Copyright © 1995, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

REAL-TIME TASK LEVEL SCHEDULING IN THE POLIS CO-DESIGN ENVIRONMENT

by

Daniel Wayne Engels

Memorandum No. UCB/ERL M95/101

6 December 1995

REAL-TIME TASK LEVEL SCHEDULING IN THE POLIS CO-DESIGN ENVIRONMENT

by

Daniel Wayne Engels

Memorandum No. UCB/ERL M95/101

6 December 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

Real-Time Task Level Scheduling in the
POLIS Co-Design Environment

by

Daniel Wayne Engels
Master of Science in Electrical Engineering and Computer Science

University of California at Berkeley

Professor Alberto Sangiovanni-Vincentelli, Chair

The correctness of real-time systems depends on logical correctness as well as correct timing behavior (temporal correctness). The temporal correctness is largely determined by the scheduling algorithm used to set the software's execution ordering.

Creating scheduling algorithms and verifying that they generate a temporally correct ordering of software executions (a schedule) is a difficult problem, and, in general, scheduling software such that all timing constraints are met is an $\mathcal{NP}$-hard problem. In order to handle large complex scheduling problems, it is desirable to automate the generation and verification of scheduling routines. Methods to automate these tasks are presented in this paper.

This work is performed in the framework of the POLIS co-design environment which assists the user in designing small real-time systems using one or more microcontrollers and automatically generates the operating system and scheduling routines. The scheduling algorithms used in POLIS and their implementation details are presented. Scheduling routines generated by these algorithms as well as theoretical bounds on their execution times are described. In conclusion, it is shown that the experimental times are within the theoretically expected bounds.

Professor Alberto Sangiovanni-Vincentelli
Dissertation Committee Chair

Real-Time Task Level Scheduling in the POLIS Co-Design Environment

Copyright 1995

by
Daniel Wayne Engels

To my parents, Keith and Georgia.

They believe.

Contents

List of Figures vii

List of Tables viii

1 Introduction 1
- 1.1 Introduction 1
  - 1.1.1 The Main Characteristics of Real-Time Systems 2
- 1.2 Predictability in Real-Time Software 4
- 1.3 The POLIS Real-Time Design Environment 6
- 1.4 Thesis Overview 7

2 Definitions and Problem Analysis 9
- 2.1 Introduction 9
- 2.2 Real-Time Task Model 9
- 2.3 Real-Time Task Scheduling Algorithms 13
- 2.4 Complexity of Task Scheduling in the Real-Time Environment 15

3 Round Robin Scheduling 17
- 3.1 Introduction 17
- 3.2 Round Robin Scheduling 17
- 3.3 Advantages of the Round Robin Approach 19
- 3.4 Disadvantages and Other Issues of the Round Robin Approach 19
- 3.5 Implementation of the Round Robin Approach Within POLIS 19

4 Pre-Run-Time Scheduling 21
- 4.1 Introduction 21
- 4.2 Cyclic Executive 22
- 4.3 Pre-Run-Time Scheduling Algorithms 24
- 4.4 Sporadic Task Scheduling in Pre-Run-Time Schedules 28
  - 4.4.1 Servicing Sporadic Tasks 28
  - 4.4.2 Schedulability Analysis with Sporadic Tasks 30
- 4.5 Advantages of the Pre-Run-Time Approach 31
- 4.6 Disadvantages and Other Issues of the Pre-Run-Time Approach 32
- 4.7 Implementation of the Pre-Run-Time Approach Within POLIS 34

4.7.1 General-Pre-Run-Time Algorithm 35
4.7.2 Pre-Run-Time Algorithm 36

5 Static Priority Scheduling **41**
5.1 Introduction 41
5.2 The Rate Monotonic Scheduling Algorithm 42
5.3 The Deadline Monotonic Scheduling Algorithm 44
5.4 The Laxity Monotonic Scheduling Algorithm 45
5.5 Task Synchronization in Static Priority Systems 47
5.6 Non-Preemptive Static Priority Scheduling 50
5.7 Sporadic Task Scheduling in Static Priority Systems 52
5.7.1 The Priority Exchange Algorithm 52
5.7.2 The Deferrable Server Algorithm 54
5.7.3 The Sporadic Server Algorithm 55
5.7.4 Summary of Sporadic Task Handling 56
5.8 Static Priority Schedulability Analysis 58
5.8.1 Utilization Based Schedulability Analysis 58
5.8.2 Synchronous Schedulability Analysis 58
5.8.3 Asynchronous Schedulability Analysis 61
5.8.4 Non-preemptive Schedulability Analysis 61
5.9 Advantages of the Static Priority Approach 64
5.10 Disadvantages and Other Issues of the Static Priority Approach 65
5.11 Implementation of the Static Priority Approach Within POLIS 67
5.11.1 Routines Implemented Within POLIS 67
5.11.2 Generated Scheduling Routines 69

6 Dynamic Priority Scheduling **73**
6.1 Introduction 73
6.2 Dynamic Best Effort Scheduling Algorithms 74
6.3 Dynamic Planning-Based Scheduling Algorithms 76
6.4 Task Synchronization in Dynamic Priority Systems 77
6.5 Non-Preemptive Dynamic Priority Scheduling 82
6.6 Sporadic Task Scheduling in Dynamic Priority Systems 87
6.7 Advantages of the Dynamic Priority Approach 89
6.8 Disadvantages and Other Issues of the Dynamic Priority Approach 90
6.9 Implementation of the Dynamic Priority Approach Within POLIS 91
6.9.1 Routines Implemented Within POLIS 91
6.9.2 Generated Scheduling Routines 91

7 Results **95**
7.1 On-Line Scheduling Overhead 95
7.1.1 Derived Bounds for the On-Line Scheduling Overhead 95
7.1.2 Average On-Line Scheduling Overhead 97
7.1.3 Comparison with an Existing Real-Time Operating System 103
7.2 Synthesized Operating System Memory Requirements 105

7.2.1 Comparison With Existing Real-Time Operating Systems 105

8 Conclusions and Future Work **107**

A Scheduling Overhead Data **109**

Bibliography **115**

List of Figures

3.1 Generated Round Robin scheduling routine. 20

4.1 Branch and bound pre-run-time algorithm presented by Xu and Parnas. . . 25
4.2 General Pre-Run-Time (GPRT) algorithm implemented within POLIS. . . 35
4.3 Pre-Run-Time (PRT) algorithm implemented within POLIS. 37
4.4 Valid Initial Solution algorithm implemented within POLIS. 38

5.1 Audsley *et. al.*'s algorithm to determine if a synchronous static priority task set τ_n is schedulable. 60
5.2 Generated non-preemptive static priority scheduling routines. 69
5.3 Generated preemptive static priority scheduling routines. 71

6.1 Generated non-preemptive dynamic priority scheduling routines. 93
6.2 Generated preemptive dynamic priority scheduling routines. 94

7.1 Average Round Robin scheduling overhead. 98
7.2 Average Pre-Run-Time scheduling overhead. 98
7.3 Average Non-Preemptive Static Priority scheduling overhead. 99
7.4 Average Preemptive Static Priority scheduling overhead. 99
7.5 Average Non-Preemptive Dynamic Priority scheduling overhead. 100
7.6 Average Preemptive Dynamic Priority scheduling overhead. 100
7.7 Average Preemptive Static Priority scheduling overhead compared with the maximum possible scheduling overhead. 101
7.8 Average Preemptive Dynamic Priority scheduling overhead compared with the maximum possible scheduling overhead. 102
7.9 Comparison of all scheduling implementations as a function of the number of events in the system for a fixed task set size. 102
7.10 Comparison of all scheduling implementations, except for Pre-Run-Time, as a function of the number of tasks in the system for a fixed number of events. 103

List of Tables

5.1 Comparison of the PE, DS, and SS sporadic server algorithms. 57

7.1 Range of possible execution cycles for non-interrupt scheduling routines synthesized by POLISwith $N_E \equiv$ number of events in the system and $N_T \equiv$ number of tasks in the system. 96
7.2 Range of possible execution cycles for interrupt scheduling routines synthesized by POLISwith $N_E \equiv$ number of events in the system, $N_T \equiv$ number of tasks in the system and $\text{Priority}_T \equiv$ the priority of the interrupted task, T. 96
7.3 Execution times for some standard routines in the pSOS+ real-time operating system for the Intel 486DX2 33MHz processor. 104
7.4 Measured memory requirements (in bytes) of the synthesized POLISoperating system utilizing specific scheduling routines for a task set of size three with eight events. 105
7.5 Measured memory requirements (in bytes) of the synthesized POLISoperating system utilizing specific scheduling routines for a task set of size fourty-eight (48) with eighty (80) events. 106
7.6 Real-Time Operating Systems' memory (ROM) requirements. 106

A.1 Average scheduling overhead for the Round Robin scheduling routines. . . . 109
A.2 Average scheduling overhead for the Pre-Run-Time scheduling routines. . . 109
A.3 Average scheduling overhead for the Non-Preemptive Static Priority scheduling routines. 110
A.4 Average scheduling overhead for the Preemptive Static Priority scheduling routines. 110
A.5 Average scheduling overhead for the Non-Preemptive Dynamic Priority scheduling routines. 111
A.6 Average scheduling overhead for the Preemptive Dynamic Priority scheduling routines. 111
A.7 Standard deviation for the average Round Robin scheduling overhead. . . . 112
A.8 Standard deviation for the average Pre-Run-Time scheduling overhead. . . . 112
A.9 Standard deviation for the average Non-Preemptive Static Priority scheduling overhead. 113

A.10 Standard deviation for the average Preemptive Static Priority scheduling overhead. 113
A.11 Standard deviation for the average Non-Preemptive Dynamic Priority scheduling overhead. 114
A.12 Standard deviation for the average Preemptive Dynamic Priority scheduling overhead. 114

Acknowledgements

I am indebted to my research advisor, Professor Alberto Sangiovanni-Vincentelli, for his encouragement and support throughout the course of this work. I am also indebted to Professor Robert Brayton for agreeing to take time out of his busy schedule to read this report.

A large 'Thank you!' goes to Gitanjali Swamy and husband Sanjay, who's generosity and patience have given me a place to sleep and many useful suggestions on my writing. I also thank Rajeev Murgai for his couch and suggestions. Isn't that a Doctorate in Psychiatry that he has? The most sarcastic comment award goes to Stephen 'A Loon A Tick From Minnesota' Edwards without who's critiques I would not have realized that I write like a five year old high on Helium that (or is it 'which'?) was being passed around at a McDonalds birthday party.

All members of the CAD-group have been helpful, and I thank them all for the many informative discussions we have had.

They piled together all the remaining letters and dropped them into the bag. They shook them up.

"Right," said Ford, "close your eyes. Pull them out. Come on, come on, come on."

Arthur closed his eyes and plunged his hand into the towel full of stones. He jiggled them about, pulled out four and handed them to Ford. Ford laid them along the ground in the he order he got them.

"W," said Ford, "H, A, T ... What!"

He blinked.

"I think it's working!" he said.

Arthur pushed three more at him.

"D, O, Y ... Doy. Oh, perhaps it isn't working," said Ford.

"Here's the next three."

"O, U, G ... Doyoug ... It's not making sense I'm afraid."

Arthur pulled another two from the bag. Ford put them in place.

"E, T, doyouget ... Do you get!" shouted Ford. " It is working! This is amazing, it really is working!"

"More here." Arthur was throwing them out feverishly as fast as he could go.

"I, F," said Ford, "Y, O, U ... M, U, L, T, I, P, L, Y ... What do you get if you multiply ... S, I, X ... six ... B, Y, by, six by ... what do you get if you multiply six by ... N, I, N, E ... six by nine ..." He paused. "Come on, where's the next one?"

"Er, that's the lot," said Arthur, "that's all there were."

He sat back, nonplussed.

He rooted around again in the knotted up towel but there were no more letters.

"You mean that's it?" said Ford.

"That's it."

"Six by nine. Forty-two."

"That's it. That's all there is."

— Douglas Adams, *The Restaurant at the End of the Universe.*

Chapter 1

Introduction

1.1 Introduction

Real-time systems are those in which the correctness of the system depends not only on its logical correctness but also on correct timing behavior. Real-time systems range from a simple microcontroller to a highly complex, distributed system. They are found everywhere and control much of what we depend on in our everyday lives. Some of the more familiar real-time systems include the engine control unit in an automobile; the motion control system controlling robots; the process control systems used in nuclear power plants; and the air-traffic control systems guiding aircraft throughout the world. All of these systems depend upon microprocessors and microcontrollers to perform correctly.

Real-time systems are becoming more dependent upon microprocessors and microcontrollers, collectively referred to as *processors.* Certain functions are performed on the processor(s) by executing multiple threads of machine instructions, or *tasks.* These tasks compete with one another for limited resources including the processor(s), memory, and I/O access, and their execution must be scheduled so that each task's individual resource and timing requirements are met. If the tasks are not scheduled correctly, timing constraints may be missed with disastrous results. Thus, scheduling tasks properly is crucial to the correctness of real-time systems.

1.1.1 The Main Characteristics of Real-Time Systems

Typically, a real-time system is used as a *controlling* subsystem, with the environment as the *controlled* subsystem. A controlling system interacts with its environment based on the information from various sensors and inputs. The information presented to the controlling subsystem must be consistent with the actual state of the environment that is being controlled; otherwise, the actions of the controlling system can be disastrous. This makes periodic monitoring of the environment and timely processing of sensed information a must.

The difficulty of scheduling tasks to perform some or all of the functionality of the real-time system is dependent upon the characteristics of the system. The main characteristics that affect task scheduling are discussed below.

Task Timing Characteristics

Tasks that must complete execution shortly after they become ready to execute (are *invoked*) have tight timing constraints. These tasks force the operating system to react quickly and the scheduling algorithm to be fast.

Tight timing constraints also arise when a task's execution time is a significant fraction of the time it has to complete execution. For example, consider a task with an execution time of 8 time units that must complete execution 10 time units after its invocation. The operating system and the scheduling algorithm can take no more than 2 time units to react to an invocation and begin executing the task. Most real-time systems have many tasks with tight timing constraints, making it difficult to meet all timing constraints. In general, the tighter the timing constraints and the more tasks with tight timing constraints, the quicker the operating system and the scheduling algorithm must react to a task invocation.

Strictness of Timing Constraints

The strictness of the timing constraints, either hard or soft, is the value of completing some task after a timing constraint is missed. For a task with a *hard timing constraint*, there is no value in performing the task after the timing constraint is missed. For a task with a *soft timing constraint*, there is some diminished value in completing the task after the timing constraint is missed, so the task should be completed.

Different techniques are used to deal with hard real-time tasks and soft real-time tasks. Hard real-time tasks are often preallocated and scheduled on the required resources to guarantee that the timing constraints on such tasks are met 100% of the time. Soft real-time tasks are often scheduled on the required resources in such a way so as to obtain good average case performance.

Reliability

The reliability requirements of the system arise when certain tasks, known as *critical tasks,* must be guaranteed to meet their timing constraints under all operating conditions. That is, all critical tasks must be guaranteed to meet their timing constraints even under the worst-case conditions. Correct timing behavior of critical tasks is often guaranteed by off-line analysis of the system. Schemes that reserve resources for the critical tasks may also be used. Note, a task with hard timing constraints is not necessarily a critical task. For example, if the task that controls the stiffness of the suspension does not meet a hard timing constraint, the passengers in the vehicle experience a slightly less enjoyable ride. However, if the task controlling the deployment of the air bags does not meet its hard timing constraint, the passengers can experience additional injury or even death. The task controlling the deployment of the air bags is a critical task, whereas the task controlling the stiffness of the suspension is not a critical task.

The distinction between critical tasks and non-critical tasks with hard timing constraints is often made to ensure that the most important tasks are executed during unusual or unexpected situations. For example, during a head-on collision the correct performance of the suspension is irrelevant, but the correct performance of the air bags can save lives.

Environment

The environment in which a real-time system operates is often the most influential factor in the design of the system. In a well-defined environment, such as an automobile engine, real-time systems are often small and static, and all timing constraints may be guaranteed *a priori.* However, if the environment is not well-defined or may change over time, different techniques must be used to design the system. It is very difficult to generate a small, static system that will work flexibly in such an environment. Therefore, all timing

constraints may not be guaranteed *a priori.*

When timing constraints are not guaranteed, the system is not predictable. That is, it is not known if or when a timing constraint will be violated in the system. Predictability is a key component of all real-time systems. For example, consider an unpredictable system controlling the reactor temperature in a nuclear power plant. For years it might work correctly, but one day it may allow a melt-down. Since the system is not predictable, it is not known when that day will come.

1.2 Predictability in Real-Time Software

The ability to guarantee the temporal correctness of a real-time system determines the predictability of the system. That is, the ability to show, demonstrate, or prove that timing requirements are met subject to any assumptions made is critical to guaranteeing that a real-time system performs correctly at all times [SR90]. General and reliable techniques do not exist for verifying the predictability of software.

To ensure that timing constraints are predictably adhered to, the schedulability of the tasks must be verified prior to software execution. This verification is often referred to as *worst-case schedulability analysis*, or simply schedulability analysis [HS91, SHH91], and is an integral part in the development process of predictable real-time software. When the software is written in a language that supports real-time timing constructs, e.g., Real-Time Euclid [KS86], the schedulability analysis may be performed on the software directly, allowing a very low-level analysis of its timing characteristics. However, most real-time software is written in high-level languages, e.g., C, that do not provide the facilities that allow the programmer to control the real-time responsiveness of a task[1]. These languages consider the real-time details to be non-essential and hide them from the programmer. Consequently, the performance of software implemented in these languages is sensitive to the resource allocation protocols used in the operating system and is outside the control of the programmer.

The restricted access to the timing details of the software forces scheduling and schedulability analysis to be performed at a level (usually the task level) that does not analyze the the exact timing details of the software. Task level scheduling is related to the

[1] It may be argued that languages such as Ada do provide this access; however, the syntax of many of the 'timing' commands, such as the `delay` command in Ada, are more suitable for soft real-time systems than for hard real-time systems [HS90].

general flow-shop problem[2] and has been studied extensively in the literature [LL73, DL78, LW82, CSR86, BSR88, BMR90, Xu93].

The schedulability analysis of the various task level scheduling algorithms, or *scheduling algorithms*, depends strongly on the task level real-time system model, or *real-time model*, an abstraction containing information on the real-time system characteristics. A general real-time model consists of a set of tasks, processors, non-processor resources, communication links, and clocks. Each task is associated with a processor requirement, i.e., computation time, and non-processor resource requirements, e.g., a shared data bus. Tasks may communicate with each other, and communication restrictions may be specified for each task or for groups of tasks. In addition, relationships, e.g., precedence constraints, may be specified between tasks and between segments of tasks.

Different scheduling algorithms for the same real-time model may require different schedulability analysis tests. For example, given the following real-time model:

- a single processor,
- no non-processor resources,
- all tasks are ready for execution at constant intervals (i.e., all tasks are periodic),
- all tasks must complete execution before they are invoked again,
- constant task execution times,
- no inter-task communication, and
- no operating system or scheduling overhead;

a scheduling algorithm that lays out the entire time-line of execution for the tasks before the tasks are executed uses an implicit schedulability test. If the tasks so scheduled always meet their timing constraints, the set of tasks can be successfully scheduled. Explicit schedulability analysis tests need not be performed. However, a scheduling algorithm that uses task priorities to determine execution order at run-time must perform an explicit schedulability analysis test. A utilization based schedulability analysis test may be used for

[2]The general flow-shop problem can be formulated as follows. Each of n jobs $J_1 \dots J_n$ has to be processed on m machines $M_1 \dots M_m$ in that order. Job J_i, $i = 1, \dots, n$, thus consists of a sequence of m operations $O_{i1}, \dots, O_{im}$; O_{ik} corresponds to the processing of J_i on M_k during an uninterrupted processing time P_{ik}. M_k, $k = 1, \dots, m$, can handle at most one job at a time. The objective is to find a processing order on each M_k such that the time required to complete all jobs is minimized [LL78].

this scheduling algorithm. If the expected processor utilization is below a threshold, then the task set is guaranteed to be schedulable.

1.3 The POLIS Real-Time Design Environment

POLIS is a hardware/software codesign environment targeted towards reactive (control dominated) real-time system design [CGJ+94]. These systems are relatively small, real-time controllers composed of software on one (or few) processor(s) and some semi-custom hardware components operating in a well-defined environment.

The designer specifies the design in a high-level language language such as Esterel [BCG91] or a high-level graphical language such as State Charts [DH89]. The high-level description is transformed into the internal representation used by POLIS. The internal representation of the real-time system is based upon the Codesign Finite State Machine (CFSM) formalism, an extension of the classical finite state machine. By manipulating the high-level description, the designer is able to indirectly control the size of the resulting CFSMs.

The basic model is a network of interacting CFSMs that communicate through a very low-level primitive: events. A CFSM, and possibly the environment in which the system operates, broadcasts events that one or more CFSMs or the environment can detect.

Events directly implement a communication protocol that does not require an acknowledgement. The receiver waits for the sender to emit the event, but the sender can proceed immediately after emission. An implicit buffer between the sender and each receiver saves exactly one event until it is detected or overwritten with another occurrence of the event. This allows each CFSM to detect an event at most once any time after the event's emission and until another event of the same type overwrites it. This approach lends itself to an efficient hardware implementation with synchronous circuits, as well as a software implementation using either polling or interrupts to detect events.

Each CFSM in the network is assigned an implementation, either hardware or software. The hardware synthesis and the software synthesis proceed from the appropriate CFSMs. The hardware is synthesized using the standard logic net list model used by logic synthesis systems such as SIS [SSL+92, SSM+92]. The software is synthesized using the software graph (s-graph) model, an abstraction of the basic instructions of a very simple computer model [CGH+94]. In addition, the operating system, including the task

level scheduler, is automatically generated with the interfaces between the hardware, the software, and the operating system automatically synthesized as well.

Each CFSM implemented in software corresponds to a task in the system. The tasks synthesized from the CFSMs can have small code sizes with short run times. This is possible since the designer has control over the size of the CFSMs in the system, and the size of the CFSM directly affects the synthesized code size and run time. The synthesized operating system may use either a polling-based or an interrupt-based event detection method, allowing for easy implementation of most scheduling algorithms in the synthesized operating system.

Validation is used to verify that a synthesized design satisfies its specification. Formal verification is used to debug both the specification with respect to high-level properties and the implementation with respect to lower-level properties. Error traces describing the reasons for failing to satisfy a desired property are provided to allow the designer to fix the errors and try alternate solutions. Simulation may also be used to verify the cases that would be difficult for formal verification techniques to handle.

1.4 Thesis Overview

Real-time systems must predictably produce logically correct results while meeting all timing constraints. POLIS is a design environment which supports the design and verification of real-time systems, and is capable of supporting many real-time models; thus, the operating system which is synthesized by POLIS may contain any one of a number of scheduling algorithms. At the time POLIS synthesizes the operating system, a schedulability analysis test is performed to verify that the desired scheduling algorithm produces a valid schedule for the given task set.

The various scheduling algorithms and their associated schedulability analysis tests supported by POLIS are presented in the following. The related theory behind each of the scheduling algorithms and the schedulability analysis tests are presented. The details of the implementation within POLIS are also presented. More specifically,

- Chapter 2 presents the representation of a task used throughout, the main types of task level schedules, and the complexity analysis for several types of schedules with different real-time models.

- Chapter 3 deals with the round robin scheduling algorithm. The ideas behind the round robin algorithm implemented in POLIS and the implementation within POLIS are presented.
- Chapter 4 deals with pre-run-time scheduling algorithms. The cyclic executive approach and an algorithmic approach are presented. The pre-run-time algorithms implemented within POLIS are presented.
- Chapter 5 deals with static priority scheduling algorithms. The theory behind the Rate Monotonic scheduling algorithm, the Deadline Monotonic scheduling algorithm, and the Laxity Monotonic scheduling algorithm is presented, and the more important extensions of the theory are discussed. The static priority algorithms implemented within POLIS are presented.
- Chapter 6 deals with dynamic priority scheduling algorithms. The theory behind the Earliest Deadline First and the Minimum Laxity First best effort scheduling algorithms are presented, and the more important extensions of the theory are discussed. The dynamic priority algorithms implemented within POLIS are presented.
- Chapter 7 analyzes the performance of the scheduling algorithms generated by POLIS.
- Chapter 8 summarizes the contributions of this work and presents directions for future work.

Chapter 2

Definitions and Problem Analysis

2.1 Introduction

The *task model* is an abstraction containing information on the characteristics of a task. The information contained in the task model affects the types of scheduling algorithms and any associated schedulability analysis tests which may be used to schedule all of the tasks in the system, the *task set*. For example, a task model which contains information only upon the rate at which tasks are invoked, i.e. their period, would preclude the use of a pre-run-time scheduling algorithm (see Chapter 4), but other types of scheduling algorithms could still be used. Information on the worst case execution times for each task is required for any schedulability analysis test to be performed.

The general task model used in the sequel is presented with relevant terminology. The general types of scheduling algorithms are then discussed. Finally, the complexity of task level scheduling under various real-time models is examined.

2.2 Real-Time Task Model

The task is represented by a task model, an abstraction containing information on the major task characteristics. This information may be used by the designer for determining which scheduling algorithm to use. The task model is also used by the chosen scheduling algorithm and its related schedulability analysis tests. The following task model is general enough for use with most task sets and most scheduling algorithms (with their associated schedulability analysis tests).

A *task* is a sequential program invoked by a particular event. An event is a stimulus generated either external or internal to the system executing the tasks at some maximum frequency. Each task will be executed at a time determined by the scheduling algorithm. A more formal definition of a task is given below.

Definition 2.2.1 *A* task *T is a 6-tuple* $(r,\ c,\ p,\ d,\ D,\ P)$ *where*

r: the release time *of task T: the time when task T is first invoked.*

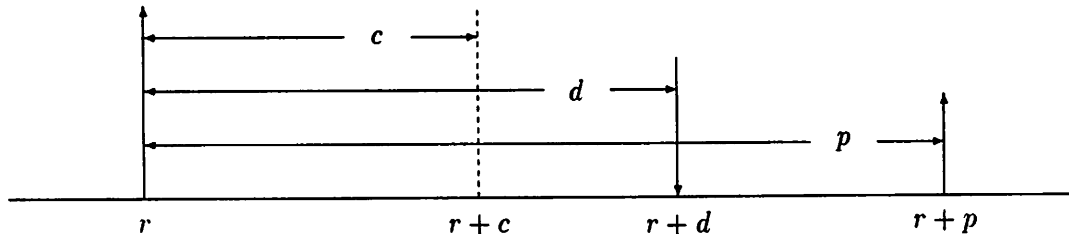
c: the maximum computation time *of task T: the maximum time required by task T to execute to completion on a dedicated uniprocessor.*

p: the period *of task T, where $p \geq c$: the minimum interval between successive invocations of task T.*

d: the deadline *of task T, where $p \geq d \geq c$: the time after the invocation by which execution of task T must be completed.*

D: the type of deadline *of task T: either a soft deadline or a hard deadline.*

P: the classification *of task T: either periodic, sporadic, or aperiodic.*



r, c, p, and d are assumed to be integer values. ■

Three broad classifications of tasks are commonly used [ABRW91, Jef92, KLR94]: periodic tasks, sporadic tasks, and aperiodic tasks.

Periodic tasks are invoked at constant intervals, i.e., invoked at a fixed frequency. In other words, a periodic task is invoked exactly p time units apart. This yields the following behavior rules for the invocation and execution of periodic task $T_p = (r_p,\ c_p,\ p_p,\ d_p,\ D_p,\ P_p)$. If t_i is the time of the ith invocation of task T_p, then

1. The first invocation of task T_p will occur at time $t_1 = r_p$.

2. The $(i+1)^{\text{th}}$ invocation of task T_p *occurs at* time $t_{i+1} = t_i + p_p$.

3. The ith invocation of task T_p may begin execution no earlier than time t_i.

4. The ith invocation of task T_p must complete execution no later than $t_i + d_p$ if its deadline is to be met.

Sporadic tasks are invoked at random intervals with a minimum time between successive invocations. Behavior rules for the invocation and execution of sporadic task $T_s = (r_s,\ c_s,\ p_s,\ d_s,\ D_s,\ P_s)$ are similar to those for a periodic task. If t_i is the time of the ith invocation of task T_s, then

1. The first invocation of task T_s will occur at time $t_1 = r_s$.

2. The $(i+1)^{\text{th}}$ invocation of task T_s *occurs no earlier than* time $t_i + p_s$.

3. The ith invocation of task T_s may begin execution no earlier than time t_i.

4. The ith invocation of task T_s must complete execution no later than $t_i + d_s$ if its deadline is to be met.

A sporadic task acts like a periodic task when it is continually invoked p_s time units apart.

Aperiodic tasks are invoked at random intervals with no minimum time between successive invocations. The arrival patterns for aperiodic tasks may be described by probability density functions. Aperiodic tasks may have hard deadlines, but the timing requirements are usually stated in terms of satisfying an average response time requirement. Since it is impossible to guarantee the schedulability of tasks with hard timing constraints that may be invoked infinitely often at a single instant in time, aperiodic tasks will not be considered.

The following definitions are related to tasks.

Definition 2.2.2 *The* response time *of an invocation of task T is the time span between the time when the task is invoked, t_i, and the time when the task has just finished executing (completion time), t_f, so $(t_f - t_i) \geq c$.*

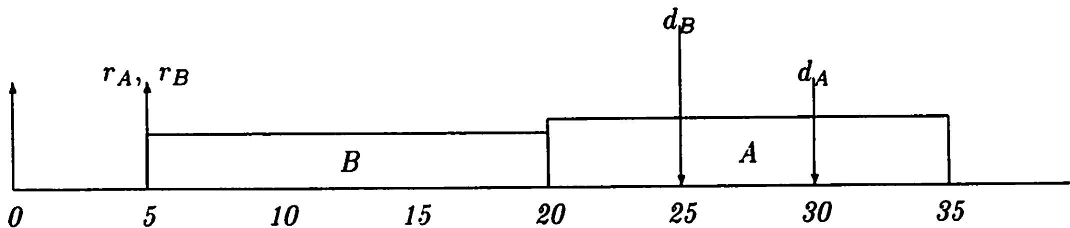
Definition 2.2.3 *The* critical instant *of a task, T, is an instant at which an invocation of T will have its largest response time.*

Definition 2.2.4 *The* critical zone *of a task, T, is the time interval between the critical instant and the completion time, t_f, of the task.*

Definition 2.2.5 *An* overflow *occurs at time t if t is the deadline of a task that has not completed execution.*

The following example illustrates these definitions.

Example 2.2.1 *Consider the task set containing the following two tasks where $T = (r, c, p, d)$: $T_A = (5, 15, 30, 25)$ and $T_B = (5, 15, 30, 20)$. Using an earliest deadline first heuristic, i.e., allocating the processor to the ready task with the earliest deadline, the following schedule is obtained for a uniprocessor system. Note that this schedule repeats every 30 time units.*



The response time *for T_A is calculated by subtracting the release time of T_A from the completion time of T_A, $35 - 5 = 30$. The* response time *for T_B is $20 - 5 = 15$.*

Since the above schedule repeats, the maximum response times for the two tasks are 30 and 15 for T_A and T_B respectively. These maximum response times occur at times $t = 5, 35, 65, \ldots$ By definition, these are the critical instants *for both T_A and T_B.*

The critical zone *of T_A occurs between time $t = 5$, the critical instant of T_A, and time $t = 35$, the completion time of T_A, for a critical zone of 30. The* critical zone *of T_B occurs between times $t = 5$ and $t = 20$ for a critical zone of 15.*

An overflow *occurs at time $t = 30$ since T_A has not completed execution by its deadline.*

Note, there is no schedule that would successfully schedule this task set in a uniprocessor system. ■

Precedence, exclusion, preempt, and *before* relations are now defined.

Definition 2.2.6 *Task T_i is said to* precede *another task T_j (T_i PRECEDE T_j or $T_i \prec T_j$) if T_j can only start execution after T_i has completed.*

Precedence relations may exist between tasks when one task invokes another task or one task requires information produced by another task.

Definition 2.2.7 *Task T_i is said to* exclude *another task T_j (T_i EXCLUDE T_j) if no execution of T_j can occur between the time T_i starts its execution and the time T_i completes.*

Exclude relations may exist between tasks when some tasks must prevent access by other tasks to shared resources such as data or I/O devices.

Definition 2.2.8 *Task T_i is said to* preempt *another task T_j (T_i* PREEMPT *T_j) if the execution of T_j is interrupted by the execution of task T_i.*

Preempt relations are generally used only in pre-run-time scheduling algorithms, forcing one task to preempt another.

Definition 2.2.9 *A task T_i is said to come* before *another task T_j (T_i* BEFORE *T_j) if the execution of T_j cannot begin before the execution of T_i has begun.*

Before relations may exist between tasks when synchronization is required. This relation is more useful in a multiprocessor environment than in a uniprocessor environment.

2.3 Real-Time Task Scheduling Algorithms

We are interested in the scheduling of sets of tasks that compete for processor and non-processor resources. Given a set of tasks $\tau_n = \{T_1, \ldots, T_n\}, T_i = (r_i,\ c_i,\ p_i,\ d_i,\ D_i,\ P_i)$, a scheduling algorithm specifies at each time t which task, if any, shall execute. If the scheduling algorithm generates a complete schedule off-line, the resulting schedule is called a pre-run-time schedule[1]. Pre-run-time schedules allow for easy schedulability analysis. If a pre-run-time schedule meeting all timing constraints cannot be generated for the task set, then the timing constraints cannot be met. Pre-run-time schedules can be efficient due to a minimal amount of overhead and a guaranteed lack of contention for resources. However, they do have some drawbacks. Any change in the task set requires that the pre-run-time schedule be recomputed and tested. Pre-run-time schedules are also not well suited to handle sporadic tasks. Therefore, they do not offer as good a response time to sporadic tasks as may be obtained by other methods.

Run-time scheduling algorithms[2] compute the schedule for the tasks on-line from the pool of invoked tasks. The run-time scheduler may or may not assume any knowledge about future invocations of the tasks in the task set.

[1]Pre-run-time schedules may also be referred to as "off-line schedules" and "static schedules."
[2]Run-time schedules may also be referred to as "on-line schedules" and "dynamic schedules."

Run-time schedules are often classified as *static priority* or *dynamic priority.* A static priority scheduling algorithm assigns a fixed priority to each task (and every instance of that task) prior to the execution of the first task. At each instant, the processor executes a ready task with the highest priority. A dynamic priority scheduling algorithm allows the priority of a task to change any time after it is invoked and before it is completed. At each instant that a task may be executed, each task is assigned a priority based upon some criteria, and a ready task with the highest priority is allocated the processor.

A second classification of run-time scheduling algorithms is *preemptive* versus *non-preemptive.* A preemptive scheduling algorithm allows the processor to be allocated to a task that is ready to run before the currently executing task has completed. Thus, the executing task is preempted by the ready task. Preemptive algorithms provide flexibility especially when dynamic priority is also employed. However, this flexibility does have a price. When a task is preempted, its current state must be saved in memory before the next task can begin execution. This context switch requires an amount of time proportional to the amount of state information that must be saved.

Preemption must not violate the exclusion relations between tasks. Preemptive algorithms must prevent high priority tasks from missing deadlines in the presence of lower priority tasks that require a non-preemptive resource required by the higher priority task. All of these problems add to the complexity and to the overhead of preemptive algorithms.

Non-preemptive scheduling algorithms are simpler than preemptive algorithms. They have an implicit exclusion relation between every pair of tasks that execute on the same processor. Non-preemptive algorithms are simple to implement and easy to analyze, but they too have some problems. The main problem with non-preemptive scheduling algorithms is that high priority tasks may miss deadlines due to long running low priority tasks. It is also possible that a low priority task will miss a deadline in the presence of higher priority tasks. In general, non-preemptive scheduling algorithms are unable to find a valid schedule for all task sets that preemptive algorithms are able to find a valid schedule.

Definition 2.3.1 *A schedule is considered* valid[3] *if and only if every task in the task set is always able to meet its deadline when the task is released at its specified release time. If the release times are not specified, then a schedule is considered valid if and only if every task in the task set is always able to meet its deadline for all possible release times.*

[3] *Feasible* is often used as a synonym for valid in the literature.

Definition 2.3.2 *A task set is considered* schedulable *if there exists a valid schedule for the task set.*

We note that some types of scheduling algorithms will not be able to produce a valid schedule for a schedulable task set. This is due to the limitations placed on the scheduling algorithm (e.g., no preemption, no inserted idle time, etc.) and the limitations of the real-time model used.

2.4 Complexity of Task Scheduling in the Real-Time Environment

We are interested in the general problem of producing a valid schedule for a task set on one or more processors. Ideally, the scheduling algorithm will find a valid schedule if the task set is schedulable. However, it is not always easy to find a schedule and verify its validity because most scheduling problems, differentiated by their real-time model, have been found to belong to the class of $\mathcal{NP}$-hard problems.

Some of the more interesting scheduling problems for which complexity constraints have been derived are given below. These include problems for which the complexity has been found to belong to the class of $\mathcal{NP}$-hard problems and those for which the complexity has been found not to belong to the class of $\mathcal{NP}$-hard problems.

Theorem 2.4.1 ([GJ79] [BS89]) *Given a set of tasks, τ_n, with arbitrary execution times and preemption is not allowed, deciding whether τ_n is schedulable on one processor is $\mathcal{NP}$-complete.*

Theorem 2.4.2 ([JSM91]) *Non-preemptive scheduling of periodic tasks when their release times are specified is $\mathcal{NP}$-hard.*

Theorem 2.4.3 ([LW82]) *Given a set of tasks, τ_n, with arbitrary release times, arbitrary deadlines, and allowing arbitrary preemption, deciding whether τ_n is schedulable on one processor is $\mathcal{NP}$-hard.*

Theorem 2.4.4 ([LW82]) *Given a set of tasks, τ_n, with arbitrary release times, arbitrary deadlines, and allowing arbitrary preemption and a priority assignment ρ, deciding whether the schedule produced by ρ is valid is $\mathcal{NP}$-hard.*

Theorem 2.4.5 ([LW82]) *Given a task set, τ_n, with all tasks having the same release time, arbitrary deadlines, and allowing arbitrary preemption, deciding whether τ_n is schedulable on one processor requires pseudo-polynomial time.*

Theorem 2.4.6 ([Ull75]) *Given a set of tasks, τ_n, each requiring a single time unit for execution, with precedence constraints, producing a valid schedule on m processors is $\mathcal{NP}$-complete.*

It follows from these theorems that the real-time system model affects the complexity of the problem to be solved. In general, real-time task scheduling may be considered to be a difficult problem with most problem definitions belonging to the class of $\mathcal{NP}$-hard problems. Since no polynomial time algorithm exists that finds the optimum schedule, heuristics are used to find a close to optimum schedule in polynomial or pseudo-polynomial time.

Fortunately, the task scheduling problem seeks only a valid schedule as a solution. This has allowed heuristic search algorithms, and even enumerative and branch and bound algorithms on some task sets, to be successful and fast. The most widely used scheduling algorithms employ heuristics as their main method of generating a schedule.

Chapter 3

Round Robin Scheduling

3.1 Introduction

Non-preemptive *Round Robin* (RR) schedules are the simplest schedules which may be constructed. They specify the order in which tasks are checked in determining the next task to execute. Constructing the RR schedule requires no information on the task characteristics, and any type of task set (periodic, sporadic, or aperiodic) may be scheduled using the RR approach. The schedule is executed repeatedly at run-time.

RR schedules are not typically used in real-time applications since it is difficult to verify their timing predictability. POLIS is capable of generating RR schedules, so they are discussed here for completeness.

We discuss the round robin approach to scheduling. The advantages and disadvantages of this approach are then examined. Finally, the round robin implementation within the POLIS co-design environment is discussed.

3.2 Round Robin Scheduling

Round Robin schedules are not true schedules; they do not specify the order of execution of the tasks. Instead, an RR schedule specifies the order in which tasks are checked to determine the next task to be executed. Since the exact order of execution of the tasks is not fixed in an RR schedule, the timing behavior of the task set is not predictable.

In its simplest form, an RR schedule is a list of the tasks in the task set. No information on the tasks is required to generate the list (a random list is acceptable). No

schedulability analysis tests exist for an RR schedule; therefore, simulation must be used to verify that all tasks meet their respective deadlines.

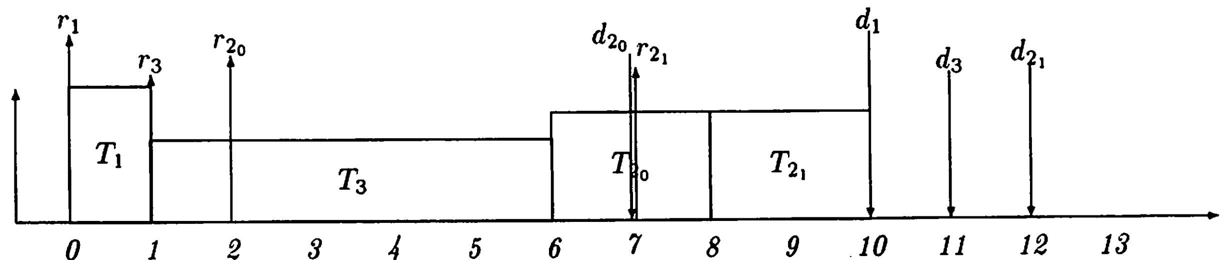
At run time, the RR schedule is repeatedly traversed until a ready task is found. The ready task is allocated the processor and run to completion. Once a task has completed execution, the RR schedule is traversed beginning with the next task in the schedule.

The following example illustrates the RR scheduling approach.

Example 3.2.1 *Consider the following periodic tasks (the tasks are represented by the 4-tuple* $T = (r, c, p, d)$ *):* $T_1 = (0, 1, 10, 10)$, $T_2 = (2, 2, 5, 5)$, *and* $T_3 = (1, 5, 10, 10)$.

One possible RR *schedule is:* T_1, T_2, T_3.

This schedule (and every other schedule produced by the round robin approach) produces the following execution ordering (repeated every 10 time units).



A better understanding of the actions of the RR *schedule at run-time may be obtained by examining the actions occurring at a specific time. Consider time unit* 8. *The following actions occur at this time (T_1 and T_3 are not ready to execute and T_2 is ready to execute).*

- *T_2 completes execution.*
- *The scheduler checks T_3 and determines that it is not ready to execute.*
- *The scheduler checks T_1 and determines that it is not ready to execute.*
- *The scheduler checks T_2 and allocates the processor to it.*

Notice that T_3 is checked after T_2 completes execution and that T_1 (the first task in the schedule) is checked after T_3 (the last task in the schedule).

This is not a valid schedule since task T_2 misses its deadline at time 7, *but the task set is schedulable.*

■

3.3 Advantages of the Round Robin Approach

The main advantage of the round robin scheduling approach is its simplicity. An RR schedule is generated without the use of task characteristics, and the run time implementation requires only to be able to keep its place in the RR schedule and detect and execute ready tasks.

The RR scheduling approach is able to handle all types of tasks, periodic, sporadic, and aperiodic. The ability of an RR schedule to generate a valid schedule is verifiable through simulation.

3.4 Disadvantages and Other Issues of the Round Robin Approach

The round robin scheduling approach, by its nature, is too simple. Complete simulation is the only method of verifying that deadlines are met. This may not be practical if there are a large number of tasks in the system, and worst-case simulation yields pessimistic results.

The run-time overhead of the RR schedule can be very large. Since the round robin approach attempts to find a ready task by checking the tasks in a fixed order, it is possible that all tasks are checked before a ready task is found.

3.5 Implementation of the Round Robin Approach Within POLIS

The round robin scheduling approach is implemented within the POLIS co-design environment. The RR schedule is a random listing of the tasks in the system. It is generated directly from the list of tasks given to the schedule generation routines within POLIS. (The given list of tasks is the RR schedule.)

No schedulability analysis tests are performed on the generated schedule. The reason for this is that the execution times for the tasks are assumed to be unknown when an RR schedule is produced. Without knowledge of the execution times of the tasks it is impossible to perform a schedulability test.

The generated RR scheduling routine is shown in Figure 3.1.

```
// main round robin scheduling routine
scheduler()
{
    while( 1 ) {
       poll_inputs_and_update_input_buffers();
       for( task =0; task < NUMBER_TASKS; task++ ) {
          if( is_ready( task ) ) {
             execute( task );
          }
       }
    }
}
```

Figure 3.1: Generated Round Robin scheduling routine.

Chapter 4

Pre-Run-Time Scheduling

4.1 Introduction

Pre-run-time schedules are computed off-line. The schedule specifies the action to be taken at each instant. Constructing the schedule requires the major characteristics of the tasks be known in advance[1]. Pre-run-time scheduling may be used to schedule periodic task sets by computing a schedule through the time period equal to the least common multiple of the periods of the tasks. The schedule is then executed repeatedly at run-time. Task sets containing sporadic tasks may be scheduled by converting the sporadic tasks to equivalent periodic tasks and then scheduling the resulting periodic task set [Mok83].

Multiple schedules may be generated for a system, corresponding to different operating 'modes.' This provides additional flexibility for the system, allowing it to adapt to its environment.

Traditionally, pre-run-time schedules are generated by hand, often using ad hoc methods. The cyclic executive approach provides a structured framework within which the designer may use heuristics to construct a valid schedule. This framework limits the solution space the designer must explore to find a valid schedule, making it easier to construct a schedule.

Algorithmic approaches have been proposed to automate pre-run-time scheduling. We discuss the cyclic executive approach and a branch-and-bound pre-run-time algorithm.

[1]Given a task set τ_n, for every task $T_i = (r_i,\ c_i,\ p_i,\ d_i,\ D_i,\ P_i) \in \tau_n$ all components of the task and the relations between this task and the other tasks in the task set must be known in order to schedule this task set off-line.

The advantages and disadvantages of these approaches are then examined, highlighting the advantages of the cyclic executive and the algorithmic approaches. Finally, the pre-run-time implementation within the POLIS co-design environment is discussed.

4.2 Cyclic Executive

The cyclic executive approach is the most widely used and best understood implementation technique for scheduling periodic task sets. A *cyclic executive* is a control structure or program that interleaves the execution of several periodic tasks on a single processor [BS86, BS89]. The interleaving is deterministic, providing predictable timing behavior for the task set.

A *cyclic schedule* specifies an interleaving of actions allowing each task to meet its deadline. The possible actions are complete execution of a task, partial execution of a task, and processor idle. The cyclic schedule is composed of one or more *major schedules* describing the sequence of actions to be performed during a fixed time period, called the *major cycle.* The major cycle is equal to the Least Common Multiple of the periods (LCM) of the tasks. Different major schedules correspond to different modes of operation of the system.

Major schedules are further divided into *minor schedules,* or *frames.* Frame boundaries correspond to points at which timing behavior may be enforced, typically via an interrupt from a timer circuit. If the actions of a frame are not completed before the end of the frame, an error, a *frame overrun,* has occurred. Typically, all frames are of equal length requiring a simple periodic timer to enforce timing behavior.

The length of a frame, called the *minor cycle* when all frames are of equal length, is restricted by the task set being scheduled. The frame length may be no longer than the shortest period of all the tasks. Tasks that cannot complete execution within one frame must be split into two or more sub-tasks, each of which can complete within one frame. This is equivalent to preempting the task at a predetermined point. There are additional restrictions on m, the minor cycle:

1. $m \leq d_i$ for $i = 1, \ldots, n$, allowing for frame overruns to be detected shortly after they occur.

2. m must be greater than or equal to the computation time of the longest action.

3. m must divide the major cycle, M, for timer simplicity.

4. $m + (m - \gcd(m, p_i)) \leq d_i \; \forall \; i \in \{1, \ldots, n\}$ where gcd stands for the greatest common divisor function; requiring that a complete frame exist between every release time and its corresponding deadline.

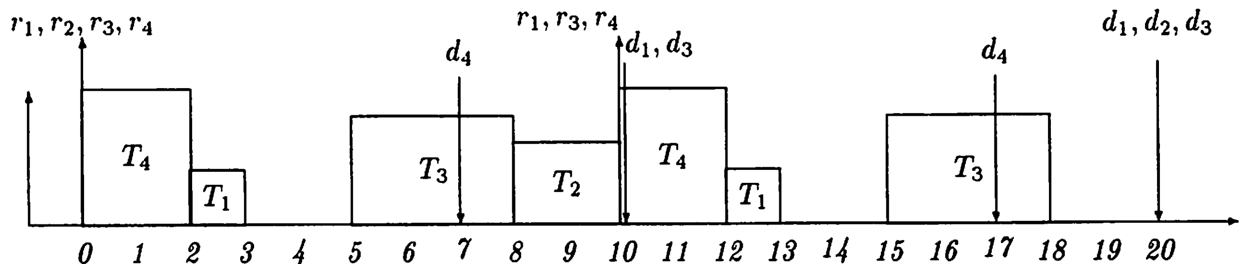
Example 4.2.1 demonstrates how a schedule is generated using the cyclic executive approach.

Example 4.2.1 *Consider the following periodic tasks with the same release time (the tasks are represented by the abbreviated triple $T = (c,\ p,\ d)$):* $T_1 = (1, 10, 10)$, $T_2 = (2, 20, 20)$, $T_3 = (3, 10, 10)$, $T_4 = (2, 10, 7)$.

The major cycle is the LCM *of the tasks. The periods are* 10, 20, *and* 10, *with an* LCM *of* 20*; therefore, the major cycle is* $M = 20$.

The minor cycle is calculated as follows. Since the shortest period is 10*, the possible minor cycle times are limited to be one of* $1, \ldots, 10$. *Requirement (1) for the minor cycle time limits the possible times to* $1, \ldots, 7$. *For simplicity, assume all tasks are a single action. Thus, the longest action is 3 time units. Requirement (2) reduces the possible times to* $3, \ldots, 7$. *Requirement (3) reduces the possible times to* 4 *or* 5. *Requirement (4) does not reduce the possibilities further; thus,* $m = 4$ *or* $m = 5$.

One possible major schedule is shown below for $m = 5$. *The order of actions within each frame are determined by an earliest deadline first heuristic with ties broken arbitrarily.*



■

Usually, heuristics are used to determine the order of actions within each frame, but these do not guarantee a valid schedule will be found if one exists. A valid schedule is usually difficult to find in a complex system. This increases the chance that a human designer will be unable to find a valid schedule or will erroneously conclude that a schedule

is valid. The framework of the cyclic executive does help to reduce the chance of errors in the generated schedule, but does not preclude the chance for error.

4.3 Pre-Run-Time Scheduling Algorithms

The cyclic executive approach provides a structured framework that facilitates the generation of a schedule. However, this framework can hinder the generation of a valid schedule. In particular, the use of frames can prevent a valid schedule from being found. Frames are an implementation structure used to ensure that deadlines are met at run-time. If the given release time, computation time, deadline, and period are accurate for all tasks, frames do not need to be used.

Pre-run-time scheduling algorithms have been presented that generate a valid schedule without the use of frames [XP90, SG91, Xu93]. Many of these algorithms are based upon *branch and bound* techniques. These techniques use a tree-structured search format and utilize bounding methods to eliminate entire branches of the search tree. These techniques are amenable to automation and will find a valid schedule if one exists.

Figure 4.1 shows a branch and bound algorithm that generates a schedule in which the lateness[2] of all segments is minimized and all precedence and exclusion relations are satisfied [XP90]. The algorithm was designed to schedule a periodic task set on a single processor. Each task T_i is divided into segments $t[0]_i$, $t[1]_i$, ..., $t[n]_i$, where $t[0]_i$ is the first segment and $t[n]_i$ is the last segment in task T_i. The characteristics of each segment, i.e., release time, computation time, deadline, and period, are determined and the set of precedence and exclusion relations on the segments are initialized to those on the tasks.

The segments are scheduled through their LCM. The schedule meets all release time, precedence, and exclusion constraints on the segments. If the minimum lateness of all possible schedules is greater than zero, then no valid schedule exists for the task set. Otherwise, a valid schedule is found by the algorithm.

The algorithm is typical of most branch and bound pre-run-time scheduling algorithms in that it generates a search tree with each node containing a complete schedule. The schedule is generated by an earliest-deadline-first heuristic, and a lower bound on the lateness of the schedule is computed.

[2]The lateness of a segment is equal to the difference between its completion time and its deadline. The lateness of a schedule is equal to the maximum lateness of all segments in the schedule.

Branch and Bound Pre-Run-Time Algorithm [XP90]

input: the set of segments to be scheduled, initial PRECEDE and EXCLUDE constraints.

output: the set of segments in scheduled order

1. Compute a schedule for the input set of segments. This schedule is the root node of the search tree, and is the first 'parent' node.
2. Determine the lower bound on the lateness for this schedule.
3. Find the latest segment $t[l]_i$ and its lateness. If its lateness equals its lower bound, then stop (the schedule is optimal). Return this node.
4. Repeat the following steps until either a valid schedule is found or the optimum schedule is found.
   (a) For the parent node, find the sets of segments S_1, those segments that may be preceded by $t[l]_i$, and S_2, those segments that may be preempted by $t[l]_i$.
   (b) For each segment in S_1, s_j, create a child node with all constraints of the parent node plus $t[l]_i$ PRECEDE s_j.
   (c) For each segment in S_2, s_k, create a child node with all constraints of the parent node plus $t[l]_i$ PREEMPT s_k.
   (d) Recompute the schedule and the least lower bound on lateness, and find the latest segment and its lateness for each child node.
   (e) If the lateness of one or more child nodes is less than or equal to zero, return the node with the smallest lateness.
   (f) For each child node, if the lateness of the node is less than or equal to the least lower bound of all unexpanded nodes (nodes without child nodes), then stop (the schedule is optimal). Return the best solution found.
   (g) Select among all unexpanded nodes the node with the least lower bound, in case of ties, select the node with least lateness. Call this node the parent node.

Figure 4.1: Branch and bound pre-run-time algorithm presented by Xu and Parnas.

The solution space is searched by incrementally adding constraints. At each node in the search tree, two sets of segments are identified, S_1 and S_2, such that the schedule found at that node may be improved on if either the latest segment is scheduled before a segment in S_1; or, the latest segment preempts a segment in S_2.

For each segment in the sets S_1 and S_2, a child node is generated that contains either an additional PRECEDE or an additional PREEMPT constraint. Thus, the latest segment in the parent node is either scheduled before a segment in S_1 or preempts a segment in S_2 in each of the child nodes. The node with the least lower bound on lateness among all unexpanded nodes (nodes without child nodes) is always expanded.

New nodes in the search tree are generated until either a valid schedule is found or the lower bound of all unexpanded nodes is greater than the least lateness of all schedules generated so far. In the latter case, the schedule with the least lateness is optimal.

The following examples illustrate this algorithm.

Example 4.3.1 *Consider the following periodic tasks represented by the abbreviated 4-tuple* $T = (r,\ c,\ p,\ d)$*:* $T_1 = (0, 1, 10, 10)$, $T_2 = (0, 2, 20, 20)$, $T_3 = (0, 3, 10, 10)$, $T_4 = (0, 2, 10, 7)$. *There are no precedence or exclusion constraints among the tasks. Each task contains only one segment.*

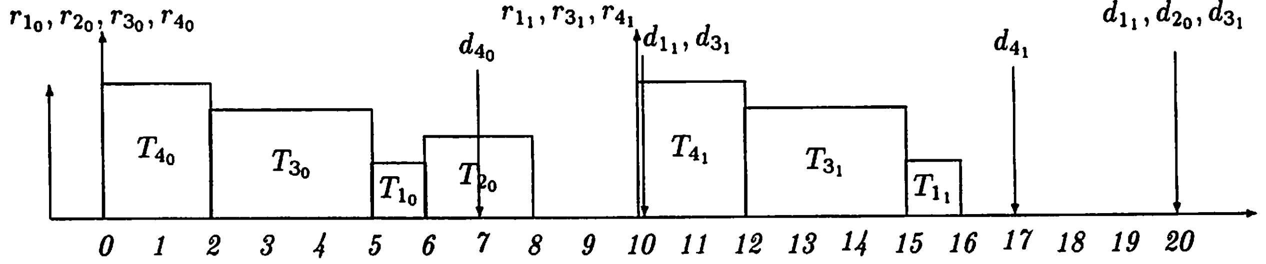
The LCM *is* $M = 20$.

The equivalent tasks that will be scheduled over M *are calculated as follows. Since the only parameter that changes between instances of the same task is the release time, only the release times need to be calculated for the successive invocations of a task. All invocations with a release time greater than or equal to* M *are not considered for obvious reasons.*

Starting with task T_1*, the first instance of* T_1 *has the given release time,* $T_{1_0} = (0, 1, 10, 10)$. *The release time of the second instance of* T_1 *is determined by adding the period of* T_1 *to the release time of* T_{1_0}, $T_{1_1} = (10, 1, 10, 10)$. *The third invocation of* T_1 *has a release time equal to* M*; thus,* T_{1_2}*, and all subsequent invocations, are not considered.*

Repeating this for the remaining three tasks, we obtain the following tasks: $T_{2_0} = (0, 2, 20, 20)$, $T_{3_0} = (0, 3, 10, 10)$, $T_{3_1} = (10, 3, 10, 10)$, $T_{4_0} = (0, 2, 10, 7)$, $T_{4_1} = (10, 2, 10, 7)$.

An initial schedule is generated for the task set and shown below.



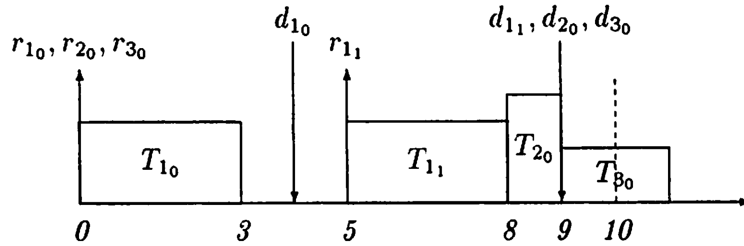
This is a valid schedule, so the algorithm returns the above schedule. ∎

Example 4.3.2 *Consider the following periodic tasks represented by the abbreviated 4-tuple* $T = (r,\ c,\ p,\ d)$*:* $T_1 = (0,3,5,4)$, $T_2 = (0,1,9,10)$, $T_3 = (0,2,10,10)$. *There are no precedence or exclusion constraints among the tasks. Each task contains only one segment.*

The LCM *is* $M = 10$.

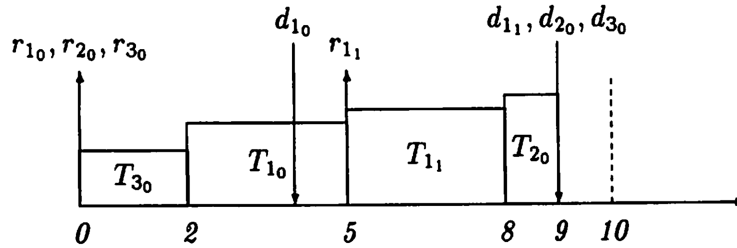
The equivalent tasks that will be scheduled over M *are calculated to be:* $T_{1_0} = (0,3,5,4)$, $T_{1_1} = (5,3,5,4)$, $T_{2_0} = (0,1,10,9)$, $T_{3_0} = (0,2,10,9)$.

An initial schedule is generated for the equivalent task set and shown below.

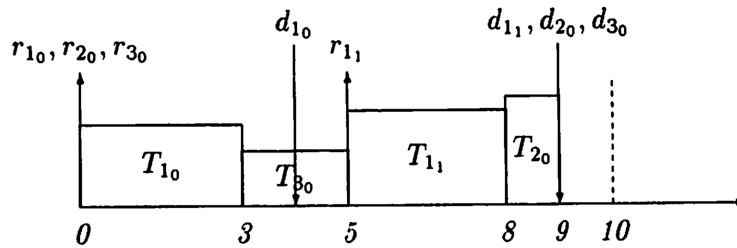


This is not a valid schedule, for task T_{3_0} *misses its deadline. It is late by two time units. Therefore, the child nodes in the search tree must be generated. Recall that* S_{1_k} *is the set of segments that must be preceded by the latest segment at node* n_k*. In this instance, the latest segment for node* n_1 *is* T_{3_0}*. Since no precedence constraints exist between* T_{3_0} *and any other segment, all other segments belong to* $S_{1_1} = \{T_{1_0},\ T_{1_1},\ T_{2_0}\}$*. Similarly for* S_{2_1}*, the set of segments that must be preempted by* T_{3_0}*, since no exclusion constraints exist between* T_{3_0} *and any other segment,* $S_{2_1} = \{T_{1_0},\ T_{1_1},\ T_{2_0}\}$.

Arbitrarily starting with set S_{1_1}*, a* PRECEDE *constraint is added between* T_{3_0} *and* T_{1_0} *such that* $T_{3_0} \prec T_{1_0}$*. The following schedule results when this constraint is added.*



This is not a valid schedule since T_{1_0} is late by one time unit. The next child node is generated for node n_1. Taking the next segment in S_{1_1}, T_{1_1}, a precedence constraint is added between T_{3_0} and T_{1_1} such that $T_{3_0} \prec T_{1_1}$. Recall that since a new child node is being generated, the precedence constraint between T_{3_0} and T_{1_0} does not exist for this new child node. The resulting schedule is shown below.



This is a valid schedule for the task set; therefore, this schedule is returned by the algorithm. ∎

4.4 Sporadic Task Scheduling in Pre-Run-Time Schedules

4.4.1 Servicing Sporadic Tasks

Thus far, only periodic task sets have been addressed; however, sporadic tasks are often present in otherwise periodic task sets. The sporadic tasks must be scheduled correctly with the pre-run-time scheduling approach used. One method of dealing with sporadic tasks is to consider the sporadic tasks to be periodic tasks with period equal to the minimum time between invocations. The resulting periodic task set is then scheduled. At run time, the scheduler executes a sporadic task only if it is ready. The worst case scenario arises when the sporadic task is invoked just after it is checked by the scheduler; possibly resulting in a missed deadline. Consequently, this approach may result in unpredictable timing behavior causing sporadic tasks to miss their deadlines. This is only acceptable if all sporadic tasks have soft deadlines.

An alternative method of handling sporadic tasks is to establish a server to provide a private resource for the exclusive use of the sporadic tasks. This approach creates a periodic server task that is scheduled like other periodic tasks. When the server task is allocated the processor, it executes any waiting sporadic tasks. The server task does not allow the total execution time of the sporadic tasks to exceed its maximum execution time. The server task exits when either no sporadic tasks are ready to execute or the maximum execution time of the server task is exceeded. In this way, the server task provides a regular service to sporadic tasks, minimizing the lateness of the sporadic tasks.

A periodic server task does not affect the schedulability of the periodic tasks.

Theorem 4.4.1 *A periodic task set that is schedulable by the pre-run-time scheduling approach with a task, T_i, is also schedulable by the pre-run-time scheduling approach if T_i is replaced by a periodic server task with the same release time, period, deadline, and maximum computation time.*

Proof By contradiction. Assume that in the periodic task set schedulable by the pre-run-time approach, task T_i is replaced by a periodic server task, T_{ps}, with the same release time, period, deadline, and maximum computation time, but the resulting task set is not schedulable by the pre-run-time approach.

Consider the pre-run-time schedule generated using the original task set. Every instance of task T_i in this schedule may be replaced with an instance of the periodic server task T_{ps} with the same task characteristics. This is possible since there are no restrictions placed upon T_{ps}; thus, T_{ps} may be preempted at arbitrary points. The resulting schedule, with T_i replaced with T_{ps}, does not cause any periodic task to miss its deadline. Therefore, the schedule is valid. Contradiction. ■

Furthermore, multiple server tasks may be used to service different, or possibly the same, sporadic tasks.

The main concern when scheduling hard sporadic tasks with a pre-run-time scheduler is that the sporadic task will be invoked just after it is tested, either by the scheduler or the periodic server task. The pre-run-time schedule must handle this case correctly, i.e., the hard sporadic task must not miss its deadline even when it is invoked just after it is tested. In order to guarantee that a hard sporadic task does not miss its deadline in the worst case,

the hard sporadic task, T_s, must be checked to see if it is ready to execute every $d_s - c_s$ time units. A periodic server task may be used for each hard sporadic task. The characteristics of the server task, T_p, for each hard sporadic task, T_s, are determined by the characteristics of T_s.

Mok [Mok83] proposed a method for translating a sporadic task $T_s = (r_s,\ c_s,\ p_s,\ d_s,\ D_s,\ P_s)$ into an equivalent periodic task $T_p = (r_p,\ c_p,\ p_p,\ d_p,\ D_p,\ P_p)$. For this translation, the following conditions are satisfied: $r_p = r_s$, $c_p = c_s$, $p_p = \min(d_s - c_s + 1, p_s)$, $d_p = c_s$, $D_p = D_s$. This transformation guarantees that a sporadic task will not miss its deadline in the worst case.

4.4.2 Schedulability Analysis with Sporadic Tasks

When sporadic tasks are scheduled in a pre-run-time schedule, the timing analysis becomes more complicated. For example, consider the sporadic tasks to be periodic tasks with period equal to the minimum time between successive invocations. Finding a valid schedule such that no periodic task ever misses a deadline does not guarantee that a sporadic task will never miss a deadline.

If Mok's method is used to determine the characteristics of every periodic server task serving a hard sporadic task, any valid pre-run-time schedule guarantees that no hard sporadic task will ever miss a deadline. However, if the pre-run-time schedule does not guarantee all deadlines, then a more exact analysis must be performed to determine if the schedule is valid. The inconsistency arises from the fact that the maximum utilization of the periodic server task created using Mok's method (the utilization that the pre-run-time schedule assumes to be used) is greater than the utilization of the corresponding hard sporadic task.

The actual processor usage of hard sporadic tasks must be taken into account when determining the validity of a schedule. More precisely, in a pre-run-time schedule, the computation time of a hard sporadic task should be counted exactly once during every time span equal to its minimum time between successive invocations, corresponding to the worst case processor utilization during each of these time spans. For example, consider the hard sporadic task T_s (with $r_s = 0$, $c_s = 2$, and $p_s = d_s = 10$) serviced by Mok's periodic server T_M. During the time interval $0, \ldots, 10$ the computation time attributed to T_s should be equal to c_s even though the computation time of T_M may be more than c_s in this interval.

During the time interval $1, \ldots, 11$ the computation time attributed to T_s should be equal to c_s, and so on. In addition, every other hard task in the task set should meet its deadline taking into account the actual processor usage characteristics of T_s.

The analysis to determine if all tasks meet their respective deadlines is complex and computationally expensive, but it must be performed to ensure that all hard tasks always meet their deadlines.

Using a single periodic server task for each soft sporadic task allows additional information to be obtained on the schedulability of the task set. All occasions when a soft sporadic task may miss a deadline or cause a hard task to miss a deadline will be identified. In addition, the probability that a soft sporadic task misses a deadline can be calculated, allowing for a more exact analysis of the schedulability of the task set.

4.5 Advantages of the Pre-Run-Time Approach

The main advantage of the pre-run-time scheduling approach is the predictability of the system's timing behavior, allowing all deadlines to be guaranteed without any explicit schedulability analysis test.

> For satisfying timing constraints in hard real-time systems, predictability of the system's behavior is the most important concern; pre-run-time scheduling is often the only practical means of providing predictability in a complex system [XP93].

When we are presented with accurate worst case timing behavior of the tasks, the timing predictability of the system is guaranteed even though the exact timing behavior of the tasks and of the system may not be predictable. Deadlines can be enforced within the precision of one frame in the cyclic executive approach.

The schedule is easily constructed such that precedence, distance, exclusion, and resource constraints are met, eliminating the possibility of deadlock and unpredictable delays. Thus, no special task synchronization protocols are required.

Pre-run-time schedules may be generated automatically, reducing the possibility of errors. In addition, branch and bound algorithms will find a valid schedule if one exists. This is especially important in complex systems where a human may not be able to find a valid schedule.

Multiple schedules may be generated for different modes of operation of the system, allowing the system to adapt to its environment.

Implementations of pre-run-time schedules are simple and efficient. The schedule[3] can be represented by a table of actions that is interpreted by the scheduler at run-time with the minor cycle timer constructed as a periodic timer. The scheduler requires minimal run-time resources to follow the schedule and perform context switching.

Sporadic tasks are handled by the pre-run-time scheduling approach using periodic server tasks.

4.6 Disadvantages and Other Issues of the Pre-Run-Time Approach

Since the general scheduling problem belongs to the class of $\mathcal{NP}$-hard problems, heuristics are needed to guide the search for a valid schedule. The most relevant heuristic used in constructing a schedule is derived from a run-time scheduling algorithm. The heuristic is derived from the earliest deadline first scheduling algorithm [LL73], an optimal algorithm in the sense that it will always produce a valid schedule if one exists. This is a dynamic priority preemptive scheduling algorithm where the task allocated the processor has the earliest deadline of all those tasks that are ready to be allocated the processor. The relevant heuristic for pre-run-time schedules is that one should try to schedule actions in earliest deadline order.

Generally, this heuristic is not used in its original form in pre-run-time scheduling due to its preemptive nature, requiring tasks to allow preemption at arbitrary points rather than at specific points. In addition, allowing arbitrary preemption would cause the software and hardware handling the context switches to be more complicated. The net result is that the context switch would require more time and space than may be acceptable, and task synchronization and resource management would be more complicated.

The amount of information required to generate a pre-run-time schedule can be prohibitively large. If the system contains unpredictable tasks, it will be difficult to determine a valid schedule off-line, and any change to the task set will necessitate the generation of a new schedule. In addition, the determination of some of the required information is

[3]The major schedule for cyclic executives.

very difficult.

The determination of specific points within a task at which it is "safe" to preempt the task is very difficult. The determination is usually done manually based on natural "regions" of code, critical sections, or functional units [BS86]. This is especially critical to the cyclic executive approach where the tasks must be "broken" (if they are to be broken) before the major schedule may be constructed.

The timing predictability of sections of code is difficult. Many of the current timing prediction techniques yield average case timing. Those timing prediction techniques that do determine worst-case timing are often overly pessimistic due to false paths. Because of the problems involved in determining the worst-case execution times, the timing predictions are often inaccurate. The overly pessimistic times cause wasted CPU cycles while the overly optimistic times may cause frame overruns and are the reason that errors occur.

The relationship between the periods in the task set can cause the major cycle time to be extremely large. When the periods in the task set are harmonics of one another, the major cycle is equal to the largest period in the task set. However, when the periods are not harmonics, the major cycle may be much larger than the largest period in the task set. This is shown in the following examples.

Example 4.6.1 *Given the following set of harmonic periods: 2, 4, 8, 16; the major cycle is 16.* ■

Example 4.6.2 *Given the following set of non-harmonic periods: 2, 5, 11, 13; the major cycle is 1430.* ■

Mode changes can cause difficulties. When the system changes its mode of operation, the currently running task must be terminated in some logical way (which may be just an abortion). All resources need to be freed, and the state of the system needs to be set to a known state. The new major schedule must then be started. All of this may necessitate that any mode change require special processing to reinitialize or make consistent certain variables. Thus, any system that has multiple modes of operation will require a more complicated run-time scheduler.

Multiple modes of operation also cause additional memory space to be used to store the additional schedules. This can be critical for memory limited systems that have relatively large pre-run-time schedules. The multiple pre-run-time schedules required for each mode of operation may not fit in the available memory.

Sporadic tasks are not easily handled by the pre-run-time scheduling approach. Although the periodic server tasks provide a method by which sporadic tasks may be scheduled and serviced, the resulting schedule is often difficult to analyze and to implement if a periodic server services more than one task. In addition, if a periodic server services multiple tasks, the state of any running task must be saved and all currently used resources released when the periodic server exhausts its execution time. This results in a complex periodic server implementation.

4.7 Implementation of the Pre-Run-Time Approach Within POLIS

The cyclic executive scheduling approach was developed assuming that a human would determine the worst-case execution time of the tasks and then manually arrange the major schedule. Algorithmic approaches are easily automated. Since POLIS automatically determines the worst-case execution time for the various tasks in the system and automatically generates a schedule, only an algorithmic pre-run-time approach is implemented within POLIS.

The algorithm presented by Xu and Parnas [XP90] discussed in Section 4.3 is the basis for the General Pre-Run-Time (GPRT) algorithm implemented within POLIS. The main difference between the algorithm in [XP90] and the GPRT algorithm is that a task consists of only one non-preemptible segment, eliminating the need to determine viable points for preemption. This is essential to keeping the size of the real-time operating system (of which the scheduler is a key part) generated by POLIS small enough to fit within the tight memory bounds of its target applications. In addition, context switching overhead is eliminated by this modification.

The following assumptions are made about the tasks scheduled by the GPRT algorithm.

A1: *Preemption is not allowed.* The target applications for POLIS have a limited amount of memory and hardware support available. Preemption in the pre-run-time scheduling approach would tax or exceed the available resources.

A2: *All tasks are periodic.* Those tasks that are sporadic may be transformed into periodic tasks by Mok's method. When the schedule is executed the sporadic tasks

General Pre-Run-Time Algorithm

input: the network of CFSMs, the set of tasks (CFSMs) to be scheduled τ_n, the target output file

output: the table of actions, the table of sporadic tasks, the main schedule loop

1. Generate a periodic server task for each soft task.
2. Convert all hard sporadic tasks into periodic tasks using Mok's method.
3. Determine the Least Common Multiple (LCM) of the periods.
4. Determine all task instances that will occur during the LCM and their appropriate initial precedence constraints.
5. Consider all instances to be the task set τ_n' that is to be scheduled.
6. Generate the schedule for the task set τ_n' using the PRT algorithm.
7. Generate the table of actions to be performed; writing the result to the output file.
8. Generate the table of sporadic tasks; writing the result to the output file.
9. Generate the main schedule loop that handles sporadic tasks properly; writing the result to the output file.

Figure 4.2: General Pre-Run-Time (GPRT) algorithm implemented within POLIS.

are executed only if they are ready.

A3: *All tasks are not independent.* Initial precedence constraints may exist between the tasks, and implicit exclusion constraints exist between all tasks.

The following sections describe the GPRT algorithm.

4.7.1 General-Pre-Run-Time Algorithm

Figure 4.2 shows the GPRT algorithm implemented within POLIS. The input to the GPRT algorithm is the network of connected CFSMs comprising the entire design, a

list of tasks (CFSMs) to be scheduled, and the target output file. The output of the GPRT algorithm is a table listing the sequence of actions to be performed, a table listing the sporadic tasks in the task set, and the main scheduler loop that determines which action to perform next; all written to the target output file.

The GPRT algorithm begins by creating a periodic server task for each soft task. Since preemption is not allowed (by **A1**), periodic server tasks can handle at most one task. The characteristics and constraints on the server task are equal to those of the task it serves. Thus, all soft tasks are assumed to behave like hard periodic tasks for simplicity.

In accordance with **A2** all hard sporadic tasks are transformed into periodic tasks using Mok's method. Initial precedence constraints are determined for the tasks in the task set by traversing the network of CFSMs in topological order.

All task instances that will occur through the LCM are determined with their appropriate precedence constraints. The set of all instances of the tasks in the original task set, τ_n, comprise a new task set τ_n'. Task set τ_n' is the input to the Pre-Run-Time (PRT) algorithm.

The PRT algorithm returns either a valid schedule or the schedule with the least maximum lateness if a valid schedule does not exist for the task set. From the schedule returned by the PRT algorithm, a table of actions to be performed by the scheduler is generated and written to the output file. A table of the sporadic tasks is generated and written to the output file. Finally, the main scheduler loop is written to the output file.

Note, even if a valid schedule is not found, the optimum schedule returned by the PRT algorithm is written to the output file. This allows the designer to examine the optimum schedule to determine what can be modified to obtain a valid schedule.

4.7.2 Pre-Run-Time Algorithm

The PRT algorithm is shown in Figure 4.3. The PRT algorithm takes as input the set of tasks to be scheduled, the best schedule found thus far, and the maximum lateness of the best schedule. The output of the PRT algorithm is either a valid schedule or the optimum schedule.

The PRT algorithm searches for a valid schedule by generating a *valid initial solution, S*. A valid initial solution is a schedule such that all release times and precedence constraints are satisfied. A valid initial solution does not imply a valid schedule.

Pre-Run-Time Algorithm

input: the set of tasks to be scheduled, τ_n, the best schedule found so far, τ_n^B, the maximum lateness of the best schedule, M.

output: the best schedule found.

1. Generate a *valid initial solution*, $\tau_n^{'}$.
2. Find the task, T_l, with the maximum lateness in $\tau_n^{'}$.
3. If the maximum lateness, m, is less than or equal to zero, return $\tau_n^{'}$.
4. If $m < M$, $\tau_n^B = \tau_n^{'}$, $M = m$.
5. For each task $T_i \in \tau_n \neq T_l$ do the following.
   (a) Add a PRECEDE constraint between T_l and T_i of $\tau_n^{'}$, $T_l \prec T_i$, to obtain $\tau_n^{''}$.
   (b) If the PRECEDE constraint causes a loop in the precedence graph, then `continue`.
   (c) Compute the lower bound on lateness for all tasks in $\tau_n^{''}$.
   (d) If the maximum lower bound on lateness for $\tau_n^{''}$ is greater than M, then continue. This added constraint will not lead to a solution that is better than the current best solution found.
   (e) Call the PRT algorithm for $\tau_n^{''}$.
   (f) If the maximum lateness, m, of the returned schedule, $\tau_n^{''}$, is less than or equal to zero, then return $\tau_n^{''}$.
   (g) If $m < M$, $\tau_n^B = \tau_n^{''}$, $M = m$.
6. Return τ_n^B.

Figure 4.3: Pre-Run-Time (PRT) algorithm implemented within POLIS.

Valid Initial Solution Algorithm

input: the set of tasks to be scheduled, τ_n.

output: the set of tasks in valid initial solution order, $\mathcal{S}$.

```
time = 0
S = emptyset
T = τn
while( T not empty )
  if( ∃Ti ∈ T : ri ≤ time )
    Among the set { Ti | Ti ∈ T and
      ri ≤ time and
      ∄Tj ∈ T : Tj  PRECEDES  Ti }
    select the task Ti that has the minimum di.
    Append Ti to S.
    T = T − Ti.
  if( Ti ≠ NULL )
    time = time + ci.
  else
    time = time + 1.
return S.
```

Figure 4.4: Valid Initial Solution algorithm implemented within POLIS.

Figure 4.4 shows the algorithm used for determining the valid initial solution of a task set. The algorithm uses an earliest deadline first heuristic to determine execution ordering of the tasks. The set of precedence constraints that exists on the input task set are satisfied while the valid initial solution is constructed. A valid initial solution constructed by this algorithm guarantees that all tasks begin execution after they are released and that if T_j PRECEDES T_i, then T_j is scheduled before T_i.

If the maximum lateness of the valid initial solution is less than or equal to zero, then a valid schedule has been found. This schedule is immediately returned. If the maximum lateness of the valid initial solution is less than the maximum lateness of the best

schedule found so far, the best schedule found is set to the current valid initial solution.

In order to search for a valid schedule, the latest task, T_l, in the valid initial solution, $\mathcal{S}$, is constrained to precede the other tasks in the task set. Therefore, for each task T_i in the task set for which the constraint T_l PRECEDE T_i does not cause a cycle in the corresponding constraint graph, a child node is created with this constraint added to the task set. The child nodes are created by calling the PRT algorithm recursively.

In order to prevent the search for a valid schedule from becoming an exhaustive search of the solution space, a child node is not generated if it is not possible for that node to lead to a schedule that is better than the best schedule found thus far. This bounding is done by using the maximum lower bound on lateness for a given task set to determine when to stop searching along a given path. The maximum lower bound on lateness is computed for each child node prior to generating a valid initial solution. The lower bound on lateness for each task, T_i, is computed as follows.

$$L_i = r_i' + c_i - d_i$$

where $r_i' = r_i$ if there does not exist a task T_j such that the constraint T_j PRECEDE T_i exists; otherwise, $r_i' = \max(r_i, r_j' + c_j | T_j \text{ PRECEDE } T_i)$.

If the maximum lower bound on lateness for a task set is greater than the maximum lateness of the best schedule found so far, then the child node is not created, since this child node and all of its successors will not find a schedule that is better than the best schedule found so far.

If a valid schedule is not found, the best schedule found (the optimum schedule) is returned.

Chapter 5

Static Priority Scheduling

5.1 Introduction

Static priority scheduling algorithms are run-time scheduling algorithms that assign a priority to each task off-line. The priority of a task does not change with time, and each invocation of a task has the same priority, allowing static priority scheduling algorithms to have a low run-time overhead. The scheduling algorithm allocates the processor to a ready task with the highest priority. For some applications it is possible to implement the scheduling in hardware by use of a priority-interrupt mechanism, effectively reducing the scheduling overhead to zero.

Many algorithms have been proposed that determine the static priority of the tasks in the task set, and comprehensive reviews of these algorithms exist [LSST91]. The most influential of these algorithms is the *Rate Monotonic Scheduling algorithm* (RMS), an optimal fixed priority scheduling algorithm for periodic tasks, presented by Liu and Layland in 1973 [LL73]. Since then, many generalizations of this algorithm have been presented that address practical problems that arise in the construction of real-time systems. These include the problems of transient overload and stochastic execution times, the scheduling of task sets containing both periodic and sporadic tasks, and task synchronization.

The *Deadline Monotonic Scheduling algorithm* (DMS), an optimal fixed priority scheduling algorithm presented by Leung and Whitehead in 1982 [LW82], is similar in concept to the RMS algorithm; the theory behind the RMS algorithm is used as the basis for the DMS theory. The DMS algorithm weakens some fundamental constraints of the RMS algorithm that allows for easier handling of sporadic tasks in the task set. This is its

main advantage over the RMS approach.

The *Laxity Monotonic Scheduling algorithm* (LMS), an optimal fixed priority scheduling algorithm, follows the theory for RMS and DMS. The LMS algorithm assigns priority based upon the laxity of the task when it is invoked, identifying critical tasks.

The Liu and Layland theory of RMS and the Leung and Whitehead theory of DMS are presented, as well as, the proof of optimality of LMS. The problems of task synchronization, non-preemptive scheduling, and sporadic task scheduling are then discussed. Finally, the static priority algorithms that are implemented within the POLIS co-design environment and the generated scheduling routines are discussed.

5.2 The Rate Monotonic Scheduling Algorithm

The RMS algorithm was first presented by Liu and Layland in 1973 [LL73] to solve the problem of scheduling periodic task sets on a uniprocessor using fixed priorities. Liu and Layland made several assumptions about the hard-real-time environment:

A1: All tasks are periodic and are ready at the beginning of each period.

A2: The deadline of each task is equal to its period.

A3: Tasks are independent, i.e., no precedence or exclusion constraints exist between tasks.

A4: The execution time for each task is constant for that task and does not vary with time.

A5: Every task may be preempted at arbitrary points.

Liu and Layland derived important results under these assumptions.

Theorem 5.2.1 ([LL73]) *A* critical instant *for any task occurs whenever the task is requested simultaneously with requests for all higher priority tasks.*

The proof of Theorem 5.2.1 follows from the observation that the processing of a request of any task can only be delayed by requests of a higher-priority task. It can be shown that the maximum interference due to a higher priority task occurs when the higher priority task is invoked at the same instant that the task is invoked. This is easily seen since a higher priority task may be requested multiple times during the period of the

task, and each request will preempt the lower priority task unless the lower priority task has completed execution. Therefore, the maximum interference that a task may experience occurs when the task is invoked simultaneously with all higher priority tasks.

Theorem 5.2.1 suggests a simple, direct method for determining whether a given priority assignment will yield a valid schedule. Specifically, if all tasks are able to meet their respective deadlines when they are invoked at their critical instants, then the priority assignment produces a valid schedule.

A critical instant for every task occurs at time r if all tasks have the same release time r. Thus, assuming that all tasks have the same release time, the schedule can be constructed through $\max_{1 \leq i \leq n}(p_i)$ to verify the validity of the priority assignment.

Theorem 5.2.1 also suggests a priority assignment algorithm. The rule for priority assignment is to assign priorities to tasks according to their request rates, independent of their run-times, i.e., assign the highest priority to a task with the shortest period and the lowest priority to a task with the longest period, ties broken arbitrarily. The priority assignment may be performed in $\mathcal{O}(n \ln n)$ time[1]. Such a priority assignment is known as the *rate monotonic priority assignment* and is optimal in the sense that no other fixed priority assignment rule can schedule a task set that cannot be scheduled by the rate monotonic priority assignment. Liu and Layland were able to prove this optimality.

Theorem 5.2.2 ([LL73]) *If a valid priority assignment exists for some task set, the rate monotonic priority assignment is valid for that task set.*

Liu and Layland went on to determine the least upper bound to processor utilization, $U^*_{\tau_n}$, for periodic task set τ_n in fixed priority systems. Task sets with utilization above this bound are not guaranteed that any priority assignment will yield a valid schedule. For task set τ_n, the processor utilization is

$$U_{\tau_n} = \sum_{i=1}^{n} \frac{c_i}{p_i}.$$

Note that if $U_{\tau_n} > 1$, then τ_n is not schedulable on a single processor by any scheduling algorithm.

[1]The priority assignment may be performed by sorting the tasks by period with the resulting sorted task list corresponding to the priority ordering of the tasks.

It is possible for a valid schedule to exist when $U_{T_n} > U^*_{T_n}$ [2], but the RMS algorithm is not guaranteed to be able to determine a valid schedule. The least upper bound to processor utilization is given in the following theorem.

Theorem 5.2.3 ([LL73]) *For a set of n tasks with fixed priority order, the least upper bound to processor utilization is* $U^*_{T_n} = n(2^{1/n} - 1)$.

The sequence of least upper bounds is given by $U^*_{T_1} = 1$, $U^*_{T_2} = 0.828$, ... , $U^*_{T_\infty} = \ln\ 2 = 0.693$. Therefore, any periodic task set is guaranteed to be scheduled by the RMS algorithm if its processor utilization is no greater than $\ln\ 2 = 0.693$. Section 5.8 discusses static priority schedulability analysis.

5.3 The Deadline Monotonic Scheduling Algorithm

The DMS algorithm was first presented by Leung and Whitehead in 1982 [LW82]. They make the following assumptions about the hard real-time environment:

A1: All tasks are periodic and are ready at the beginning of each period.

A2: The deadline of each task is less than or equal to its period.

A3: Tasks are independent, i.e., no precedence or exclusion constraints exist between tasks.

A4: The execution time for each task is constant for that task and does not vary with time.

A5: Every task is at arbitrary points.

Notice that Leung and Whitehead make the same assumptions about the hard real-time environment as do Liu and Layland except that the deadlines of the tasks may be less than or equal to the period instead of strictly equal to the period. Theorem 5.2.1 (the critical instant for a task) is valid under these assumptions. However, the RMS algorithm is not optimal under these assumptions. As a result, a different priority assignment rule must be used to obtain an optimal priority assignment.

The *inverse-deadline priority assignment* assigns a higher priority to a task with a smaller deadline, i.e., the priority of a task is inversely proportional to its deadline. The

[2] For example, a task set containing tasks that are harmonics of one another may have a utilization of 1 and still be schedulable (see Theorem 5.8.2).

inverse-deadline priority assignment reduces to the rate-monotonic priority assignment when $d_i = p_i \ \forall \ T_i, \ 1 \leq i \leq n$. The inverse-deadline priority assignment may be performed in $\mathcal{O}(n \ln n)$ time[3]. Leung and Whitehead proved that such a priority assignment is optimal.

Theorem 5.3.1 ([LW82]) *The inverse-deadline priority assignment is an optimal priority assignment for one processor in the sense that no other priority assignment can schedule a task set that can not be scheduled by the inverse-deadline priority assignment.*

Leung and Whitehead did not provide any schedulability analysis tests for the DMS approach. However, schedulability analysis based upon critical instants may be used for the DMS approach. Just as in the RMS approach, if all tasks are able to meet their respective deadlines when they are invoked at their critical instants, the priority assignment produces a valid schedule.

Schedulability analysis tests based upon the processor utilization may be used to guarantee the schedulability of a task set. The processor utilization schedulability analysis test proposed by Liu and Layland for the RMS approach may be used for the DMS approach. Theorem 5.2.3 (the least upper bound to processor utilization) is valid for all fixed priority assignments.

5.4 The Laxity Monotonic Scheduling Algorithm

The LMS algorithm solves the problem of scheduling periodic task sets on a uniprocessor using fixed priorities. The following assumptions are made about the hard real-time environment:

A1: All tasks are periodic and are ready at the beginning of each period.

A2: The deadline of each task is less than or equal to its period.

A3: Tasks are independent, i.e., no precedence or exclusion constraints exist between tasks.

A4: The execution time for each task is constant for that task and does not vary with time.

A5: Every task is preemptable at arbitrary points.

[3]The priority assignment may be performed by sorting the tasks by deadline with the resulting sorted task list corresponding to the priority ordering of the tasks.

These are the same assumptions that Leung and Whitehead make for the DMS algorithm.

The *laxity monotonic priority assignment* assigns a higher priority to a task with a smaller laxity, ties broken in favor of the task with the smaller deadline, where the laxity, l_i, of task T_i is $l_i = d_i - c_i$. The laxity monotonic priority assignment may be performed in $\mathcal{O}(n \ln n)$ time.

The notion of a critical instant from Theorem 5.2.1 is applicable to the laxity monotonic priority assignment. Therefore, if every task is able to meets its deadline when released at its critical instant, then the task set is schedulable. By Theorem 5.2.1, the schedule produced by a particular priority assignment is valid if and only if the deadline of the first invocation of each task is met in the schedule when all tasks are invoked at the same time.

The optimality of the laxity monotonic priority assignment is now proven. Let $\rho_n = (T_1, \ldots, T_k, T_{k+1}, \ldots, T_n)$ denote a priority assignment for task τ_n such that the priority of T_i is higher than the priority of T_{i+1}, $\forall 1 \leq i \leq n-1$. Let $\rho_n^k = (T_1, \ldots, T_{k+1}, T_k, \ldots, T_n)$ denote the priority assignment obtained from ρ_n by interchanging the priority of tasks T_k and T_{k+1} for any $1 \leq k \leq n-1$.

Lemma 5.4.1 *Let $\rho_n = (T_1, \ldots, T_k, T_{k+1}, \ldots, T_n)$ be a given priority assignment such that the schedule produced by ρ_n is valid. If $l_k \geq l_{k+1}$ for some T_k, T_{k+1} $1 \leq k \leq n-1$, then the schedule produced by $\rho_n^k = (T_1, \ldots, T_{k+1}, T_k, \ldots, T_n)$ is also valid.*

Proof Let ρ_n be a given priority assignment and let S_n denote the schedule produced by ρ_n. Suppose $l_k \geq l_{k+1}$ for some $T_k, T_{k+1}, 1 \leq k \leq n-1$, and let S_n^k denote the schedule produced by the priority assignment ρ_n^k. We need to show that if S_n is a valid schedule, then S_n^k is also valid. To do this, all we need to show is that the deadline of the first request of each task is met in S_n^k. It is easy to see that the response time of the first request of each task $T_j, j \neq k, k+1$ is the same in both S_n and S_n^k, since the processing of these tasks is not affected by the priority reordering of T_k and T_{k+1}. Moreover, the response time of T_{k+1} can only be smaller in S_n^k than in S_n, since the processing of T_{k+1} in S_n^k is delayed by one less task than in S_n, namely T_k. Thus our proof is reduced to showing that the deadline of the first request of T_k is also met in S_n^k.

Since T_{k+1} meets its deadline in S_n, but has a lower priority than T_k,

$$\mathcal{C} + c_k + c_{k+1} \leq d_{k+1} \tag{5.1}$$

$$\Rightarrow C + c_k \leq l_{k+1} \tag{5.2}$$

where C is the amount of computation done in S_n by all tasks T_i, $1 \leq i \leq k-1$, in the interval $[0, d_{k+1}]$. By Theorem 5.2.1 we know that the amount of computation done for all tasks T_i, $1 \leq i \leq k-1$, in any interval of length d_{k+1} is at most C. The following condition is sufficient to guarantee that the deadline of the first request of T_k is met in S_n^k,

$$\left\lfloor \frac{l_k}{l_{k+1}} \right\rfloor C + c_k \leq \left\lfloor \frac{l_k}{l_{k+1}} \right\rfloor l_{k+1}. \tag{5.3}$$

The above follows from the observations that the amount of computation done in S_n^k for all tasks T_i, $1 \leq i \leq k-1$, in the interval $[0, \lfloor l_k/l_{k+1} \rfloor d_{k+1}]$ is at most $\lfloor l_k/l_{k+1} \rfloor C$ and that the number of requests made by T_{k+1} in the interval $[0, \lfloor l_k/l_{k+1} \rfloor d_{k+1}]$ is at most $\lfloor l_k/l_{k+1} \rfloor$. Since 5.2 implies 5.3, it follows that the deadline of the first request of T_k is met in S_n^k. ∎

Theorem 5.4.1 *If a valid priority assignment exists for some task set, the laxity monotonic priority assignment is valid for that task set.*

Proof By contradiction, assume that there is a task set, τ_n, such that the schedule produced by the priority assignment ρ_n is valid and yet the schedule produced by the laxity monotonic priority assignment is not valid. ρ_n can be transformed into the laxity monotonic priority assignment by a sequence of adjacent priority reorderings. By Lemma 5.4.1 the schedule produced by the laxity monotonic priority assignment is also valid. Contradiction. ∎

Schedulability analysis based upon critical instants or simulation may be used to ensure that the LMS approach yields a valid schedule. If simulation is used, the simulation must be through the LCM of the tasks.

5.5 Task Synchronization in Static Priority Systems

Thus far, tasks have been assumed to be completely preemptable i.e., no exclusion constraints exist between segments of the tasks. This is not always a realistic assumption since it is often the case that tasks contain sections of code that must be executed atomically

and/or the tasks utilize non-preemptable resources. When exclusion constraints exist in a static priority system, it is possible that *priority inversion* occurs. Priority inversion occurs when the execution of a high priority task is blocked by a lower priority task. For example, consider tasks T_1 and T_2 (where T_1 has a higher priority than T_2) that share data. To maintain the consistency of the shared data, accesses to it must be serialized. If T_1 gains access to the shared data first, the proper priority order is maintained; however, if T_2 gains access to the shared data first, T_1 must wait until T_2 completes its access of the data. By blocking T_1, T_2 has caused a priority inversion to occur.

Exclusion constraints are typically enforced via task synchronization methods such as semaphores, locks, and monitors. Care must be taken to prevent unbounded priority inversion and deadlock when these methods are employed. To ensure that deadlocks are prevented and schedulability analysis tests may be performed, a task synchronization protocol must address these problems when it is developed.

A task synchronization protocol that was developed to prevent deadlock and still allow for schedulability analysis tests was presented by Sha *et. al.* [SRL90]. The *Priority Ceiling Protocol* (PCP) uses the notion of the *priority ceiling* of a semaphore. The definition of priority ceiling and priority ceiling protocol are given below.

Definition 5.5.1 ([SRL90]) *The* priority ceiling *of a semaphore is the priority of the highest priority task that may lock this semaphore. The priority ceiling of a semaphore S_i, denoted $C(S_i)$, represents the highest priority that a critical section guarded by S_i can execute, either by normal or inherited priority.*

Definition 5.5.2 (Priority Ceiling Protocol [SRL90])

1. *Task T, that has the highest priority among the tasks ready to execute, is assigned the processor, and let S^* be the semaphore with the highest priority ceiling of all semaphores currently locked by tasks other than task T. Before task T enters its critical section, it must first obtain the lock on the semaphore S guarding the shared data structure. Task T will be blocked and the lock on S will be denied if the priority of task T is not higher than the priority ceiling of semaphore S^*. In this case, task T is said to be blocked on semaphore S^* and to be blocked by the task that holds the lock on S^*. Otherwise, task T will obtain the lock on semaphore S and enter its critical section. When a task T exits its critical section, the binary semaphore associated with*

the critical section will be unblocked and the highest priority task, if any, blocked by task T will be awakened.

2. *A task T uses its assigned priority, unless it is in its critical section and blocks higher priority tasks. If task T blocks higher priority tasks, T* inherits *P_H, the highest priority of the tasks blocked by T. When T exits a critical section, it resumes the priority it had at the point of entry into the critical section. That is, when T exits a critical section, it resumes its previous priority that may not be its initial priority. Priority inheritance is transitive. Finally, the operations of priority inheritance and of the resumption of previous priority must be atomic.*

3. *A task T, when it does not attempt to enter a critical section, can preempt another task T_L if its priority is higher than the priority at which task T_L is currently executing.*

Under the priority ceiling protocol a high priority task may be blocked by a lower priority task in one of three situations. First, the high priority task may be directly blocked by the lower priority task. In this case, the high priority task attempts to lock a locked semaphore. This type of blocking is necessary to ensure the consistency of shared data. Second, a medium priority task, T_m, may be blocked by a low priority task, T_l, that is executing at the priority of a higher priority task, T_h, that it is blocking. This type of blocking is necessary to avoid having a high priority task, T_h, being indirectly preempted by the execution of a medium priority task, T_m. Third, a task may be blocked by the priority ceiling of a semaphore. This type of blocking is necessary to avoid deadlock and chained blocking.

Sha *et. al.* went on to prove many properties about the priority ceiling protocol. The more important of these properties are given below.

Theorem 5.5.1 ([SRL90]) *The priority ceiling protocol prevents deadlocks.*

Theorem 5.5.2 ([SRL90]) *Under the priority ceiling protocol, a task T can experience priority inversion for at most the duration of one critical section.*

In addition, Sha *et. al.* were able to develop a schedulability analysis test when the priority ceiling protocol is used in conjunction with the RMS algorithm. For this test, B_i, $1 \leq i \leq n$, is defined to be the longest duration of blocking that can be experienced

by task T_i. Note, $B_n = 0$ since T_n has the lowest priority and, hence, cannot experience a priority inversion. Theorem 5.5.3 gives a sufficient condition for the schedulability of a task set in which PCP is used in task synchronization on a uniprocessor.

Theorem 5.5.3 ([SRL90]) *A set of n periodic tasks with $d_i = p_i$ using the priority ceiling protocol can be scheduled by the* RMS *algorithm for all task phasings if*

$$\max_{1 \leq i \leq n} \min_{0 \leq t \leq p_i} \left(\sum_{j=1}^{i} \frac{c_i}{t} \left\lceil \frac{t}{p_j} \right\rceil + \frac{B_i}{t} \right) \leq 1.$$

5.6 Non-Preemptive Static Priority Scheduling

The RMS, DMS, and LMS algorithms assume arbitrary preemption is allowed. When arbitrary preemption is not allowed, the PCP may be used to ensure the schedulability of a task set. In the extreme case, no preemption is allowed, and the PCP causes an unacceptable amount of overhead.

The critical instant for a task set when preemption is not allowed, a non-preemptive task set, is different from the critical instant for a preemptive task set. The following theorem formally presents the notion of a critical instant for a non-preemptive task set.

Theorem 5.6.1 *A* critical instant *for any task when preemption is not allowed occurs whenever the task is requested simultaneously when all higher priority tasks are requested and the processor is allocated to the lower priority task with the longest computation time.*

Proof Let $\tau_n = \{T_1, \ldots, T_n\}$ denote a set of priority ordered tasks with T_n being the lowest priority task. Consider the request for task T_i at time t_1. Let T_j, $i < j$, be the lower priority task with the longest computation time. Let T_k be the task allocated the processor at time t_1. The maximum response time corresponds to the maximal interference due to other tasks.

If T_k equals T_i, corresponding to either the processor being idle at time t_1 or a task completing execution at time t_1 and T_i being the highest priority task ready to run, then the response time is zero. However, if T_k is not equal to T_i, then the response time for T_i is non-zero. Therefore, the response time can only be maximal if T_k is not equal to T_i. Furthermore, the maximum interference experienced by T_i due to T_k occurs when T_k is allocated the processor at time t_1.

Suppose that T_h, $h < i$, is the first higher priority task requested at or after time t_1. Then, between times t_1 and $t_1 + d_i$, T_h is requested at times t_2, $t_2 + p_h$, $\cdots$, $t_2 + xp_h$. In order for T_h to affect the response time of T_i, t_2 must be less than or equal to $t_1 + c_k$. The interference due to T_h is αc_h where α is the number of times T_h is invoked before T_i is allocated the processor.

Suppose that T_g, $g < i$, is the second higher priority task requested after t_1. Then, between times t_1 and $t_1 + d_i$, T_g is requested at times t_3, $t_3 + p_g$, $\cdots$, $t_3 + yp_g$. In order for T_g to affect the response time of T_i, t_3 must be less than or equal to $t_1 + c_k + c_h$. The interference due to T_g is βc_g where β is the number of times T_g is invoked before T_i is allocated the processor.

Repeating the argument for all T_m, $m = 1 \ldots i-1$ it can be seen that the maximum interference due to higher priority tasks is $ac_1 + \cdots + bc_{i-1}$ where the coefficients to the computation times are dependent upon the number of times the task is invoked before T_i is allocated the processor. Moreover, the maximum number of invocations of a higher priority task occurs when the higher priority task is invoked at time t_1.

Suppose that $1 \leq k \leq i-1$ (T_k corresponds to a higher priority task). Then $ac_1 + \cdots + bc_{i-1}$ is the maximum response time for T_i. Instead, suppose that $k > i$ (T_k corresponds to a lower priority task). The response time for T_i is then $c_k + ac_1 + \cdots + bc_{i-1}$ which is greater than $ac_1 + \cdots + bc_{i-1}$. Thus, T_k must be a task with a lower priority than T_i. Furthermore, to obtain a maximum response time, T_k must be T_j. ■

As in the preemptive case, if all tasks in a task set are able to meet their deadlines at their respective critical instants, then the task set is schedulable. Thus, to determine if a particular priority assignment will yield a valid schedule, the schedule may be constructed through $\max_{1 \leq i \leq n}(p_i)$ assuming that all tasks have the same release time r. This is a pessimistic test since a critical instant may never occur for some tasks due to the phasing and periods of the tasks.

As in the preemptive case, the RMS, DMS, and LMS algorithms are optimal algorithms in the sense that they will find a valid priority assignment if one exists for a task set satisfying their assumptions. The proofs of the optimality in the preemptive case for these algorithms do not rely upon the preemptive nature of the algorithms. Lemma 5.4.1

is valid for the non-preemptive case, and similar Lemmas for both RMS [LL73] and DMS [LW82] can be used to prove the optimality of these algorithms.

5.7 Sporadic Task Scheduling in Static Priority Systems

Only periodic task sets have been addressed for static priority systems to this point. However, many real-time systems contain sporadic tasks. Most traditional approaches for handling sporadic tasks are inadequate for guaranteeing that deadlines are met. Two such approaches are (1) allowing the sporadic tasks to interrupt the periodic tasks and run to completion, and (2) force the sporadic tasks to background service at a lower priority level than all of the periodic tasks. In approach (1) it is possible that a periodic task will miss a deadline. This condition cannot be checked for unless all possible schedules are generated and checked off-line. In approach (2) servicing of sporadic requests occurs whenever the processor is idle; thus, it is possible that a sporadic task will miss its deadline, unacceptable behavior if the sporadic task has a hard deadline.

An alternative to these approaches is to establish a polling server to provide a private resource for the exclusive use of the sporadic tasks. This approach creates a periodic task that is scheduled like other periodic tasks. When the polling task is invoked, it services any pending sporadic requests. However, if no sporadic requests exist, the polling task suspends itself until its next period, and the time originally allocated for sporadic service is not preserved but is instead used by periodic tasks. This is an improvement, but is still rather inflexible in that it offers regular service to a stream of tasks whose demand for that service is irregular. By modifying the sporadic server to serve the tasks when they are ready, significant improvements in performance and guaranteed schedulability may be obtained.

5.7.1 The Priority Exchange Algorithm

The *Priority Exchange algorithm* (PE) [LSS87] provides a method by which sporadic tasks are serviced in a static priority scheduling environment. The PE algorithm minimizes sporadic response times without causing periodic tasks to miss their deadlines. In the PE algorithm, a sporadic server is assigned a high priority. If sporadic tasks are ready when the server is allocated the processor, the sporadic tasks are executed at the priority of the server. Once the server time is consumed, no more sporadic tasks may be executed until the server is invoked and allocated the processor again. If there are no sporadic tasks ready

to execute when the sporadic server is allocated the processor, then the sporadic server exchanges its high priority execution time with the execution time of the highest priority periodic task that is ready to execute. If there are no periodic or sporadic tasks ready to execute, the server time is lost. In this way, the sporadic server time is preserved, although at a lower priority level, allowing the PE algorithm to accumulate deferred run-time across period boundaries.

The PE algorithm does not adversely affect the schedulability of a task set.

Theorem 5.7.1 ([LSS87]) *For a set of schedulable fixed-priority ordered tasks, τ_n, with T_1 the sporadic server, T_1 can trade run-time c_1, at priority ρ_1 for run-time c_i, at priority ρ_i, with any underlying periodic task without degrading the schedulability of the underlying periodic task set, $T_2, \ldots, T_n$.*

The proof follows from the one-to-one trading that at best advances the underlying periodic task T_i's execution; thus, guaranteeing its deadline and delaying the service for the sporadic tasks that are not guaranteed.

Lehoczky *et. al.* were able to derive a utilization bound below which a task set that contains the sporadic server task T_1 is guaranteed to be schedulable.

Theorem 5.7.2 ([LSS87]) *For a set of fixed-priority ordered tasks, τ_n, scheduled using the* RMS *algorithm where T_1 has the highest priority, the least upper schedulability bound as a function of the utilization, U_1, of task T_1 is*

$$U^* = U_1 + (n+1)\left(\left(\frac{2}{U_1+1}\right)^{\frac{1}{m-1}} - 1\right)$$

that converges to

$$U^* = U_1 + \ln \frac{2}{U_1+1}$$

as $n \to \infty$.

The above theorem applies not only to the PE algorithm, but also to the more general polling case. Hard sporadic tasks are not guaranteed to meet their deadlines. They are only guaranteed to have a low average response time.

5.7.2 The Deferrable Server Algorithm

The PE algorithm may be difficult to implement due to the priority exchange of execution times. A simpler sporadic server implementation may be obtained with the *Deferrable Server algorithm* (DS) [LSS87]. The DS algorithm is similar to the PE algorithm with the exception that it does not exchange its high priority run-time with that of lower priority tasks. Thus, the sporadic server execution time is only consumed when there is a sporadic task ready to execute. Any server execution time left at the end of the server period is lost. This simplifies the implementation while preserving the bandwidth.

The DS algorithm does not adversely affect the schedulability of a task set; however, the DS algorithm creates a task that does not satisfy one of the assumptions made by Liu and Layland: namely, requests for the deferrable server task are not necessarily separated by constant intervals. This is important because the DS algorithm was created for use with the RMS algorithm. For this reason, two cases must be analyzed to determine the least upper bound on utilization: (1) the DS task is requested three times during the period for task T_n, (2) the DS task is requested twice during the period for task T_n. Here, T_n is the task with the largest period.

For case (1), the least upper bound to utilization is given by the following theorem.

Theorem 5.7.3 ([LSS87]) *For a set of n fixed priority ordered tasks $\tau_n = \{T_1, \ldots, T_n\}$ with a critical zone length greater than $p_1 + c_1$ where T_1 is the Deferrable Server task, the least upper utilization bound as a function of the utilization of task T_1, U_1, is*

$$U = U_1 + (n-1)\left[\left(\frac{U_1 + 2}{U_1 + 1}^{\frac{1}{n-1}}\right) - 1\right]$$

that converges to

$$U = U_1 + \ln\left(\frac{U_1 + 2}{2U_1 + 1}\right) \quad \textit{as } n \to \infty.$$

This function has a minimum of 0.6518 when $U_1 = 0.186$.

For case (2), the least upper bound to utilization is much worse. The utilization bound is strongly dependent upon the utilization of the Deferrable Server task. The following equation gives the utilization bound as a function of U_1 [LSS87].

$$U = \begin{cases} 1 - U_1 & U_1 < 0.5 \\ U_1 & U_1 \geq 0.5 \end{cases}$$

The least upper bound to utilization in this case is 0.500.

Due to the much lower least upper utilization bound experienced in case (2), the designer should attempt to avoid this case. The designer will need to ensure that the longest period, p_n, is greater than $p_1 + c_1$ where T_1 is the Differable Server task.

5.7.3 The Sporadic Server Algorithm

Although the PE and DS algorithms are able to obtain a low average response time for sporadic tasks, they are not able to handle hard sporadic tasks. The *Sporadic Server algorithm* (SS) [SSL89] improves upon the PE and DS algorithms by drawing upon the advantages of each.

The SS algorithm creates a high-priority task for servicing sporadic tasks. The sporadic server's execution time is preserved at its high priority level until a sporadic request occurs. The SS algorithm differs from the PE and the DS algorithms in the way it replenishes its server execution time. The sporadic server execution time is replenished after some or all of the execution time is consumed by sporadic task execution; in comparison to periodically replenishing the server execution time to full capacity as in the PE and DS algorithms. The rules of execution time replenishment for a sporadic server executing at priority level i are as follows [SSL89].

1. If the sporadic server has execution time available, the replenishment time, RT, is set when a task of priority level i or greater begins execution. If, on the other hand, the sporadic server capacity has been exhausted, then RT cannot be set until the sporadic server's available execution time becomes greater than zero and a task of priority level i or greater is executing. In either case, the value of RT is set equal to the current time plus the period of the sporadic server task.

2. The amount of execution time to be replenished can be determined when either a task with a priority level less than i executes or when the sporadic server's available execution time has been exhausted. The amount to be replenished at RT is equal to the amount of sporadic server execution time consumed since the last time at which a task of priority level i or greater began executing after a task of priority level less than i was executing.

Since a sporadic server task may defer its execution when no sporadic tasks are ready, it violates a basic assumptions made by the RMS algorithm, once a periodic task is

the highest-priority task that is ready to execute, it must execute. If a periodic task defers its execution when it otherwise could execute immediately, it may be possible that a lower-priority task will miss its deadline. The SS's replenishment method compensates for any deferred execution of the sporadic server. This allows it to provide an average response time that is better than that of the DS algorithm with approximately the same implementation complexity.

Sprunt, Sha, and Lehoczky were able to prove that a sporadic server that follows the above replenishment policy does not affect the schedulability of a periodic task set.

Theorem 5.7.4 ([SSL89]) *A periodic task set that is schedulable with a task, T_i, is also schedulable if T_i is replaced by a sporadic server, using the SS algorithm, with the same period and execution time.*

Furthermore, several sporadic server tasks may be defined at different priority levels to handle different sporadic streams.

The SS algorithm provides a low average response time to sporadic tasks with soft deadlines. In addition, it is able to guarantee that all hard sporadic tasks meet their deadlines if their deadlines are greater than their minimum time between successive invocations. However, when the deadline of a hard sporadic task is less than the minimum time between invocations, the SS algorithm does not guarantee that all deadlines are met. To ensure that a hard sporadic task will not miss a deadline the sporadic server is assigned a priority based upon the deadline of the task that it is to serve. This is equivalent to the deadline monotonic priority assignment. Note, only a single sporadic server is created for all soft sporadic tasks while one sporadic server is created for each hard sporadic task.

If preemption is not allowed in the system, then the SS algorithm is not guaranteed to work.

5.7.4 Summary of Sporadic Task Handling

Table 5.1 summarizes the major characteristics of the PE, DS, and SS algorithms. These algorithms were developed for use with the RMS algorithm as a way to handle soft sporadic tasks. The SS algorithm is the only one that allows the RMS algorithm to handle hard sporadic tasks correctly.

All three of the sporadic server algorithms may be used with both the DMS and LMS algorithms to handle soft sporadic tasks. Hard sporadic tasks are handled by the

	PE	DS	SS
Implementation Complexity	High	Low	Medium
Low Average Response Time For Soft Tasks	Yes	Yes	Yes
Handles Hard Tasks	No	No	Yes
Server Run Time Replenishment	Periodic	Periodic	One Period After Use
Exchange Run Time	Yes	No	No
Preserves Run Time Across Period Boundaries	Yes	No	Yes
Preserves Bandwidth	Yes	Yes	Yes
Requires Special Schedulability Analysis Test	Yes	Yes	No
Multiple Servers Allowed	No	No	Yes

Table 5.1: Comparison of the PE, DS, and SS sporadic server algorithms.

DMS and LMS algorithms. When the DMS algorithm is used, sporadic tasks are assigned a priority based upon their deadline just as the periodic tasks. When the LMS algorithm is used, sporadic tasks are assigned a priority based upon their laxity just as the periodic tasks.

5.8 Static Priority Schedulability Analysis

5.8.1 Utilization Based Schedulability Analysis

To guarantee the predictability of a task set scheduled using a static priority assignment, the schedulability of the task set under the given priority assignment must be verified. Many utilization based schedulability analysis tests have been developed. Lui and Layland were the first to present a schedulability analysis test based upon the total utilization of the task set. The test presented in Theorem 5.2.3 is a sufficient test for schedulability.

Theorem 5.2.3 was improved upon by Lehoczky [Leh90] to handle the more general case where the deadline is less than or equal to the period. The more general utilization bound test is given be the following theorem.

Theorem 5.8.1 ([Leh90]) *A periodic task set, τ_n, with $d_i = \Delta p_i$, $1 \leq i \leq n$ is schedulable if*

$$\sum_{i=1}^{n} U_i \leq U_n^* = \begin{cases} n((2\Delta)^{1/n} - 1) + (1 - \Delta) & \frac{1}{2} \leq \Delta \leq 1 \\ \Delta & 0 \leq \Delta \leq \frac{1}{2} \end{cases}$$

This is a sufficient utilization bound, and it may be used for any static priority scheduling algorithm that allows arbitrary preemption.

A less general utilization bound was derived by Lehoczky *et.al.* [LSST91] for the RMS algorithm when the periods of the tasks in the task set are harmonics of one another.

Theorem 5.8.2 ([LSST91]) *If a task set τ_n is scheduled using the* RMS *algorithm and p_j evenly divides p_i for $1 \leq j \leq i$, then T_i meets all its deadlines if and only if $\sum_{k=1}^{i} U_k \leq 1$. If p_j evenly divides p_i for all $j \leq i$, $1 \leq i \leq n$, then the task set is schedulable if and only if $\sum_{k=1}^{n} U_k \leq 1$.*

5.8.2 Synchronous Schedulability Analysis

Utilization is not the only factor upon which schedulability analysis tests may be based. Theorem 5.2.1 suggests that for a task $T_i \in \tau_n$ (where τ_n is sorted in non-increasing priority order) to be schedulable, the sum of its computation time and the time spent waiting while tasks of higher priority execute on the processor (the *interference* time) must be no more than d_i. For a task set where all of the tasks are released at the same time (a synchronous task set), the above test may be performed in pseudo-polynomial time.

Theorem 5.8.3 ([LW82]) *For a synchronous task set there is a pseudo-polynomial time algorithm to decide whether or not the schedule produced by a particular priority assignment is valid.*

Audsley *et. al.* [ABRW91] formalized this critical instant test with:

$$\forall i : 1 \leq i \leq n : \frac{c_i}{d_i} + \frac{I_i}{d_i} \leq 1$$

where I_i is a measure of the interference caused by higher priority tasks:

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{d_i}{p_j} \right\rceil c_j.$$

The interference is composed of the computation time of all higher priority tasks. It does not take into account the release times of the tasks, and it includes the computation time of higher priority tasks that will occur only after d_i. A less pessimistic measure of the interference is given by:

$$I_i = \sum_{j=1}^{i-1} \left[\left\lfloor \frac{d_i}{p_j} \right\rfloor c_j + \min\left(c_j, d_i - \left\lfloor \frac{d_i}{p_j} \right\rfloor p_j \right) \right].$$

This measurement does not include the parts of executions of higher priority tasks that could not occur before d_i. However, it is a sufficient but not necessary test.

Audsley *et.al.* [ABRW91] derived a schedulability analysis test that is both necessary and sufficient. This test constructs a schedule such that the exact interleaving of higher priority task executions is known. Figure 5.1 presents the static priority schedulability analysis test given in [ABRW91]. This test is valid for synchronous task sets and runs in pseudo-polynomial time. The value for the interference for this test is defined to be

$$I_i^t = \sum_{j=1}^{i-1} \left\lceil \frac{t}{p_j} \right\rceil c_j.$$

```
// Static priority schedulability analysis for a synchronous task set.

StaticPrioritySchedAnal( ) {
   foreach T_i {
      t = Σ_{j=1}^{i} c_j
      continue = TRUE
      while ( continue ) {
         if (I_i^t/t + c_i/t ≤ 1) {
            // T_i is schedulable
            continue = FALSE
         } else {
            t = I_i^t + c_i
         }
         if (t > d_i) {
            // T_i is not schedulable
            EXIT
         }
      }
   }
}
```

Figure 5.1: Audsley *et. al.*'s algorithm to determine if a synchronous static priority task set τ_n is schedulable.

5.8.3 Asynchronous Schedulability Analysis

Not all task sets are synchronous. Some task sets are asynchronous, i.e., not all tasks have the same release time. Leung and Whitehead derived the complexity bound for determining if a priority assignment yields a valid schedule for an asynchronous task set.

Theorem 5.8.4 ([LW82]) *Given a task set τ_n with arbitrary release times and a priority assignment ρ, the problem to decide whether or not the schedule produced by ρ is valid is $\mathcal{NP}$-hard.*

Leung and Whitehead were able to extend this theorem to cover the more general case of scheduling on $m \geq 1$ processors.

Lehoczky, Sha, and Ding [LSD89] derived necessary and sufficient conditions for determining if a static priority assignment yields a valid schedule for an asynchronous task set. This schedulability analysis test is based on Theorem 5.2.1. The demands made by the task set as a function of time are considered to determine if a task can meet its deadline. This is formally stated in the following theorem.

Theorem 5.8.5 ([LSD89]) *Let a periodic task set τ_n be given in priority order and scheduled by a fixed priority scheduling algorithm using those priorities. If $d_i \leq p_i$, then T_i will meet all its deadlines under all release times if and only if*

$$\min_{0 \leq t \leq d_i} \sum_{j=1}^{i} \frac{c_j}{t} \left\lceil \frac{t}{p_j} \right\rceil \leq 1$$

The entire task set is schedulable under the worst case release time assignment if and only if

$$\max_{1 \leq i \leq n} \min_{0 \leq t \leq d_i} \sum_{j=1}^{i} \frac{c_j}{t} \left\lceil \frac{t}{p_j} \right\rceil \leq 1.$$

5.8.4 Non-preemptive Schedulability Analysis

When preemption is not allowed, simple utilization based schedulability analysis tests can not be used. Computation time-based tests and simulation-based tests must be used.

Theorem 5.6.1 suggests a direct, computation time based test to determine whether a particular priority assignment yields a valid schedule. Intuitively, this test checks that the processor is not overloaded during the interval from when the task is invoked at its critical instant until its deadline. Theorem 5.8.6 formally presents this schedulability analysis test.

Theorem 5.8.6 *A periodic task set, τ_n, arranged in non-decreasing priority order, that does not allow preemption is schedulable if*

$$\left(\frac{c_k}{d_k}\right) + \max_{1 \leq i < k}\left(\frac{c_i}{d_k}\right) + \sum_{j=k+1}^{n}\left(\left\lceil\frac{d_k}{p_j}\right\rceil\frac{c_j}{d_k}\right) \leq 1 \;\forall\; T_k \in \tau_n.$$

Proof This theorem places a bound on the processor demand that may occur from the critical instant of a task to the deadline of that task, for every task, T_k, without that task missing a deadline.

Let task T_k be invoked at time t_k, its critical instant. The processor demand, D_{T_k}, in the interval $[t_k, t_k + d_k]$ due to task T_k is c_k. The processor demand, D_{lower}, in the interval $[t_k, t_k + d_k]$ due to the lower priority task with the largest computation time is $\max_{1 \leq i < k}(c_i)$. The processor demand, D_{higher}, in the interval $[t_k, t_k + d_k]$ due to the higher priority tasks is bounded by

$$D_{\text{higher}} \leq \sum_{j=k+1}^{n}\left(\left\lceil\frac{d_k}{p_j}\right\rceil c_j\right).$$

$\lceil d_k/p_j \rceil$ is the maximum number of times that each higher priority task may be invoked during $[t_k, t_k + d_k]$.

The total processor demand during the interval $[t_k, t_k + d_k]$ must be less than d_k if T_k is to meet its deadline. Therefore, assuming worst-case demand due to all higher priority tasks, the processor demand during $[t_k, t_k + d_k]$ must satisfy the following equation

$$c_k + \max_{1 \leq i < k}(c_i) + \sum_{j=k+1}^{n}\left(\left\lceil\frac{d_k}{p_j} = \right\rceil c_j\right) \leq d_k.$$

Rearranging this equation by dividing through by d_k yields

$$\left(\frac{c_k}{d_k}\right) + \max_{1 \leq i < k}\left(\frac{c_i}{d_k}\right) + \sum_{j=k+1}^{n}\left(\left\lceil\frac{d_k}{p_j}\right\rceil\frac{c_j}{d_k}\right) \leq 1.$$

Thus, if the above equation holds, then T_k is guaranteed to meet its deadline. Since T_k may be any task, the theorem is proved. ■

This is a pessimistic test as it assumes that all tasks are independent and have the same release time. It also includes the computation time of higher priority tasks that will occur

only after d_k. Theorem 5.8.7 presents a slightly less pessimistic schedulability analysis test that does not include the computation time of higher priority tasks that will only occur after d_k.

Theorem 5.8.7 *A periodic task set arranged in priority order, τ_n, that does not allow preemption is schedulable if*

$$\left(\frac{c_k}{d_k}\right) + \max_{1 \leq i < k}\left(\frac{c_i}{d_k}\right) + \sum_{j=k+1}^{n}\left[\left\lfloor\frac{d_k}{p_j}\right\rfloor\frac{c_j}{d_k} + \min\left(\frac{c_j}{d_k}, 1 - \left\lfloor\frac{d_k}{p_j}\right\rfloor\frac{p_j}{d_k}\right)\right] \leq 1 \;\forall\; T_k \in \tau_n.$$

Proof Let task T_k be invoked at time t_k, its critical instant. The processor demand due to task T_k in the interval $[t_k, t_k + d_k]$ is c_k. The processor demand do to the lower priority task with the maximum execution time in the interval $[t_k, t_k + d_k]$ is $\max_{1 \leq i < k}(c_i)$.

The processor demand do to the higher priority tasks in the interval $[t_k, t_k + d_k]$ is calculated as follows. The maximum number of periods that a higher priority task, T_j, can have during the interval $[t_k, t_k + d_k]$ is d_k/p_j. For each complete period, T_j is completely executed once. The processor demand due to the complete executions of task T_j is $\lfloor d_k/p_j \rfloor c_j$.

For the partial period, the processor demand may or may not be equal to the computation time of task T_j. If the difference $(t_k + d_k) - (t_k + \lfloor d_k/p_j \rfloor p_j)$ is less than c_j, then the processor demand due to task T_j is equal to $d_k - \lfloor d_k/p_j \rfloor p_j$. The processor demand due to the partial period of task T_j is therefore $\min(c_j, d_k - \lfloor d_k/p_j \rfloor p_j)$.

To ensure that T_k meets its deadline, the processor demand during $[t_k, t_k + d_k]$ must satisfy the following equation

$$c_k + \max_{1 \leq i < k}(c_i) + \sum_{j=k+1}^{n}\left[\left\lfloor\frac{d_k}{p_j}\right\rfloor c_j + \min\left(c_j, d_k - \left\lfloor\frac{d_k}{p_j}\right\rfloor p_j\right)\right] \leq d_k.$$

Rearranging this equation by dividing through by d_k yields

$$\left(\frac{c_k}{d_k}\right) + \max_{1 \leq i < k}\left(\frac{c_i}{d_k}\right) + \sum_{j=k+1}^{n}\left[\left\lfloor\frac{d_k}{p_j}\right\rfloor\frac{c_j}{d_k} + \min\left(c_j, 1 - \left\lfloor\frac{d_k}{p_j}\right\rfloor\frac{p_j}{d_k}\right)\right] \leq 1.$$

Thus, if the above equation holds, then T_k is guaranteed to meet its deadline. Since T_k may be any task, the theorem is proved. ■

This is a sufficient test to determine the validity of a particular priority assignment.

5.9 Advantages of the Static Priority Approach

The key advantage of the static priority approach is the simplicity of the implementation. A simple implementation maintains the priority of each task in a table with the ready list kept in priority order, requiring a minimal amount of code to implement. Therefore, little ROM space is used, critical for memory limited applications.

The information required to make a priority assignment is typically small, e.g., the RMS algorithm only requires the period of the tasks. In addition, the assignment of priorities is accomplished by a simple sort of the tasks.

The LMS algorithm automatically identifies the most time critical tasks and assigns them the highest priorities. This is important when the task set contains sporadic tasks with very tight deadlines.

The schedulability of a task set by the static priority approach may be determined off-line by the use of utilization-based, critical instant-based, or simulation-based schedulability analysis tests. Many of these schedulability analysis tests are easy to compute and may be performed with accurate worst case execution times, release times, deadlines, and periods.

Although the worst case schedulability analysis test in Theorem 5.2.3 suggests a utilization of less than $\ln 2$ is required for a static priority algorithm to work, a utilization of approximately 88% can usually be achieved [Leh90]. Critical instant-based and simulation-based schedulability analysis tests can provide less pessimistic schedulability analysis tests.

Many of the practical problems that arise in actual real-time systems have been addressed with possible solutions formulated. Task synchronization may be accomplished by the Priority Ceiling Protocol that guarantees deadlock avoidance and provides a schedulability analysis test, eliminating the possibility of unpredictable delays at run time. The Sporadic Server algorithm may be used to schedule both hard and soft sporadic tasks without compromising the schedulability of the task set.

In sporadic task sets there is always the possibility for a large number of tasks to be ready to execute at the same time, t. When the demand for the processor exceeds the available processor time for an interval starting at time t, we say that a *transient overload* occurred at time t. Static priority algorithms provide a way to guarantee that critical tasks will meet their deadlines during a transient overload. The critical tasks are assigned a high priority; thus, they will be guaranteed to execute during the transient overload.

Both preemptive and non-preemptive static priority approaches may be used. This allows the static priority approach to be used in a wide variety of environments.

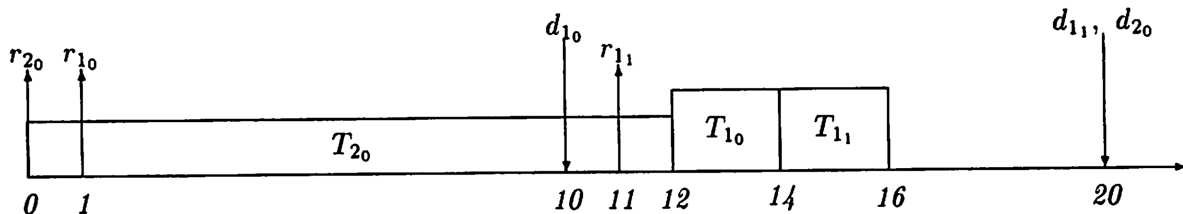
5.10 Disadvantages and Other Issues of the Static Priority Approach

Static priority-driven schemes are only capable of producing a very limited subset of the possible schedules for a given task set, severely restricting the capability of priority-driven schemes to satisfy timing and resource sharing constraints at run time. Therefore, it is possible that the static priority scheduling approach will not yield a valid schedule for a schedulable task set. For example, there are situations where, in order to satisfy all timing constraints, it is necessary to let the processor be idle for a certain interval in time, even though there are tasks that have arrived and are requesting use of system resources. This is especially true when preemption is not allowed either because the system does not allow preemption or because a task is utilizing a non-preemptable resource or executing in a critical section.

The following example illustrates a case when inserted idle time is the only method by which a valid schedule may be obtained.

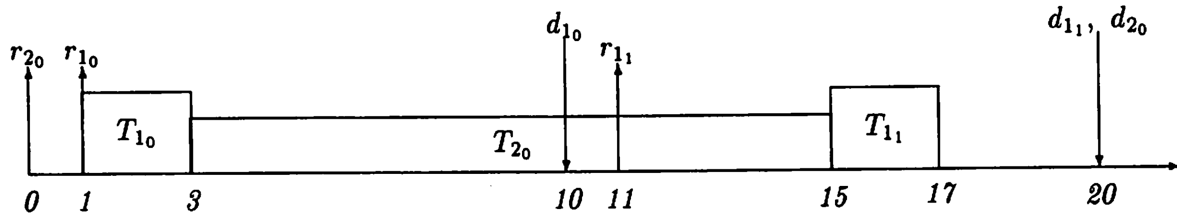
Example 5.10.1 *Consider the following two tasks represented by the 4-tuple,* $T = (r, c, p, d)$*:* $T_1 = (1, 2, 10, 9)$ *and* $T_2 = (0, 12, 20, 20)$.

Scheduling these tasks with a non-preemptive static priority algorithm that does not use inserted idle time yields the following schedule (repeated every 20 time units).



Notice that T_1 *misses its deadline at time 10 even though* T_1 *has a higher priority than* T_2 *if either* RMS, DMS, *or* LMS *is used.*

The following valid non-preemptive schedule is obtained with inserted idle time.



■

Clearly, some cases exist when a static priority scheduler must be able to insert idle time in order to find a valid schedule. Knowledge of all future invocations of tasks is required for a run-time scheduler to determine when and how much idle time to insert.

Actual execution orderings of tasks may not adhere to the original priorities of the tasks due to priority inversion, making it difficult to verify that all timing and resource constraints will be satisfied. The Priority Ceiling Protocol attempts to deal with this problem for the special case when all tasks are independent; however, it is not efficient for the general scheduling problem where precedence constraints often exist.

Due to the static nature of the static priority assignment, the addition of tasks to the task set may invalidate the given priority assignment, forcing the priority assignments to be recalculated. Recalculating the priorities requires the system to be halted. Thus, dynamic systems are not efficiently handled by the static priority scheduling algorithms.

Mode changes are not easily handled by static priority scheduling algorithms; the same task may have different priorities in different modes of operation. One possible way of dealing with mode changes is to consider all tasks that will ever be run by the system and assign a unique priority to each task. Thus, a single table may hold all priorities instead of one table for each mode. Unfortunately, a task may have different characteristics in different modes of operation requiring it to have multiple priorities. For example, task T may have period $p_A = 10$ in mode A and period $p_B = 5$ in mode B. If all other tasks remain unchanged, a different priority assignment must be calculated for modes A and B.

As has been noted previously, the determination of the worst-case execution time for a task is difficult due to false paths and loops. Pessimistic estimates of worst-case execution time will cause the utilization schedulability analysis tests to be even more pessimistic while optimistic estimates may cause the tests to falsely determine that a task set is schedulable.

In addition, the timing behavior of such complex run time synchronization mechanisms as rendezvous and monitors is often extremely difficult to predict with any certainty.

The use of such synchronization mechanisms by individual tasks allows individual tasks to make important scheduling decisions even though individual tasks do not possess global information about the system, making it virtually impossible for the scheduler to prevent deadlocks and guarantee deadlines. In general, deadlock avoidance at run time requires that the run time synchronization mechanism be conservative, resulting in situations where a process is blocked by the run time synchronization mechanism even though it could have proceeded without causing deadlock.

The static priority theory does not directly address the problem of distance constraints [HL92]. For example, under the presented static priority algorithms it is possible that two consecutive invocations of a task are executed consecutively in time. This corresponds to the first invocation being executed at the end of the period and the second invocation being executed at the beginning of the period. The opposite is also possible, the first invocation is executed at the beginning of its period while the second invocation is executed at the end of its period. This situation arises since the execution of a task in one period is independent of the execution of the same task in any other period. The only way to enforce distance constraints in a static priority schedule is to make those tasks that have distance constraints a high priority, forcing the task to be executed near the beginning of its period. Unfortunately, if there are multiple tasks with distance constraints, this approach will not work.

5.11 Implementation of the Static Priority Approach Within POLIS

5.11.1 Routines Implemented Within POLIS

Within the POLIS environment, the RMS, DMS, and LMS algorithms are implemented. The assignment of priorities is performed using the standard quick sort routine in the C libraries. No assumptions are made about the tasks when they are assigned priorities, allowing the assignment of priorities to be performed in $\mathcal{O}(n \ln n)$ time. The designer must guarantee that the task set satisfies the assumptions made by the algorithm being used.

Sporadic servers are not implemented within POLIS due to the large code size required to implement them. Consequently, the characteristics of soft sporadic tasks must be manipulated by the designer to force them to a low priority. In addition, if hard spo-

radic tasks are present, the RMS algorithm should not be used. Only the DMS and LMS algorithms correctly assign priorities to hard sporadic tasks when sporadic servers are not used.

Since PoLIS must be able to handle sporadic task sets, the implemented schedulability analysis tests guarantee the schedulability of both periodic and sporadic task sets. Soft tasks are ignored in the preemptive schedulability analysis tests since they do not affect the schedulability of the task set. Soft tasks are not ignored in the non-preemptive schedulability analysis tests since they affect the schedulability of the task set.

If the task set is completely periodic and preemption is allowed, a utilization based schedulability analysis test is performed. The utilization based test is from Theorem 5.8.1. This allows the same utilization based schedulability analysis test to be performed regardless of which static priority algorithm is used to assign the priorities. In order to determine a Δ for the schedulability analysis test, the Δ_i for each task $T_i \in \tau_n$ is determined. The smallest Δ is used in the formula for determining the maximum possible utilization above which the task set is not guaranteed to be schedulable. This is the worst case behavior, since the smaller the Δ, the smaller the utilization bound.

If preemption is not allowed, the schedulability analysis test presented in Theorem 5.8.7 is used to determine the schedulability of the task set.

If the utilization based test can not guarantee the schedulability of the periodic task set, the task set is simulated through the LCM of the task set, allowing for a tighter check on the schedulability of the task set at the expense of processing time. Thus, it is unlikely that a periodic task set will be labeled as unschedulable when it is truly schedulable. The simulation of the task set identifies those tasks that will miss deadlines, as well as, the time at which they will miss them.

If the task set contains sporadic tasks, then a simple schedulability analysis test is not possible to guarantee the schedulability of the task set. A complete simulation of the task set is also not feasible due to the almost infinite number of possible invocation times of the sporadic tasks. In order to guarantee the schedulability of the task set through simulation, every possible schedule must be generated.

The schedulability analysis test used to guarantee the schedulability of sporadic task sets is based upon the critical instant of a task. All tasks are assumed to be independent with the same release time, and the task set is simulated through $\max_{1 \leq i \leq n}(p_i)$. The assumptions that all tasks are independent and that the critical instant for each task

```
// non-preemptive static priority scheduling routine; tasks are arranged
// in order by priority
scheduler()
{
    task = 0;
    while( 1 ) {
        if(task == 0 )
            poll_inputs_and_update_input_buffers();
        if( is_ready( task ) ) {
            execute( task );
            task = 0;
        } else {
            if( (++task) ≥ NUMBER_TASKS ) task = 0;
        }
    }
}
```

Figure 5.2: Generated non-preemptive static priority scheduling routines.

will occur causes this test to be pessimistic, but it guarantees that problem tasks will be identified.

5.11.2 Generated Scheduling Routines

POLIS generates a priority ordered array of tasks, referred to whenever a task of a given priority is desired.

The non-preemptive static priority scheduling algorithm shown in Figure 5.2 uses a polling based approach to determine the highest priority ready task. Starting with the highest priority task and proceeding in priority order, tasks are checked until either a ready task is found and executed to completion or all tasks have been checked, at which point the checking begins again with the highest priority task.

Before the highest priority task is checked, all input buffers, the contents of which indicate whether or not a task is ready to execute, are updated, essentially, polling all input

events.

The preemptive static priority scheduling routines shown in Figure 5.3 rely upon interrupts to perform preemption. The interrupt handling routine is not provided any information as to which event caused the interrupt; therefore, it updates all input buffers and, starting with the highest priority task, searches for a ready task.

If the interrupted task is the highest priority task ready to execute, the interrupt handler returns; otherwise, all ready higher priority tasks are executed before the interrupt handler returns, allowing nested interrupts.

```
// main preemptive static priority scheduling routine; tasks are arranged
// in order by priority
scheduler()
{
    task = 0;
    while( 1 )
       task = check_and_execute_task( task );
       current_task = NUMBER_TASKS;
       if( ( ++task ) ≥ NUMBER_TASKS ) task = 0;
}

// checks given task and executes it if it is ready
check_and_execute_task( task )
{
    if( is_ready( task ) ) {
       current_task = task;
       execute( task );
       update_input_buffers();
       task = -1;
    }
    return task;
}

// interrupt handling routine; updates input buffers and executes
// all higher priority tasks before returning to the interrupted task
interrupt_scheduler()
{
    interrupted_task = current_task;
    poll_inputs_and_update_input_buffers();
    task = 0;
    while( task < interrupted_task ) {
       task = check_and_execute_task( task );
       task++;
    }
    current_task = interrupted_task;
    return; // return from interrupt
}
```

Figure 5.3: Generated preemptive static priority scheduling routines.

Chapter 6

Dynamic Priority Scheduling

6.1 Introduction

Dynamic priority scheduling algorithms are run-time scheduling algorithms that assign a priority to each task at run time. The priority of a task may change at any time. The continual reevaluation of task priorities causes dynamic priority scheduling algorithms to have a high implementation overhead. To minimize the overhead, the priority assignment is performed only when the priorities of the tasks may change, at the completion of a task and the invocation of a task. Hardware support may be used to decrease or eliminate the scheduling overhead. For example, a dedicated co-processor for the dynamic scheduler may reduce the scheduling overhead to nearly zero [Mok83].

Typical dynamic priority scheduling algorithms do not assume any information about future invocations of tasks, making it difficult to guarantee the schedulability of a (possibly varying) task set. Dynamic priority scheduling algorithms that do not perform any type of schedulability analysis at run-time are referred to as *dynamic best effort scheduling algorithms.* Dynamic priority scheduling algorithms that do perform some type of schedulability analysis at run-time are referred to as *dynamic planning-based scheduling algorithms.*

Dynamic best effort scheduling algorithms employ a purely priority-driven scheduling approach. In a simple approach, at each instant that the priorities of the tasks may change, every task that is either ready or running is assigned a priority. A task with the highest priority is allocated the processor. This approach does not perform any type of schedulability analysis. Therefore, it is not known when a timing constraint will be violated

until the constraint is actually violated.

Dynamic planning-based scheduling algorithms provide the flexibility of the dynamic best effort scheduling algorithms with the predictability of schedulability analysis. In a simple approach, every dynamically arriving task is accepted for execution only if it is found to be able to meet its deadline. The set of tasks in the system that have not completed execution comprise a schedulable task set, τ_n, and a newly invoked task is accepted into the system if and only if the resulting task set, τ_{n+1}, is schedulable. This allows for predictability with respect to individual task arrivals.

The two most popular dynamic best effort scheduling algorithms, *Earliest Deadline First* (EDF) and *Minimum Laxity First* (MLF), are presented. The use of these algorithms in a dynamic planning based approach is then discussed. The problems of task synchronization, non-preemptive scheduling, and sporadic task scheduling are discussed. Finally, the dynamic priority algorithms that are implemented within the POLIS co-design environment are discussed.

6.2 Dynamic Best Effort Scheduling Algorithms

In dynamic best effort scheduling algorithms, a priority value is computed for each task in the system based on the task's characteristics, and the system schedules tasks according to their priority. The validity and predictability of the system is determined by extensive simulations, in conjunction with modifying task characteristics (usually by recoding the tasks) to adjust priorities, allowing the designers to be confident that the system will perform correctly under the tested operating conditions.

A commonly used dynamic best effort scheduling algorithm is the *Earliest Deadline First algorithm* (EDF), an optimal scheduling algorithm that is guaranteed to find a valid schedule if the task set is schedulable, presented by Liu and Layland in 1973 [LL73]. The ready task with the nearest deadline is assigned the highest priority and is allocated the processor.

Liu and Layland make the same assumptions about the EDF algorithm as they do about the RMS algorithm.

A1: All tasks are periodic and are ready at the beginning of each period.

A2: The deadline of each task is equal to its period.

A3: Tasks are independent, i.e., no precedence or exclusion constraints exist between tasks.

A4: The execution time for each task is constant for that task and does not vary with time.

A5: Arbitrary preemption is allowed.

Liu and Layland were able to prove the following theorem under these assumptions.

Theorem 6.2.1 ([LL73]) *When the* EDF *scheduling algorithm is used to schedule a set of periodic tasks on a single processor, there is no processor idle time prior to a missed deadline.*

Liu and Layland used this theorem to determine an upper bound on processor utilization, below which the EDF algorithm is guaranteed to schedule a periodic task set.

Theorem 6.2.2 ([LL73]) *For a given set of n periodic tasks, the* EDF *scheduling algorithm is valid on a uniprocessor if and only if*

$$\sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1$$

.

Thus, so long as the total utilization of the tasks in the task set does not exceed 1 and arbitrary preemption is allowed, the EDF algorithm is guaranteed to find a valid schedule on a uniprocessor.

Dertouzos [Der74] was able to prove the optimality of the EDF algorithm on a single processor for an arbitrary distribution of requests, i.e., the tasks are not necessarily periodic.

Theorem 6.2.3 ([Der74]) *The* EDF *algorithm is optimal in that if there exists any algorithm that can achieve scheduling of a single processor on an arbitrary distribution of requests, hard deadlines, execution times, and arbitrary preemption is allowed then the* EDF *algorithm will also achieve scheduling.*

The proof follows from the ability to swap execution times of tasks in a valid schedule such that the tasks are executed in EDF order without violating the validity of the schedule.

A second commonly used dynamic best effort scheduling algorithm is the *Minimum Laxity First algorithm* (MLF) (also known as the *Least Laxity First algorithm*), an optimal scheduling algorithm in the same sense as the EDF scheduling algorithm. This algorithm allocates the processor to the ready task with the minimum laxity, or time that the task may wait to begin execution before it is guaranteed to miss its deadline. Essentially, the MLF algorithm attempts to observe the time constraints that are placed upon the beginning of service (the time by which the task must begin execution) for a task.

Mok [Mok83] proved the optimality of the MLF algorithm.

Theorem 6.2.4 ([Mok83]) *The* MLF *algorithm is optimal in that if there exists any algorithm that can achieve scheduling of a single processor on an arbitrary distribution of requests, hard deadlines, execution times, and arbitrary preemption is allowed then the* MLF *algorithm will also achieve scheduling.*

Both the EDF and the MLF algorithms have a complexity of either $\mathcal{O}(m)$ or $\mathcal{O}(\ln m)$ depending upon the implementation, where m is the number of tasks that are ready to execute.

Neither the EDF nor the MLF algorithm makes assumptions about future task invocations. Nor do they perform any type of schedulability analysis test at run time. This makes them very simple to implement, but it does not provide any guarantees as to the schedulability of task sets.

6.3 Dynamic Planning-Based Scheduling Algorithms

Dynamic planning-based scheduling algorithms perform a schedulability analysis test at run-time in conjunction with a dynamic best-effort scheduling algorithm. In this way these algorithms attempt to guarantee the schedulability of a set of tasks.

The most common type of dynamic planning-based scheduling algorithm is one in which a schedulability analysis test (often a utilization based test) is performed at each task invocation on the set of active tasks[1] plus the invoked task. If the set of active tasks plus the invoked task are found to yield a valid schedule, then the invoked task is accepted into the active set and placed on the ready queue; otherwise, the task is rejected. In this

[1]A task is considered to be *active* at time t if it was invoked before t, but has not completed execution by t.

way, the dynamic planning-based algorithm identifies tasks that will miss their deadline or will cause other tasks to miss their deadlines.

Complex and exact schedulability analysis tests may be performed, but the execution time of the test limits the viable tests. A utilization based test, e.g., checking that the utilization of all active tasks plus the utilization of the invoked task is less than 1, used in conjunction with the EDF algorithm has a low execution time. However, a simulation-based test that determines the exact sequence of executions of the tasks has a large execution time. In general, the larger the expected maximum processor utilization by the tasks, the smaller the allowable execution time of the schedulability analysis test.

6.4 Task Synchronization in Dynamic Priority Systems

Thus far tasks have been assumed to be completely preemptable, i.e., no exclusion constraints exist between segments of the tasks. This is not always a realistic assumption since it is often the case that tasks contain sections of code that must be executed atomically and/or the tasks utilize non-preemptive resources. As in static priority systems, priority inversion may occur in dynamic priority systems when exclusion constraints exist between segments of tasks. Unfortunately, the PCP cannot be used if tasks do not have fixed priorities.

A task synchronization protocol based on the PCP is used to prevent deadlocks and still allow for schedulability analysis tests. The *Dynamic Priority Ceiling Protocol* (DPCP) [CL90] uses the notion of the *dynamic priority ceiling* of a semaphore to prevent deadlock and chained blocking.

Definition 6.4.1 *The* dynamic priority ceiling *of a semaphore S at time t is defined to be the dynamic priority of the highest dynamic priority task that currently locks or may lock S, at that time. The dynamic priority ceiling of a semaphore S_i, denoted $\mathcal{D}(S_i)(t)$, represents the highest dynamic priority that a critical section guarded by S_i can execute at time t, either by normal or inherited priority.*

Note, in determining the dynamic priority ceiling of a semaphore, the dynamic priority of non-active tasks is determined and used in determining the dynamic priority of a semaphore. For non-active tasks, the characteristics of the earliest possible task invocation are used to determine the dynamic priority of a task.

Definition 6.4.2 (Dynamic Priority Ceiling Protocol [CL90])

1. *Task T, which has the highest dynamic priority among the tasks ready to execute, is allocated the processor. Let S^* be the semaphore with the highest dynamic priority ceiling of all semaphores currently locked by tasks other than T. Before task T enters its critical section, it must first obtain the lock on the semaphore S guarding the shared data structure. Task T will be blocked and the lock on S will be denied if the dynamic priority of task T is not higher than the dynamic priority ceiling of semaphore S^*. In this case, task T is said to be blocked on semaphore S^* and to be blocked by the task that holds the lock on S^*. Otherwise, task T will obtain the lock on semaphore S and enter its critical section. When a task T exits its critical section, the binary semaphore associated with the critical section will be unlocked and the highest dynamic priority task, if any, blocked by task T will be awakened.*

2. *A task T uses its dynamic priority, as determined by the original characteristics of the task, unless it is in its critical section and blocks a task, T_H, with task characteristics, either initial or inherited, that would make T have a lower dynamic priority than T_H. If task T blocks a task that would normally be assigned a higher dynamic priority than T, T inherits T_H, the set of task characteristics of task T_H, either initial or inherited, that make T_H the highest priority task of the tasks blocked by T. This is equivalent to saying that T inherits the dynamic priority of the task T_H, the task with the highest dynamic priority blocked on S whose dynamic priority is greater than that of task T. When T exits a critical section, it resumes the set of task characteristics that it had at the point of entry into the critical section. That is, when T exits a critical section, it resumes its previous task characteristics that may not be its initial task characteristics. Dynamic priority inheritance is transitive. Finally, the operations of dynamic priority inheritance and of the resumption of previous dynamic priority must be atomic.*

3. *A task T, when it does not attempt to enter a critical section, can preempt another task T_L if its dynamic priority is higher than the dynamic priority at which task T_L is currently executing.*

As with the PCP, under the DPCP a high dynamic priority task may be blocked by a lower dynamic priority task in one of three situations. First, the high dynamic priority task may be directly blocked by the lower priority task; a high dynamic priority task attempts

to lock a locked semaphore. This *direct* blocking is necessary to ensure the consistency of shared data. Second, a medium dynamic priority task, T_m, may be blocked by a low dynamic priority task, T_l, that is executing at the dynamic priority of a higher dynamic priority task, T_h, that it is blocking. This *indirect* blocking is necessary to avoid having a high dynamic priority task, T_h, being indirectly preempted by the execution of a medium dynamic priority task, T_m. Third, a high dynamic priority task T_h may be blocked when it attempts to lock on a semaphore and a lower priority task T_l already has the lock on a semaphore that T_h may attempt to lock. This *ceiling* blocking is necessary to avoid deadlock and chained blocking.

In order to prove some properties about the DPCP, the following definitions and notation regarding semaphores and critical sections are made.

A binary semaphore guarding shared data and/or non-preemptive shared resources is denoted by S. The j $^{\text{th}}$ critical section of task T_i is denoted by s_{ij}. The semaphore that guards critical section s_{ij} is denoted by S_{ij}. The computation of critical section s_{ij} is denoted by $c_{s_{ij}}$.

Critical sections are assumed not to be nested, i.e., for any pair of critical sections s_{ij} and s_{ik}, $s_{ij} \cap s_{ik} = \emptyset$.

Definition 6.4.3 ([SRL90]) *A task T_i is said to be* blocked *by the critical section s_{kj} of task T_k if T_k has a lower priority than T_i but T_i has to wait for T_k to exit s_{kj} in order to continue execution.*

Definition 6.4.4 ([SRL90]) *A task T_i is said to be blocked by task T_k through semaphore S, if the critical section s_{kj} blocks T_i and $S_{kj} = S$.*

The schedulability of a task set is dependent upon the amount of blocking that may be experienced by the tasks in the task set. Since a dynamic priority scheduling algorithm is being used, it is possible that task T_i may be blocked by any task T_j such that $d_i < d_j$ that may lock a semaphore S that may also be locked by T_i. Let $\mathcal{B}_{ij} = \{s_{jk} | (s_{jk} \text{ can block } T_i) \wedge (\nexists s_{jm} \in \mathcal{B}_{ij} \text{ such that } s_{jk} \subset s_{jm})\}$ denote the set of non-nested critical sections of task T_j that can block T_i. The set $\mathcal{B}_{ij}$ contains the longest critical sections of T_j that can block T_i. Note, $\mathcal{B}_{ij} = \mathcal{B}_{ji}$. Let $\mathcal{B}_I = \bigcup_j \mathcal{B}_{ij}$ denote the set of non-nested critical sections of all tasks T_j such that $d_i < d_j$ that can ever block T_i.

Lemma 6.4.1 *A task T_h can be blocked by a lower dynamic priority task T_l only if T_l is executing within a critical section $s_{lj} \in \mathcal{B}_{hl}$ when T_h acquires the highest dynamic priority of all tasks.*

Proof By the definition of the dynamic priority ceiling protocol and $\mathcal{B}_{hl}$, T_l may block T_h only if it holds the semaphore upon which T_h is blocked or has its dynamic priority raised above that of T_h through priority inheritance or holds a semaphore in $\mathcal{B}_{hl}$. In any case, the critical section currently being executed by T_l is in $\mathcal{B}_{hl}$ since T_h is the highest dynamic priority task. If T_l is not within a critical section $s_{lj} \in \mathcal{B}_{hl}$, then T_l can be preempted by T_H and can never block T_h. ■

Lemma 6.4.2 ([CL90]) *A task can be blocked only before it enters its first critical section.*

Lemma 6.4.3 *A task T_h can be blocked by a lower dynamic priority task T_l only if the dynamic priority of task T_h is no higher than the highest dynamic priority ceiling of all the semaphores that are locked by all lower dynamic priority tasks when T_h becomes the highest dynamic priority task ready to execute.*

Proof Suppose that when T_h is invoked, the dynamic priority of task T_h is higher than the highest dynamic priority ceiling of all the semaphores that are currently locked by all lower dynamic priority tasks. Thus, by definition of the dynamic priority ceiling of a semaphore, T_h does not require any of the semaphores that are currently locked. Since T_h is the highest dynamic priority task ready to execute, it will execute until either it completes execution or a higher dynamic priority task T_H becomes ready to execute.

If T_h executes until completion, it will never be blocked since no task holds the lock on a semaphore that it requires and no lower dynamic priority task will execute before it completes. If T_H preempts T_h and never becomes blocked, T_h can never be blocked since all semaphores that were locked by T_H will be unlocked by T_H before T_h resumes execution. If T_H preempts T_h and then becomes blocked, the blocking task T_l will execute for at most the span of one critical section at the dynamic priority of T_H. After T_l exits its critical section it reverts to its previous dynamic priority.

If, at the time that T_H preempts T_h, T_h has already entered its first critical section, then by Lemma 6.4.2 T_h will not be blocked by any task. If, at the

time that T_H preempts T_h, T_h has not entered its first critical section, then until T_H completes its execution, T_h will not be the task with the highest dynamic priority. Thus, T_h will not execute until after T_H has completed execution. During the time between the invocation of T_H and the completion of T_H a lower priority task can only unlock a semaphore. This follows from the definition of dynamic priority ceiling protocol. Thus, no semaphores that may be required by T_h will be locked by a lower priority task, and T_h will never be blocked by a lower priority task. ■

Lemma 6.4.4 *Under the dynamic priority ceiling protocol a high dynamic priority task T_h can be blocked by a lower dynamic priority task T_l for at most the duration of one critical section of $\mathcal{B}_{hl}$ regardless of the number of semaphores T_h and T_l have in common.*

Proof By Lemma 6.4.1, for T_l to block T_h, T_l must be currently executing a critical section $s_{lj} \in \mathcal{B}_{hl}$. Once T_l exits s_{lj}, it can be preempted by T_h, and T_h cannot be blocked by T_l again. ■

Lemma 6.4.5 ([CL90]) *The priority inherited by a job cannot be higher than the highest priority ceiling of those critical sections the job has locked.*

Lemma 6.4.6 ([CL90]) *At any time, among the semaphores that are currently locked by tasks with non-inherited dynamic priorities lower than P, at most one semaphore has dynamic priority ceiling higher than or equal to P.*

Theorem 6.4.1 ([CL90]) *Chained blocking is impossible using the dynamic priority ceiling protocol.*

Theorem 6.4.2 ([CL90]) *The dynamic priority ceiling protocol prevents deadlocks.*

Chen and Lin derived a utilization based schedulability analysis test for the DPCP.

Theorem 6.4.3 ([CL90]) *A set of n periodic tasks can be scheduled by* EDF *using the dynamic priority ceiling protocol if the following condition is satisfied:*

$$\sum_{i=1}^{n} \frac{c_i + B_i}{p_i} \leq 1$$

where B_i is the duration of the longest critical section in $\mathcal{B}_I$.

6.5 Non-Preemptive Dynamic Priority Scheduling

The problem of scheduling a set of tasks non-preemptively on a single processor was studied by Jeffay *et.al.* [JSM91]. Jeffay *et.al.* were able establish necessary and sufficient conditions for the non-preemptive scheduling of a task set when the period is equal to the deadline. The following theorem establishes necessary conditions for the schedulability of a periodic task set with period equal to deadline.

Theorem 6.5.1 ([JSM91]) *Let $\tau_n = \{T_1, T_2, \ldots, T_n\}$ be a set of periodic tasks with arbitrary release times sorted in non-decreasing order by deadline. If τ_n is schedulable then*

1. $\sum_{i=1}^{n} \frac{c_i}{p_i} \leq 1$.
2. $\forall i, 1 < i \leq n; \forall L, p_1 < L < p_i;$
   $L \geq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor c_j$.

The first requirement states that the processor cannot be overloaded. In a single processor system, the sum of the utilization of all the tasks in the task set τ_n must be less than or equal to one (assuming there is no scheduling overhead).

The second requirement states that for a set of tasks to be schedulable, the demand for processor execution time in the interval L must always be less than or equal to the length of the interval. This is equivalent to saying that the processor may not be overloaded during any interval L. This requirement appears to be similar to the requirement that the total utilization of the processor not exceed one, but it can be shown that these two requirements are not related. It is possible to conceive of both schedulable task sets that have a processor utilization of one, and unschedulable task sets that have arbitrarily small processor utilization.

Requirements (1) and (2) from Theorem 6.5.1 are also necessary for scheduling a sporadic task set non-preemptively.

Corollary 6.5.1 ([JSM91]) *If a set of sporadic tasks $\tau_n = \{T_1, T_2, \ldots, T_n\}$, sorted in non-decreasing order by deadline, is schedulable, then τ_n satisfies requirements (1) and (2) from Theorem 6.5.1.*

Jeffay *et.al.* went on to demonstrate the existence of a non-preemptive scheduling algorithm that is guaranteed to schedule any periodic or sporadic task set that satisfies the

necessary conditions. The chosen algorithm is the non-preemptive EDF algorithm. In this formulation, the EDF algorithm assigns the ready task with the earliest deadline to the processor (ties broken arbitrarily). Once a task is assigned to the processor, the task is allowed to run to completion.

The following theorem demonstrates the universality of the non-preemptive EDF scheduling algorithm for sporadic task sets. That is, if any non-preemptive scheduling algorithm schedules a set of sporadic tasks, then the non-preemptive EDF scheduling algorithm will schedule the same set of tasks as well. To prove universality, it suffices to show that requirements (1) and (2) from Theorem 6.5.1 are sufficient to ensure that the non-preemptive EDF scheduling algorithm schedules any set of sporadic tasks with specified release times.

Theorem 6.5.2 ([JSM91]) *Let $\tau_n = \{T_1, T_2, \ldots, T_n\}$ be a set of sporadic tasks sorted in non-decreasing order by deadline. If τ_n satisfies requirements (1) and (2) from Theorem 6.5.1, then the non-preemptive* EDF *scheduling algorithm will schedule τ_n.*

The following corollary shows that the non-preemptive EDF scheduling algorithm is universal for scheduling periodic tasks.

Corollary 6.5.2 ([JSM91]) *Let $\tau_n = \{T_1, T_2, \ldots, T_n\}$ be a set of periodic tasks sorted in non-decreasing oder by deadline. If τ_n satisfies requirements (1) and (2) from Theorem 6.5.1, then the non-preemptive* EDF *scheduling algorithm will schedule τ_n.*

It can also be shown that the non-preemptive MLF scheduling algorithm is universal for sporadic and periodic task sets. To prove universality, it suffices to show that requirements (1) and (2) from Theorem 6.5.1 are sufficient to ensure that the non-preemptive MLF scheduling algorithm schedules any set of sporadic tasks with specified release times.

For following definition is used for the proof of Theorem 6.5.3.

Definition 6.5.1 *The* laxity deadline *of a task, T_i, is equal to the deadline of the task, d_i, minus the computation time of the task, c_i.*

Theorem 6.5.3 *Let $\tau_n = \{T_1, T_2, \ldots, T_n\}$ be a set of sporadic tasks sorted in non-decreasing order by deadline. If τ_n satisfies requirements (1) and (2) from Theorem 6.5.1, then the non-preemptive* MLF *scheduling algorithm will schedule τ_n.*

Proof By contradiction. Assume that τ_n satisfies conditions (1) and (2) from Theorem 6.5.1, and there exists an assignment of release times such that a task misses a deadline at some point in time when τ_n is scheduled by the non-preemptive MLF scheduling algorithm. The proof proceeds by deriving upper bounds on the processor demand for an interval ending at the time at which a task misses a laxity deadline.

Let t_d be the earliest time at which a laxity deadline is missed. τ_n can be partitioned into three disjoint subsets:

$\mathcal{S}_\infty$ = the set of tasks that have an invocation with a laxity deadline at time t_d.

$\mathcal{S}_\in$ = the set of tasks that have an invocation occurring prior to time t_d with a laxity deadline after t_d.

$\mathcal{S}_\ni$ = the set of tasks not in $\mathcal{S}_\infty$ or $\mathcal{S}_\in$.

To bound the processor demand prior to t_d it suffices to examine only $\mathcal{S}_\in$. Let $r_1, r_2, \cdots, r_k$ be the invocation times of the tasks in $\mathcal{S}_\in$. There are two cases to consider.

Case 1: None of the invocations of tasks in $\mathcal{S}_\in$ occurring at times $r_1, r_2, \cdots, r_k$ are scheduled prior to t_d.

Let t_0 be the end of the last interval prior to t_d in which the processor was idle. If the processor has never been idle let $t_0 = 0$. In the interval $[t_0, t_d]$, the processor demand, $D_{[t_0,t_d]}$, is the total computation requirement of the tasks that are invoked at or after t_0 with laxity deadline at or before time t_d. This yields

$$D_{[t_0,t_d]} \leq \sum_{j=1}^{n} \left\lfloor \frac{t_d - t_0}{p_j} \right\rfloor c_j.$$

Since there is no interval during which the processor is idle in the interval $[t_0, t_d]$ and since a task misses a deadline at t_d, it follows that $D_{[t_0,t_d]} > t_d - t_0$. Thus,

$$t_d - t_0 < \sum_{j=1}^{n} \left\lfloor \frac{t_d - t_0}{p_j} \right\rfloor c_j \leq \sum_{j=1}^{n} \frac{t_d - t_0}{p_j} c_j$$

which yields the following relation

$$1 < \sum_{j=1}^{n} \frac{c_j}{p_j}.$$

However, this contradicts the assumption that condition (1) is met.

Case 2: Some of the invocations of tasks in $\mathcal{S}_\in$ occurring at times $r_1, r_2, \cdots, r_k$ are scheduled prior to t_d.

Let T_i, allocated the processor at time $t_i < t_d$, be the last task in $\mathcal{S}_\in$ scheduled prior to time t_d. Note that if the processor is ever idle during the interval $[t_i, t_d]$, then the analysis of *Case 1* can be used where $t_i < t_0 < t_d$. Therefore, assume that the processor is never idle during the interval $[t_i, t_d]$. Let T_k be a task that misses a laxity deadline at time t_d. Due to the choice of T_i and the use of the MLF scheduling algorithm, if follows that $t_i < t_d - p_k + c_k$. It also follows that every task other than task T_i executed in $[t_i, t_d]$ must have a laxity deadline at or before t_d due to the MLF algorithm. Therefore, other than task T_i, no task that is scheduled in $[t_i, t_d]$ could have been invoked at time t_i. If such a task exists, then that task would have been allocated the processor at time t_i and not task T_i.

Two cases arise depending upon the computation time of task T_i.

Case 2.1: The computation time of task T_i, c_i, is less than the computation time of task T_k, c_k, $c_i < c_k$.

The relationship between the periods of task T_i, p_i, and the period of task T_k, p_k, is unknown. Because of this, the processor demand during the interval $[t_i, t_d + c_k]$ must be calculated by considering all tasks in the task set.

In the interval $[t_i, t_d + c_k]$, the processor demand, $D_{[t_i, t_d + c_k]}$, is the total computation requirement of the tasks that are invoked at or after time t_i with a deadline at or before time $T_d + c_k$. This yields

$$D_{[t_i, t_d + c_k]} \leq \sum_{j=1}^{n} \left\lfloor \frac{t_d + c_k - t_i}{p_j} \right\rfloor c_j.$$

Since there is no interval during which the processor is idle in the interval $[t_i, t_d + c_k]$ (this is easily seen since the processor is not idle in

the interval $[t_i, t_d]$ and T_k misses its laxity deadline at time t_d, i.e., T_k is not allocated the processor before time t_d, T_k may be allocated the processor after t_d ensuring that there is no idle time in the interval $[t_i, t_d + c_k]$) and since task T_k will miss its deadline at time $t_d + c_k$ (this follows from T_k missing its laxity deadline at time t_d), it follows that $D_{[t_i, t_d+c_k]} > t_d + c_k - t_i$. Thus,

$$t_d + c_k - t_i < \sum_{j=1}^{n} \left\lfloor \frac{t_d + c_k - t_i}{p_j} \right\rfloor c_j \leq \sum_{j=1}^{n} \frac{t_d + c_k - t_i}{p_j} c_j$$

which yields the following relation

$$1 < \sum_{j=1}^{n} \frac{c_j}{p_j}.$$

This contradicts the assumption that condition (1) is met.

Case 2.2: The computation time of task T_i, c_i, is greater than or equal to the computation time of task T_k, c_k, $c_i \geq c_k$.

The period of task T_i is larger than the period of task T_k.

Since $p_i > t_d + c_k - t_i$ only tasks $T_1 \cdots T_i$ need to be considered in computing the processor demand during $[t_i, t_d + c_k]$, $D_{[t_i, t_d+c_k]}$. Since the invocation of task T_i that is scheduled at time t_i has a laxity deadline after time t_d, all task invocations occurring prior to time t_i with laxity deadlines at or before t_d must have been completed by time t_i; thus, they do not contribute to $D_{[t_i, t_d]}$. Since none of the invocations of tasks $T_1 \cdots T_{i-1}$ that are scheduled in the interval $[t_i, t_d + c_k]$ occurred at or before time t_i, the demand due to these tasks during $[t_i, t_d + c_k]$ is the same as the demand during the interval $[t_i + 1, t_d + c_k]$.

These observations, plus the fact that the invocation of task T_i scheduled at time t_i must be completed before time t_d, indicate that the processor demand in $[t_i, t_d + c_k]$ is bounded by

$$D_{[t_i, t_d+c_k]} \leq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{t_d + c_k - (t_i + 1)}{p_j} \right\rfloor c_j.$$

Since there is no idle time in $[t_i, t_d + c_k]$ and a task missed a laxity deadline at t_d, it follows that $D_{[t_i, t_d+c_k]} > t_d + c_k - t_i$.

Let $L = t_d - t_i + c_k$. Substituting L into the above equation yields

$$D_{[t_i, t_d + c_k]} \leq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor c_j.$$

Since a task misses its deadline at time $t_d + c_k$, $D_{[t_i, t_d + c_k]} > t_d + c_k - t_i$. Combining this with the above equation yields

$$L < D_{[t_i, t_d]} \leq c_i + \sum_{j=1}^{i-1} \left\lfloor \frac{L-1}{p_j} \right\rfloor c_j.$$

Since $p_i > t_d + c_k - t_i = L$ and $t_i < t_d - p_k + c_k$, $t_d + c_k - t_i > p_k \geq p_1$; thus $L > p_1$. Therefore, condition (2) is violated. ■

6.6 Sporadic Task Scheduling in Dynamic Priority Systems

Only periodic task sets have been addressed for dynamic priority systems to this point. Due to the existence of sporadic tasks in real-world applications, dynamic priority scheduling algorithms must be able to correctly handle sporadic tasks.

Hard sporadic tasks are easily handled by both dynamic best effort scheduling algorithms and dynamic planning based scheduling algorithms. Dynamic best effort scheduling algorithms schedule hard sporadic tasks as if they were hard periodic tasks. Thus, neither the EDF nor the MLF algorithms needs to be modified to handle sporadic tasks with hard deadlines. However, the given schedulability analysis tests for these algorithms are unable to handle sporadic tasks. Specifically, simple utilization based tests do not account for the possibility of transient overload.

Critical instant-based schedulability analysis tests must be used when sporadic tasks are present. The following theorem establishes necessary conditions for schedulability for a task set that contains sporadic tasks.

Theorem 6.6.1 *Let $\tau_n = \{T_1, \ldots T_n\}$, arranged in non-decreasing order by deadline, d, be a set of sporadic and periodic tasks. If τ_n is schedulable, then*

1. $\sum_{j=1}^{n} \frac{c_j}{p_j} \leq 1$.
2. $\forall i, 1 \leq i \leq n; \forall L, d_1 \leq L \leq d_i$

$$L \geq \sum_{k=1}^{i} \frac{L}{p_k} c_k.$$

Proof The contrapositive of the theorem is proven: if a set of tasks τ_n does not satisfy condition (1) or condition (2), then there exists a task set, τ_{nr}, with specified release times, generated from τ_n, that is not schedulable.

The worst case processor utilization of a sporadic task occurs when it acts like a periodic task.

Consider the set of tasks $\tau_{nr} = \{T_1, \ldots, T_n\}$ where each task has a specified release time of $r = 0$. Let $t = p_1 \cdot p_2 \cdot \ldots \cdot p_n$. In the interval $[0, t]$, task i must receive $(t/p_i)c_i$ units of processor time to ensure it does not miss a deadline in the interval. Therefore, the total processor demand, D, for all tasks during the interval $[0, t]$ is

$$D = \sum_{j=1}^{n} \frac{t}{p_j} c_j.$$

Dividing this equation by t yields

$$\frac{D}{t} = \sum_{j=1}^{n} \frac{c_j}{p_j}.$$

If condition (1) does not hold, then $D > t$, and hence τ_{nr} is not schedulable.

For condition (2), consider the set of tasks $\tau_{nr} = \{T_1, \ldots, T_n\}$ where each task has a specified release time of $r = 0$. Consider the interval L, where $d_1 \leq L \leq d_i$. The processor demand, D, in the interval $[0, L]$ is given by

$$D = \sum_{j=1}^{i} \frac{L}{p_j} c_j.$$

The demand consists of the processor demand due to tasks that will have a deadline within $[0, L]$. Note that tasks with deadlines greater than d_i have no invocations with deadlines in the interval $[0, L]$, and hence do not contribute to the processor demand in the interval $[0, L]$.

If condition (2) does not hold, then $D > L$, and hence τ_{nr} is not schedulable. ■

Dynamic planning based scheduling algorithms handle hard sporadic tasks. The off-line schedulability analysis tests are the same as for the dynamic best effort scheduling algorithms. If the dynamic planning based scheduling algorithm uses knowledge about

future invocations of a task in deciding schedulability at run-time, then the worst case processor demand and the worst case task phasings for the sporadic tasks must be assumed.

Soft sporadic tasks may be handled by use of a sporadic server task. Spuri and Buttazzo [SB94] present five sporadic server algorithms for use with EDF. The Dynamic Priority Exchange (DPP), Dynamic Sporadic Server (DSS), Total Bandwidth (TB), EDL, and Improved Priority Exchange (IPE) algorithms minimize the average response times of the soft sporadic tasks.

6.7 Advantages of the Dynamic Priority Approach

The main advantage of the dynamic priority approach is its ability to obtain a high processor utilization. When arbitrary preemption is allowed, dynamic priority scheduling algorithms may obtain a processor utilization of one for any task set satisfying the assumptions of the scheduling algorithm. Static priority scheduling algorithms may obtain this utilization only when the periods of the tasks are harmonics.

The major characteristics of the tasks do not need to be known in advance. Only the information required to determine priorities (and perform a schedulability analysis test if a dynamic planning based algorithm is used) needs to be known, and this information may be provided by the task when it is invoked.

Dynamic priority algorithms are flexible. They can adapt to changes in their environment, requiring no special processing for mode changes or changes in the task set.

Utilization based schedulability analysis tests may be used to determine *a priori* whether a task set is schedulable by the dynamic priority approach. These same tests may also be used by the dynamic planning based approach.

Sporadic tasks are easily handled by the dynamic priority approach. This is important especially when the sporadic tasks have hard deadlines. Sporadic servers may be used to give soft sporadic tasks a low average response time.

Task synchronization may be accomplished by use of the Dynamic Priority Ceiling Protocol that guarantees deadlock avoidance and prevents unpredictable delays at run time.

6.8 Disadvantages and Other Issues of the Dynamic Priority Approach

In order to obtain a high processor utilization the dynamic priority approach must be able to handle multiple levels of preemption efficiently. The physical characteristics of the system can limit the number of nested interrupts that are allowed, e.g., the stack size and the way it is used can limit the number of nested interrupts. Preemption causes the scheduler to have a high overhead, due mainly to context switches, and causes unpredictable delay in task execution.

Dynamic best effort scheduling algorithms have a limited knowledge of the task characteristics of the active tasks in the system. In addition, they do not have information on the future invocations of tasks, increasing the chances that a valid schedule will not be found.

> If the scheduler does not have [any knowledge about the major characteristics of tasks that have not yet arrived in the system] then it is impossible to guarantee that all timing constraints will be satisfied, because no matter how clever the scheduling algorithm is, there is always the possibility that a newly arrived task possesses characteristics that will make that task either miss its own deadline, or cause other tasks to miss their deadlines. This is true even if the processor capacity was sufficient for the task at hand. [XP90]

As with the static priority approach, the dynamic priority scheduling approach sometimes has difficulty handling practical problems. Task synchronization is possible, but the run time task synchronization mechanism must be conservative to avoid deadlock. This results in situations where a task is blocked by the run time synchronization mechanism even though it could have proceeded without causing deadlock, reducing the level of processor utilization.

The dynamic priority approach does not directly address the problem of distance constraints. The only possible way to guarantee distance constraints is to give those tasks with distance constraints a high priority. Since the designer does not have direct control over the run-time priority of a task, the pertinent task characteristics must be modified to ensure that distance constraints will be met.

6.9 Implementation of the Dynamic Priority Approach Within POLIS

6.9.1 Routines Implemented Within POLIS

The preemptive and non-preemptive EDF and MLF best-effort scheduling algorithms are implemented within POLIS. Sporadic servers are not implemented due to their large code. Consequently, soft sporadic tasks are relegated to background service.

The schedulability of the task set is guaranteed off-line. For purely periodic task sets, the preemptive implementation uses the utilization-based test presented in Theorem 6.2.2. For the non-preemptive implementation, the task set is verified to meet the conditions in Theorem 6.5.1.

When sporadic tasks are present, all tasks are verified to meet their deadlines when invoked at their critical instant, assuming all tasks are independent. Any tasks that are not guaranteed to meet their deadlines are identified, providing useful feedback to the designer.

6.9.2 Generated Scheduling Routines

The deadlines (and computation times for MLF) of all tasks in the system are stored in a table. A table containing all soft tasks is also present. The scheduling routines refer to these tables when determining the deadline times and laxities of tasks as well as identifying soft tasks.

The dynamic schedulers generated by POLIS rely upon interrupts to perform correctly. It is only with the use of interrupts that deadline times and laxities may be correctly determined. Knowing the invocation time of a task is critical to determining its deadline time and laxity.

For the non-preemptive scheduler shown in Figure 6.1, the interrupt handling routine identifies all invoked tasks and places them on a ready queue in dynamic priority order. The main scheduling routine non-preemptively executes the highest priority task on the ready queue.

For the preemptive scheduler shown in Figure 6.2, the interrupt handling routine performs all functions of the scheduler. All invoked tasks are identified and placed on a ready queue. The highest priority task in the ready queue is then executed. The interrupt routine exits when the highest priority task is the task that was interrupted. This interrupt

handling implementation prevents the need for complex context switching software, reducing the memory requirements of the scheduler.

```
// main scheduling routine that executes the highest priority task
scheduler()
{
   while( 1 ) {
      execute( highest_priority_task() );
   }
}

// interrupt handling routine
interrupt_handler()
{
   poll_inputs_and_update_input_buffers();
   // find the ready tasks and put them on the ready list
   for( task = 0; task < NUMBER_TASKS; task++ ) {
      if( is_ready( task ) ) {
         if( not_on_ready_list( task ) ) {
            place_on_read_list_in_order( task );
         }
      }
   }
   return; // return from interrupt
}
```

Figure 6.1: Generated non-preemptive dynamic priority scheduling routines.

```
// main scheduling routine that should never execute a task
// it is used as a safety net
scheduler()
{
   while( 1 ) {
      execute( current_task = highest_priority_task() );
      current_task = NUMBER_TASKS;
   }
}

// interrupt handling routine; all tasks should be executed from
// this routine
interrupt_handler()
{
   interrupted_task = current_task;
   poll_inputs_and_update_input_buffers();

   // find the ready tasks and put them on the ready list
   for( task = 0; task < NUMBER_TASKS; task++ ) {
      if( is_ready( task ) ) {
         if( not_on_ready_list( task ) ) {
            place_on_read_list_in_order( task );
         }
      }
   }
   place_on_ready_list_in_order( current_task );
   while( ( current_task = highest_priority_task() ) != interrupted_task )
      execute( current_task );
   return; // return from interrupt
}
```

Figure 6.2: Generated preemptive dynamic priority scheduling routines.

Chapter 7

Results

7.1 On-Line Scheduling Overhead

7.1.1 Derived Bounds for the On-Line Scheduling Overhead

Bounds on the execution cycles for each of the generated scheduling algorithms were determined via hand simulation. The calculated execution times do not take into account context switching overhead or interrupt latency time, i.e., time to react to an interrupt. The execution cycles of blocks of code were determined using the cycle time estimator within POLIS. The cycle time estimator calculates the number of cycles required to execute the given block of code on the Motorola HC11 microcontroller with a maximum error of 20%.

By simulating the code, upper and lower bounds on the number of cycles required by the scheduler were obtained. Table 7.1 presents the derived bounds for the presented scheduling algorithms that do not use interrupts, Round Robin, Pre-Run-Time, and Non-Preemptive Static Priority.

Table 7.2 presents the derived bounds for the presented scheduling algorithms that use interrupts, Preemptive Static Priority, Non-Preemptive Dynamic Priority, and Preemptive Dynamic Priority. Upper and lower bounds on the execution cycles of the interrupt handling routines are also presented.

The cycle timing dependence upon the number of events in the system, N_E, arises from the method in which events are detected. When the occurrence of an event is checked for, all events are checked. 135 clock cycles are required to check for the presence of an

	Minimum Execution (cycles)	Maximum Execution (cycles)
Round Robin	$80 + 135 * N_E$	$80 + 260 * N_E + 80 * N_T$
Pre-Run-Time	$70 + 135 * N_E$	$70 + 260 * N_E$
Non-Preemptive Static Priority	$150 + 135 * N_E$	$40 + 260 * N_E + 160 * N_T$

Table 7.1: Range of possible execution cycles for non-interrupt scheduling routines synthesized by POLIS with $N_E \equiv$ number of events in the system and $N_T \equiv$ number of tasks in the system.

	Minimum Interrupt (cycles)	Maximum Interrupt (cycles)	Minimum Execution (cycles)	Maximum Execution (cycles)
Preemptive Static Priority	$360 + 135 * N_E$	$285 + 135 * N_E$ $+75 * \text{Priority}_T$	150	$285 + 135 * N_E$ $+75 * \text{Priority}_T$
Non-Preemptive Dynamic Priority	$460 + 135 * N_E$	$300 + 135 * N_E$ $+160 * N_T$	80	$300 + 135 * N_E$ $+160 * N_T$
Preemptive Dynamic Priority	$265 + 135 * N_E$ $+80 * N_T$	$265 + 135 * N_E$ $+240 * N_T$	80	$265 + 135 * N_E$ $+240 * N_T$

Table 7.2: Range of possible execution cycles for interrupt scheduling routines synthesized by POLIS with $N_E \equiv$ number of events in the system, $N_T \equiv$ number of tasks in the system, and $\text{Priority}_T \equiv$ the priority of the interrupted task, T.

event, and an additional 125 clock cycles are required to update the appropriate input buffers if an event is present.

The cycle timing dependences upon the number of tasks in the system, N_T, and the priority of the interrupted task, Priority_T, arise from the order in which tasks are checked to find the highest priority task that is ready to run. In the worst-case, all tasks must be checked before the highest priority task that is found.

7.1.2 Average On-Line Scheduling Overhead

The derived bounds for the on-line scheduling overhead allow for a pessimistic worst-case analysis, guaranteeing the predictability of the system. However, this worst-case analysis does not provide any insight into what the average scheduling overhead might be. The average scheduling overhead is of interest since it provides information as to how pessimistic an estimate the worst-case scheduling overhead is in the typical case.

To determine an average scheduling overhead per executed task, various task sets consisting of independent periodic tasks were simulated through 100000 cycles (where one cycle corresponds to a time period equal to the least common multiple of the task periods) for each of the scheduling algorithms. The total execution time of the scheduling routines was divided by the total number of tasks executed during the simulation to arrive at the scheduling overhead per executed task (*scheduling overhead*). This overhead does not include context switching time or interrupt latency.

Figures 7.1, 7.2, 7.3, 7.4, 7.5, and 7.6 show the observed average scheduling overhead for the Round Robin, Pre-Run-Time, Non-Preemptive Static Priority, Preemptive Static Priority, Non-Preemptive Dynamic Priority, and Preemptive Dynamic Priority scheduling algorithms respectively. The observed average scheduling overhead is not representative of all task sets or even all periodic task sets. The observed scheduling overheads only indicate what the average scheduling overhead might be for a 'typical' task set.

All scheduling algorithms exhibit a strong relation between scheduling overhead and number of events in the system. This is expected because of the event detection routine used in all implementations.

Figures 7.7 and 7.8 show the relationship between the average scheduling overhead and the worst-case scheduling overhead as a function of the number of tasks in the system for a fixed number of events. As the figures show, the larger the number of tasks in the

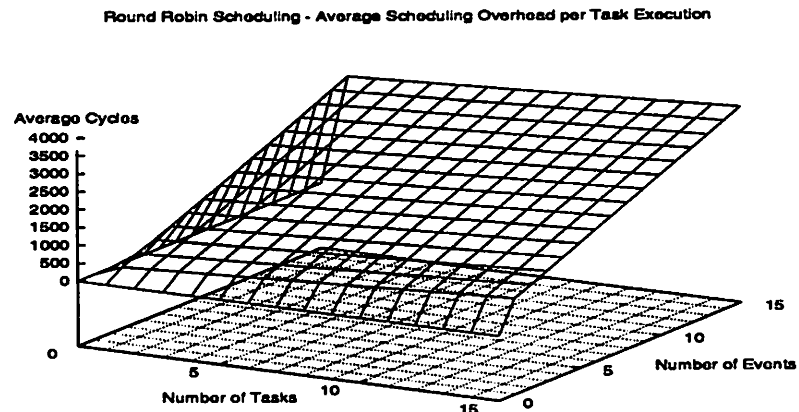


Figure 7.1: Average Round Robin scheduling overhead.

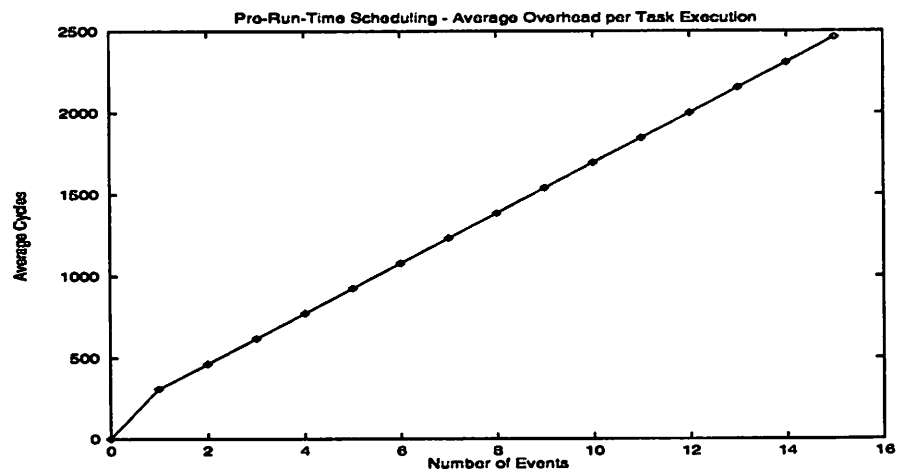


Figure 7.2: Average Pre-Run-Time scheduling overhead.

Non-Preemptive Static Priority Scheduling - Average Scheduling Overhead per Task Execution

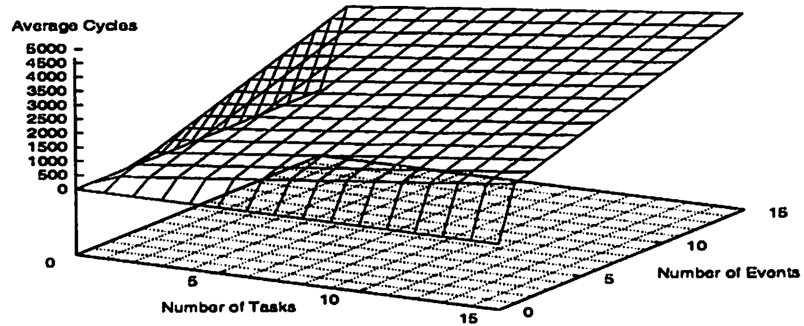


Figure 7.3: Average Non-Preemptive Static Priority scheduling overhead.

Preemptive Static Priority Scheduling - Average Scheduling Overhead per Task Execution

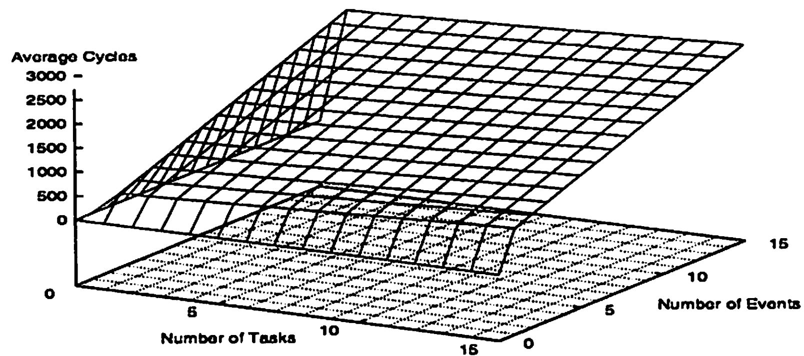


Figure 7.4: Average Preemptive Static Priority scheduling overhead.

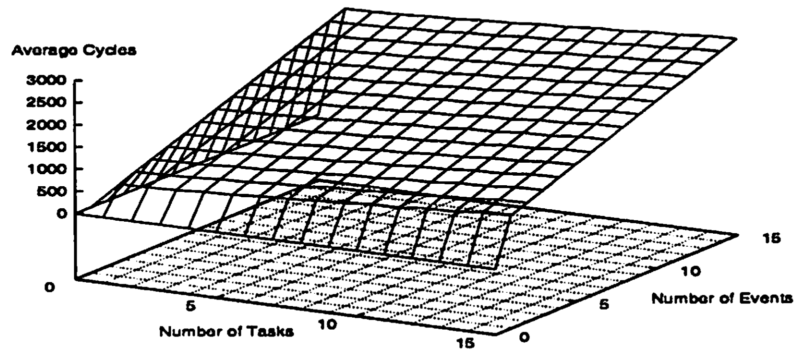


Figure 7.5: Average Non-Preemptive Dynamic Priority scheduling overhead.

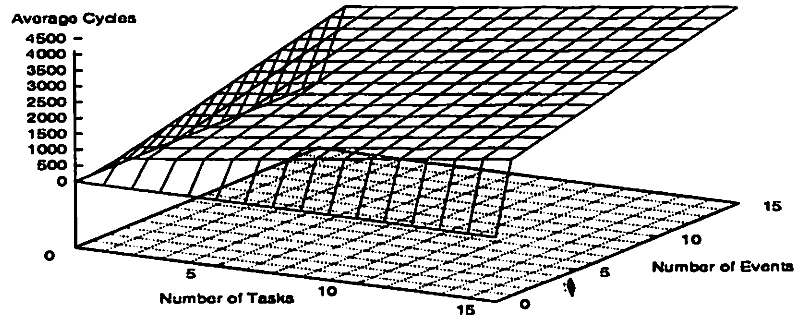


Figure 7.6: Average Preemptive Dynamic Priority scheduling overhead.

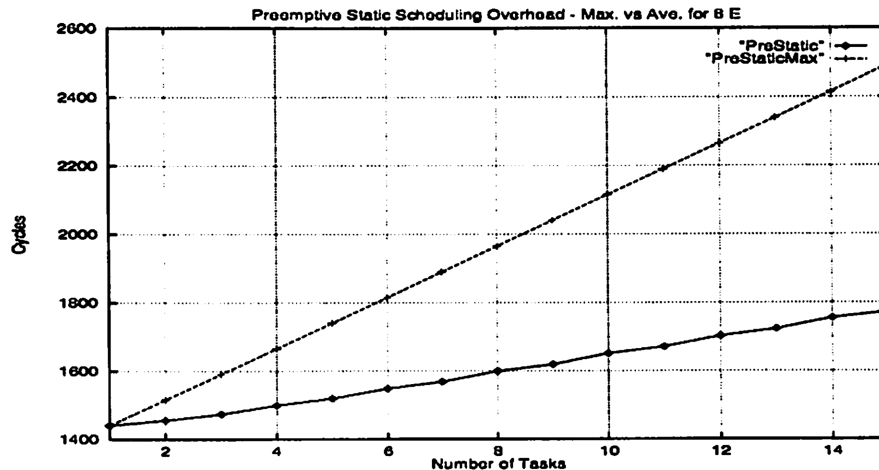


Figure 7.7: Average Preemptive Static Priority scheduling overhead compared with the maximum possible scheduling overhead.

system, the more pessimistic the worst-case scheduling overhead is when compared with the average scheduling overhead. This relationship is evident in all of the scheduling algorithms except the Pre-Run-Time scheduling algorithm; the Pre-Run-Time scheduling overhead is independent of the number of tasks in the system.

Figure 7.9 compares the average scheduling overhead for each of the scheduling algorithms as a function of number of events in the system for a task set size of eight.

Figure 7.10 compares the average scheduling overhead for all scheduling algorithms, except for the Pre-Run-Time scheduling algorithm, as a function of the number of tasks in the system with eight events in the system.

Figures 7.9 and 7.10 show that the PRT scheduling implementation has the lowest average scheduling overhead of all the scheduling algorithms (for eight events, the PRT scheduling implementation has an average overhead of 1386 cycles). However, it is clear that the PRT implementation is not optimal. An optimal PRT implementation would have an almost constant overhead, with fluctuations due only to variations in the number of tasks dependent upon an input event and the number of input events upon which a task depends. The current implementation checks all input events, not just the input events the next task in the schedule depends upon, causing the scheduling overhead to be larger than it could be.

The Non-Preemptive Static Priority scheduling routines have the largest average scheduling overhead. The Preemptive Static Priority scheduling routines have a much lower

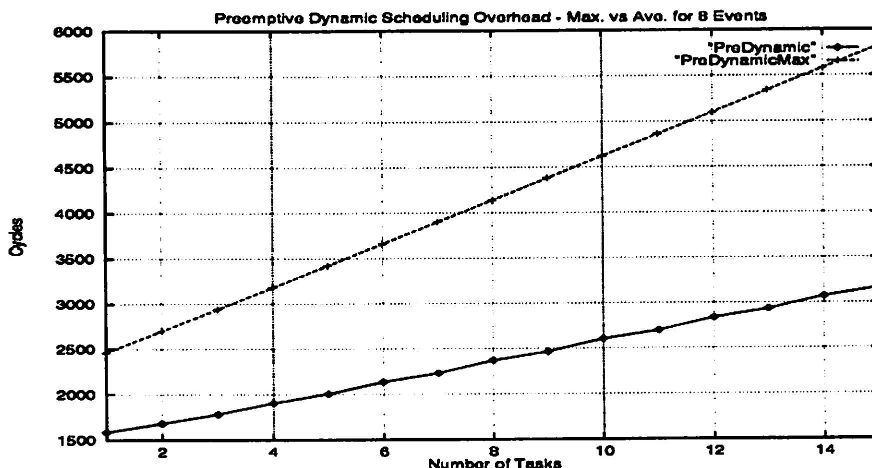


Figure 7.8: Average Preemptive Dynamic Priority scheduling overhead compared with the maximum possible scheduling overhead.

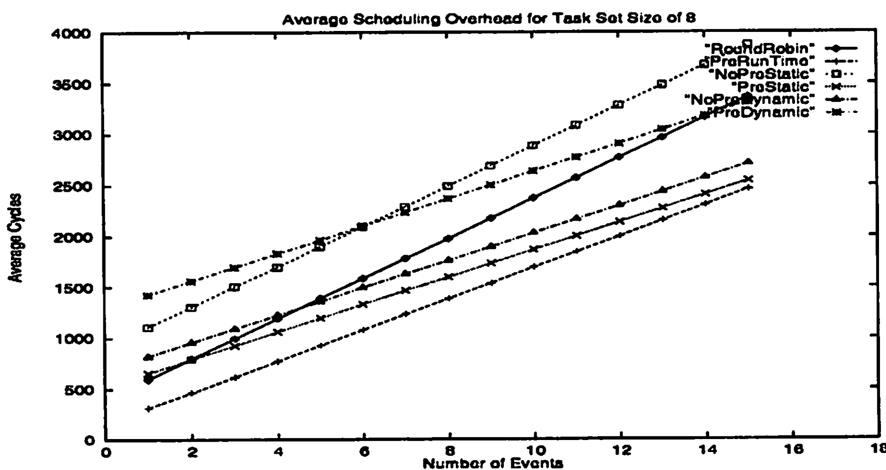


Figure 7.9: Comparison of all scheduling implementations as a function of the number of events in the system for a fixed task set size.

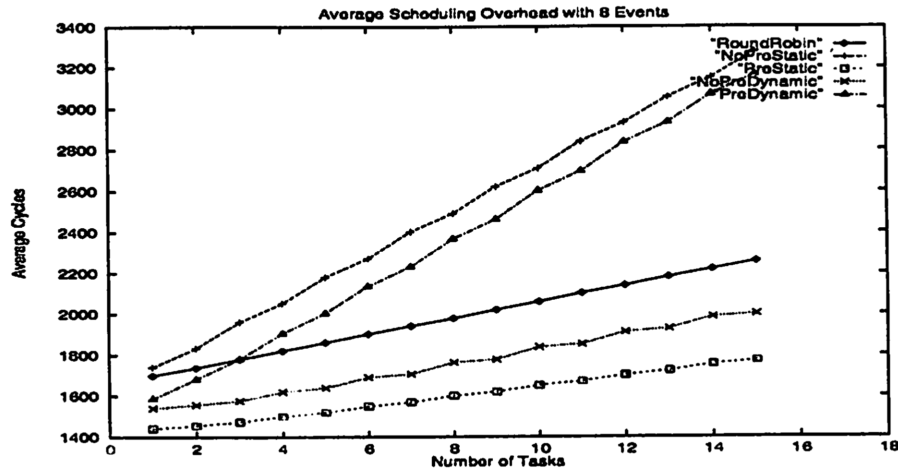


Figure 7.10: Comparison of all scheduling implementations, except for Pre-Run-Time, as a function of the number of tasks in the system for a fixed number of events.

average scheduling overhead (the second lowest average scheduling overhead); suggesting that the Non-Preemptive Static Priority average scheduling overhead could be decreased by using interrupts to detect events instead of polling.

The Non-Preemptive Dynamic Priority scheduling routines have a relatively low average scheduling overhead; further evidence that the use of interrupts would decrease the average scheduling overhead for the Non-Preemptive Static Priority scheduling routines. The Preemptive Dynamic Priority scheduling routines have an expectedly high average scheduling overhead.

7.1.3 Comparison with an Existing Real-Time Operating System

Many commercial Real-Time Operating Systems (RTOSs) require the designer to write interrupt routines causing the RTOS to have a variable amount of scheduling overhead. In these instances, the context switching time, the interrupt latency, and the maximum execution timings of pertinent, highly optimized, routines are published.

The pSOS+ multitasking operating system from Integrated Systems, Inc. is typical of many commercial RTOSs. It employs a preemptive static priority scheduler. The designer is required to develop the interrupt handling routines using the provided operating system calls.

For the Intel 486DX2 33MHz processor with 256KB of cache, pSOS+ has a

Procedure	Execution Time (μs)	Estimated Cycles
T_START	8.36	552
T_RESTART	13.00	858
T_SUSPEND	5.80	384
T_RESUME	6.07	400
T_SETPRI	7.74	512
T_MODE	3.15	208
T_SETREG	3.33	220
T_GETREG	3.36	220
I_ENTER	1.88	124
I_RETURN	1.73	114
TIME_GET	3.03	200

Table 7.3: Execution times for some standard routines in the pSOS+ real-time operating system for the Intel 486DX2 33MHz processor.

claimed[1] interrupt latency of less than 4.00μs and a claimed context switching time of 6.27μs. Table 7.3 shows the claimed execution times, both in μs and estimated clock cycles, for some of the operating system calls available to tasks running under pSOS+.

A typical interrupt handling routine might read the contents of some register, update another register based on the contents of the first register, and then return from the interrupt. At the least, this interrupt routine would require 554 cycles using pSOS+. If the interrupt handling routine performs any scheduling activities, such as updating a ready task list, the total interrupt time would increase further. In addition to this time, the static priority scheduling routines within pSOS+ must be executed after the interrupt returns, further increasing the run-time overhead.

In comparison, for the preemptive static priority scheduling routines synthesized by POLIS, the interrupt routine identifies the ready tasks in the system and begins execution of the highest priority task; performing both interrupt handling and scheduling. This requires a minimum of 495 cycles to perform and increases as the number of events in the system increases, due to the input event update routine.

[1] All timing data were obtained from `http://www.isi.com/Products/pSOS/386.html`.

Scheduling Algorithm	RAM		ROM	
	data	bss	const	text
RR	100	78	0	944
PRT	100	78	6	969
NoPreStatic	100	78	6	981
PreStatic	100	78	6	1048
NoPreDynamic	118	84	12	1511
PreDynamic	118	84	12	1665

Table 7.4: Measured memory requirements (in bytes) of the synthesized POLIS operating system utilizing specific scheduling routines for a task set of size three with eight events.

7.2 Synthesized Operating System Memory Requirements

The size (in bytes) of the operating system is important to the target application of POLIS, small embedded controllers. These applications typically have a limited amount of memory for code storage. (Typical embedded systems contain a fixed amount of Read-Only-Memory (ROM) in which the code for the operating system and the application tasks is stored.) Therefore, the operating system should be as small as possible to allow for it and the tasks to fit into the available memory.

The complete operating system synthesized by POLIS consists of the scheduler and I/O routines. The operating system may contain information, required by the scheduler, on each task (e.g., task priority, task deadlines, task periods), causing the size of the operating system to be dependent upon the number of tasks in the system.

Table 7.4 shows the memory requirements of the generated operating system for a task set size of three with eight events. The larger operating systems in terms of RAM and ROM usage are those which utilize interrupts.

Table 7.5 shows the memory requirements of the generated operating systems for the shock absorber controller example described in [CEG+95]. This example contains fourty-eight (48) tasks and eighty (80) events.

7.2.1 Comparison With Existing Real-Time Operating Systems

Table 7.6 shows the minimum memory (ROM) requirements for various commercial and research RTOSs. All of these RTOSs utilize a static priority scheduling methodology.

Scheduling Algorithm	RAM		ROM	
	data	bss	const	text
RR	4762	1161	488	3380
PRT	4762	1161	584	3405
NoPreStatic	4762	1161	584	3417
PreStatic	4762	1161	584	3484
NoPreDynamic	5050	1167	680	3947
PreDynamic	5050	1167	680	4101

Table 7.5: Measured memory requirements (in bytes) of the synthesized POLIS operating system utilizing specific scheduling routines for a task set of size fourty-eight (48) with eighty (80) events.

Operating System	Min Size (kB)
HI68K [TSiH87]	24
REALOS/286 [Shi87]	13.5
Maruti [SdA93]	14
pSOS (Integrated Systems Inc.)	15
RTEMS 3.1.0 (U.S. Military)	11.6
QNX 4.21 (QNX Software Systems)	10

Table 7.6: Real-Time Operating Systems' memory (ROM) requirements.

They contain many features which are not present in the operating systems synthesized by POLIS, e.g., semaphores. The price for these additional features, and the generality required to allow their use in many different systems, is additional memory usage.

In the shock absorber example above, the synthesized operating system, regardless of the scheduling method used, is smaller than any of these more general RTOSs. The application specific nature of the synthesized operating system in POLIS allows for the RTOS to use a minimal amount of memory (ROM) while guaranteeing the schedulability of the task set; an advantage when the amount of memory in a system is limited.

Chapter 8

Conclusions and Future Work

Multiple real-time task level scheduling algorithms and their associated schedulability analysis tests have been implemented within the POLIS co-design environment. The theory behind each of these algorithms was presented, and extensions to this theory were made.

- The Laxity Monotonic static priority scheduling algorithm (LMS) was presented, and its optimality proven.
- The notion of a critical instant for a non-preemptive static priority scheduling algorithm was presented, and its validity proven.
- Two sufficient non-preemptive static priority schedulability analysis tests were developed.
- The optimality of the non-preemptive dynamic priority Minimum Laxity First scheduling algorithm was proven.
- Necessary conditions for preemptive dynamic priority scheduling of sporadic task sets were derived.

The generated scheduling routines were found to have a much larger scheduling overhead than commercial and other research RTOSs due to the event detection mechanism; however, the generated scheduling routines are considerably smaller in size (measured by ROM usage) than these RTOSs.

There are many future directions for the run-time schedulers generated by POLIS. These include, but are not limited to, the following.

- The generated scheduling routines can be optimized for speed, decreasing the scheduler overhead. The current routines are simple, and were developed for logical correctness, ignoring execution time.

- Task synchronization protocols will be added for both the static and dynamic scheduling approaches. The use of task synchronization protocols, e.g., semaphores, will increase the required ROM usage of the generated routines. The ability to correctly schedule more complex task sets makes the increased ROM usage acceptable.

- Sporadic servers will be added to minimize the response times of soft tasks.

Appendix A

Scheduling Overhead Data

	Number of Events														
Tasks	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	317	514	712	909	1106	1305	1502	1699	1898	2093	2293	2488	2687	2886	3084
2	356	554	752	949	1147	1345	1541	1737	1939	2133	2329	2531	2727	2924	3121
3	397	595	792	989	1187	1385	1582	1779	1980	2174	2372	2568	2770	2965	3162
4	437	634	831	1029	1227	1425	1622	1820	2018	2213	2411	2609	2810	3003	3202
5	477	674	872	1068	1267	1464	1661	1860	2058	2254	2453	2646	2846	3046	3240
6	517	715	911	1111	1307	1502	1703	1901	2096	2295	2493	2688	2889	3083	3284
7	557	754	953	1151	1346	1546	1742	1940	2137	2336	2532	2729	2926	3126	3317
8	596	795	993	1190	1389	1585	1782	1977	2176	2374	2572	2771	2966	3165	3363
9	637	834	1032	1231	1427	1624	1821	2020	2218	2413	2611	2810	3010	3202	3400
10	676	875	1072	1270	1468	1665	1862	2059	2259	2455	2654	2849	3046	3242	3443
11	715	916	1112	1310	1508	1705	1902	2102	2297	2494	2692	2891	3088	3280	3484
12	758	955	1151	1351	1549	1744	1943	2140	2337	2536	2732	2930	3128	3321	3519
13	799	996	1192	1390	1588	1785	1986	2182	2374	2575	2775	2966	3167	3364	3565
14	836	1034	1231	1430	1628	1823	2022	2220	2415	2615	2814	3011	3204	3403	3601
15	876	1076	1274	1471	1666	1865	2061	2259	2456	2659	2854	3048	3243	3445	3643

Table A.1: Average scheduling overhead for the Round Robin scheduling routines.

Number of Events														
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
311	464	618	772	926	1079	1233	1386	1541	1696	1847	2000	2156	2308	2463

Table A.2: Average scheduling overhead for the Pre-Run-Time scheduling routines.

Tasks	Number of Events														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	357	555	751	950	1147	1344	1543	1740	1937	2135	2331	2529	2726	2927	3121
2	450	648	845	1042	1239	1437	1636	1833	2029	2227	2426	2623	2820	3020	3215
3	577	775	973	1170	1366	1565	1761	1959	2157	2354	2554	2751	2944	3144	3341
4	670	868	1065	1262	1459	1656	1853	2051	2251	2448	2643	2842	3037	3235	3435
5	797	995	1191	1389	1587	1785	1981	2179	2375	2575	2772	2970	3165	3364	3561
6	891	1086	1284	1480	1677	1875	2075	2271	2469	2667	2866	3057	3257	3454	3657
7	1015	1214	1412	1610	1807	2002	2200	2402	2597	2795	2993	3187	3388	3580	3781
8	1108	1303	1504	1696	1899	2096	2289	2491	2689	2887	3085	3284	3479	3676	3873
9	1238	1433	1631	1828	2026	2228	2424	2621	2816	3013	3215	3413	3608	3803	3998
10	1325	1526	1721	1921	2117	2318	2513	2711	2909	3104	3304	3498	3700	3897	4093
11	1457	1653	1854	2046	2244	2444	2642	2841	3036	3235	3428	3633	3824	4024	4219
12	1546	1744	1942	2140	2340	2534	2731	2931	3131	3324	3525	3718	3915	4115	4312
13	1680	1875	2071	2272	2468	2665	2860	3057	3256	3452	3652	3845	4053	4245	4440
14	1766	1969	2161	2361	2559	2752	2952	3151	3347	3550	3742	3941	4141	4336	4535
15	1898	2093	2293	2485	2681	2887	3078	3277	3478	3678	3870	4075	4268	4459	4662

Table A.3: Average scheduling overhead for the Non-Preemptive Static Priority scheduling routines.

Tasks	Number of Events														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	495	630	765	900	1035	1170	1305	1440	1575	1710	1845	1980	2115	2250	2385
2	509	644	779	915	1050	1184	1319	1455	1590	1725	1860	1995	2130	2265	2399
3	528	663	798	933	1069	1203	1338	1473	1608	1743	1878	2013	2148	2283	2418
4	554	689	825	959	1095	1229	1364	1499	1635	1769	1904	2039	2175	2310	2444
5	575	710	845	980	1114	1250	1384	1519	1655	1790	1924	2059	2195	2330	2465
6	603	739	874	1009	1144	1279	1413	1548	1684	1819	1954	2090	2224	2359	2494
7	625	759	894	1029	1164	1300	1434	1568	1704	1840	1974	2109	2244	2380	2514
8	654	790	924	1060	1194	1330	1466	1599	1735	1869	2004	2139	2274	2410	2544
9	675	810	945	1080	1215	1349	1484	1619	1755	1889	2025	2159	2294	2431	2565
10	706	841	977	1110	1246	1381	1516	1651	1786	1921	2056	2192	2326	2460	2596
11	727	861	995	1133	1267	1402	1534	1671	1806	1941	2076	2210	2346	2480	2616
12	758	893	1028	1162	1298	1433	1569	1702	1837	1973	2106	2243	2377	2512	2647
13	776	913	1048	1181	1317	1453	1589	1723	1858	1994	2127	2263	2396	2533	2666
14	809	942	1081	1214	1348	1486	1620	1755	1891	2023	2159	2294	2429	2564	2698
15	829	966	1099	1236	1372	1504	1640	1773	1910	2045	2180	2313	2449	2586	2719

Table A.4: Average scheduling overhead for the Preemptive Static Priority scheduling routines.

	Number of Events														
Tasks	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	595	730	865	1000	1135	1270	1405	1540	1675	1810	1945	2080	2215	2350	2485
2	611	745	880	1016	1151	1285	1420	1556	1690	1826	1961	2096	2230	2366	2500
3	630	765	900	1036	1171	1306	1441	1575	1710	1845	1980	2115	2251	2386	2521
4	674	809	944	1079	1214	1349	1484	1619	1755	1890	2025	2159	2295	2430	2564
5	693	828	963	1098	1233	1369	1503	1638	1774	1907	2043	2178	2315	2449	2584
6	745	880	1016	1151	1286	1420	1556	1690	1826	1960	2096	2231	2366	2501	2634
7	764	897	1032	1168	1302	1439	1573	1706	1844	1980	2113	2247	2383	2517	2654
8	818	955	1090	1225	1357	1496	1630	1763	1897	2033	2168	2301	2438	2574	2709
9	835	971	1105	1241	1376	1508	1644	1778	1914	2050	2184	2318	2454	2591	2724
10	896	1028	1164	1299	1431	1567	1703	1838	1972	2108	2242	2377	2514	2647	2784
11	908	1041	1176	1315	1451	1583	1718	1852	1988	2121	2258	2390	2529	2662	2799
12	967	1101	1237	1372	1508	1644	1778	1912	2045	2182	2316	2454	2586	2722	2857
13	982	1118	1253	1386	1522	1657	1792	1928	2061	2198	2334	2466	2601	2737	2872
14	1043	1176	1312	1449	1579	1720	1856	1985	2125	2254	2392	2528	2664	2794	2933
15	1057	1193	1327	1463	1598	1730	1868	2000	2138	2273	2410	2535	2675	2815	2946

Table A.5: Average scheduling overhead for the Non-Preemptive Dynamic Priority scheduling routines.

	Number of Events														
Tasks	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	640	775	910	1045	1180	1315	1450	1585	1720	1855	1990	2125	2260	2395	2530
2	736	870	1005	1141	1276	1410	1545	1681	1815	1951	2086	2221	2355	2491	2625
3	835	970	1105	1241	1376	1511	1646	1780	1915	2050	2185	2320	2456	2591	2726
4	959	1094	1229	1364	1499	1634	1769	1904	2040	2175	2310	2444	2580	2715	2849
5	1058	1193	1328	1463	1598	1734	1868	2003	2139	2272	2408	2543	2680	2814	2949
6	1190	1325	1461	1596	1731	1865	2001	2135	2271	2405	2541	2676	2811	2946	3079
7	1289	1422	1557	1693	1827	1964	2098	2231	2369	2505	2638	2772	2908	3042	3179
8	1423	1560	1695	1830	1962	2101	2235	2368	2502	2638	2773	2906	3043	3179	3314
9	1520	1656	1790	1926	2061	2193	2329	2463	2599	2735	2869	3003	3139	3276	3409
10	1661	1793	1929	2064	2196	2332	2468	2603	2737	2873	3007	3142	3279	3412	3549
11	1753	1886	2021	2160	2296	2428	2563	2697	2833	2966	3103	3235	3374	3507	3644
12	1892	2026	2162	2297	2433	2569	2703	2837	2970	3107	3241	3379	3511	3647	3782
13	1987	2123	2258	2391	2527	2662	2797	2933	3066	3203	3339	3471	3606	3742	3877
14	2128	2261	2397	2534	2664	2805	2941	3070	3210	3339	3477	3613	3749	3879	4018
15	2222	2358	2492	2628	2763	2895	3033	3165	3303	3438	3575	3700	3840	3980	4111

Table A.6: Average scheduling overhead for the Preemptive Dynamic Priority scheduling routines.

Tasks	*Number of Events*														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	74	109	145	181	217	252	290	326	360	395	434	469	509	541	576
2	91	122	154	188	223	261	295	331	364	400	436	471	509	542	576
3	108	134	167	199	229	267	300	336	369	405	442	474	512	545	585
4	128	152	181	209	242	273	308	343	377	409	446	481	515	548	584
5	149	170	195	224	253	287	317	351	383	416	455	491	522	554	588
6	170	188	213	236	265	298	326	360	393	428	464	493	524	558	598
7	192	210	230	254	282	312	338	372	402	436	463	503	541	572	607
8	216	229	248	272	296	325	353	384	412	444	479	512	544	576	611
9	239	253	271	289	310	339	363	400	427	452	487	518	551	589	622
10	261	271	289	309	330	356	382	408	436	469	498	532	558	597	628
11	283	296	314	327	347	374	397	425	452	481	509	540	580	607	642
12	306	316	331	348	369	389	417	441	469	494	528	548	582	613	650
13	327	335	350	368	389	408	438	454	485	509	539	563	597	625	660
14	350	359	369	389	404	425	452	469	498	525	551	584	607	638	670
15	373	384	394	412	425	450	468	488	512	537	564	599	624	654	686

Table A.7: Standard deviation for the average Round Robin scheduling overhead.

Number of Events														
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
45	55	81	110	143	178	211	250	280	318	341	388	419	448	489

Table A.8: Standard deviation for the average Pre-Run-Time scheduling overhead.

	Number of Events														
Tasks	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	92	123	156	190	224	258	295	330	365	401	436	472	511	547	577
2	134	158	184	212	246	279	310	344	379	414	447	484	514	554	587
3	180	196	217	245	273	303	330	367	396	426	460	498	531	565	605
4	223	238	253	278	301	329	356	389	419	446	478	514	547	580	615
5	271	285	298	317	338	363	390	419	441	472	501	540	569	604	634
6	315	328	340	358	376	396	421	447	474	502	531	556	593	622	655
7	365	377	385	404	423	441	460	484	507	533	557	593	617	650	689
8	408	417	434	443	452	476	496	521	543	565	593	619	645	673	713
9	462	468	480	486	501	522	535	563	575	607	627	649	687	706	732
10	509	508	523	534	541	565	575	595	623	638	658	684	709	732	766
11	557	564	564	581	598	608	631	645	660	681	705	724	749	777	795
12	605	609	616	628	638	642	663	677	699	721	742	764	784	809	834
13	655	663	663	667	693	708	705	727	745	754	792	792	825	845	869
14	700	701	707	717	730	741	760	766	775	801	822	844	857	888	897
15	750	758	754	766	777	780	800	806	821	837	862	877	891	918	948

Table A.9: Standard deviation for the average Non-Preemptive Static Priority scheduling overhead.

	Number of Events														
Tasks	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	2	2	2	2	2	2	2	1	2	2	2	2	2	2
2	22	22	22	22	22	21	22	22	22	22	22	22	22	22	23
3	43	43	42	43	41	43	42	42	42	42	42	42	42	42	42
4	63	64	64	64	62	64	64	64	63	64	63	64	63	63	64
5	83	83	82	83	83	84	85	82	84	83	84	83	83	83	83
6	104	105	105	106	107	105	102	103	104	104	106	105	104	105	105
7	123	124	123	125	124	123	125	123	125	126	123	127	123	124	125
8	143	145	147	145	145	145	146	148	147	146	148	145	146	144	147
9	165	164	166	165	162	165	163	164	165	166	162	164	166	164	163
10	185	182	188	187	183	187	185	188	188	185	185	186	185	186	188
11	207	207	205	206	204	204	207	207	203	205	203	206	208	207	207
12	228	226	228	226	231	226	228	227	226	227	228	230	226	228	228
13	248	250	244	244	248	246	245	246	245	242	247	245	245	247	245
14	270	267	269	272	269	269	269	269	264	267	268	268	268	270	268
15	285	288	283	286	287	284	290	286	285	285	286	287	286	287	290

Table A.10: Standard deviation for the average Preemptive Static Priority scheduling overhead.

	Number of Events														
Tasks	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	2	3	2	2	3	3	2	2	3	3	2	4	2	3	3
2	23	23	24	23	24	23	23	24	24	24	23	23	24	24	24
3	46	46	44	46	44	45	44	45	45	45	45	45	45	45	45
4	68	68	68	68	66	68	68	69	67	68	67	68	67	67	68
5	89	89	87	89	88	89	91	87	89	89	90	88	89	88	89
6	111	112	112	113	114	112	109	110	111	111	113	112	111	113	112
7	131	132	132	133	133	131	133	131	133	134	132	135	131	132	133
8	153	154	157	155	154	155	155	158	157	155	158	155	156	154	157
9	176	175	178	176	173	176	174	175	176	177	173	175	177	175	174
10	198	195	201	200	196	200	197	200	201	197	198	199	198	198	201
11	220	221	218	220	218	218	220	221	216	219	216	220	222	221	220
12	243	242	243	242	247	241	243	242	241	249	243	245	241	244	245
13	264	267	261	260	261	263	261	263	261	253	263	262	261	263	260
14	288	285	287	290	288	287	287	287	282	282	286	286	285	288	286
15	304	307	302	306	304	303	309	305	304	305	305	306	305	306	308

Table A.11: Standard deviation for the average Non-Preemptive Dynamic Priority scheduling overhead.

	Number of Events														
Tasks	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	6	7	7	6	7	7	6	6	7	7	5	8	5	8	7
2	47	47	48	47	48	46	47	48	48	48	47	47	47	48	49
3	92	91	89	91	88	91	89	91	91	90	90	91	90	91	90
4	136	136	136	137	132	136	136	138	135	137	135	137	135	135	137
5	178	178	175	178	177	179	182	175	179	178	180	177	178	177	178
6	222	224	224	226	228	225	219	221	223	223	226	225	223	226	224
7	263	265	264	267	266	263	266	263	266	269	264	271	263	265	266
8	306	309	314	310	309	311	311	317	314	311	317	310	312	309	315
9	353	350	356	352	346	352	349	351	353	355	347	350	355	351	348
10	396	390	402	400	392	400	395	401	402	395	396	398	396	397	402
11	441	442	437	441	436	436	441	443	433	438	433	440	445	442	441
12	487	484	487	484	494	482	486	485	483	484	486	491	482	488	486
13	529	534	522	521	530	526	523	526	523	518	527	524	522	527	524
14	577	571	575	581	574	575	574	575	564	571	572	573	571	577	573
15	609	615	604	612	612	607	619	611	609	609	611	612	610	612	618

Table A.12: Standard deviation for the average Preemptive Dynamic Priority scheduling overhead.

Bibliography

[ABRW91] N.C. Audsley, A. Burns, M.F. Rishardson, and A.J. Wellings. Hard real-time scheduling: The deadline-monotonic approach. In *Real Time Programming: Proceedings of the IFAC/IFIP Workshop*, pages 127 - 132. Pergamon Press, June 1991.

[BCG91] G. Berry, P. Corunne, and B. Gonthier. The synchronous approach to reactive and real-time systems. In *IEEE Proceedings*, September 1991.

[BMR90] Sanjoy K. Baruah, Aloysius K. Mok, and Louis E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 182 - 190. IEEE Computer Society Press, December 1990.

[BS86] Theodore P. Baker and Gregory M. Scallon. An architecture for real-time software systems. *IEEE Software*, pages 50 - 58, May 1986.

[BS89] T.P. Baker and A. Shaw. The cyclic executive model and Ada. *Real-Time Systems*, 1(1):7 - 25, June 1989.

[BSR88] Sara R. Biyabani, John A. Stankovic, and Krithi Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 152 - 160. IEEE Computer Society Press, December 1988.

[CEG+95] M. Chiodo, D. Engels, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, K. Suzuki, and A. Sangiovanni-Vincentelli. A case study in computer-aided codesign of embedded controllers. to appear, 1995.

[CGH+94] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich, and K. Suzuki. Synthesis of Software Programs for Embedded Control Applications. Technical Report UCB/ERL M94/87, ERL, Univ. of California, Berkeley, CA 94720, November 1994.

[CGJ+94] M. Chiodo, P. Giusto, A. Jurecska, H. Hsieh, A. Sangiovanni-Vincentelli, and L. Lavagno. Hardware-software codesign of embedded systems. *IEEE Micro*, 14(4):26 - 36, August 1994.

[CL90] Min-Ih Chen and Kwei-Jay Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *The Journal of Real-Time Systems*, 2:325 - 346, 1990.

[CSR86] Shengchang Cheng, John A. Stankovic, and Krithivasan Ramamritham. Dynamic scheduling of groups of tasks with precedence constraints in distributed hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 166 - 174. IEEE Computer Society Press, December 1986.

[Der74] Michael L. Dertouzos. Control robotics: The procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807 - 813, 1974.

[DH89] D. Druzinski and D. Hare. Using statecharts for hardware description and synthesis. *IEEE Transactions on Computer-Aided Design*, 8(7), July 1989.

[DL78] Sudarshan K. Dhall and C.L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127 - 140, January 1978.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[HL92] Ching-Chih Han and Kwei-Jay Lin. Scheduling distance-constrained real-time tasks. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 300 - 308. IEEE Computer Society Press, December 1992.

[HS90] Wolfgang A. Halang and Alexander D. Stoyenko. Comparative evaluation of high-level real-time programming languages. *The Journal of Real-Time Systems*, 2:365 - 383, 1990.

[HS91] Wolfgang A. Halang and Alexander D. Stoyenko. *Constructing Predictable Real-Time Systems.* Kluwer Academic Publisher, 1991.

[Jef92] Kevin Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 89 - 99. IEEE Computer Society Press, December 1992.

[JSM91] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 129 - 139. IEEE Computer Society Press, December 1991.

[KLR94] Mark H. Klein, John P. Lehoczky, and Ragunathan Rajkumar. Rate-monotonic analysis for real-time industrial computing. *Computer*, 27(1):24 - 33, January 1994.

[KS86] Eugene Kligerman and Alexander D. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Transactions on Software Engineering*, SE-12(9):941–949, September 1986.

[Leh90] John P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 201 - 209. IEEE Computer Society Press, December 1990.

[LL73] C.L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46 - 61, January 1973.

[LL78] B.J. Lageweg and J.K. Lenstra. A general bounding scheme for the permutation flow-shop problem. *Operations Research*, 26(1):53 - 67, January 1978.

[LSD89] John Lehoczky, Lui Sha, and Ye Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 166 - 171. IEEE Computer Society Press, December 1989.

[LSS87] John P. Lehoczky, Lui Sha, and Jay K. Strosnider. Enhanced aperiodic responsiveness in hard real-time environments. In *Proceedings of the IEEE Real-Time*

Systems Symposium, pages 261 – 270. IEEE Computer Society Press, December 1987.

[LSST91] John P. Lehoczky, Lui Sha, J.K. Strosnider, and Hide Tokuda. Fixed priority scheduling theory for hard real-time systems. In André M. van Tilborg and Gary M. Koob, editors, *Foundations of Real-Time Computing: Scheduling and Resource Management*, chapter 1, pages 1 – 30. Kluwer Academic Publishers, 1991.

[LW82] Joseph Y.-T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4):237 – 250, December 1982.

[Mok83] A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment.* PhD thesis, Dept. of Electrical Engineering and Computer Science, The Massachusetts Institute of Technology, Cambridge, MA, May 1983.

[SB94] Marco Spuri and Giorgio Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. Technical Report ARTS Lab 94-06, Scuola Superiore Di Studi Universitari E Di Perfezionamento S. Anna, April 1994.

[SdA93] M. Saksena, J. da Silva, and A.K. Agrawala. Design and implementation of maruti-ii. Technical Report UMD CS-TR-3181, UMICA TR-93-122, University of Maryland, 1993.

[SG91] Terry Shepard and J.A. Martin Gagné. A pre-run-time scheduling algorithm for hard real-time systems. In *IEEE Transactions on Software Engineering*, volume 17, pages 669 - 677. IEEE Computer Society Press, July 1991.

[SHH91] Alexander D. Stoyenko, Carl Hamacher, and Richard C. Holt. Analyzing hard-real-time programs for guaranteed schedulability. In *IEEE Transactions on Software Engineering*, volume 17, pages 737 – 750. IEEE Computer Society Press, August 1991.

[Shi87] Akira Shimohara. REALOS/286: An implementation of ITRON/MMU on 80286. In *TRON Project 1987 (Proceedings of the Third TRON Project Symposium)*, pages 45–56. Springer-Verlog, 1987.

[SR90] John A. Stankovic and Krithi Ramamritham. What is predictability for real-time systems? *The Journal of Real-Time Systems*, 2:247 – 254, 1990.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175 – 1185, September 1990.

[SSL89] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *The Journal of Real-Time Systems*, 1(1):27 – 60, June 1989.

[SSL+92] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. SIS: A System for Sequential Circuit Synthesis. Technical Report UCB/ERL M92/41, ERL, Univ. of California, Berkeley, CA 94720, May 1992.

[SSM+92] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Sequential Circuit Design Using Synthesis and Optimization. In *ICCD*, pages 328–333, Oct 1992.

[TSiH87] Hiroshi Takeyama, Tsuyoshi Shimizu, and Ken ichi Horikoshi. The HI series of operating systems of the ITRON architecture. In *TRON Project 1987 (Proceedings of the Third TRON Project Symposium)*, pages 57–71. Springer-Verlog, 1987.

[Ull75] J.D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, June 1975.

[XP90] Jia Xu and David Lorge Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. In *IEEE Transactions on Software Engineering*, volume 16, pages 360 – 369. IEEE Computer Society Press, March 1990.

[XP93] Jia Xu and David Lorge Parnas. On satisfying timing constraints in hard-real-time systems. In *IEEE Transactions on Software Engineering*, volume 19, pages 70 – 84. IEEE Computer Society Press, January 1993.

[Xu93] Jia Xu. Multiprocessor scheduling of processes with release times, deadlines, precedence, and exclusion relations. In *IEEE Transactions on Software Engineering*, volume 19, pages 139 – 154. IEEE Computer Society Press, February 1993.