristic $\mathcal{S}$.

Each graph is identified by the number of channels considered and the parameter κ. The x-axis gives the total utilization of the workload and the y-axis gives the success of the heuristic. For low utilizations (less than 50%) there is no need to do a complex test because the demand can be easily met. We chose four different values of the number of channels (4, 8, 12, 16) and varied the parameter κ between 0.5 to 1.0. It was observed that values of κ less than 0.5 resulted in too many channels missing their deadlines.

Observations

- For low utilizations (less than 0.7) we observe that both the heuristics have a similar admissibility. Given that the heuristic $\mathcal{R}$ is less expensive (computation time-wise) than $\mathcal{S}$, under conditions of low utilizations one can choose the heuristic $\mathcal{R}$.

- For a given value of n and κ we observe that the admissibility of heuristic $\mathcal{R}$ falls abruptly beyond a point on the x-axis given by the utilization bound. For example, in Figure A.6 we can see that the heuristic $\mathcal{R}$ begins to reject channels when the total utilization crosses beyond 0.72.

- The performance of $\mathcal{S}$ degrades gracefully beyond the utilization bound. For example. in Figure A.6 we can see that the heuristic $\mathcal{S}$ continues to admit channels up to a total utilization of 0.92. The probability of acceptance decreases gradually (and steadily) however. This implies that the heuristic has a better ability to adapt to temporary overloads [43, 26] (increased demand from one of the channels) in the network traffic.

- As the number of channels increases, the performance degradation beyond the utilization bound is slower in the case of heuristic $\mathcal{S}$. This goes on to support the ability of the heuristic to adapt to temporary overloads (increase in the number of channels). The two sources of overload have been successfully handled by the heuristic $\mathcal{S}$.

- As the number of channels increases the success of the heuristic $\mathcal{S}$ improves compared to the heuristic $\mathcal{R}$.

- In conclusion we can say that for low utilizations both heuristics have similar performance (however one should prefer the heuristic $\mathcal{R}$ due it computational ease) but, at high utilizations $\mathcal{S}$ far outperforms $\mathcal{R}$. Further, we can justify the cost of computation involved in $\mathcal{S}$ by noting that the computation can be done before the actual channel request is made.

Chapter 9

Summary of Results

As an example to demonstrate the results reported in this thesis, we choose the "Olympus Attitude and Orbital Control System"(AOCS). A detailed case study of this real-time system can be found in [5, 46]. The AOCS subsystem of the Olympus satellite[1] acquires and maintains spacecraft positions as desired. A detailed analysis of this system was performed by A. Burns and his colleagues, as a result of which they have summarized a list of tasks (Appendix B, Figures B.1, B.2 and B.3) that capture the system's functionality. They have identified mainly two classes of tasks viz., periodic (Figures B.1, B.2) and sporadic tasks (Figure B.3).

The class of periodic tasks in the AOCS case-study are consistent with our definition and treatment of periodic tasks in this thesis. Sporadic tasks on the other hand are tasks whose periodicity and arrival time are not known. However, there is a known minimum interval between successive arrivals of these tasks. Also the arrival time parameter of a sporadic task is not known a priori due to the nature of these tasks. Sporadic tasks typically occur due to events such as exceptions and interrupts which are triggered by a logical state

[1]The Olympus satellite was launched in July 1989 as the world's largest and most powerful civil three-axis-stabilized communications satellite. It provides direct broadcast TV and 'distance learning' experiments to Italy and Northern Europe.

of the system or an external event. These events are therefore a function of the run-time characteristic of the system.

The treatment in this thesis has been restricted to handling only periodic tasks, however we can accommodate sporadic tasks by making a few observations about their behavior. The minimum inter-arrival time parameter associated with a sporadic task is a lower bound on its periodicity. For the purpose of this chapter we choose the periods of sporadic tasks to have values ranging from the minimum to the average periods of periodic tasks. Accordingly the chosen values of periods for sporadic tasks have been listed in the tables. Further, we have chosen the arrival times of these tasks to be zero, in other words that the first occurrence of these tasks is at time $t = 0$. Clearly, this is only one of the many possibilities but is sufficient to demonstrate our point of interest here.

The following sections use this task-set to demonstrate the results reported in chapters 5 to 7.

9.1 Scalability in Uniprocessor Systems

The above task-set (say T) is given for a uniprocessor system, where all the tasks are known to execute on a central control computer. In order to apply the result given in Chapter 5 we have to choose a subset (say S) of tasks in the task-set that are to undergo scaling. For a lack of better knowledge about the tasks we pick $S = T$, i.e., we are interested in finding the maximum common scaling factor for all tasks in the task-set. Table 9.1 gives the results of this analysis:

Table 9.1: Task Table with Scaling Factors

Task Name	Priority	Period	Arrival	Exec	Deadline	Scale Factor
BUS_INTERRUPT	62	50	0.00	0.18	1.00	5.5556
REAL_TIME_CLOCK	27	50	0.00	0.28	9.00	19.5652
READ_BUS_IP	23	10	0.00	1.76	10.00	4.5045
COMMAND_ACUTUATORS	20	200	50.00	2.13	14.00	2.2989
REQUEST_DSS_DATA	19	200	150.00	1.43	17.00	2.2546
REQUEST_WHEEL_SPEEDS	18	200	0.00	1.43	22.00	2.2296
REQUEST_IRES_DATA	17	100	0.00	1.43	24.00	1.9736
TELEMETRY_RESPONSE	15	200	0.00	3.19	30.00	1.9543
PROCESS_IRES_DATA	14	100	50.00	8.21	50.00	1.8463
READ_YAW_GYRO	12	500	0.00	4.08	100.00	2.4740
CONTROL_LAW	8	200	50.00	22.84	200.00	2.18770
PROCESS_DSS_DATA	6	1000	200.00	5.16	400.00	2.1748
CALIBRATE_GYRO	5	1000	200.00	6.91	900.00	2.1645
TELECOMMANDS	4	500	0.00	2.50	187.00	1.7941
Scaling Factor for S					=	1.7941

The mechanism used to find the scaling factor in the uniprocessor scenario is based on the critical instant assumption. This result can be easily improved by using a more precise mechanism that is based on the results in chapter 7. However, as discussed in chapter 4 the critical instant assumption is more suitable in uniprocessor systems. Further, it makes the scaling factor computation more efficient and cheaper (in terms of processing time).

Another perspective of the scaling factor can be expressed in terms of the utilization. The utilization of a task T_i is given by the ratio of its execution time to its periodicity, $\frac{e_i}{p_i}$. The total utilization of the task-set before scaling is given by:

$$U = \frac{e_1}{p_1} + \frac{e_2}{p_2} + \ldots + \frac{e_n}{p_n}$$

For the task-set in our case study this is given by: 0.4619. The utilization of the task-set after the scaling is performed is given by:

$$U' = sf \times \sum_{i \in S} \frac{e_i}{p_i} + \sum_{j \in T-S} \frac{e_j}{p_j}$$

Where, sf is the maximum common scaling factor for the task in S. In our example, $S = T$, therefore the second term in the above equation is zero. The new utilization is now given by $1.7941 \times 0.4619 = 0.8287$. This achievable improvement in utilization is promising for the application with regards to, scalability, execution time estimation, portability and fault-tolerance as described in chapter 3.

9.2 Schedulability of Task-Sets with Arrivals

As described in chapter 4, solving the problem of scalability in end-to-end real-time systems involves solving the two problems of (i) schedulability of tasks on a single component without ignoring arrival times and, (ii) scalability of tasks with non-zero arrival times. The first of these problems was discussed in chapter 6.

This section discusses this result by applying it the AOCS case-study. Our first example involves, treating the AOCS as an end-to-end task system with each task comprising only one sub-task which runs on the only component in the system, i.e., the processor. Now, the determining the schedulability of the tasks involves computing their worst-case response times. For comparison purposes, the following table (9.2) gives the worst-case response times using two different mechanisms, i,e., the critical instant approach (WC^c) and, the approach described in chapter 6 (WC^r). The third column gives the percentage improvement in the response time obtained by using our precise approach as opposed to the critical instant approach. The fourth column gives the slack savings achieved by using our approach. Slack savings (in percentage) is given by the formula:

$$ss_i = \frac{WC_i^c - WC_i^r}{d_i - e_i} \times 100$$

While the percentage improvement obtained does have some merit in explaining the need for a more precise approach, the slack savings parameter qualifies the ability of a task to accommodate task interdependence (e.g., precedence), withstand temporary overloads, accommodate aperiodics in the system and restrict jitter in end-to-end systems.

Table 9.2: Response times of Tasks

Task Name	Resp Time		% Improvement	Slack Savings (in %)
	WC^c	WC^r		
BUS_INTERRUPT	0.18	0.18	0.0	0.0
REAL_TIME_CLOCK	0.46	0.46	0.0	0.0
READ_BUS_IP	2.22	2.22	0.0	0.0
COMMAND_ACUTUATORS	4.35	4.35	0.0	0.0
REQUEST_DSS_DATA	5.78	3.65	36.85	13.60
REQUEST_WHEEL_SPEEDS	7.21	3.65	49.37	17.30
REQUEST_IRES_DATA	8.64	5.08	41.20	15.77
TELEMETRY_RESPONSE	13.59	8.27	39.14	19.84
PROCESS_IRES_DATA	23.56	14.32	39.21	22.11
READ_YAW_GYRO	27.64	14.11	48.95	13.06
CONTROL_LAW	56.22	42.44	24.51	7.77
PROCESS_DSS_DATA	63.14	15.19	75.94	12.14
CALIBRATE_GYRO	71.81	23.86	66.77	5.36
TELECOMMANDS	74.31	16.61	77.64	31.27

As a second demonstration of the results in Chapter 6, we consider an actual decomposition of the task-set into sub-tasks. The chosen decomposition is only one of the possible decompositions obtained by arbitrarily dividing and assigning tasks to four components. The decomposed task-set is given in Figure 9.3.

Table 9.3: Decomposition of tasks

Task Name	Resource(s)
BUS_INTERRUPT	R_1
REAL_TIME_CLOCK	R_2
READ_BUS_IP	R_3
COMMAND_ACUTUATORS	$R_1 \rightarrow R_4$
REQUEST_DSS_DATA	$R_1 \rightarrow R_2$
REQUEST_WHEEL_SPEEDS	$R_1 \rightarrow R_3$
REQUEST_IRES_DATA	R_4
TELEMETRY_RESPONSE	R_4
PROCESS_IRES_DATA	$R_1 \rightarrow R_2 \rightarrow R_3$
READ_YAW_GYRO	$R_1 \rightarrow R_3$
CONTROL_LAW	$R_1 \rightarrow R_2 \rightarrow R_4$
PROCESS_DSS_DATA	$R_1 \rightarrow R_3$
CALIBRATE_GYRO	$R_2 \rightarrow R_4$
TELECOMMANDS	$R_1 \rightarrow R_2$

The following tables (9.4, 9.5, 9.6, 9.7) give details of the analysis of each component in turn. The parameter of the sub-tasks that run on the first component R_1, are directly inherited from the parent. Further, the deadline parameter is not required in this problem since we are only interested in finding the worst-case response times of tasks, which are given by the sum of the response times of their individual sub-tasks. The arrival time parameter of sub-tasks on component R_2 (and subsequently R_3 and R_4) are obtained by the phase adjustment mechanism.

The following table (9.8) compares the resulting worst-case response times of tasks with their deadlines. Clearly, all tasks meet their deadlines.

Table 9.4: Analysis of Component R_1

Task Name	Priority	Period	Arrival	Exec	Response Time
BUS_INTERRUPT	62	50	0.00	0.18	0.18
COMMAND_ACUTUATORS	20	200	50.00	1.13	1.31
REQUEST_DSS_DATA	19	200	150.00	0.43	0.61
REQUEST_WHEEL_SPEEDS	18	200	0.00	0.70	1.88
PROCESS_IRES_DATA	14	100	50.00	3.21	4.52
READ_YAW_GYRO	12	500	0.00	1.08	2.96
CONTROL_LAW	8	200	50.00	5.00	9.52
PROCESS_DSS_DATA	6	1000	200.00	2.10	3.98
TELECOMMANDS	4	500	0.00	1.00	3.96

Table 9.5: Analysis of Component R_2

Task Name	Priority	Period	Arrival	Exec	Resp Time
REAL_TIME_CLOCK	27	50	0.00	0.28	0.28
REQUEST_DSS_DATA	19	200	150.61	1.00	1.00
PROCESS_IRES_DATA	14	100	54.82	3.00	3.00
CONTROL_LAW	8	200	59.52	5.00	5.00
CALIBRATE_GYRO	5	1000	200.00	3.0	3.28
TELECOMMANDS	4	500	3.96	1.50	1.50

Further, by comparing these response times against those in table 9.2 we observe the enormous improvement in response times of tasks.

9.3 Scalability in End-to-End Systems

As mentioned in the previous section, the second issue to be addressed in solving the scalability problem in end-to-end systems is: scalability of tasks on a single component with non-zero arrival times. This was the subject of Chapter 7. In this section, we first compare the scaling factor obtained by incorporating task arrival times against, that obtained by using the critical instant assumption

Table 9.6: Analysis of Component R_3

Task Name	Priority	Period	Arrival	Exec	Resp Time
READ_BUS_IP	23	10	0.00	1.76	1.76
REQUEST_WHEEL_SPEEDS	18	200	1.88	0.73	0.73
PROCESS_IRES_DATA	14	100	57.82	2.00	2.00
READ_YAW_GYRO	12	500	2.96	3.00	3.00
PROCESS_DSS_DATA	6	1000	203.98	3.06	3.06

Table 9.7: Analysis of Component R_4

Task Name	Priority	Period	Arrival	Exec	Resp Time
COMMAND_ACUTUATORS	20	200	51.31	1.0	1.0
REQUEST_IRES_DATA	17	100	0.00	1.43	1.43
TELEMETRY_RESPONSE	15	200	0.00	3.19	4.62
CONTROL_LAW	8	200	64.52	7.84	7.84
CALIBRATE_GYRO	5	1000	203.28	3.91	4.68

(chapter 5). Table 9.9 gives the summary of this comparison. The maximum common scaling factor by the precise approach is under the second column ($sf(actual)$) and that obtained by the critical instant assumption in chapter 5 is under the third column ($sf(orig)$). The task-set is assumed to run on a single component and accordingly each task has a single sub-task. The subset S that has to be scaled is same as T. The common scaling factor $sf(actual)$ is 2.1295 which is clearly greater than 1.7941 obtained by the other mechanism. In terms of utilization the resultant task-set utilization is 0.9836 or 98.36%. Note that, ideally one would expect to be able to obtain 100% utilization on scaling, however, this is not achievable in the case of static fixed priority schedulers.

Recall that in chapter 4 problem of scalability of task-sets in end-to-end real-times systems comes in two different forms: task changes and component

Table 9.8: Schedulability of the End-to-End Tasks

Task Name	Response Time	Deadline
BUS_INTERRUPT	0.18	1.00
REAL_TIME_CLOCK	0.28	9.00
READ_BUS_IP	1.76	10.00
COMMAND_ACUTUATORS	2.31	14.00
REQUEST_DSS_DATA	1.61	17.00
REQUEST_WHEEL_SPEEDS	2.61	22.00
REQUEST_IRES_DATA	1.43	24.00
TELEMETRY_RESPONSE	4.62	30.00
PROCESS_IRES_DATA	9.52	50.00
READ_YAW_GYRO	5.96	100.00
CONTROL_LAW	22.36	200.00
PROCESS_DSS_DATA	7.04	400.00
CALIBRATE_GYRO	9.68	900.00
TELECOMMANDS	5.46	187.00

changes. The following modification of the case study demonstrates how our approach to finding the precise scaling factor can be applied in an end-to-end scenario where component changes can occur. The same decomposition used in the previous is used here. The following tables (9.10, 9.11, 9.12 and 9.13) give the details of the scaling factor computation for each of the components. The deadline parameter for each sub-task is obtained by using a heuristic that divides the slack of a task among its sub-tasks in a weighted proportion of their execution times. Now, For example, if the set of components that undergo change are $\{R_2, R_4\}$ then the scaling factor is *min* $\{5.3949, 6.4935\}$ which is 5.3949.

Table 9.9: Task Table with Scaling Factors

Task Name	Scaling Factor	
	$sf(actual)$	$sf(orig)$
BUS_INTERRUPT	5.5556	5.5556
REAL_TIME_CLOCK	19.5652	19.5652
READ_BUS_IP	4.5045	4.5045
COMMAND_ACUTUATORS	2.2989	2.2988
REQUEST_DSS_DATA	3.1423	2.2546
REQUEST_WHEEL_SPEEDS	3.6969	2.2296
REQUEST_IRES_DATA	2.9240	1.9736
TELEMETRY_RESPONSE	2.5445	1.9543
PROCESS_IRES_DATA	2.5510	1.8463
READ_YAW_GYRO	2.5786	2.4740
CONTROL_LAW	2.1877	2.1877
PROCESS_DSS_DATA	2.2119	2.1748
CALIBRATE_GYRO	2.1885	2.1645
TELECOMMANDS	2.1295	1.7941
$CommonScalingFactor$	2.1295	1.7941

Table 9.10: Scaling on Component R_1

Task Name	Priority	Period	Arrival	Exec	Deadline	SF
BUS_INTERRUPT	62	50	0.00	0.18	1.00	5.5556
COMMAND_ACUTUATORS	20	200	50.00	1.13	7.43	5.6696
REQUEST_DSS_DATA	19	200	150.00	0.43	5.11	8.3800
REQUEST_WHEEL_SPEEDS	18	200	0.00	0.70	10.77	5.7283
PROCESS_IRES_DATA	14	100	50.00	3.21	19.55	4.3251
READ_YAW_GYRO	12	500	0.00	1.08	26.47	8.9427
CONTROL_LAW	8	200	50.00	5.00	43.78	4.5990
PROCESS_DSS_DATA	6	1000	200.00	2.10	162.79	11.4286
TELECOMMANDS	4	500	0.00	1.00	74.80	8.4890
Common Scaling Factor					=	4.3251

Table 9.11: Scaling on Component R_2

Task Name	Priority	Period	Arrival	Exec	Deadline	SF
REAL_TIME_CLOCK	27	50	0.00	0.28	9.00	32.1429
REQUEST_DSS_DATA	19	200	150.61	1.00	11.89	9.7641
PROCESS_IRES_DATA	14	100	54.82	3.00	18.27	5.3949
CONTROL_LAW	8	200	59.52	5.00	43.78	5.8554
CALIBRATE_GYRO	5	1000	200.00	3.0	390.73	13.6799
TELECOMMANDS	4	500	3.96	1.50	112.20	11.2510
Common Scaling Factor					=	5.3949

Table 9.12: Scaling on Component R_3

Task Name	Priority	Period	Arrival	Exec	Deadline	SF
READ_BUS_IP	23	10	0.00	1.76	10.00	5.6818
REQUEST_WHEEL_SPEEDS	18	200	1.88	0.73	11.23	4.0161
PROCESS_IRES_DATA	14	100	57.82	2.00	12.18	3.2394
READ_YAW_GYRO	12	500	2.96	3.00	73.53	4.0462
PROCESS_DSS_DATA	6	1000	203.98	3.06	237.21	4.6894
Common Scaling Factor					=	3.2394

Table 9.13: Scaling on Component R_4

Task Name	Priority	Period	Arrival	Exec	Deadline	SF
COMMAND_ACUTUATORS	20	200	51.31	1.0	6.57	6.5727
REQUEST_IRES_DATA	17	100	0.00	1.43	24.00	16.7832
TELEMETRY_RESPONSE	15	200	0.00	3.19	30.00	6.4935
CONTROL_LAW	8	200	64.52	7.84	112.43	9.0236
CALIBRATE_GYRO	5	1000	203.28	3.91	509.26	12.3508
Common Scaling Factor					=	6.4935

Chapter 10

Conclusions

The significant contributions of this thesis can be broadly summarized as follows:

- We have addressed the need to handle complexity in real-time systems in all phases of system design, viz., design, development and maintenance.
- We have presented a novel perspective to analyzing real-time systems that in addition to ascertaining the ability of a system to meet task deadlines also qualifies these guarantees.
- The need to qualify guarantees was shown to arise from the following scenarios pertinent in the design, development and maintenance of real time systems:
  - *Scaling application requirements*: As a system evolves the functionalities of tasks expand, reflecting in terms of increase in code size and/or improvement in data handling of tasks. This increase affects the schedulability guarantees made using the previous execution times. Therefore, what we are interested in is, finding a factor by which the execution times can be scaled (capturing the data handling change) without invalidating the schedulability guarantees.

– *Task execution time estimation*: Using mean task execution times as opposed to worst-case execution times in schedulability analysis reduces the pessimism (leading to over design and under-utilization of resources) inherent in the computation. Unfortunately however, using the mean could lead to cases where the guarantees provided by the schedulability analysis could be invalid (The number of such cases being determined directly by the variance in the computed mean execution time). Therefore, it is necessary to accommodate the variance information along with the mean (for task execution times).

– *Porting applications*: Any analysis performed (to guarantee performance) assuming particular values of task execution times becomes invalid once the target platform changes. For example, a faster processor could result in a lower execution time (not invalidating the analysis), but a slower processor would surely have an adverse affect on the schedulability analysis. As a system evolves, though in general the overall system is likely to improve, the performance of individual components (some processors for example) might not always improve. Another instance where a target platform is in general slower, arises in the case of prototype building and testing [51].

– *Fault Tolerance*: It is common practice to provide fault-tolerant operation by the use of redundant components (often at least one secondary component). In general, secondary components provide only a minimal functionality (sufficient to keep the system operational till the primary is fixed) and therefore tend to be slower. Any schedulability analysis guarantees provided with the primary component as the target will be invalid once the system falls back onto the secondary.

- The scaling factor problem (refer to Chapter 4) defines a quantitative measure that in essence captures the above mentioned scenarios under a uniform framework. The problem is generic in the sense that it leaves such particulars as:

  - the scheduling mechanism,
  - deadline to period relationship, and,
  - arrival information,

  open. For example an instance of the problem could be to find the scaling factor when the assumed scheduling mechanism is a static fixed rate monotonic priority assignment, the task deadlines are less than or equal to their periods, and, the task arrivals are arbitrary.

- The scaling factor problem was first formulated in the context of uniprocessor real-time systems. This scenario can be more generally referred to as the single component scenario. The tasks running on the single component in question are evaluated with regards to their ability to meet their requirements (processing and deadline). Further, we compute a measure that gives us the ability of these tasks to scale-up without violating their guarantees. One important assumption made in this context was that the arrival times of the various tasks can be assumed to be zero. This assumption has helped us in using the critical instant argument to ascertain task schedulability and also in finding the scaling factor. We demonstrated some justifications for the use of this argument, particularly in the context of single component systems with independent tasks.

- Unlike uniprocessor systems, in end-to-end systems, the scaling factor problem appears in two different scenarios, viz., component changes and task changes. We showed how both these scenarios arise and how they can be reduced to solving the following fundamental problems:
  - Compute sub-task parameters of periodicity and deadline.
  - Given a task-set T of n tasks (with non-zero arrivals) executing on a single component, find the worst-case completion times of all tasks in the task-set.
  - Solve the scaling problem when the tasks have arbitrary non-zero arrivals.

The first of the above problems involved finding sub-task periodicities by a technique called phase adjustment and sub-task deadlines by using a heuristic based on proportional division of the total slack of a task among its sub-tasks. Our solution to the second problem is the subject of Chapter 6. This problem has been observed to be relevant in many other contexts in real-time systems, and a discussion to this end can be found in the same Chapter. Chapter 7 presents a solution to the third problem listed above. The complexity is introduced mainly by having to accommodate task arrivals into the analysis. However, this consideration adds validity to our work and also bridges the gap between theory and practice by better modeling the behavior of current complex real-time systems.

- Finally we presented an application of the scaling factor problem in the context of real-time communication. The problem considered is the admission control of real-time channels (Ferrari et. al. [9]). Admission control is the mechanism by which multiple real-time connections can simultaneously share the resources of a packet switching network without resulting in congestion. The mechanism used to determine the admissibility of a real-time channel involves verifying at each intermediate link (along the path) in turn whether the RT channel's QoS requirements can be guaranteed. If a channel's requirements can be met at each of the intermediate links then we can accept the channel. If however, the channel's requirements cannot be met at any of the intermediate link then we can reject the channel. In fact the first such link that deems the channel inadmissible is sufficient to confirm that the channel would not be admissible.

  This problem is shown to be analogous to the end-to-end schedulability problem with the exception that the solution cannot be based on evaluating a channels admissibility by doing a complete (expensive) schedulability test. To this end, we proposed a heuristic approach that is based on the scaling factor computation. The room for accommodating a new channel into a system is expressed in terms of the maximum scaling factor with which the requirements of the channels already in the system can be scaled without violating their guarantees. This expression is then compared against the requirements of the new channel that is to be considered for admission. The expression being of a heuristic nature, we resorted to a simulation study (details in Chapter 8), the results of which have demonstrated the effectiveness of our approach.

Bibliography

[1] R. Arnold, F Mueller, D. B. Whalley, and M. Harmon. Bounding Worst-case Instruction Cache Performance. *Proceedings of IEEE Real Time Systems Symposium*, pages 172–181, December 1994.

[2] N. C. Audsley, A. Burns, M. Richardson, and A. Wellings. Hard Real-Time Scheduling:The Deadline Monotonic Approach. *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, 1991.

[3] N. C. Audsley, R. I. Davis, and A. Burns. Mechanisms for Enhancing the Flexibility and Utility of Hard Real-Time Systems. *Proceedings of the Real Time Systems Symposium*, pages 12–22, December 1994.

[4] Ricardo Bettati. End-To-End Scheduling to meet Deadlines in Distributed Systems. *PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign*, 1994.

[5] A. Burns, A. J. Wellings, C. M. Bailey, and E. Fyfe. The Olympus Attitude and Orbital Control System: A Case Study in Hard Real-Time System Design. *Technical Report YCS190, Department of Computer Science, University of York*, 1993.

[6] M. H. Klien et. al. *A Practitioners Handbook for Real-Time Analysis.* Kluwer Academic Publishers, 1993.

[7] Reza Etemadi. End-To-End Scheduling in Hard Real-Time Multiprocessor Systems. *Candidacy Report, Department of Computer Systems and Engineering, Carleton University, Canada*, 1995.

[8] D. Ferrari. Real-Time Communication in an Internetwork. *Journal of High Speed Networks*, 1(1):79–103, 1992.

[9] D. Ferrari. A New Admission Control Method for Real-Time Communication in an Internetwork. *S. Son, Ed., Advances in Real-Time Systems, Prentice Hall Englewood Cliffs, NJ*, pages 105–116, 1995.

[10] D. Ferrari. Real-Time Communication in an Internetwork. *Technical Report TR-92-072, International Computer Science Institute, Berkeley CA*, January 1992.

[11] D. Ferrari and C. C. Verma. A scheme for Real-Time Channel Establishment in Wide-area Networks. *IEEE Journal on Selected areas in Communications*, SAC-8(3):368–379, 1990.

[12] M. Garey and D. Johnson. Complexity Results for Multiprocessor Scheduling with Resource Constraints. *SIAM Journal of Computing*, 4(4):396–411, 1975.

[13] M. Garey and D. Johnson. *Computers and Intractability.* W. H. Freeman and Co., San Francisco, 1979.

[14] R. Gerber, S. Hong, and M. Saksena. Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes. *Proceedings of the Real-Time Systems Symposium*, pages 192–205, December 1994.

[15] T. Gonzales and S. Sahni. Flowshop and Jobshop scheduling: Complexity and Approximation. *Operations Research*, 26(1):220–244, 1978.

[16] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley and Sons, Inc; Wiley Professional Computing, 1991.

[17] D. D. Kandlur. *Networking in Distributed Real-Time Systems.* PhD thesis, University of Michigan, 1991.

[18] D. D. Kandlur, K. G. Shin, and D. Ferrari. Real-Time Communication in Multi-hop Networks. *Proceedings of the 11th International Conference on Distributed Computing Systems*, 1(2):184–194, May 1991.

[19] J. P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. *Proceedings of the IEEE Real-Time Systems Symposium*, pages 201–209, 1990.

[20] J. P. Lehoczky, L. Sha, and Y Ding. Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case. *Proceedings of the IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[21] J. Y. Leung and M. L. Merill. A Note Preemptive Scheduling of Periodic, Real-Time Tasks. *Information Processing Letters*, 11(3):115–118, November 1980.

[22] J. Y. Leung and J. Whitehead. On Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2(4):237–250, 1982.

[23] S. S. Lim, Y. H. Bae, G. T. Jang, B. D. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An Accurate Worst-case Timing Analysis for RISC Processors. *Proceedings of IEEE Real Time Systems Symposium*, pages 97–108, December 1994.

[24] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of ACM*, 20(1):46–61, 1973.

[25] J. W. S. Liu, K. J. Lin, W. K. Smith, A. C. Yu, J. Y. Chung, and W. Zhao. Algorithms for Scheduling Imprecise Computations. *IEEE Computer*, pages 58–68, May 1991.

[26] C. D. Locke. *Best-effort Decision Making for Real-Time Scheduling.* PhD thesis, Carnegie-Mellon University, 1986.

[27] C. D. Locke. Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. *Real Time Systems*, 4(1):37–53, March 1992.

[28] Nicholas Malcolm and Wei Zhao. Advances in Hard Real-Time Communication with Local Area Networks. *IEEE Trans on Computers*, pages 548–557, 1992.

[29] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.* PhD thesis, MIT., 1983.

[30] M. Di Natale and J. A. Stankovic. Dynamic End-to-End Guarantees in Distributed Real-Time Systems. *Proceedings of the Real Time Systems Symposium*, pages 216–227, December 1994.

[31] C. Y. Park. Predicting Deterministic Execution Times of Real-Time Programs. *Technical Report 92-08-02, Department of Computer Science and Engineering, University of Washington*, 1992.

[32] D. Picker and R. D. Fellman. Scaling and Performance of a Packet Queue for Real-Time Applications. *Proceedings of the Real-Time Systems Symposium*, pages 56–62, December 1994.

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

[34] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans on Computers*, 39(9):1175–1184, September 1990.

[35] L. Sha and S. S. Sathaye. A Systematic Approach to Designing Distributed Real-Time Systems. *IEEE Computer*, pages 68–78, September 1993.

[36] Wei-Kuan Shih. Scheduling in Real-Time Systems to Ensure Graceful Degradation: The Imprecise Computation and the Deferred-Deadline Approaches. *PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign*, 1992.

[37] Wei-Kuan Shih and J. W. S. Liu. On-line Scheduling of Imprecise Computations to Minimize Error. *Proceedings of the Real-Time Systems Symposium*, pages 280–289, December 1992.

[38] W. Stallings. *Data and Computer Communications.* Macmillan Publishing Company, New York, 1994.

[39] J. A. Stankovic, M. Di Natale M. Spuri, and G. C. Buttazo. Implications of Classical Scheduling Results for Real-Time Systems. *Department of CS, University of Massachussetts: Technical Report 95-23*, June 1994.

[40] J. A. Stankovic, M. Di Natale M. Spuri, and G. C. Buttazo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, 28(6):16–25, June 1995.

[41] J. A. Stankovic and K. Ramamritham. *Advances in Real-Time Analysis.* IEEE Computer Society Press, 1992.

[42] A. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, N.J., 1989.

[43] P. Thambidurai and K. S. Trivedi. Transient Overloads in Fault-Tolerant Real-Time Systems. *Proceedings of the Real-Time Systems Symposium*, pages 126–133, December 1989.

[44] S. R. Thuel and J. P. Lehoczky. On-Line Scheduling of Hard Deadline Aperiodic tasks in Fixed Priority Systems. *Proceedings of the Real-Time Systems Symposium*, pages 160–171, December 1993.

[45] K. Tindell. Adding Timing Offsets to Schedulability Analysis. *Technical Report YCS221, Department of Computer Science, University of York*, January 1994.

[46] K. Tindell. Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. *Technical Report YCS197, Department of Computer Science, University of York*, January 1994.

[47] C. Venkatramani and T. C. Chiueh. Supporting Real-Time Traffic on Ethernet. *Proceedings of IEEE Real Time Systems Symposium*, pages 282–286, December 1994.

[48] J. Xu. On Satisfying Timing Constraints in Hard Real Time Systems. *IEEE Trans on Software Engg*, 19(1):70–84, January 1993.

[49] R. Yerraballi. Replication in Distributed Real-Time Systems: Candidacy Report. *Department of CS, Old Dominion University*, 1994.

[50] R. Yerraballi and R. Mukkamala. Scalability of Real-Time Systems. *Submitted to the Special Issue of the Euromicro Journal on Real-Time Systems: Journal of System Architecture*, February 1995.

[51] R. Yerraballi and R. Mukkamala. Schedulability Related Issues in End-to-End Systems. *Proceedings of the First International Conference on Engineering of Complex Computer Systems*, November 1995.

[52] R. Yerraballi, R. Mukkamala, K. Maly, and H. Abdel-Wahab. Issues in Schedulability Analysis of Real-Time Systems. *Proceedings of the 7th Euromicro Workshop on Real Time Systems*, pages 87–92, June 1995.

[53] Q. Zheng. Real-Time Fault-Tolerant Communication in Computer Networks. *PhD thesis, Electrical Engineering: Systems, University of Michigan*, 1993.

[54] Q. Zheng and K.G. Shin. Fault-tolerant real-time communication in distributed computing systems. *Proceedings of 22nd Annual International Symposium on Fault-tolerant Computing*, pages 86–93, 1992.

Appendices

Appendix A

Simulation Results for Admission Control

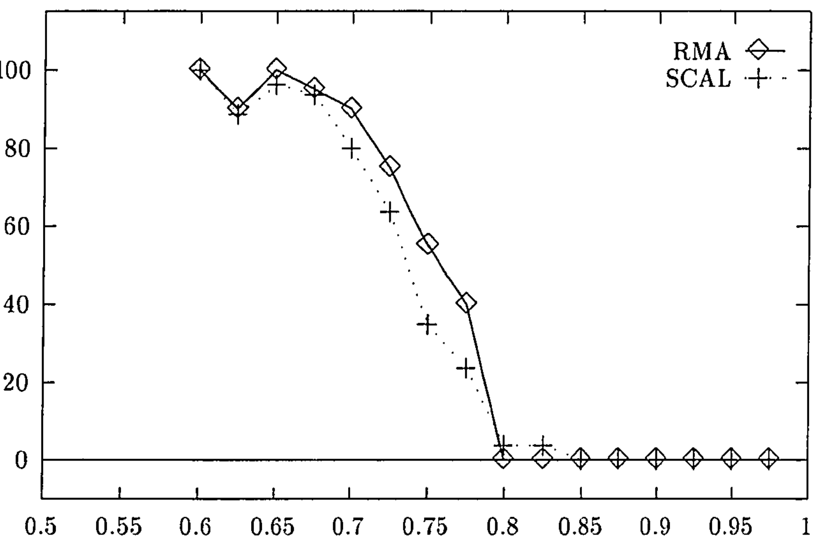

Figure A.1: $n = 4$ and $\kappa = 0.5$

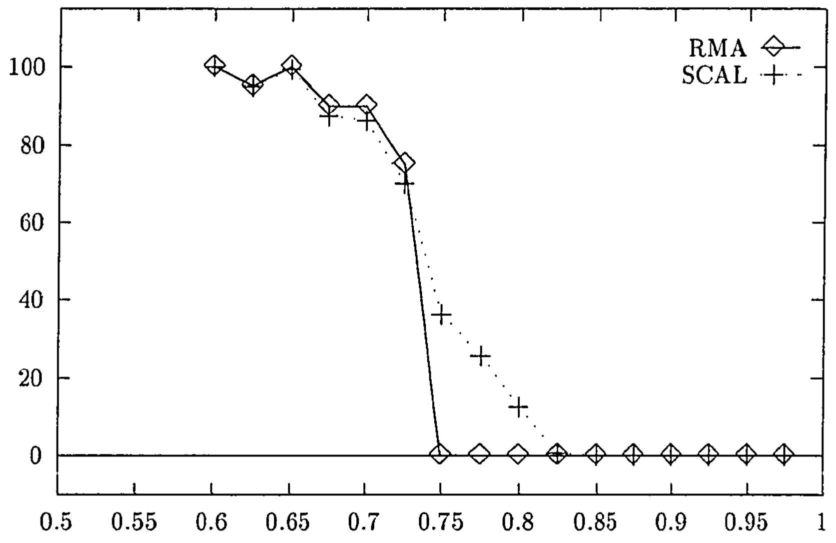

Figure A.2: $n = 8$ and $\kappa = 0.5$

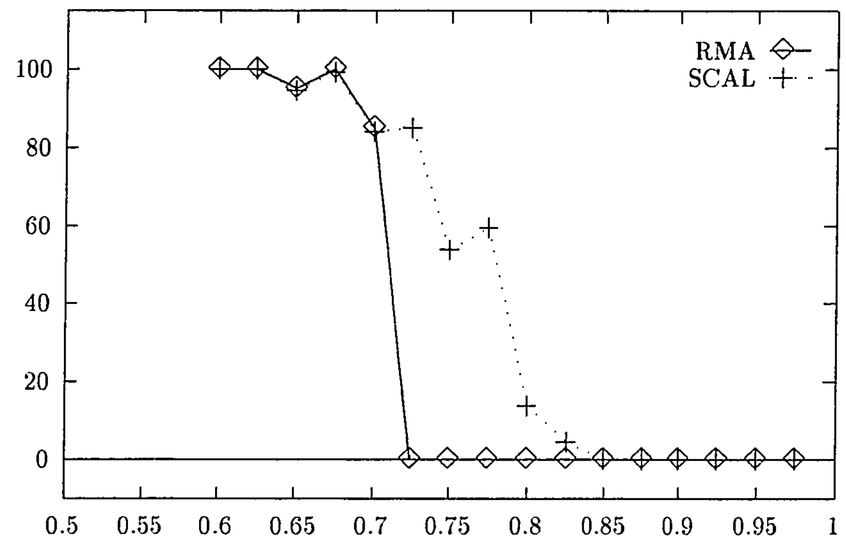

Figure A.3: $n = 12$ and $\kappa = 0.5$

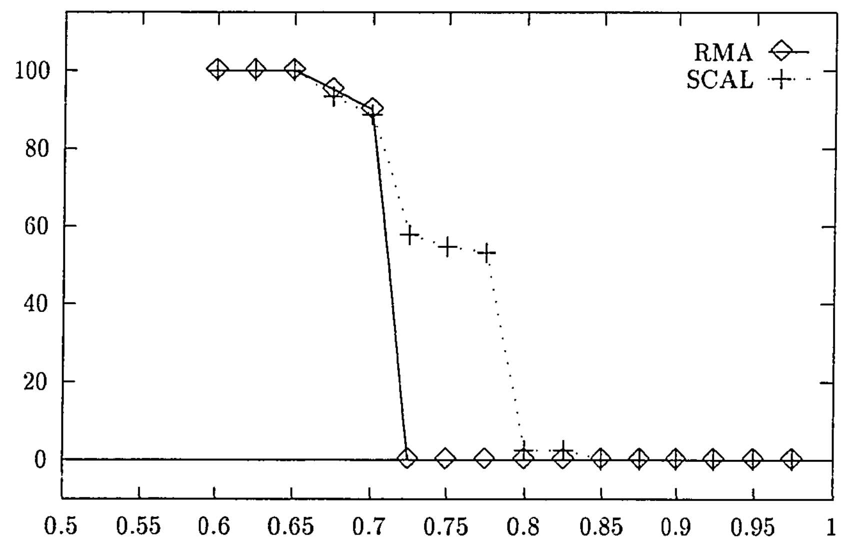

Figure A.4: $n = 16$ and $\kappa = 0.5$

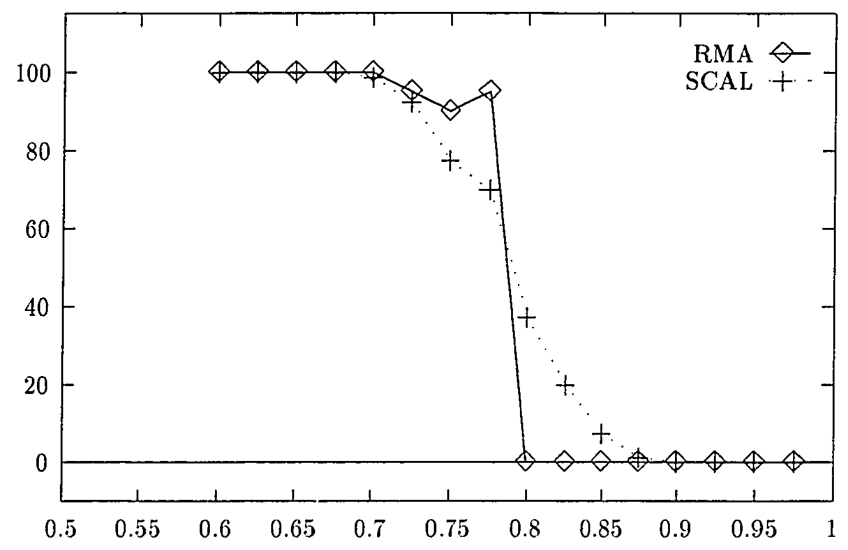

Figure A.5: $n = 4$ and $\kappa = 0.6$

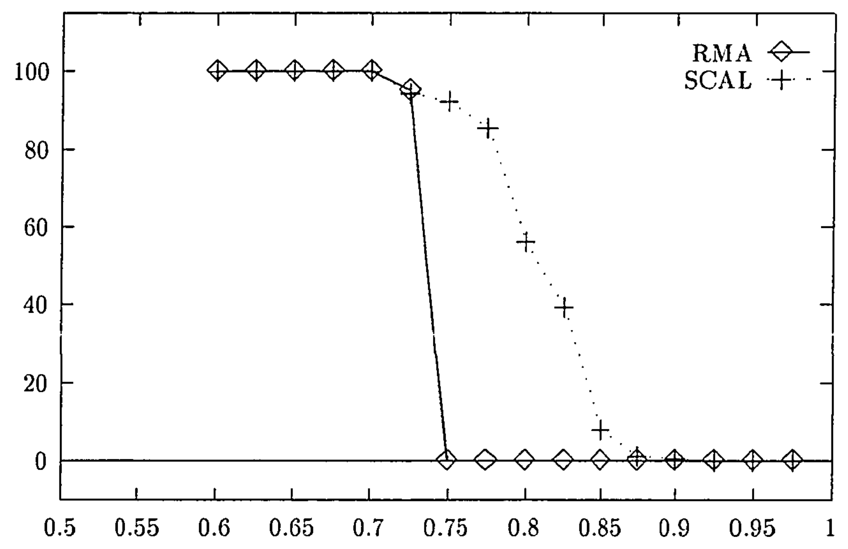

Figure A.6: $n = 8$ and $\kappa = 0.6$

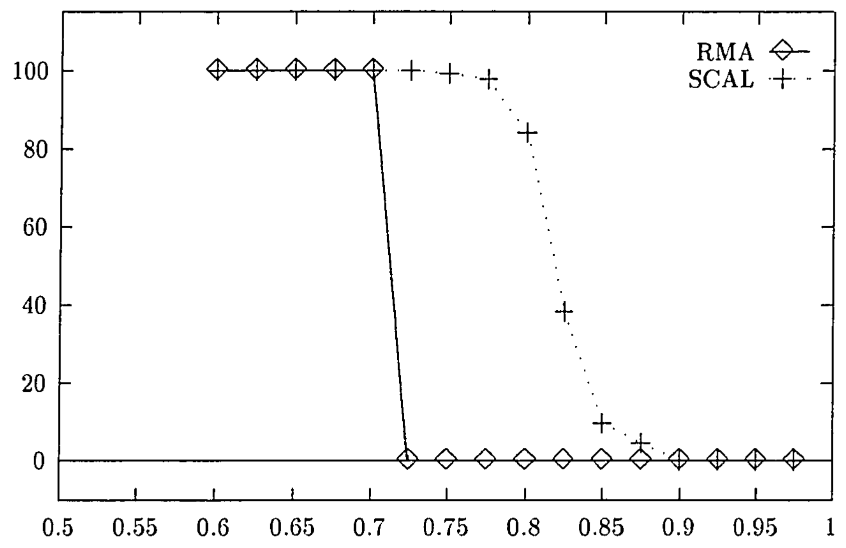

Figure A.7: $n = 12$ and $\kappa = 0.6$

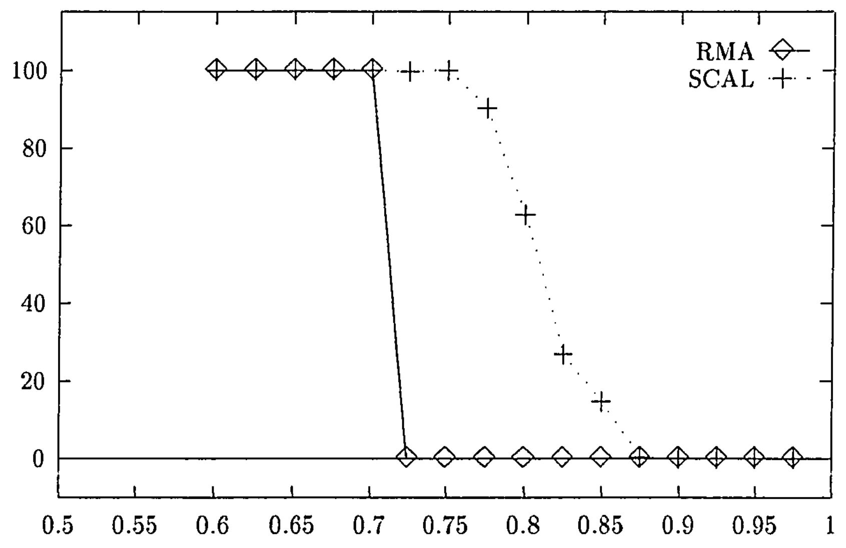

Figure A.8: $n = 12$ and $\kappa = 0.6$

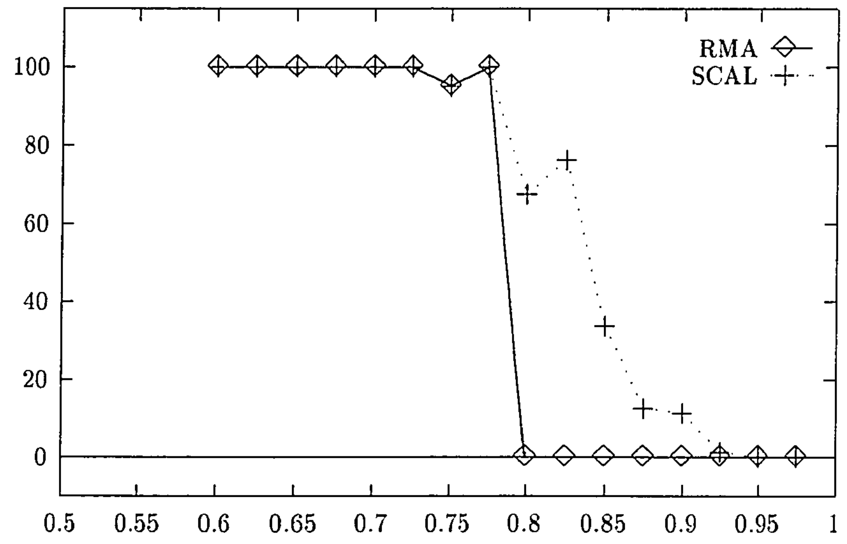

Figure A.9: $n = 4$ and $\kappa = 0.7$

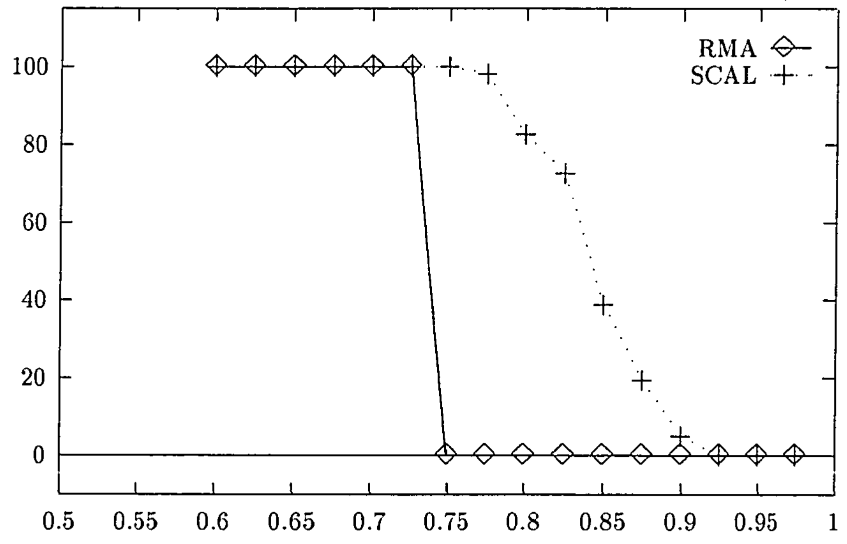

Figure A.10: $n = 8$ and $\kappa = 0.7$

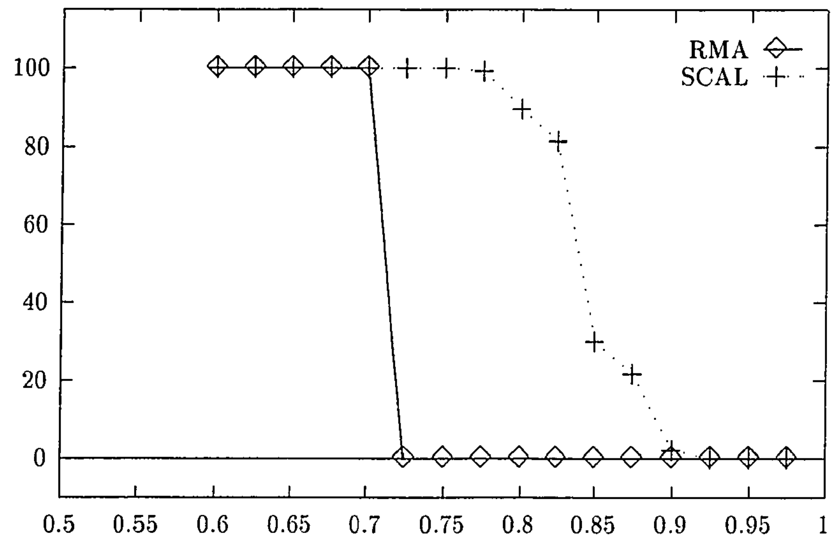

Figure A.11: $n = 12$ and $\kappa = 0.7$

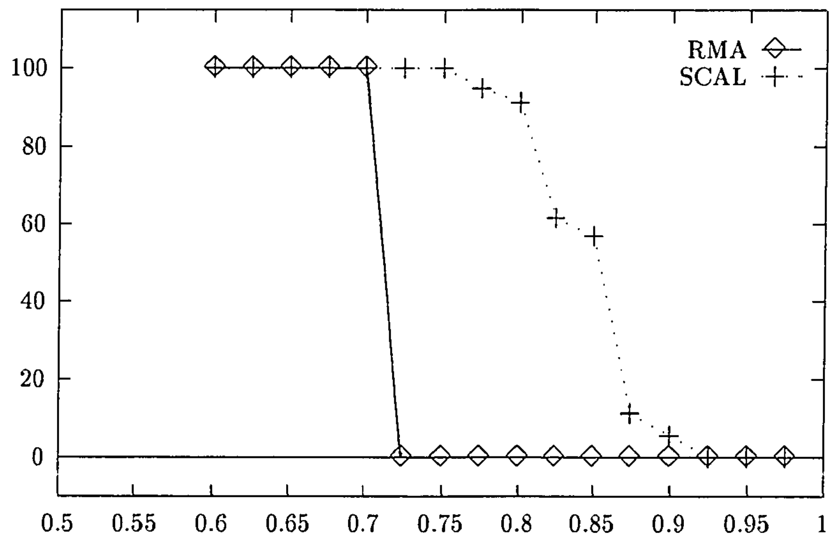

Figure A.12: $n = 16$ and $\kappa = 0.7$

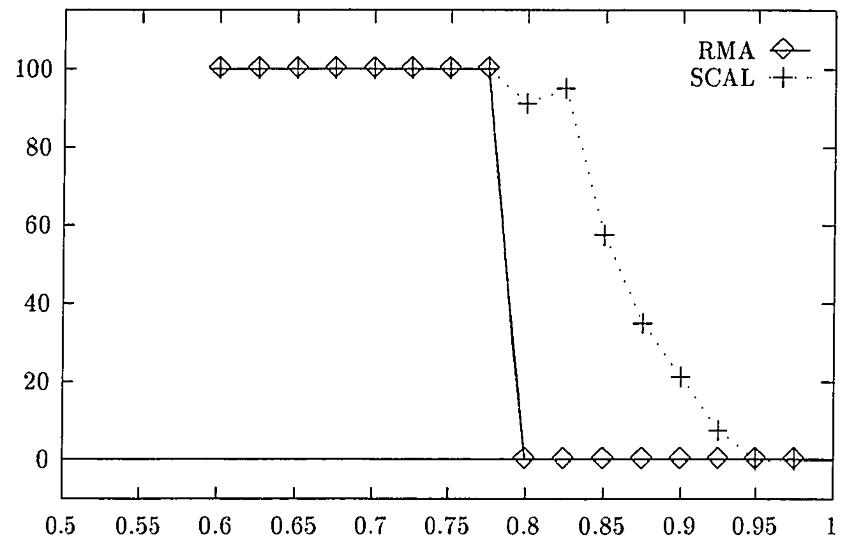

Figure A.13: $n = 4$ and $\kappa = 0.8$

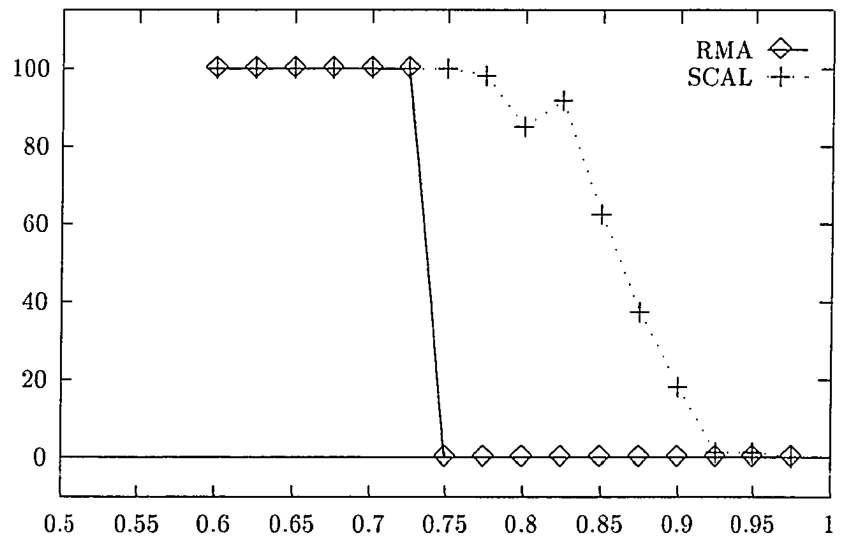

Figure A.14: $n = 8$ and $\kappa = 0.8$

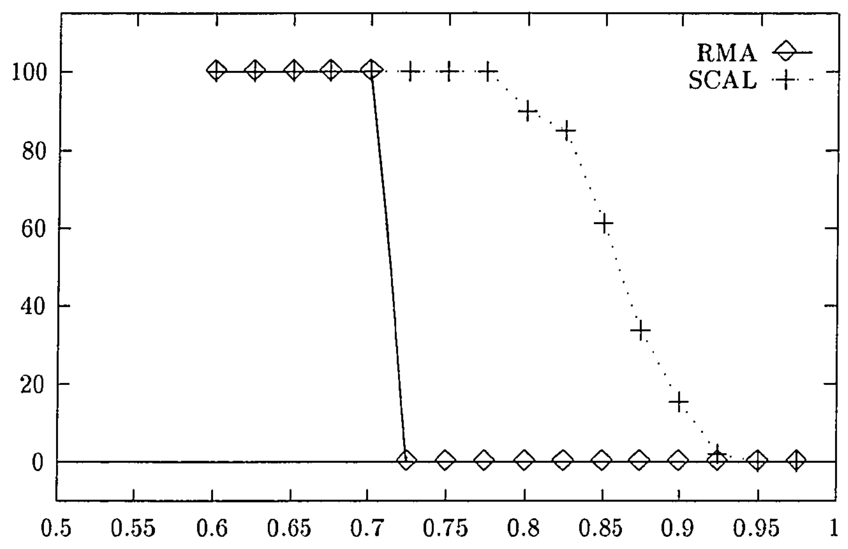

Figure A.15: $n = 12$ and $\kappa = 0.8$

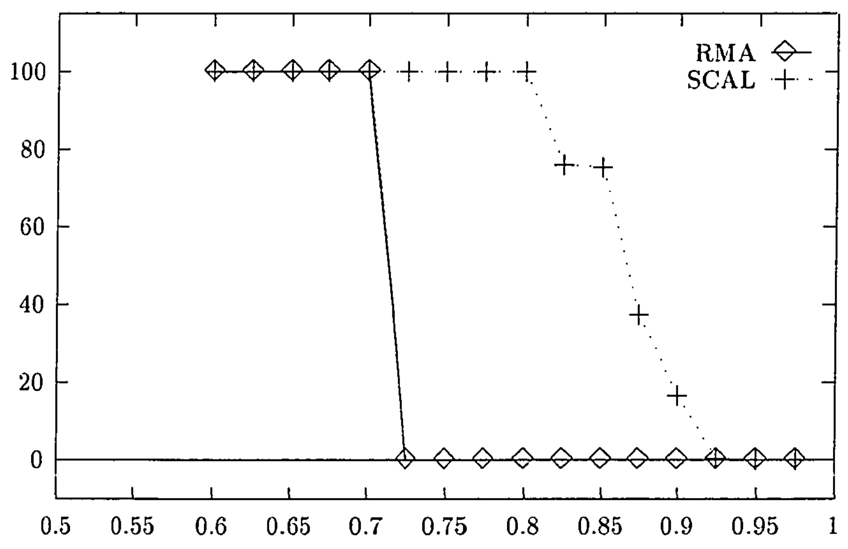

Figure A.16: $n = 16$ and $\kappa = 0.8$

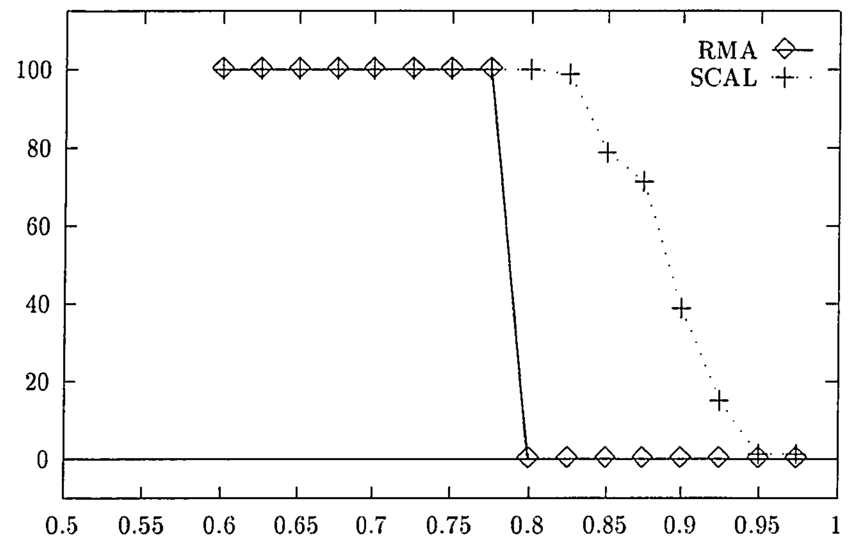

Figure A.17: $n = 4$ and $\kappa = 0.9$

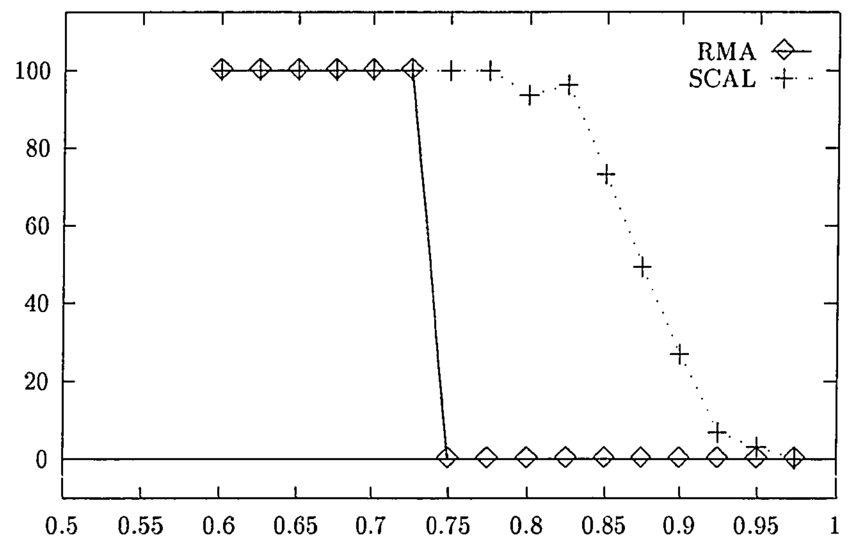

Figure A.18: $n = 8$ and $\kappa = 0.9$

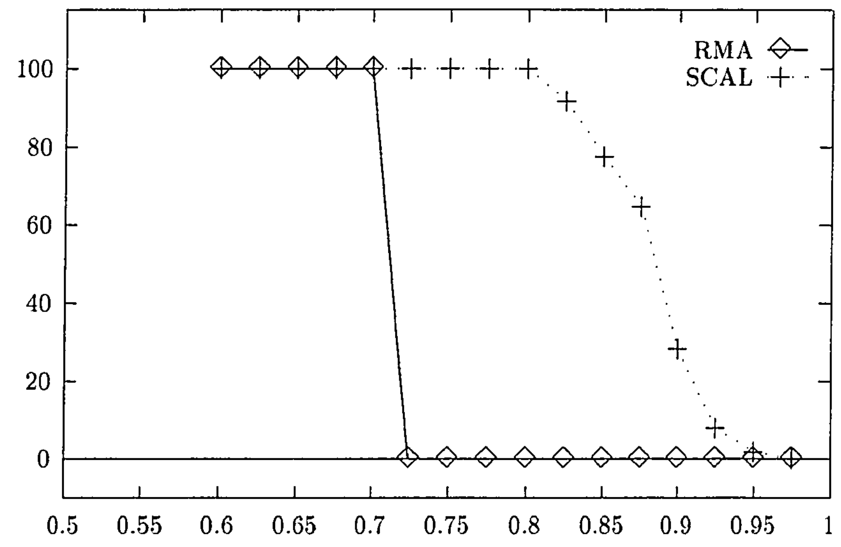

Figure A.19: $n = 12$ and $\kappa = 0.9$

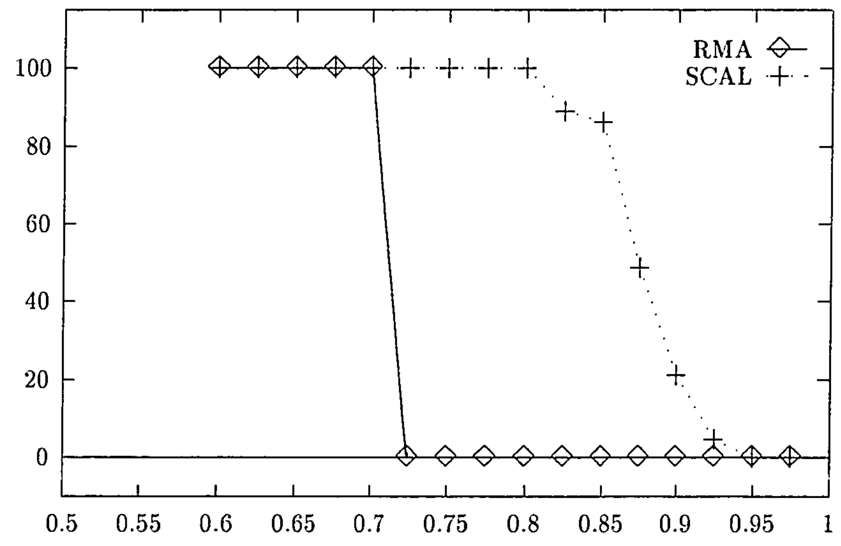

Figure A.20: $n = 16$ and $\kappa = 0.9$

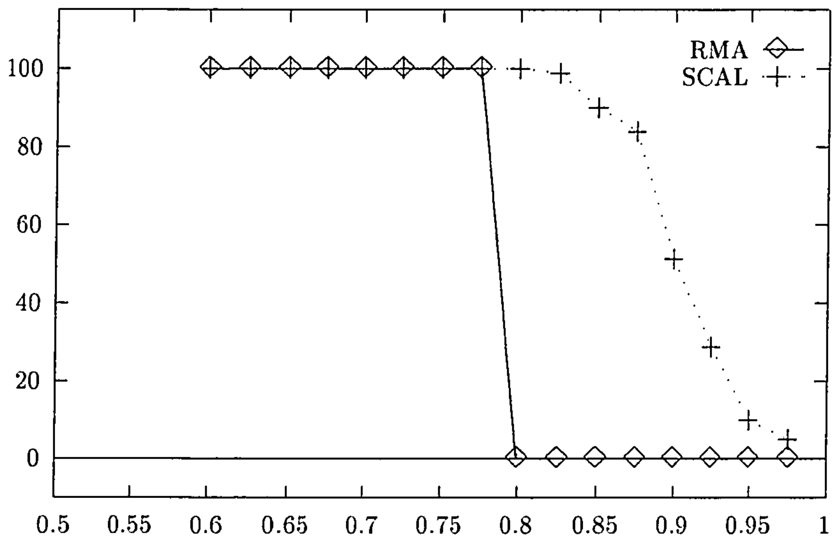

Figure A.21: $n = 4$ and $\kappa = 1.0$

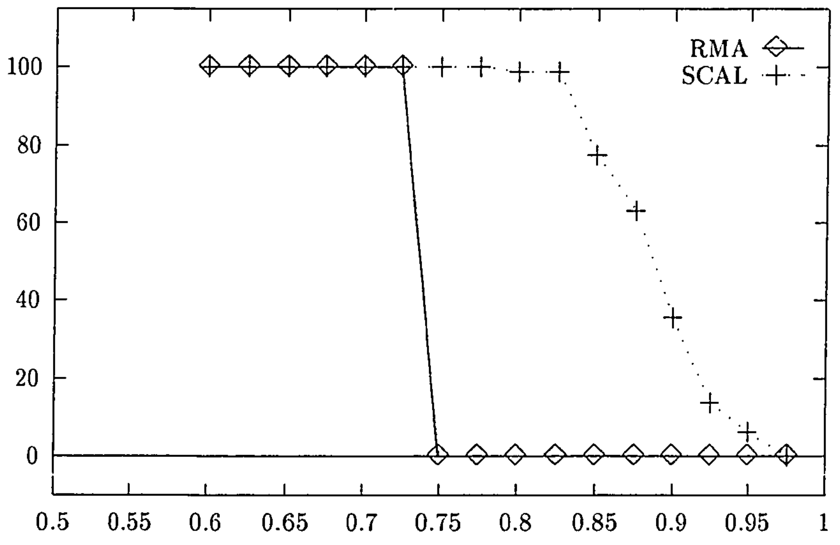

Figure A.22: $n = 8$ and $\kappa = 1.0$

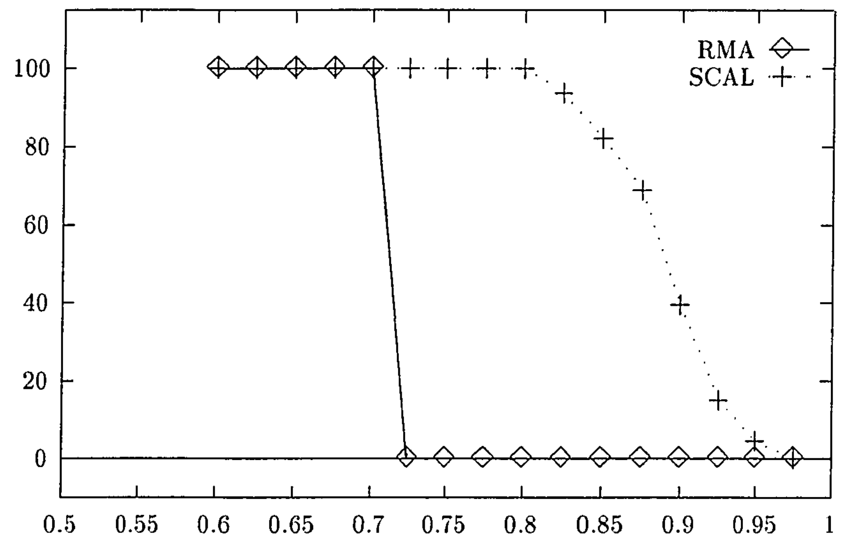

Figure A.23: $n = 12$ and $\kappa = 1.0$

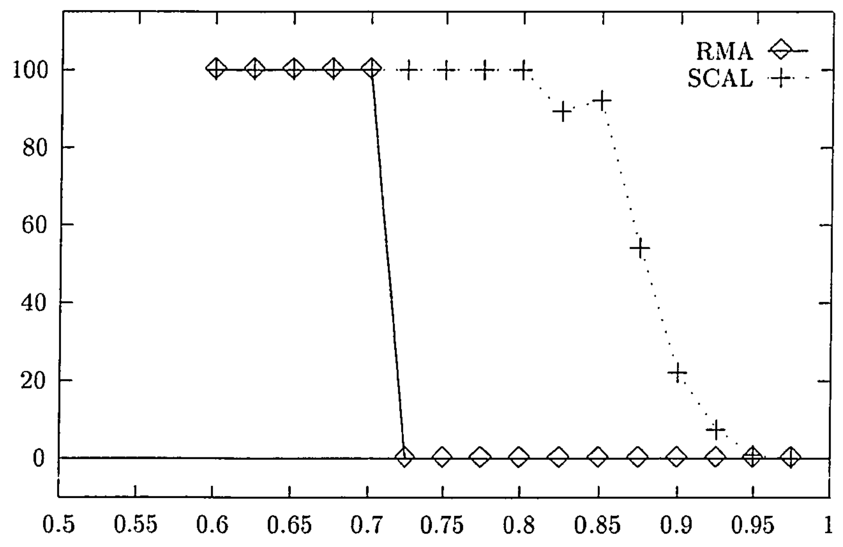

Figure A.24: $n = 16$ and $\kappa = 1.0$

Appendix B

The Olympus Attitude and Orbital Control System

The first two tables list the periodic tasks in the system and the last table lists the sporadic tasks. The parameter of periodicity of sporadic tasks is a derived parameter chosen for our study and not specified in the original study. The first parameter, critical level, is not used in our study, but essentially adds to the priority information of tasks. In general we could have had HARD, SOFT or FIRM categories of criticality and a special category called INTERRUPT, that implies that the corresponding task should be executed non-preemptively. In this case study there is only one task that is not categorized as HARD. Since this task (BUS_INTERRUPT) is assigned the highest priority, it is guaranteed to run un-preempted, satisfying the requirement of tasks that are categorized as INTERRRUPT.

The periodicity of sporadic tasks was chosen randomly to lie between the minimum inter-arrival and the average periodicity of periodic tasks. The minimum inter-arrival time parameter of sporadic tasks gives a lower bound on successive arrivals and is very rarely encountered in practice. Therefore, even if two successive arrivals of a sporadic task do occur at this minimum interval the probability of the next instance also occurring at this interval is very remote.

Table B.1: Periodic Tasks

Task Name	Characteristic	Value
REAL_TIME_CLOCK	Critical Level	HARD
	Priority	27
	Period	50.00
	Arrival Time	0.00
	Execution Time	0.28
	Deadline	9.00
READ_BUS_IP	Critical Level	HARD
	Priority	23
	Period	10.00
	Arrival Time	0.00
	Execution Time	1.76
	Deadline	10.00
COMMAND_ACUTUATORS	Critical Level	HARD
	Priority	20
	Period	200.00
	Arrival Time	50.00
	Execution Time	2.13
	Deadline	14.00
REQUEST_DSS_DATA	Critical Level	HARD
	Priority	19
	Period	200.00
	Arrival Time	150.00
	Execution Time	1.43
	Deadline	17.00
REQUEST_WHEEL_SPEEDS	Critical Level	HARD
	Priority	18
	Period	200
	Arrival Time	0.00
	Execution Time	1.43
	Deadline	22.00

Table B.2: Periodic Tasks - continued

Task Name	Characteristic	Value
REQUEST_IRES_DATA	Critical Level	HARD
	Priority	17
	Period	100.00
	Arrival Time	0.00
	Execution Time	1.43
	Deadline	24.00
PROCESS_IRES_DATA	Critical Level	HARD
	Priority	14
	Period	100.00
	Arrival Time	50.00
	Execution Time	8.21
	Deadline	50.0
CONTROL_LAW	Critical Level	HARD
	Priority	8
	Period	200.00
	Arrival Time	50.00
	Execution Time	22.84
	Deadline	200.00
PROCESS_DSS_DATA	Critical Level	HARD
	Priority	6
	Period	1000.00
	Arrival Time	200.00
	Execution Time	5.16
	Deadline	400.00
CALIBRATE_GYRO	Critical Level	HARD
	Priority	5
	Period	1000.00
	Arrival Time	200.00
	Execution Time	6.91
	Deadline	900.00

Table B.3: Sporadic Tasks

Task Name	Characteristic	Value
BUS_INTERRUPT	Critical Level	INTERRUPT
	Priority	62
	Min Inter-arrival	10.00
	Period	50.00
	Arrival Time	0.0
	Execution Time	0.18
	Deadline	0.63
TELEMETRY_RESPONSE	Critical Level	HARD
	Priority	15
	Min Inter-arrival	100.00
	Period	200
	Arrival Time	0.00
	Execution Time	3.19
	Deadline	30.00
READ_YAW_GYRO	Critical Level	HARD
	Priority	12
	Min Inter-arrival	100.00
	Period	500.00
	Arrival Time	0.00
	Execution Time	4.08
	Deadline	100.0
TELECOMMANDS	Critical Level	HARD
	Priority	4
	Min Inter-arrival	200.00
	Period	500.00
	Arrival Time	0.00
	Execution Time	2.50
	Deadline	200.00

Vita

Ramesh Yerraballi was born in 1970 in Hyderabad, India. He aquired his Bachelors in Computer Science and Engineering from Osmania University, Hyderabad, India, in 1991. He is due to receive his PhD degree in Computer Science from Old Dominion University, Norfolk, VA, in August 1996. Starting from the Fall of 96 he will be an Assistant Professor at Midwestern State University, Wichita Falls, TX. Dr. Yerraballi's research interests include real-time systems, distributed systems, high speed networks and performance issues in operating systems and network protocols. His teaching interests encompass all areas of Computer Science.

Permanent address: Department of Computer Science
Old Dominion University
Norfolk, VA 23529
U.S.A.

This dissertation was typeset with LaTeX[‡] by the author.

[‡] LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.