REAL-TIME SYSTEMS: WELL-TIMED SCHEDULING AND SCHEDULING WITH PRECEDENCE CONSTRAINTS

Goran Zlokapa

CMPSCI Technical Report 93-51

February 1993

REAL-TIME SYSTEMS: WELL-TIMED SCHEDULING AND SCHEDULING WITH PRECEDENCE CONSTRAINTS

A Dissertation Presented

by

GORAN ZLOKAPA

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

DOCTOR OF PHILOSOPHY

February 1993

Department of Electrical and Computer Engineering

© Copyright by Goran Zlokapa 1993

All Rights Reserved

REAL-TIME SYSTEMS:
WELL-TIMED SCHEDULING AND
SCHEDULING WITH PRECEDENCE CONSTRAINTS

A Dissertation Presented

by

GORAN ZLOKAPA

Approved as to style and content by:

____
John A. Stankovic, Chair

____
Krithi Ramamritham, Member

____
Mani C. Krishna, Member

____
Dhiraj K. Pradhan, Member

____
Lewis E. Franks, Department Head
Department of Electrical and
Computer Engineering

To those whose knowledge contributed
to reaching thus far, and
to those for whom this research
will help to reach even further

ACKNOWLEDGMENTS

Being there is nice, but being there for someone is what separates a friend from an acquaintance. I'm thankful to all the people that were there for me. Without your support I would not have had neither the strength nor knowledge necessary to successfully complete this dissertation. Thank you all.

My special thanks to Prof. Jack Stankovic and Prof. Krithi Ramamritham, my two advisors and great researchers whose guidance, vision, positive critique, and enormous editorial patience helped me to overcome difficulties associated with a project of this magnitude. I would also like to extend my gratitude to Prof. Mani Krishna and Prof. Dhiraj Pradhan for their help during the conceptualization and design phase of this dissertation. In particular, my thanks to Prof. Krishna for his moral support and greatly appreciated technical expertise.

I would like to thank Betty Hardy, the secretary. Her cheerful attitude, prompt and efficient help would always brighten even the most gloomy day. Also, thanks to my colleges and office mates—in particular to Chia, Lory, Gary, Fuxing, Doug, Cris, Panos and Victor. What would life be if we did not share the good and the bad through all these years; it would probably be a boring place.

My special thanks to Deepak, Dennis, Lisa, Stefan and Pat, my true friends. Their warmth, understanding and unselfish behavior made my home closer and my transition to a new world much easier.

Being far away from home, it is hard not to remember all the good things that my family and my friends gave to me. Thank you for your enormous help and influence on my life, for without you this dissertation would not be possible. My

parents, Djuro and Zora, my brother Boban, and my special friends Tripa, Mile, Nešo and Teša, thank you all so much.

Last, but not least, I would like to thank Janie who gave meaning to my life in these hard times when my homeland is raging in civil war. Her love, care, understanding and patience are unsurpassed.

Thank you all for being there with me and for me.

ABSTRACT

REAL-TIME SYSTEMS:

WELL-TIMED SCHEDULING AND

SCHEDULING WITH PRECEDENCE CONSTRAINTS

FEBRUARY 1993

GORAN ZLOKAPA

B.S., UNIVERSITY OF SARAJEVO, BOSNIA-HERZEGOVINA

M.S., PH.D., UNIVERSITY OF MASSACHUSETTS

Directed by: Professor John A. Stankovic

This dissertation attempts to bridge the gap between recent theoretical results from Scheduling Theory, Queueing Theory, and Operations Research and actual requirements of current and future real-time systems.

The first part develops a novel approach for the *timely* scheduling of dynamically arriving tasks, and it is, primarily, designed for on-line schedulers of non-deterministic complex real-time systems that perform in temporary or permanent overloads. In these systems, to predict whether a task will complete by its deadline, an on-line schedulability analysis has to be carried out. The quality of the analysis, as well as its computational overheads, depends on *when* the analysis is performed and *how many* tasks are involved. Our approach to schedulability analysis, called *Well-Timed Scheduling*, is based on analytically derived control parameters. This approach presents a *framework* for on-line real-time schedulers, and it lends itself to use with different scheduling policies. Well-Timed Scheduling provides a methodical approach to quantifiable guarantees of timing constraints with potentially low scheduling overhead and high system performance. Using this approach, the ready-to-execute tasks are scheduled at an "opportune" time, rather than at arrival time or at dispatch time as in the traditional approaches. The analytical derivation

of the "opportune" time is based on recent theoretical results, and it is validated through simulation. Aside from run-time benefits (e.g., low scheduling overheads), Well-Timed Scheduling is useful as a design tool. It can, for example, be used to determine the number of processors needed to achieve the required level of system's guaranteed performance for a given $M/G/c$ real-time system.

In the second part of the dissertation, we develop off-line preprocessing algorithms that enable effective and efficient on-line scheduling of task groups with different contributing values, timing constraints, resource constraints, and arbitrary precedence constraints. These algorithms derive new value densities that *reflect* how valuable the individual tasks and their successors are. By utilizing these reflective value densities, on-line schedulers are not required to examine the successors of ready-to-execute tasks at run-time to select the best task to schedule next. This approach greatly reduces the computational complexity of on-line schedulers, and it extends the applicability range of existing on-line scheduling algorithms for independent tasks to scheduling of task groups with arbitrary precedence constraints. Due to the separation of value density preprocessing and deadline preprocessing, the developed algorithms are equally applicable to real-time and non real-time systems.

The overall goal of this dissertation is the development of efficient and effective scheduling methods for on-line scheduling of complex real-time systems in very demanding non-deterministic environments, where the best-effort algorithms are used to maximize the total accrued system value.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS v

ABSTRACT vii

LIST OF TABLES xiii

LIST OF FIGURES xvii

Chapter

1. INTRODUCTION 1

   1.1 Motivation 3
   1.2 Our Approach 5

      1.2.1 Well-Timed Scheduling 5

         1.2.1.1 The Punctual Point 7

      1.2.2 Scheduling with Precedence Constraints 8

   1.3 Dissertation Organization 11

2. WELL-TIMED SCHEDULING: A FRAMEWORK FOR DYNAMIC REAL-TIME SCHEDULERS 14

   2.1 Introduction 14
   2.2 Punctual Point Derivation 17

      2.2.1 Punctual Point Derivation for $\rho < 1$ 18
      2.2.2 Punctual Point Derivation for $\rho > 1$ 20

   2.3 Comparison of Analytical and Experimental Results 28
   2.4 Laxity Analysis 35

      2.4.1 Zero Laxity Systems 35

2.4.2 Probabilistic Guarantee for $M/G/c$ Systems 37
2.5 Summary 40
3. INTEGRATION OF WELL-TIMED SCHEDULING AND REAL-TIME SCHEDULERS 43
3.1 Introduction 43
3.2 Model Description 44
3.2.1 DLVD Algorithm 48
3.2.2 RDS Algorithm 50
3.3 Simulation Description 54
3.3.1 System Parameters 55
3.3.2 Task Parameters 55
3.3.3 The Punctual Points 57
3.3.4 Simulation Parameters 59
3.4 Integration Analysis 61
3.4.1 Analysis of DLVD and RDS Algorithms 61
3.4.2 The Limitations of DLVD and RDS Algorithms 69
3.4.2.1 Two Processor System 70
3.4.2.2 Eight Processor System 74
3.4.3 The Effects of Integration 78
3.4.3.1 Two Processor System 78
3.4.3.2 Eight Processor System 85
3.4.4 The Robustness of the Integration 90
3.4.4.1 Two Processor System 91
3.4.4.2 Eight Processor System 102
3.5 Summary 111
4. PRECEDENCE CONSTRAINS SCHEDULING 120
4.1 Introduction 120
4.2 Related Work 124
4.2.1 Notation and Problem Classification 125

4.2.2 Metric . . . 126
4.2.3 Single Processor Scheduling . . . 128
4.2.4 Multiple Processor Scheduling . . . 131

4.3 Model Description . . . 133
4.4 Design Strategy . . . 135
4.5 VDP-R: An Algorithm for Rooted Tree Task Groups . . . 137
4.6 VDP-G: An Algorithm for Arbitrary Task Groups . . . 145

4.6.1 The Notation . . . 146
4.6.2 The Algorithm Description . . . 148
4.6.3 Three Illustrative Examples . . . 150

4.6.3.1 The Out-Tree Example . . . 151
4.6.3.2 The In-Tree Example . . . 154
4.6.3.3 The Example with Arbitrary Precedence Constraints 155

4.7 Summary . . . 156

5. VDP-G ALGORITHM: APPLICABILITY ANALYSIS . . . 159

5.1 Introduction . . . 159
5.2 Model Description . . . 161
5.3 Simulation Testbed Description . . . 162

5.3.1 The Task Group Generator . . . 163
5.3.2 The Load Generator . . . 164
5.3.3 The Simulator . . . 164

5.4 The Analysis . . . 165

5.4.1 Analysis of Out-Tree Task Groups . . . 167

5.4.1.1 Groups with Random Values . . . 168
5.4.1.2 Groups with Linear Value Assignment . . . 171
5.4.1.3 Large Groups with Random Values . . . 178
5.4.1.4 Groups with Diminishing Contributing Values . . . 179
5.4.1.5 Groups with Random Values: A Two Processor System 181

5.4.2 Analysis of In-Tree Task Groups . . . 183

5.4.2.1 Groups with Random Values . . . 184

5.4.2.2 Groups with Diminishing, Random Values 185
5.4.3 Analysis of Arbitrary Task Groups 187
5.4.3.1 Groups with Random Values 187
5.4.3.2 Large Groups with Random Values 190
5.4.3.3 Groups with Diminishing, Random Values 193
5.4.3.4 Large Groups with Diminishing, Random Values . . . 194
5.5 Summary 196
6. OBSERVATIONS AND CONCLUSIONS 201
6.1 Contributions 202
6.1.1 Well-Timed Scheduling 202
6.1.2 Scheduling with Precedence Constraints 204
6.2 Extensions 204
APPENDICES
A. NOTATION 207
B. WAITING TIME DISTRIBUTION FOR AN M/G/c SYSTEM 208
C. REJECTION PROBABILITIES FOR AN $M/G/c/K$ SYSTEM 209
D. $M/E_k/c/K$: A CASE STUDY 211
BIBLIOGRAPHY 213

LIST OF TABLES

Table Page

2.1 Punctual Points for $\psi = 0.99$, and 0.999 Probabilities. 18

2.2 Punctual Points for $M/M/2/K$ Systems with $\rho = 1.6$ Load. 26

2.3 Analytically Derived Punctual Points for $M/M/2$ system. 30

2.4 Analytically Derived Punctual Points for $M/M/8$ system. 32

2.5 Punctual Points for $\rho = 0.9$ Load. 38

2.6 Punctual Points for $\rho = 0.7$ Load. 39

2.7 Punctual Points for $\rho = 0.9$ Load. 39

3.1 System Parameters. 55

3.2 Task Parameters. 56

3.3 Punctual Points for $M/M/2$ System 58

3.4 Punctual Points for $M/E_3/2$ System 58

3.5 Punctual Points for $M/E_3/8$ System 59

3.6 Simulation Parameters. 60

3.7 Applicability Range for a Traditional Approach (2 CPUs). 115

3.8 Applicability Range for an Integrated Approach (2 CPUs). 115

3.9 Applicability Range for a Traditional Approach (8 CPUs). 118

3.10 Applicability Range for an Integrated Approach (8 CPUs). 118

4.1 Parameters for Three Independent Tasks. 127

4.2 Derivation of $\vec{S} \times \vec{V}$ Space for Figure 4.9 153

4.3 Derivation of $\vec{S} \times \vec{V}$ Space for Figure 4.10 155

4.4 Derivation of $\vec{S} \times \vec{V}$ Space for Figure 4.11 156

5.1 Task Group Generator Parameters. 164

5.2 Load Generator Parameters. 165

5.3 Simulator Parameters. 166

5.4 Groups with Random Values: I-heuristic vs. G-heuristic. 171

5.5 Groups with Random Values: R-heuristic vs. G-heuristic. 171

5.6 Groups with Top-Heavy Values: I-heuristic vs. G-heuristic. 175

5.7 Groups with Top-Heavy Values: R-heuristic vs. G-heuristic. 175

5.8 Groups with Bottom-Heavy Values: I-heuristic vs. G-heuristic. 177

5.9 Groups with Bottom-Heavy Values: R-heuristic vs. G-heuristic. . . . 177

5.10 Large Groups with Random Values: I-heuristic vs. G-heuristic. 178

5.11 Large Groups with Random Values: R-heuristic vs. G-heuristic. . . . 179

5.12 Groups with Diminishing Values: I-heuristic vs. G-heuristic. 180

5.13 Groups with Diminishing Values: R-heuristic vs. G-heuristic. 180

5.14 Two Processor Systems: I-heuristic vs. G-heuristic. 183

5.15 Two Processor Systems: R-heuristic vs. G-heuristic. 183

5.16 Groups with Random Values: I-heuristic vs. G-heuristic. 184

5.17 Groups with Random Values: R-heuristic vs. G-heuristic. 185

5.18 Groups with Diminishing Values: I-heuristic vs. G-heuristic. 186

5.19 Groups with Diminishing Values: R-heuristic vs. G-heuristic. 186

5.20 Groups with Random Values: I-heuristic vs. G-heuristic. 189

5.21 Groups with Random Values: R-heuristic vs. G-heuristic. 190

5.22 Large Groups with Random Values: I-heuristic vs. G-heuristic. 192

5.23 Large Groups with Random Values: R-heuristic vs. G-heuristic. . . . 193

5.24 Groups with Diminishing Values: I-heuristic vs. G-heuristic. 194

5.25 Groups with Diminishing Values: R-heuristic vs. G-heuristic. 194

5.26 Large Groups with Diminishing Values: I-heuristic vs. G-heuristic. . . 195

5.27 Large Groups with Diminishing Values: R-heuristic vs. G-heuristic. . 195

5.28 Summary of Performance Trends for Out-Tree Task Groups. 197

5.29 Summary of Performance Trends for In-Tree Task Groups. 199

5.30 Summary of Performance Trends for Arbitrary Task Groups. 199

LIST OF FIGURES

Figure | Page

2.1 Approaches to Dynamic Scheduling Analysis. 15

2.2 Complementary Waiting Time Distribution for $\rho < 1$. 19

2.3 Task Loss Ratios for Real-time Systems [65] 21

2.4 K-size Distribution. 23

2.5 Mapping Procedure. 25

2.6 Complementary Waiting Time Distribution for Finite Buffer Systems. 26

2.7 Task Loss Ratio with Punctual Points. 1: $T_P(1^-)$, 2: $T_P(0.999)$, and 3: $T_P(0.95)$. 30

2.8 Task Loss Ratio with Punctual Points. 1: $T_P(1^-)$, 2: $T_P(0.999)$, and 3: $T_P(0.95)$. 31

2.9 Task Loss Ratio with Punctual Points. 1: $T_P(1^-)$, 2: $T_P(0.999)$, and 3: $T_P(0.95)$. 32

2.10 The Average Number of "Relevant" Tasks per Task Arrival. 34

2.11 The Average Number of "Relevant" Tasks per Task Arrival. 35

2.12 Task Loss Ratios for Zero Laxity Systems—Erlang Loss 36

2.13 Complementary Waiting Time Distribution 37

3.1 DLVD Algorithm. 49

3.2 DLVD—Rejection Procedure. 49

3.3 RDS Algorithm. 51

3.4 RDS—Parameter Assignment and Feasibility Check. 52

3.5 RDS—Heuristic Function. 53

3.6 DLVD—Value Loss Ratios for Exponential Distributions. 63

3.7 RDS—Value Loss Ratios for Exponential Distributions. 63

3.8 DLVD—Value Loss Ratios for Erlang-3 Distributions. 64

3.9 RDS—Value Loss Ratios for Erlang-3 Distributions. 64

3.10 Average Laxity/Computation for Exp. and Erlang-3 Distributions. . 66

3.11 DLVD—Task Loss Ratios for Exp. and Erlang-3 Distributions. 68

3.12 DLVD—Value Loss Ratios for $\rho = 2.0$. 71

3.13 RDS—Value Loss Ratios for $\rho = 2.0$. 71

3.14 DLVD—Maximum S-pool Size for $\rho = 2.0$. 73

3.15 RDS—Maximum S-pool Size for $\rho = 2.0$. 73

3.16 DLVD—Value Loss Ratios for 8 CPUs and $\rho = 1.2$. 75

3.17 RDS—Value Loss Ratios for 8 CPUs and $\rho = 1.2$. 75

3.18 DLVD—Value Loss Ratios for 8 CPUs and $\rho = 0.9$. 76

3.19 RDS—Value Loss Ratios for 8 CPUs and $\rho = 0.9$. 76

3.20 RDS—Value Loss Ratios for $\rho = 2.0$. 80

3.21 RDS—Value Loss Ratio for $\rho = 2.0$ and $T_P(0.999)$. 80

3.22 RDS—Value Loss Ratios for $\rho = 1.2$. 83

3.23 RDS—Value Loss Ratio for $\rho = 1.2$ and $T_P(0.999)$. 83

3.24 RDS—Value Loss Ratios for $\rho = 0.9$. 84

3.25 RDS—Value Loss Ratio for $\rho = 0.9$ and $T_P(0.999)$. 84

3.26 RDS—Value Loss Ratios for 8 CPUs and $\rho = 2.0$. 86

3.27 RDS—Value Loss Ratio for 8 CPUs, $\rho = 2.0$ and $T_P(0.999)$. 86

3.28 RDS—Value Loss Ratios for 8 CPUs and $\rho = 1.2$. 88

3.29 RDS—Value Loss Ratio for 8 CPUs, $\rho = 1.2$ and $T_P(0.999)$. 88

3.30 RDS—Value Loss Ratios for 8 CPUs and $\rho = 0.9$. 89

3.31 RDS—Value Loss Ratio for 8 CPUs, $\rho = 0.9$ and $T_P(0.999)$. 89

3.32 RDS—Maximum S-pool Size for $\rho = 2.0$ and $SCF = 10^{-6}ES$. 92

3.33 RDS—Maximum S-pool Size for $\rho = 1.2$ and $SCF = 10^{-6}ES$. 92

3.34 RDS—Value Loss Ratios for $\rho = 2.0$ and $SCF = 10^{-6}ES$. 95

3.35 RDS—Value Loss Ratios for $\rho = 1.2$ and $SCF = 10^{-6}ES$. 95

3.36 RDS—Value Loss Ratios for $\rho = 2.0$ and $SCF = 0.004ES$. 97

3.37 RDS—Value Loss Ratios for $\rho = 1.2$ and $SCF = 0.004ES$. 97

3.38 RDS—Value Loss Ratios for $\rho = 2.0$ and $SCF = 0.008ES$. 98

3.39 RDS—Value Loss Ratios for $\rho = 1.2$ and $SCF = 0.008ES$. 98

3.40 RDS—Value Loss Ratios for $\rho = 2.0$ and $SCF = 0.016ES$. 99

3.41 RDS—Value Loss Ratios for $\rho = 1.2$ and $SCF = 0.016ES$. 99

3.42 RDS—Maximum S-pool Size for 8 CPUs, $\rho = 2.0$ and $SCF = 10^{-6}ES$. 104

3.43 RDS—Maximum S-pool Size for 8 CPUs, $\rho = 1.2$ and $SCF = 10^{-6}ES$. 104

3.44 RDS—Value Loss Ratio for 8 CPUs, $\rho = 2.0$ and $SCF = 10^{-6}ES$. . . 105

3.45 RDS—Value Loss Ratio for 8 CPUs, $\rho = 1.2$ and $SCF = 10^{-6}ES$. . . 105

3.46 RDS—Value Loss Ratio for 8 CPUs, $\rho = 2.0$ and $SCF = 0.002ES$. . 107

3.47 RDS—Value Loss Ratio for 8 CPUs, $\rho = 1.2$ and $SCF = 0.002ES$. . 107

3.48 RDS—Value Loss Ratio for 8 CPUs, $\rho = 2.0$ and $SCF = 0.004ES$. . 108

3.49 RDS—Value Loss Ratio for 8 CPUs, $\rho = 1.2$ and $SCF = 0.004ES$. . 108

3.50 RDS—Value Loss Ratio for 8 CPUs, $\rho = 2.0$ and $SCF = 0.008ES$. . 110

3.51 RDS—Value Loss Ratio for 8 CPUs, $\rho = 1.2$ and $SCF = 0.008ES$. . 110

3.52 RDS—Value Loss Ratios for a Traditional Approach (2 CPUs) 113

3.53 RDS—Value Loss Ratios for an Integrated Approach (2 CPUs) 114

3.54 RDS—Value Loss Ratios for a Traditional Approach (8 CPUs) 116

3.55 RDS—Value Loss Ratios for an Integrated Approach (8 CPUs) 117

4.1 Reduction to a DAG with a Transitive Closure. 134

4.2 The Task Group Scheduling Approach. 135

4.3 Initial and Reflective Parameters with Decreasing Ratios 138

4.4 Initial and Reflective Parameters with Increasing Ratios 139

4.5 VDP-R: The Value Density Propagation Algorithm for Rooted Trees. 142

4.6 VDP-R Applied to an Example from [2]. 143

4.7 VDP-G: The Main Body. 149

4.8 VDP-G: Update $\vec{S} \times \vec{V}$ Space Procedure. 150

4.9 A Task Group with Out-Tree Precedence Constraints. 151

4.10 A Task Group with In-Tree Precedence Constraints. 155

4.11 A Task Group with Arbitrary Precedence Constraints. 156

5.1 Value Loss Ratios for Groups with Random Values, I-heuristic. 170

5.2 Value Loss Ratios for Groups with Random Values, G-heuristic. . . . 170

5.3 Value Loss Ratios for Groups with Random Values, R-heuristic. . . . 170

5.4 Value Loss Ratios for Groups with Top-Heavy Values, I-heuristic. . . 174

5.5 Value Loss Ratios for Groups with Top-Heavy Values, G-heuristic. . . 174

5.6 Value Loss Ratios for Groups with Top-Heavy Values, R-heuristic. . . 174

5.7 Value Loss Ratios for Groups with Bottom-Heavy Values, I-heuristic. 176

5.8 Value Loss Ratios for Groups with Bottom-Heavy Values, G-heuristic. 176

5.9 Value Loss Ratios for Groups with Bottom-Heavy Values, R-heuristic. 176

5.10 Value Loss Ratios for Two Processor Systems, I-heuristic. 182

5.11 Value Loss Ratios for Two Processor Systems, G-heuristic. 182

5.12 Value Loss Ratios for Two Processor Systems, R-heuristic. 182

5.13 Value Loss Ratios for Groups with Random Values, I-heuristic. 188

5.14 Value Loss Ratios for Groups with Random Values, G-heuristic. . . . 188

5.15 Value Loss Ratios for Groups with Random Values, R-heuristic. . . . 188

5.16 Value Loss Ratios for Large Groups with Random Values, I-heuristic. 191

5.17 Value Loss Ratios for Large Groups with Random Values, G-heuristic. 191

5.18 Value Loss Ratios for Large Groups with Random Values, R-heuristic. 191

CHAPTER 1

INTRODUCTION

Real-time systems are systems where the tasks must be both logically correct and timely. In these systems, the tasks are typically assigned deadlines and/or periods to complete their execution. Executing tasks beyond the given timing constraints can result in both incorrect computation, and unstable and unpredictable behavior of the entire system. Therefore, when tasks cannot meet their timing constraints, a system must reject these tasks.

To achieve high system performance, besides the timing constraints, the system must consider the relative importance of tasks when determining which tasks to reject and which tasks to execute. This relative importance is usually given as a time-value function that specifies the contributing value of a task to the system upon its successful completion. However, the assignment of contributing values to individual tasks is not required for all types of tasks found in real-time systems.

The *critical* tasks do not require explicit assignment of contributing values[43]. For these tasks, the value assignment is irrelevant because by definition, all critical tasks *must* complete within given timing constraints, or catastrophic results might occur. Critical tasks are usually the tasks vital for the overall system operation and for the overall safety of the mission. The number of safety-critical tasks is very small when compared to the total number of tasks in the system.

Besides critical tasks, the *essential* and *non-essential* tasks are commonly found in real-time systems. Essential tasks are the tasks that carry the main function of the mission, they have timing constraints, and they do not cause catastrophic

results if not completed on time. However, missing a deadline of an essential task might have a significant impact on the overall system performance.

The number of essential tasks can be very large. To cope with this potentially large number of active tasks—caused by non-deterministic environments—a dynamic, on-line scheduling approach is required.

Non-essential tasks, on the other hand, are, usually, treated as background tasks that execute in spare time regardless of their timing constraints.

The Spring Project at University of Massachusetts [43] has a long history of investigating the scheduling problems involving critical and essential tasks in complex, dynamic, and non-deterministic real-time environments. In this dissertation, we focus on scheduling aspects of essential tasks in multiprocessor environment. However, the presented concepts and algorithms are inherently applicable to non-essential tasks as well.

In the first half of the dissertation, we develop a methodological approach to schedulability analysis named *Well-Timed Scheduling*. The outstanding features of this approach are:

- The most opportune moment at which a task becomes considered schedulable is an analytically derived parameter, named the *punctual point*.
- Well-Timed Scheduling presents a framework for on-line real-time schedulers, and can be used as a design and as a run-time tool.
- Well-Timed Scheduling is applied to a wide range on $M/G/c$ real-time systems.

The major benefits of utilizing Well-Timed Scheduling are excellent performance and control of the system under temporary and permanent overloads, as well as graceful degradation.

In the second half of the dissertation, we develop off-line preprocessing algorithms that enable effective and efficient on-line scheduling of task groups with

different contributing values, timing constraints, resource constraints, and arbitrary precedence constraints. Specifically, we develop value density[1] propagation algorithms which derive value densities that reflect the value densities of individual tasks and their successors. By utilizing these *reflective* value densities, on-line schedulers do not need to traverse the all task groups to select the best task to schedule next. As a result, the on-line schedulers do not increase in computational complexity when scheduling tasks with precedence constraints.

In short, the main theme of this dissertation is the development of scheduling approaches that enable efficient and effective on-line scheduling real-time systems that operate in highly dynamic and non-deterministic environments, and where the issues of reliability, flexibility, and predictability are of utmost importance. Specifically, the methods proposed in this dissertation improve efficiency of dynamic scheduling algorithms used in real-time systems by reducing the number of tasks considered in making scheduling decision.

1.1 Motivation

An example of a real-time application of the future that operates in very complex and non-deterministic environment is the robot Dante[2] (named after the central character of Dante's classic *The Divine Comedy: Inferno*[3]). This spiderlike robot was developed for the task of exploring the active volcano at Mount Erebus, in the Antarctic. This project sponsored by NASA and the National Science Foundation is a part of an extensive preparation for a much more complex mission to the planet Mars—where robots will perform without human guidance for all but long term tasks.

---

[1]A task's value density is defined as its contributing value over its processing time.

[2]Dante is a result of the joined effort between Carnegie-Mellon University and New Mexico Institute of Mining and Technology.

On January 2, 1993, Dante started its trip down the side of the volcanic crater connected by a fiber-optic umbilical cable [4]. Clearly, the site chosen for this test is highly dynamic and non-deterministic, where the external environment can potentially trigger a large number of essential tasks. For example, Dante has to stay away from the lava lakes, and it has to avoid the flying rocks. The contributing values and the timing characteristics of the tasks invoked depend not only on the predefined mission objective but also on the environment's behavior and Dante's physical characteristics. It is easy to envision that due to Dante's maximum speed of 3.3 feet per minute, its need to process the images and signals from the very active environment, as well as the limitations of available processing power, a potentially large number of complex tasks with very stringent timing constraints can be simultaneously invoked. To successfully cope with such requirements—with minimal or no human guidance—it is necessary to provide methods that can efficiently deal with a potentially large number of complex tasks. That is, it is necessary to provide methods that do not use complex, high overhead mechanisms at run-time and that provide sufficient preparatory work that will lead to efficient and effective use of system's resources.

Expected to perform in non-deterministic environments, real-time systems of the future *must* continue to operate in temporary and permanent overloads, must be capable of controlling the overall system performance, and must exhibit graceful degradation capabilities. To achieve this, these systems must not only detect that some task did not meet its deadline, but they also must foresee that an essential task will not meet its deadline, signal it on time, and substitute it with one or more contingency tasks. To identify that the task cannot meet its timing constraints, an on-line schedulability analysis with an early warning feature is required. The early recognition that the task will not meet its deadline should provide enough *lead* time

for the timely invocation of contingency tasks, and system level planning activities to adapt the system to a continuously changing environment.

1.2 Our Approach

This dissertation presents our contribution to the problem of comprehensive scheduling capabilities of real-time systems of future. There are two major parts of this dissertation: (1) the Well-Timed Scheduling part, and (2) the scheduling with precedence constraints.

1.2.1 Well-Timed Scheduling

Many real-time systems are still being built today under the misconception that real-time computation is fast computing [54]. Furthermore, designers of these systems sometimes assume that computation times and resource requirements are too difficult to assess, so this information is not utilized in scheduling. Rather, a myopic scheduling algorithm is used which simply selects the next task to run and immediately executes it—based on some simple task characteristic such as deadline. There is no understanding of the schedulability of the task or of the projected overall performance for given system loads and tasks' timing requirements. In other words, for all but the simplest real-time systems it is often assumed that schedulability analysis is too difficult.

However, schedulability analysis can be performed if task information is *required* to be known *a priori*. Given tasks' timing constraints, their computation time, resource requirements and other constraints, schedulability analysis can be performed either off-line or on-line. For example, in the case of static priority schedulers, such as rate-monotonic schedulers [31, 47], an off-line schedulability analysis that provides a 100% guarantee is performed. This type of analysis is well suited to small, highly periodic applications. On the other hand, assumptions and restrictions imposed

by off-line approaches can lead to the creation of systems which are inflexible, not expandable, and prone to overdesign.

As recently pointed out in [26], if today's real-time systems are designed in a simple and static manner, what good is a 100% guarantee if there is a high probability that the assumptions and imposed restrictions, upon which the off-line guarantee is based, will be violated? In other words, if the assumptions are likely to be violated, a system using a static approach will not perform as designed, thus, causing tasks to miss their deadlines. What is even worse, the control of the system can be lost completely due to the *limited* scheduling capabilities.

When constraints imposed by an off-line schedulability analysis are likely to be violated, dynamic scheduling approaches are the solution. They offer a higher degree of flexibility and can be made quite general. Very often, the objective of dynamic algorithms is not to provide a 100% guarantee, but rather to maximize the system's accrued value.

Each task contributes some value to the system. Typically, a value is described using a time-value function, which relates a task's value to the timing requirements [32, 7]. In the case when all tasks have the same value, maximizing the accrued value is the same as maximizing the number of completed tasks. Examples of such algorithms can be found in [42] and [32].

The approach presented in [42] employs the notion of guarantee on a per task basis. That is, in this approach, a newly arrived task is scheduled and accepted for execution only if it can be guaranteed to complete by its deadline—exemplifying the scheduling-at-arrival-time policy. On the other hand, by associating the notion of guarantee with the approach presented in [32], for example, the scheduling-at-dispatch-time is obtained.

Guaranteeing at dispatch time might be too late to take alternative actions, while guaranteeing at arrival time can be unnecessarily too early. In spite of

the applicability of both approaches to complex real-time systems, one of their shortcomings is the lack of an analytical technique to *quantify* the overall guarantee. That is, they lack the capability to calculate the probability of meeting all given constraints rather than producing a simple guaranteed/not-guaranteed response on a per task basis.

In the first half of the dissertation, we provide an analytical technique to quantify the guarantees for dynamic real-time systems with on-line scheduling algorithms. Furthermore, we present how to utilize the analytically obtained results in on-line schedulers to obtain effective and efficient scheduling over a wide range of system loads.

1.2.1.1 The Punctual Point

One of the primary concerns of on-line schedulers is their computational efficiency. Our approach is driven by this concern. The efficiency of on-line schedulers is increased by preventing the unnecessary scheduling (or rescheduling) of "irrelevant" tasks—the tasks with very large laxities[3] that have no impact on the order of tasks at the top of the schedule. Specifically, instead of scheduling tasks when they arrive or when they are dispatched, tasks are scheduled somewhere in between. They are scheduled at the so-called *punctual point.*

The main benefit of scheduling using the punctual points is the reduced scheduling overhead when compared to scheduling-at-arrival-time. This is due to the smaller number of "relevant" tasks (the tasks with laxities smaller than or equal to the punctual point) that are scheduled at any given time. Clearly, when the computational complexity of a scheduling algorithm is higher than the complexity of

[3]Task laxity, also known as task slack time, is the maximum time a task can be delayed before its execution begins. In our model, we assume that for each task the worst case computation time and the laxity are known at scheduling time; thus, an absolute task deadline is obtained from the sum of arrival time, computation time, and laxity.

maintaining the list of "relevant" tasks, the separation into "relevant"/"irrelevant" tasks reduces the overall scheduling cost; that is, the scheduling becomes more efficient.

Scheduling-at-opportune-time (that is at the punctual point) is more flexible, more effective, and more tolerant to timing errors than scheduling-at-dispatch-time, primarily, due to its early warning characteristics.

The punctual point as a control parameter of the scheduling overhead for on-line real-time schedulers is the most prominent component of our Well-Timed Scheduling.

Due to its characteristics, Well-Timed Scheduling is useful both at design time and at run-time. In the design phase, by knowing the system parameters, the number of processors required to achieve a desired level of guaranteed performance can be determined. Specifically, at design time, the number of processors can be determined by calculating the punctual point that matches the given mean laxity and produces the desired system performance. On the other hand, at run-time, the punctual point can be dynamically adjusted to respond to changing system loads.

In summary, Well-Timed Scheduling does not enforce any particular scheduling policy *per se*, it rather provides a *framework* for dynamic scheduling algorithms based on a methodical approach to schedulability analysis.

1.2.2 Scheduling with Precedence Constraints

Beside independent tasks, complex real-time systems usually have tasks interrelated by precedence constraints. This interrelation is commonly described by a given directed graph where nodes present tasks and arcs present the execution dependences.

If a task group requires that all of its tasks must complete, the task group is characterized as an *atomic* task group. These task groups must be scheduled end-to-end; and once they have started execution, they can not be "unscheduled". If

the atomic task group is aborted—due to the arrival of more important task groups, for example—all already executed tasks must be rolled back. Examples of such task groups can be found in systems where monolithic processes are decomposed into tasks to provide more efficient resource use.

On the other hand, the *non-atomic* task groups have well-defined tasks that execute within their own context, and do not require end-to-end scheduling. These tasks inherently provide high level of parallelism, resource utilization, and do not necessarily require roll back if severed (that is, if the execution of a task group, starting from some task and including all its successors, is canceled). Examples of such task groups are found in systems that support imprecise computations. The contributed value of non-atomic task groups is the sum of contributing values of all completed tasks, rather than the contributed value of the entire task group as for atomic task groups.

In contrast to atomic task groups, non-atomic task groups are better suited for dynamic, non-deterministic environments, where for a good system performance very valuable tasks should not be rejected because some low value, but very long task group started its execution. Non-atomic task groups provide more flexibility in scheduling; they do not have any other interrelation constraints but the precedence constraints; and they can be partially executed, i.e., severed.

Greater flexibility and the larger number of feasible schedules of non-atomic task groups has its drawbacks—a potentially high scheduling complexity, for example. Specifically, aside from focusing on the groups' characteristics only (such as precedence, resource, and timing constraints), a scheduler must anticipate the arrival of more valuable tasks. It has to select not only the best task group but also the best branch within the selected group. In this case, the scheduler must examine all active task groups and all their branches just to schedule the next task. This extensive search is necessary because in non-atomic task groups a very valuable task

can easily be preceded by tasks with very low values. In this case, if a scheduler does not examine each branch, a very valuable task might miss its deadline because of the myopic scheduling approach that focuses only on the eligible set of tasks, where eligible tasks are the tasks whose all predecessors have already been executed or scheduled.

Clearly, the system value is maximized if a scheduler either traverses down the unscheduled branches or if each task contains reflective parameters that describe its successors. From the real-time perspective, the letter choice provides a better solution. It is less computationally expensive, and it does not necessarily perform worse than the former choice.

Recognizing this, we develop the off-line preprocessing algorithms that prepare task groups for on-line schedulers that select a task to schedule next based on the eligible set of tasks only. The advantages of this approach are multiple. The scheduling algorithm has the computational complexity of the scheduling algorithm for independent tasks. Already developed real-time scheduling algorithms for independent tasks that are demonstrated as effective can be utilized to obtain a high system performance.

Our preprocessing algorithms assign so-called *reflective* parameters to individual tasks, based on given precedence constraints and initial task parameters. The obtained reflective parameters embed the information about the tasks' initial parameters and the initial parameters of their more valuable successors. If a task is more valuable than its successors its reflective parameters are the same as its initial parameters.

It is important to observe that the separation of value density preprocessing from the other preprocessing procedures, such as deadline preprocessing, provides an additional advantage of our approach. Due to this separation, our preprocessing algorithms are applicable to both real-time and non real-time task groups.

1.3 Dissertation Organization

The remainder of this dissertation is organized as follows. In Chapter 2, an analytical derivation of the punctual points for $M/G/c$ systems is presented, and validated through simulation. This is followed by the analysis of the effects of the punctual points on the number of "relevant" (schedulable) tasks, and by the analysis of the relation between the punctual points and laxities. Some specific design considerations for $M/M/c$ real-time systems are presented at the end of this chapter.

The benefits of an integration of the Well-Timed Scheduling and real-time scheduling algorithms is examined in Chapter 3. Specifically, two real-time scheduling algorithms are developed and integrated with Well-Timed Scheduling. The first algorithm is the DLVD algorithm that schedules the tasks in increasing *deadline* order, and in the case of an infeasible schedule, it rejects a task with *minimum value density*. The second algorithm is the RDS algorithm where a *linear combination* of task's value density, deadline, and earliest possible start time is used to select a task to schedule next. In the case of an infeasible schedule, a selected task is rejected. The presented integration analysis consists of four parts: (1) the analysis of both real-time schedulers without the benefits of the Well-Timed Scheduling and with negligibly small scheduling costs—a traditional application with a minimal scheduling cost, (2) the analysis of the effects of the increasing scheduling costs, and still without the benefits of the Well-Timed Scheduling, (3) the effects of the Well-Timed Scheduling with realistic scheduling costs, and finally, (4) the applicability range and robustness analysis of the proposed integration. This chapter concludes the Well-Timed Scheduling part of the dissertation.

Our approach to the problem of scheduling the tasks with precedence constraints is presented in the remainder of the dissertation. The overall strategy, and the development of two off-line preprocessing algorithms: (1) the VDP-R algorithm,

developed for the rooted tree precedence constraints, and (2) the VDP-G algorithm, developed for the arbitrary precedence constraints are presented in Chapter 4.

In Chapter 5, we experimentally evaluate the effects of the off-line derived reflective parameters—applied to complex scheduling problem, such as problems of scheduling arbitrary task groups with different contributing values, timing and resource constraints in a multiprocessor environment. The evaluation is performed along multiple dimensions, including tests for:

- a wide range of task group laxities,
- a number of system loads (ranging from moderate and high loads to heavy overloads),
- different task group sizes,
- different number of application processors,
- three types of task groups: in-tree, out-tree, and task groups with arbitrary precedence constraints,
- three types of value assignments: (1) a top-heavy assignment, where the source tasks are the most valuable tasks and the terminal tasks are the least valuable tasks, (2) a bottom-heavy assignment, a reverse of the top-heavy assignment, and (3) a random value assignment,
- two types of time-value functions: a step value function, and a linearly diminishing value function, and finally,
- tasks are scheduled using: (1) initially assigned parameters, (2) the group parameters, and (3) off-line derived reflective parameters.

The analysis of the simulation results consists of three parts: (1) the analysis of systems with out-tree task groups, (2) the analysis of systems with in-tree task groups, and (3) the analysis of systems with arbitrary task groups.

The precedence constraints scheduling part is concluded by presenting the summary of general learnings and observed trends.

Chapter 6 summarizes the main contributions of the dissertation, and it outlines the directions for future research.

CHAPTER 2

Well-Timed Scheduling: A Framework for Dynamic Real-Time Schedulers

2.1 Introduction

Most of the systems that use an on-line schedulability analysis perform the analysis on a task by task basis or by simulation. In the simplest form, an assessment, whether a task will meet its deadline or not, is postponed until dispatch time. One alternative is to attempt to *plan* the execution of a newly arrived task, in conjunction with previously scheduled tasks as soon as the task arrives. Both classes of scheduling algorithms—whether performing a simple form of feasibility check at dispatch time, or a more elaborate guarantee at task arrival—avoid wasting resources, and execute tasks in a non-preemptive fashion.

Typically, algorithms that perform scheduling-at-arrival-time are computationally more expensive but, in return, they offer very high flexibility. These systems employ a guarantee routine that checks feasibility of a schedule in order to accommodate a newly arrived task, thus, intrinsically they provide a notion of early warning for the tasks that cannot meet their timing constraints [55]. On the other hand, scheduling-at-dispatch-time, while usually being very fast and simple is less flexible and fails to provide a system view of the load; it announces task rejection too late—either when its deadline has already expired or when its laxity becomes negative while waiting for execution. Consequently, it does not provide sufficient "lead" time for scheduling alternative actions when a task can not meet its timing constraints. This indicates that support for negotiability, based on *a priori* available

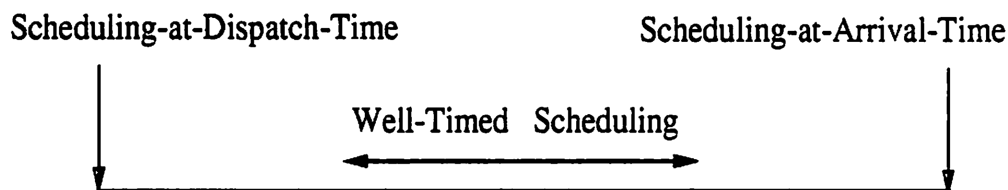

Figure 2.1 Approaches to Dynamic Scheduling Analysis.

knowledge of a task's timing and other constraints, is inherent to early warning schedulers, and not to scheduling-at-dispatch-time schedulers.

Well-Timed Scheduling is adaptive with respect to when the schedulability analysis is performed. It does not require that scheduling be performed at the earliest possible point (at task arrival time), or the latest possible point (at dispatch time). Scheduling is performed at the most opportune moment, which is primarily a function of the system load. This ensures that a scheduling decision is made earlier than with scheduling-at-dispatch-time, but not necessarily as early as with scheduling-at-arrival-time (see Figure 2.1). Consequently, "lead" time for alternative actions is adjustable, and it is based on design and run-time parameters.

Controlling the number of schedulable tasks by using the timing constraints, rather than by explicitly limiting the number of schedulable tasks, is a novel approach to achieving low scheduling overhead and high performance, for on-line scheduling algorithms. Our approach is especially beneficial for systems where tasks have different contributing values. In these systems, rejecting a task without considering it for scheduling might result in a large value loss. This can easily happen in approaches with fixed number of schedulable tasks. On the other hand, our approach guarantees that every task is considered for scheduling when its laxity reaches the most opportune moment, the punctual point.

The approaches that use simulation to bound the number of scheduled tasks are presented in [18, 22]. Both papers examine the performance of variants of the Minimum Laxity First scheduling policy—the policy that has been shown to be optimal, with respect to minimizing the long-term, steady-state fraction of lost jobs, over all work-conserving non-preemptive policies [36, 37]. The analyzed scheduling algorithms use a two part queue: the first part of the queue is a dispatch queue, called $Q_1(n)$, maintained in minimum laxity order, while the second part is a variant of the $FCFS$ queue. The laxity of a newly arrived task is compared with laxities of n tasks in $Q_1(n)$ and the task with the largest laxity is placed at the end of the $FCFS$ queue. When a task in Q_1 is executed, the top task from $FCFS$ queue is enqueued in Q_1. Their analysis indicates that a performance within 5% of the optimal (Minimum Laxity First) algorithm is achieved even for the small n, namely for $n = 5$.

A more experimental and less theoretical approach to the control of scheduled overhead, also based on bounding the number of scheduled tasks, is presented in [19]. They propose two scheduling algorithms for Real-Time Database Systems especially designed to handle overloaded situations. Namely, they develop the Adaptive Earliest Deadline and Hierarchical Earliest Deadline algorithms. In this approach, the overall queue is, again, divided into two parts, called the HIT queue and the $MISS$ queue. The number of scheduled tasks, located in the HIT queue, is continuously adjusted according to the so-called *HitRatio*. As a result, this method is very adaptive, handles deadlines and values, and appears to be very practical.

However, the weakness of both of these approaches is the lack of analytical methods to adjust the number of scheduled tasks. In both approaches, the parameters that control the number of schedulable tasks are obtained through simulation; and in both approaches a newly arrived task can miss its deadline before it gets considered for execution. In contrast, in Well-Timed Scheduling, the mechanism

that controls scheduling overhead—the punctual point—is derived analytically, and it ensures that every arrived task will be considered for execution.

The rest of this chapter is organized as follows. An analytical derivation of the punctual points for $M/G/c$ systems is presented in Section 2.2. This is followed, in Section 2.3, by a validation through simulation and the analysis of the effects of the punctual points on the number of "relevant" tasks. Section 2.4 relates the punctual points to the laxity values, and presents some specific design considerations for $M/M/c$ real-time systems. The conclusions is presented in Section 2.5.

2.2 Punctual Point Derivation

The punctual point is designed to prevent early scheduling of "irrelevant" tasks. Any task that has a laxity larger than the punctual point is considered too early for immediate scheduling, whereas tasks with laxities smaller than the punctual point are subjected to scheduling. This approach is especially important for systems where an overload is expected to occur, or for systems which are designed to perform in overloaded environments[1]. In this paper, the term *load* has the same meaning as in standard queueing theory: a load, ρ, also called traffic intensity, represents the expected number of task arrivals per mean service time in the limit.

The derivation of the punctual points is divided into two cases: (1) derivation for loads less than 1, and (2) derivation for loads greater than 1. The second case reduces to the first one by using a mapping procedure that pairs a given load $\rho_h > 1$ with a load $\rho_l < 1$. The idea behind the mapping is to find two loads whose task loss ratios saturate at the same point. (Task loss ratio is defined as the ratio of tasks that miss their deadlines to the total number of submitted tasks.)

[1]An overload in real-time systems that operate in dynamic environments can occur due to the failure of processor(s), the increased number of requested actions, or simply due to environment changes. On the other hand, by assigning the values to the tasks according to their functionality and importance, real-time systems can be designed to deal with overloads.

Table 2.1 Punctual Points for $\psi = 0.99$, and 0.999 Probabilities.

ρ	0.5	0.7	0.9	0.99
$T_P(0.99)$	10	16	50	374
$T_P(0.999)$	14	22	70	396

2.2.1 Punctual Point Derivation for $\rho < 1$

The derivation of the punctual point for loads less than 1 is a simple and straightforward procedure based on recent queueing theory results—specifically, on the results for waiting time distribution in $M/G/c$ queueing systems.

The computation of the waiting time distribution of $M/G/c$ queueing system, for given load and given service time distribution, was first presented in [60]. (For more details on the computation of the waiting time distribution refer to Appendix B.) As an example, a complementary waiting time distribution for the $M/M/1$ system is given in Figure 2.2. The graph presents the probability with which an incoming task will wait for service more than a given time t, i.e., $1 - \psi = Prob\{W_q > t\}$. The time obtained for any given probability ψ is called the ***punctual point***, denoted as $T_P(\psi)$.

The waiting time distributions, and thus the punctual points, are directly related to real-time systems. When the *minimum* laxity of any task in a given real-time system is at least as large as the punctual point $T_P(\psi)$, tasks will meet their deadlines under the $FCFS$ scheduling policy with probability ψ. (The general interpretation of the ψ probability is discussed in Section 2.4.)

To illustrate how the increase of ψ near the saturation point—the point that has very low impact on the overall system performance—affects the $T_P(\psi)$, we analyze the punctual points for the $\psi = 0.99$ and $\psi = 0.999$ probabilities. Table 2.1 lists the punctual points that are derived for moderate and heavy traffic intensities, i.e., for $\rho = 0.5, 0.7, 0.9, 0.99$ loads. As shown in Table 2.1, when the load is $\rho = 0.5$, it is

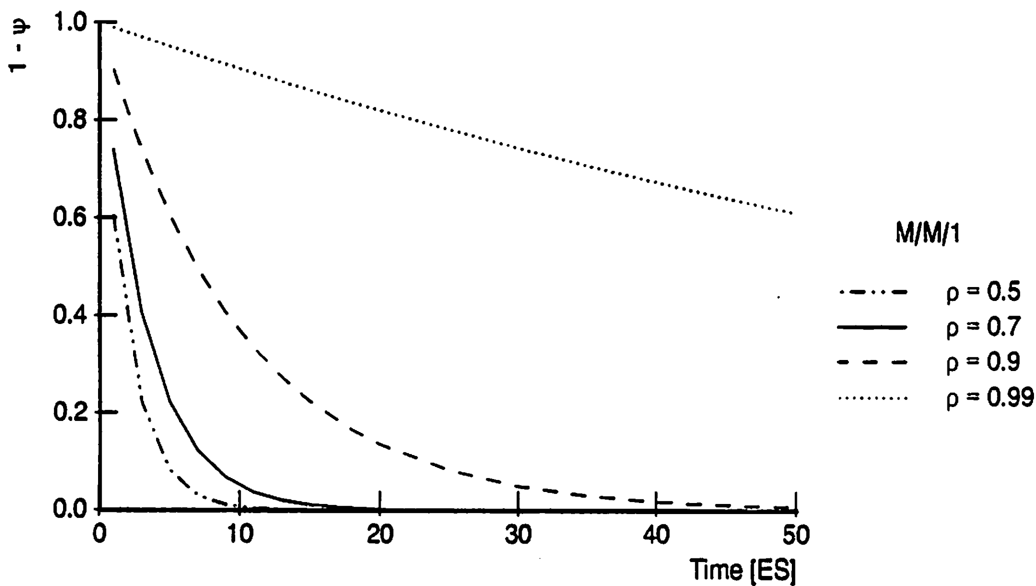

Figure 2.2 Complementary Waiting Time Distribution for $\rho < 1$.

enough to have the minimum laxity of $10ES^2$ in order to guarantee that all deadlines will be met with a probability of 99%. Further, when the load approaches $\rho = 1$, the waiting time increases exponentially. For example, when $\rho = 0.9$, the punctual point $T_P(0.99) = 50ES$ while when $\rho = 0.99$, $T_P(0.99) = 374$. This implies that real-time systems with increasing load and constant ψ probability of meeting deadlines must increase the minimum laxity requirements, correspondingly (i.e., exponentially).

Another way to look at the data in Table 2.1 is to assume that the average load is fixed and that the requirement for ψ probability changes from $\psi = 0.99$ to $\psi = 0.999$. Here the increase in the minimum laxity requirements is relatively stable. Specifically, for $\rho = 0.5, 0.7, 0.9$, the increase is circa 40%, while for $\rho = 0.99$, it is only 5.9%.

---

[2]In this case, as in all following examples, the normalized expected service time is assumed ($ES = 1$), and for brevity ES is omitted when giving the punctual points.

This concludes the discussion of the punctual point for loads $\rho < 1$. The next subsection describes the mapping procedure necessary to perform the punctual point derivation for loads $\rho > 1$.

2.2.2 Punctual Point Derivation for $\rho > 1$

When the load is greater than one, the calculation of the punctual point using the above given waiting time distribution is not feasible. The number of tasks in a waiting area grows to infinity. To overcome this, we developed a unique mapping procedure that successfully pairs loads $\rho_h > 1$ with loads $\rho_l < 1$. Once this mapping is done, the actual punctual point is easily derived using the procedure described in the previous subsection. The motivation and intuition behind our mapping procedure is presented next. Firts, we discuss the existing analytical results for $M/M/1 + M + FCFS$ systems. Second, we present a new interpretation of the well known results for $M/G/c/K$ systems. Third, the actual description of the mapping procedure is presented. The section is concluded by presenting another, simpler and less accurate method for the derivation of the punctual points. The experimental results that support the mapping method are presented in Section 2.3.

In [65], Zhao and Stankovic analyze the performance of FCFS and improved FCFS real-time scheduling algorithms. They derive a useful expression for task loss ratio, R, for the $M/M/1 + M + FCFS$ real-time queueing system (the system with Poisson arrival times, exponential service times, single processor, exponential laxity distribution and a FCFS scheduling policy). The task loss ratio R was shown to be:

$$R = R^+ \left(1 - \frac{1 - \rho(1 - R^+)}{1 + \rho R^+}\right), \tag{2.1}$$

where R^+ is defined as

$$R^+ = 1 - \frac{\int_0^\infty e^{-(\mu+l)x - \frac{\lambda}{l}e^{-lx}}\,dx}{\int_0^\infty e^{-\mu x - \frac{\lambda}{l}e^{-lx}}\,dx}, \tag{2.2}$$

and where l is the mean laxity, ρ is the traffic intensity, λ is the arrival rate and μ is the service rate, for all possible values of the unfinished work x. In [65], the

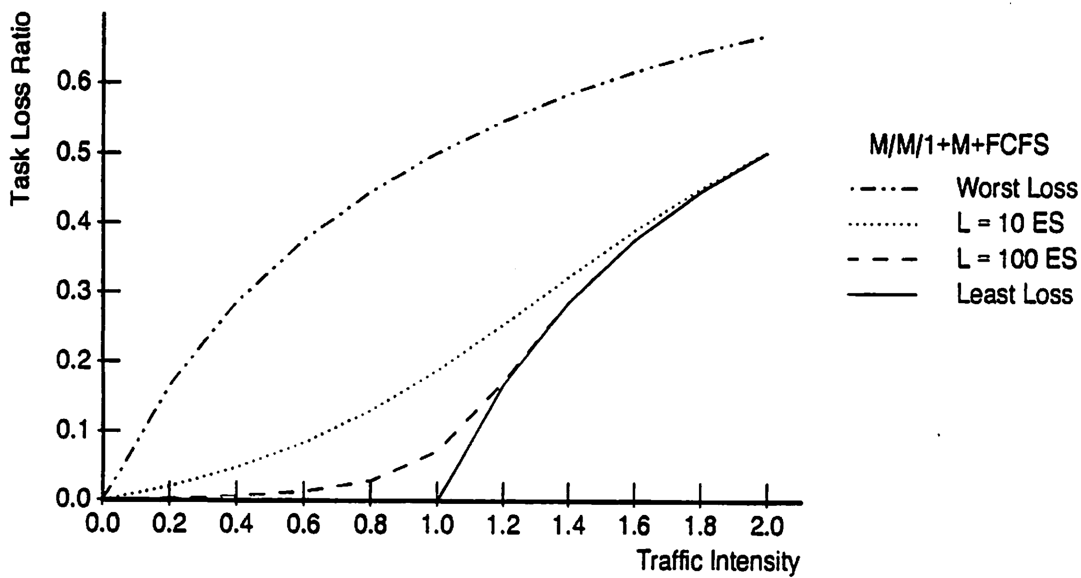

Figure 2.3 Task Loss Ratios for Real-time Systems [65]

above expression is refined and given in the form of an incomplete gamma function. However, notice that the above integrals can be directly solved by Gauss-Laguerre quadrature[3]. For details of this numerical integration see Appendix C. Furthermore, the boundaries for task loss ratio are given by two standard expressions. The worst expected task loss ratio, assuming zero laxity, is given by

$$R_{worst} = \frac{\rho}{1+\rho}.$$

The least possible task loss ratio, assuming laxities approaching infinity, is given by

$$R_{least} = \begin{cases} 0 & \text{if } \rho < 1 \\ 1 - \frac{1}{\rho} & \text{if } \rho \geq 1. \end{cases}$$

Figure 2.3 reproduces the results from [65] with a slight modification. The task loss ratios are plotted for large laxities, such as $l = 10ES$ and $l = 100ES$, rather than for small laxities, such as $l = 0.1ES$, $l = 1ES$, and $l = 10ES$ which appeared

[3]The numerical integration of equation 2.2 gives an approximation of the solution. This is due to the non-polynomial function of the given integrals; for example, $F(z) = e^{-\mu z - \frac{\lambda}{l} e^{-lz}}$ for the integral in the denominator.

in the original paper. Our intention is to illustrate that even for very large laxities and with zero scheduling cost, the task loss ratio is relatively far from R_{least} for loads around $\rho = 1$ (see curve for $l = 100ES$) Note that increase in processor speed mainly influences the execution times of real-time tasks. On the other hand, deadlines are usually driven by physical constraints of the process under control, and as such, they are not influenced by the constantly increasing processing speeds. An increase of computation power produces shorter worst case computation times, and thereby increases laxities, creating more room for the applicability of the fundamental results for large laxities.

Analyzing the task loss ratio curves from Figure 2.3, it is clear that a system with larger laxities has a smaller task loss ratio. This result is not surprising. However, a closer observation shows that $l = 10ES$ and $l = 100ES$ curves converge at loads $\rho \geq 2$, at the high end, and at loads $\rho \leq 0.02$, at the low end. This indicates that for loads after the convergence point, the $l = 10ES$ system has the same loss ratio as the $l = 100ES$ system, in spite of the fact that its laxity is one order of magnitude smaller. Intuitively, it can be concluded that a system with larger laxities can be treated as the system with lower laxities after the point of convergence, without a significant decrease in overall performance. Loosely speaking, in the example with laxities $l = 100$, if the load is $\rho \geq 2$ then an incoming task will not be subjected to scheduling until its laxity drops to $10ES$. That is, a task will be scheduled when its laxity drops to the *punctual point.* Similarly, the punctual points for other loads can be determined by determining the laxity values at which the task loss ratios, for given laxity distributions, converge to R_{least}.

To apply this idea beyond the $M/M/1 + M + FCFS$ systems, we next present the theoretical results for $M/G/c/K$ queueing systems—the systems with finite buffers. By giving a fresh look to these classical results, we provide the basis for

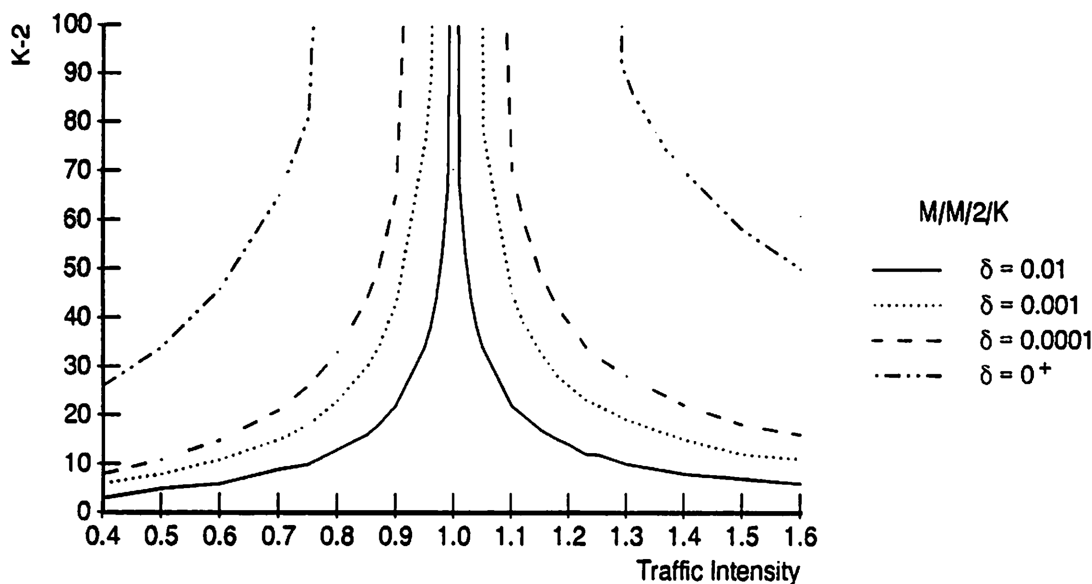

Figure 2.4 K-size Distribution.

our mapping procedure. (For clarity of presentation, all relevant mathematics is presented in Appendix C.)

Let us define δ to be a user provided parameter that specifies the difference between the task loss probability p_K and the least possible task loss ratio R_{least}, i.e., $\delta = p_K - R_{least}$. Using a fixed value of parameter δ we can easily obtain a buffer size distribution, over the entire range of loads. This distribution is referred to as the K-size distribution, and it presents the smallest required buffer sizes such that $p_K - R_{least} \leq \delta$. An example for the $M/M/2/K$ queueing system, where the waiting area size is plotted against the offered load, for four very demanding δ values, namely 0^+, 0.0001, 0.001, and 0.01, is given in Figure 2.4. (Note that K is the total number of tasks in the system, and so, $K-2$ is the number of tasks in the waiting area of a two processor system.)

Intuitively, an interpretation of the K-size distribution indicates the following:

- As the load approaches zero, the number of arrived tasks decreases, and with it, the waiting area size needed to keep a task loss ratio within δ distance from R_{least} decreases, too.

- When load $\rho \rightarrow 1^{-}$, a large waiting area size is needed to maintain a desired performance. This is caused by a dramatic increase in task arrivals, and by the fact that in the R_{least} case all the tasks would be served. Thus, a large buffer area is required to accommodate all incoming tasks.

- In the case of high overloads ($\rho > 1$), new tasks arrive faster than they can be served. Thus, the waiting area is going to be "full" at all times. The larger the load, the smaller buffer size is meeded.

The $\rho > 1$ case indicates that for increasing overloads a reduced waiting area size is sufficient to always keep the system busy, and thus to perform as specified. Maintaining the loss probability δ distance, an almost mirror image of K-size distribution is created around $\rho = 1$, with a point of discontinuity at $\rho = 1$, where $K \rightarrow \infty$.

By revisiting the results presented in Figure 2.3, we notice that in this case only systems with laxity $l \rightarrow \infty$ are going to perform the same as R_{least}. This indicates that for any finite laxity values, the scheduling algorithm must schedule *all* ready-to-run tasks in order to achieve its best performance, thus yielding a very large scheduling overhead. This further suggests that the most difficult scheduling problems, especially for FCFS scheduling, are for loads around 1, where more complex scheduling algorithms should be used.

As mentioned above, to maintain a given tolerance δ under increasing loads (for $\rho > 1$), the needed waiting area size decreases. This is consistent with the earlier observation for $M/M/1 + M + FCFS$ system (Figure 2.3). That is, as the load increases beyond the point of convergence, systems with lower laxities do not

GIVEN: (1) Parameters for $M/G/c$ system, and (2) load $\rho_h > 1$

STEP 1

GOAL: Find rejection probabilities p_K of an $M/G/c/K$ system, such that $(p_K - R_{least}) \leq \delta$.

PROCEDURE: While increasing K and decreasing δ, calculate p_K using equations C.1 and C.2 from Appendix D. Stop when the largest possible K is reached for the smallest possible δ.

RESULT: The buffer size K and parameter δ are determined.

STEP 2

GOAL: Find $\rho_l < 1$, such that $(p_K - R_{least}) \leq \delta$ for K and δ from STEP 1.

PROCEDURE: Calculate p_K using equations C.1 and C.2 from Appendix C by varying the load. Stop when $p_K - R_{least}$ is closest to δ.

RESULT: The peer load ρ_l is found.

Figure 2.5 Mapping Procedure.

perform significantly worse (as compared to the best that they can do) than systems with larger laxities.

The actual mapping procedure that utilizes the above observation is outlined in Figure 2.5. Its goal is to find a peer load $\rho_l < 1$, for a given load $\rho_h > 1$ of an $M/G/c$ system, such that the task loss probabilities saturate at the same punctual point. For greater accuracy of the mapping procedure, the minimum possible value of the δ parameter should be used at all times. This value should be such that the computation always provides the largest possible value of a buffer size K. The expression that limits the largest possible value is presented in Appendix C.

The mapping procedure from Figure 2.5 is best used for punctual point calculations during design time. To make the mapping method computationally effective

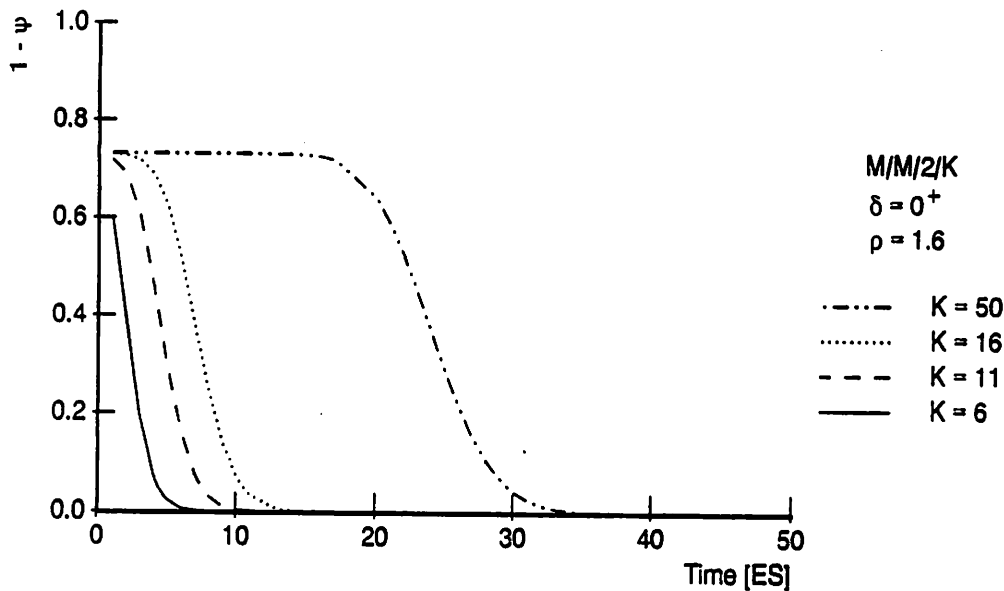

Figure 2.6 Complementary Waiting Time Distribution for Finite Buffer Systems.

Table 2.2 Punctual Points for $M/M/2/K$ Systems with $\rho = 1.6$ Load.

δ	0^+	0.0001	0.001	0.01
K	50	16	11	6
$T_P(1^-)$	44	20	16	11.6
$T_P(0.999)$	36.2	14.7	11.2	7.5
$T_P(0.95)$	29.7	10.5	7.5	4.4

during run-time, two methods are recommended. First method, where the pairs of loads that require the same buffer size in a K-size distribution are tabulated off-line. During run-time, if the loads are not found in the table, an interpolation method can be used. Second, much faster and very accurate method where $\rho_l = 1/\rho_h$. This method exploits the almost mirror like characteristic of the K-size distribution to determine peer loads. For all practical overloads, the loads up to several times the system capacity, this last method provides an excellent approximation.

To conclude this section, it is important to mention that the waiting time distribution for $M/M/c/K$ systems offers an alternative algorithm for the punctual point calculation over all loads. As reported in [56], the numerical investigation indicates that for the case of $\rho \geq 1$ the $M/M/c/K$ queue provides a reasonably good approximation to the $M/G/c/K$ queue. (State of the art of queueing theory does not provide an expression that describes waiting time distribution of $M/G/c/K$ queueing systems.) In [56], the following expression is used to compute the waiting time distribution for $M/M/c/K$ system:

$$P\{W_q > t\} = \frac{1}{1-p_K} \sum_{j=c}^{K-1} p_j \sum_{i=0}^{j-c} e^{-c\mu t} \frac{(c\mu t)^i}{i!}, \qquad t \geq 0. \qquad (2.3)$$

The state probabilities are calculated according to equations C.1 and C.2 from Appendix C. Clearly, using this method the calculation of the punctual points, for any given probability ψ, can be found directly from equation 2.3. An illustration of the waiting time distributions for different K values and load $\rho = 1.6$ is given in Figure 2.6. For a given δ, K value is derived using the K-size distribution method as in Figure 2.4. Table 2.2 gives the K values, the calculated punctual points for given ψ probabilities of $\psi = 0.999$. It is important to notice the dramatic impact of the tolerance parameter δ on the calculation of the punctual points. Due to the high variation of computed punctual points, this method, even though simpler and faster, is considered to be less accurate than the former one. The high variance is mainly due to the heavy dependence on a chosen buffer size K, which in turn depends on a choice of the parameter δ. Therefore, we recommend the earlier described mapping method based on the K-size distribution and mapping procedure from Figure 2.5. Specifically, the $\rho_l = 1/\rho_h$ mapping approximation.

2.3 Comparison of Analytical and Experimental Results

To observe the behavior and to understand the relationship among laxity, schedule length and the punctual point, analytical results were compared with experimental ones. For this purpose, a shared memory multiprocessor system with work-conserving $FCFS$ scheduling policy was used. By varying the task laxities, we measured task loss over a wide range of loads. In general, our experiments for these $M/G/c$ real-time systems indicate the following: (1) for every load $\rho > 1$ there exists a load $\rho < 1$ whose task loss ratio saturates at the same laxity value, and (2) the saturation point of experimentally obtained task loss ratio corresponds to the analytically derived punctual points.

Specifically, in this section, the results for $M/M/c+M+FCFS$ and $M/M/c+E_k+FCFS$ systems are presented. In these real-time systems, a task will start its execution only if its deadline can be met. Each task is assigned a laxity from some given exponential or Erlang-k distribution[4]. To cover a wide range of applications, a "shape parameter k" is varied as $k = 1, 3, 100$. The results show that it is sufficient to focus only on the extreme cases, namely, on Erlang-1 and Erlang-100 distributions. (The Erlang-1 distribution reduces to the exponential distribution, achieving standard deviation equal to its mean; while the Erlang-100 approaches the degenerate distribution with standard deviation of only 0.1 times its mean.) Additionally, In addition to varying the laxity distribution, the number of processors is varied, too. The results for systems with 2 and 8 processors are presented. All experimental results are obtained from runs of over 5000 task arrivals, and the results are plotted with 95% confidence intervals.

---

[4]Erlang distribution has two very useful properties: (1) Aside from the "scale factor", which determines the mean, the distribution has the "shape parameter k" that determines the degree of variability. Therefore, many empirically observed distributions can be reasonably approximated by an Erlang distribution. (2) A random variable with an Erlang distribution can be represented as the sum of k independent random variables having a common exponential distribution.

To validate our mapping procedure and derivation of the punctual points, the following method is used. Each simulation set starts by running a given $M/M/c$ system under given load $\rho_h > 1$, and for a given laxity distribution. Next, the mapping procedure is performed to analytically determine the peer loads, as described in Section 2.2—that is, by using the mapping procedure from Figure 2.5. For all obtained peer loads, the corresponding simulations are run to produce task loss ratio curves for $\rho_l < 1$. As a final step of the validation through simulation, the comparison of the saturation points of the task loss ratios of paired loads (ρ_h, ρ_l) and their analytically derived punctual points is performed.

For convenient identification in the next 3 figures, all paired loads are plotted using the same line type with the filled line symbol (e.g., $\bullet$) representing an overload (ρ_h) and the empty line symbol (e.g., $\circ$) representing the mapped load (ρ_l).

Figure 2.7 presents experimental results of task loss ratios plotted against a wide range of normalized mean laxities, for a 2 processor system and laxities from an Erlang-100 distribution (a distribution with very low standard deviation). All tested systems reached a stable performance, indicating that the chosen laxity range successfully brings the task loss ratios arbitrarily close to the optimal loss.

For example, the task loss ratio for $\rho = 2.0$, and the task loss ratio of its mapped peer load $\rho = 0.49$, saturate around laxity value $l = 4$. Similarly, the task loss ratios for $\rho = 1.6$ and $\rho = 0.62$ saturate around $l = 8$; and the task loss ratios for $\rho = 1.2$ and $\rho = 0.831$ saturate around $l = 16$.

The analytically calculated punctual points for this $M/M/2$ system, carried out according to the procedure outlined in Section 2.2, are presented in Table 2.3. The analytical results are then superimposed on the experimental results in Figure 2.7. This is done by positioning corresponding designations of the punctual points (namely, numbers 1, 2 and 3) under corresponding task loss ratio curves of overloaded systems.

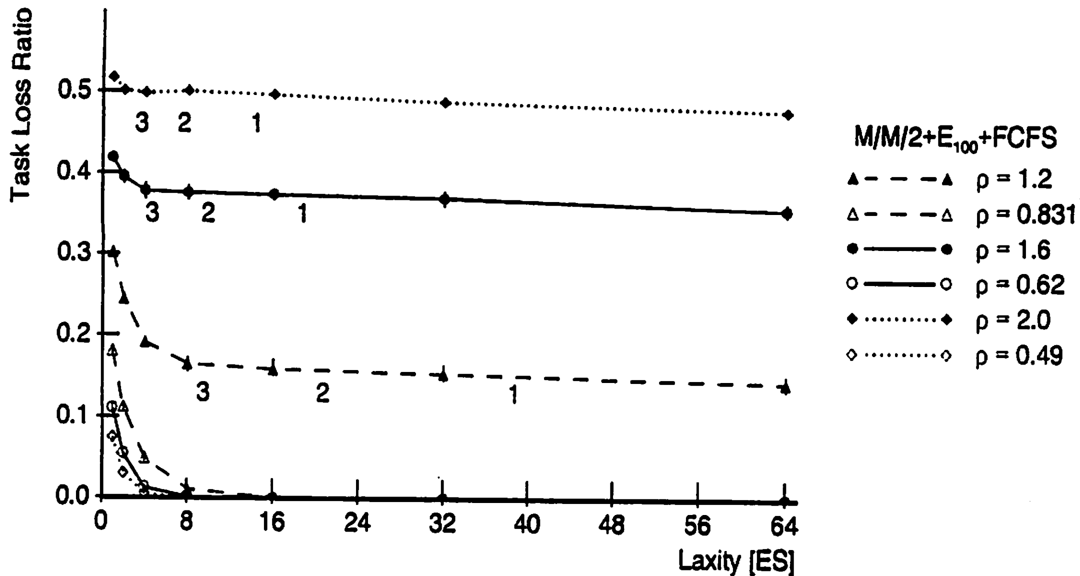

Figure 2.7 Task Loss Ratio with Punctual Points. 1: $T_P(1^-)$, 2: $T_P(0.999)$, and 3: $T_P(0.95)$.

Table 2.3 Analytically Derived Punctual Points for $M/M/2$ system.

(ρ_h, ρ_l)	(1.2, 0.831)	(1.6, 0.62)	(2.0, 0.49)
1: $T_P(1^-)$	30.1	18.1	13.7
2: $T_P(0.999)$	20.0	9.2	6.9
3: $T_P(0.95)$	8.8	4.0	3.0

The main observation, obtained from the superimposed chart (Figure 2.7), indicates that when the $\psi \to 1$, the punctual point $T_P(\psi)$ provides a very conservative saturation point. That is, the task loss ratio curve is stable for all laxities beyond the punctual point $T_P(1^-)$. (An exact value of $T_P(1^-)$, used in all calculations, is $T_P(1-10^{-6})$.) From this, a preliminary conclusion is that the analytical calculation of the punctual points matches the experimental results, when the laxities have very low standard deviation.

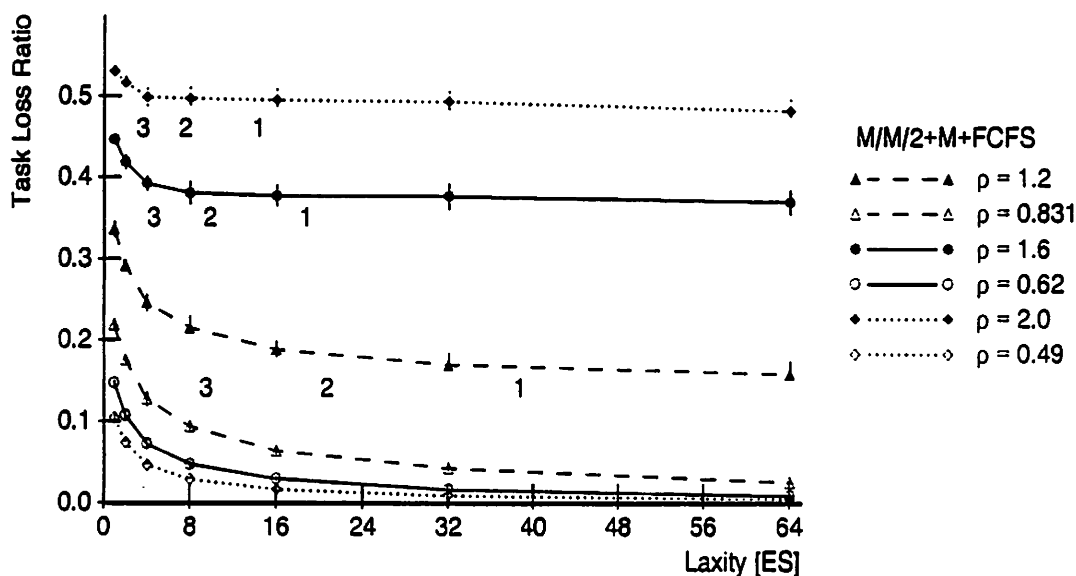

Figure 2.8 Task Loss Ratio with Punctual Points. 1: $T_P(1^-)$, 2: $T_P(0.999)$, and 3: $T_P(0.95)$.

To model systems where laxities have very high standard deviation, the laxities are drawn from the exponential distribution. The experimental results of such a system are presented in Figure 2.8. Similarly as above, and due to the fact that the punctual point calculation is independent of laxity distribution, we superimpose the same punctual points already tabulated in Table 2.3.

Observation of the Figure 2.8 indicates that, again, the analytically derived punctual points provide a very good estimate of the saturation points, even for systems with very high standard deviations. The approximation is very good but not conservative as in the previous case.

Furthermore, it is apparent that the calculation of peer loads is not as accurate as in the case of systems with low standard deviation of laxities. As shown in Figure 2.8, when $\rho = 1.2$ only one punctual point, the punctual point $T_P(1^-)$ (labeled 1), is located at the saturated part of the task loss ratio curve; $T_P(0.999)$ (labeled 2) is positioned at the bottom of the curve knee; while $T_P(0.95)$ (labeled 3) is positioned at the knee of the curve. On the other end, when $\rho = 2.0$ all three punctual

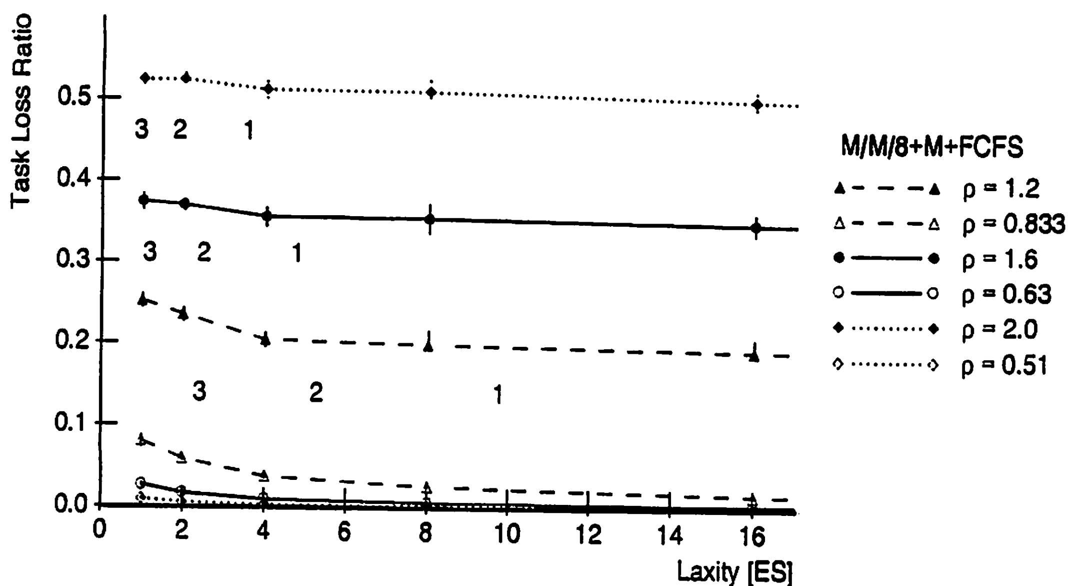

Figure 2.9 Task Loss Ratio with Punctual Points. 1: $T_P(1^-)$, 2: $T_P(0.999)$, and 3: $T_P(0.95)$.

Table 2.4 Analytically Derived Punctual Points for $M/M/8$ system.

(ρ_h, ρ_l)	(1.2, 0.833)	(1.6, 0.63)	(2.0, 0.51)
1: $T_P(1^-)$	9.6	4.6	3.4
2: $T_P(0.999)$	5.1	2.3	1.7
3: $T_P(0.95)$	2.25	1	0.77

points are located within the saturated part of the task loss ratio curve. This indicates that even under very high standard deviation of the laxity distribution, the punctual points are very usable as approximations of the saturation points. If standard deviation is large, the ψ should be close to 1, while if the standard deviation is small, a smaller value of ψ will give accurate results.

When the number of processors is increased, the saturation point is reached at much lower laxity values. The case of 8 processors with superimposed punctual points is given in Figure 2.9; and the actual values of the derived punctual points for

an $M/M/8$ system are tabulated in Table 2.3. The analysis of this system leads to the same conclusions as the ones drawn from the previous two examples (Figure 2.7 and Figure 2.8). That is, the analysis indicates that the analytical results closely match the experimental results.

Therefore, by matching the experimental results to the analytical results we have validated our analytical approach. From the above experiments, we conclude the following: the analytically derived punctual points $T_P(1^-)$ present a very good approximation of the saturation points for all tested task loss ratios. This is true even when the laxity distribution exhibits a very high standard deviation, such as for the exponential distribution. This concludes the verification of the analytically derived punctual points.

To illustrate how the punctual point reduces the scheduling overhead, let's examine the number of tasks in the waiting area. At each arrival, the laxities of all queued tasks are compared with the analytically derived punctual points, and the "relevant" tasks—the tasks with positive laxities that are smaller or equal to the punctual points—are counted. The obtained total number of the "relevant" tasks in the entire run is then averaged by the total number of arrivals.

Figure 2.10 and Figure 2.11 present the averaged number of "relevant" tasks plotted against the mean laxities, for $M/M/2+M+FCFS$ and $M/M/8+M+FCFS$ systems, respectively. The punctual points for traffic intensity of $\rho = 2$ are taken from the respective tables, i.e., from Table 2.3 and Table 2.3, given in previous examples.

The characteristic of the overloaded systems is that the number of queued tasks increases proportionally with the increase in laxities. This trend, in both figures, is presented by the curve labeled as "No T_P". Observe that for mean laxity of $l = 64$, the average number of queued tasks for the two processor system is around 124, while for the eight processor system it is around 443. On the other hand, the number

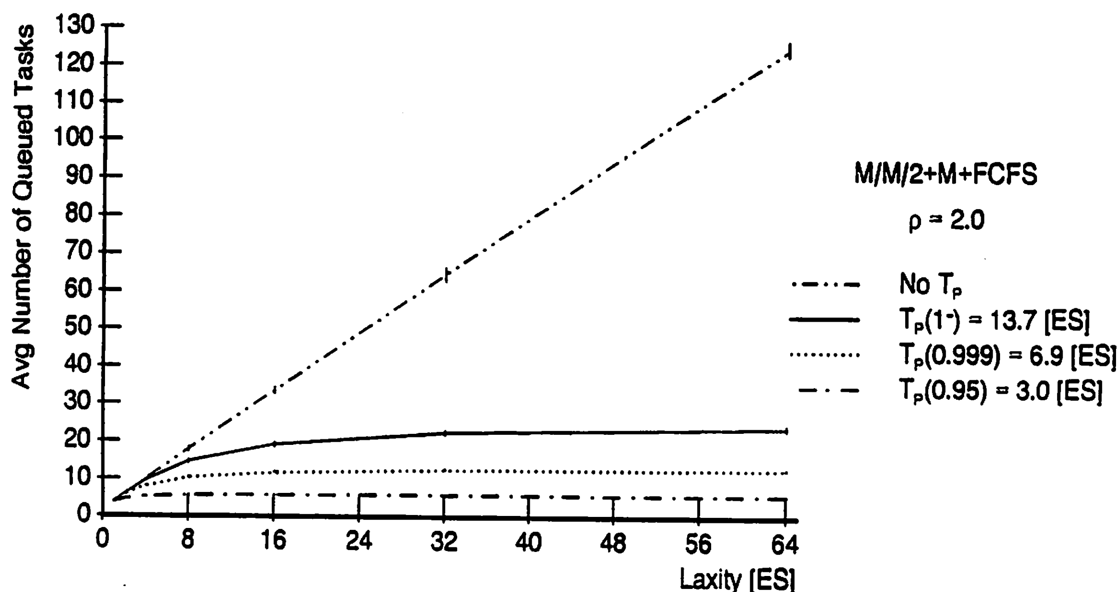

Figure 2.10 The Average Number of "Relevant" Tasks per Task Arrival.

of "relevant" tasks for $T_P(1^-)$ is around 24 and 23, respectively; for $T_P(0.999)$ it is around 13 and 12; and for $T_P(0.95)$ it is around 6 and 5. This indicates the dramatic decrease in the average number of tasks subjected to scheduling. The actual savings are easily obtained from the computational complexity of the scheduling algorithm, the complexity of maintaining the laxity order, the per task scheduling cost and the average number of "relevant" tasks.

From the same figures, Figure 2.10 and Figure 2.11, it is evident that the number of "relevant" tasks reaches the saturation point very fast. That is, it does not increase proportionally with the increase of laxities, as in the case where separation of "relevant" and "irrelevant" tasks is not performed (the "No T_P" curve).

These characteristics indicate that using the punctual points under overloads potentially provides tremendous savings in scheduling cost, especially, for systems with large laxities and large number of processors. Further exploration on the effects of the punctual points on the overall scheduling cost is subject of the next chapter, Chapter 3.

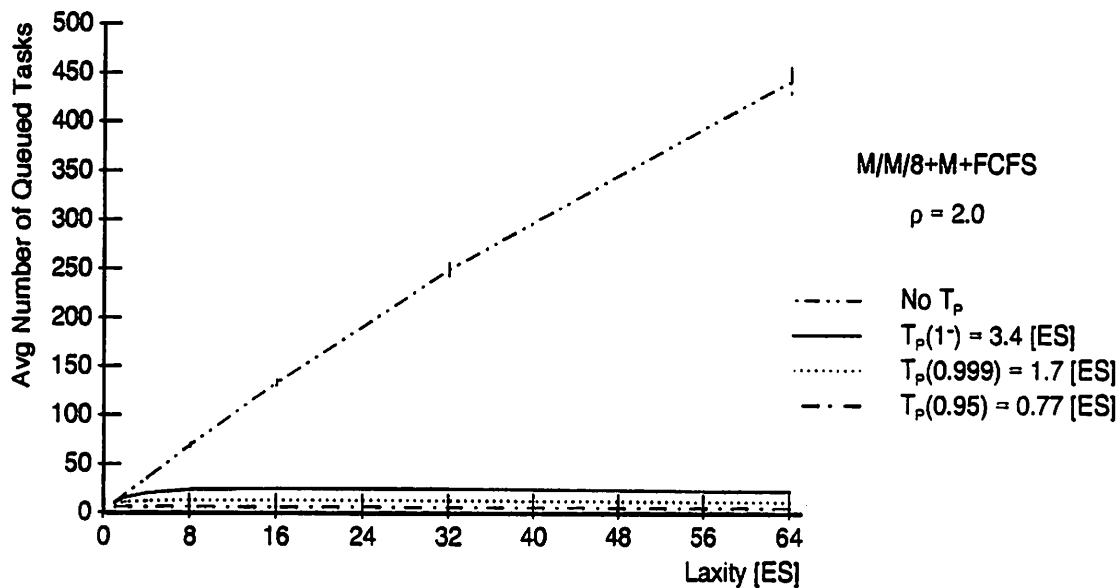

Figure 2.11 The Average Number of "Relevant" Tasks per Task Arrival.

2.4 Laxity Analysis

An important feature of punctual points is their use in the system design phase of real applications (for example, for determining the number of processors according to the given laxity requirements), and in the design of simulation experiments (for determining the laxity ranges of significance). Furthermore, due to its applicability to $M/G/c$ systems, Well-Timed Scheduling presents a very powerful design tool as well as run-time tool.

This section presents an analysis of systems with zero laxity tasks, and concludes with an introduction of probabilistic guarantee and its use in $M/G/c$ real-time systems.

2.4.1 Zero Laxity Systems

Theoretically, the most demanding real-time system is a system where all laxities are equal to zero. The distinct feature of these systems is that a task is rejected (and therefore lost) if it cannot start its execution immediately, i.e., if it cannot find

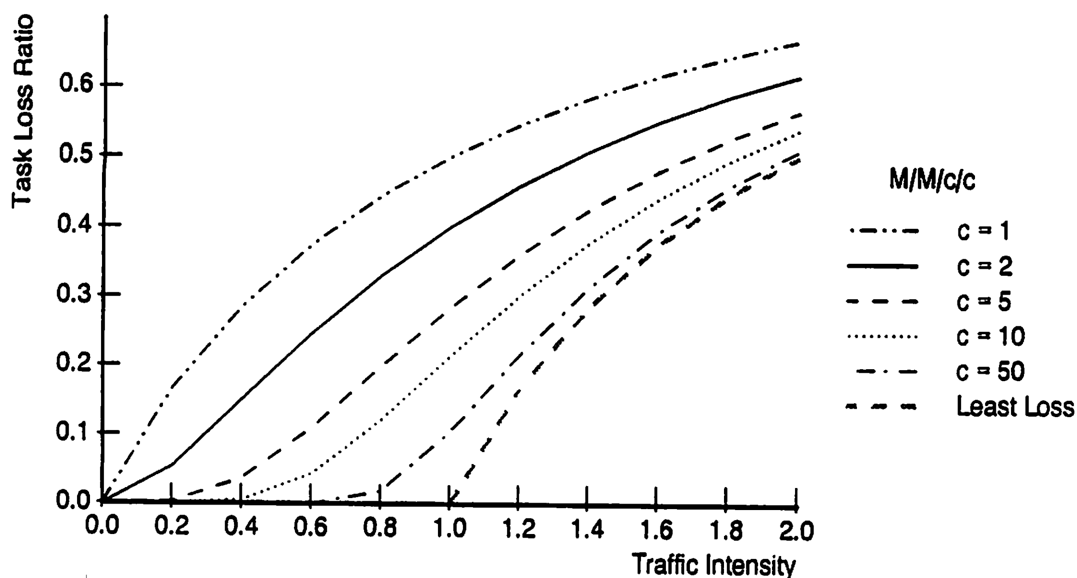

Figure 2.12 Task Loss Ratios for Zero Laxity Systems—Erlang Loss

an idle processor upon its arrival. A design question often asked for such systems is: what is the maximum load for which the tasks will complete their executions, or with what probability will they be executed upon their arrivals?

The answer to this question in queueing theory is the Erlang loss formula. The applicability of the Erlang loss formula is restricted to $M/M/c/c$ systems—the systems where the maximum number of tasks in the whole system is equal to the total number of processors, and where the service time is exponentially distributed.

An example of the loss ratios for multiprocessor systems with $c = 1, 2, 5, 10, 50$ processors, plotted over the wide range of loads, is given in Figure 2.12. When the loss ratios for $M/M/c/c$ systems (Figure 2.12) are compared with the loss ratios for $M/M/1 + M + FCFS$ system (Figure 2.3) the following can be observed. First, the worst expected losses are identical, because both systems assume zero laxity and a single processor. Second, the loss system with a large number of processors, for example $c = 50$, and loads up to the $\rho = 0.6$ have extremely small number of lost tasks. When this loss is compared with the $M/M/1 + M + FCFS$ (for laxities of

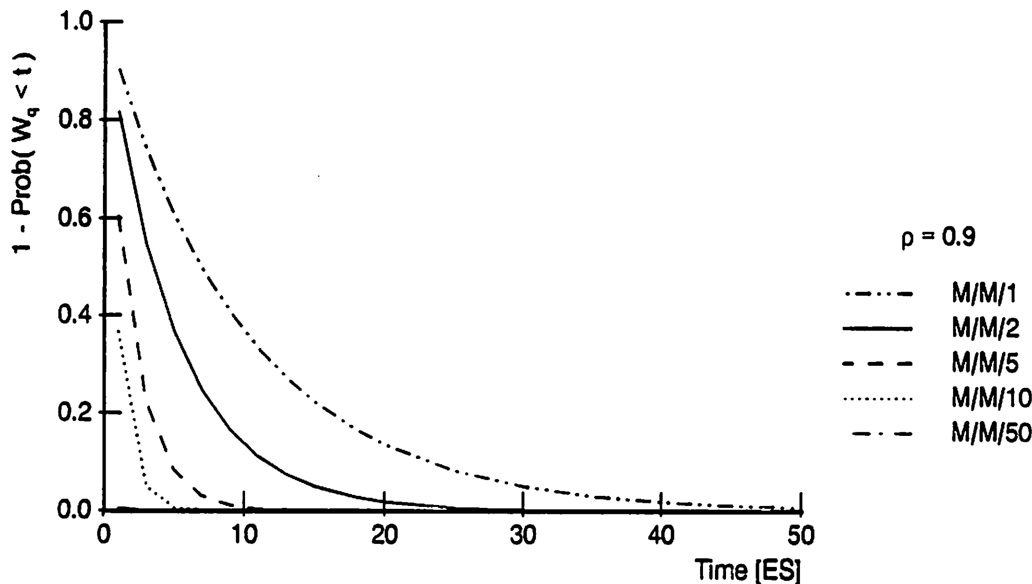

Figure 2.13 Complementary Waiting Time Distribution

$l = 100ES$ and for loads below $\rho = 0.8$), the $M/M/50/50$ system performs better while for higher loads, it is outperformed by the $M/M/1 + M + FCFS$ system.

The slightly better performance of a single processor system, at heavy loads and overloads, does not overshadow the much more important difference between these two systems: $M/M/c/c$ operates with *zero* laxities while a single processor system requires ***very large*** laxities, $l = 100ES$, in order to stay competitive. Consequently, these results suggest the need for multiprocessor systems when timing constraints are very demanding.

In what follows, we consider multiprocessor systems in the context of tasks with non-zero laxities.

2.4.2 Probabilistic Guarantee for $M/G/c$ Systems

When the waiting time distribution is observed from the viewpoint of the punctual point, that is, when the results for general $M/G/c$ system are interpreted from the real-time perspective, a notion of probabilistic guarantee can be developed. The intention is to develop a quantitative method that provides the probabilities that the

Table 2.5 Punctual Points for $\rho = 0.9$ Load.

	$M/M/1$	$M/M/2$	$M/M/5$	$M/M/10$	$M/M/25$	$M/M/50$
$T_P(0.99)$	50	25	10	5	2	1
$T_P(0.999)$	70	35	15	7	3	2

timing constraints of real-time tasks will be met. This subsection analyzes $M/G/c$ systems in light of such probabilistic guarantees.

Let us examine $M/M/c$ systems for $c = 1, 2, 5, 10, 50$, where $\rho = 0.9$. The punctual points rounded to integers, for a load of $\rho = 0.9$ and for all considered multiprocessor systems are given in Table 2.5. These tabulated results can be interpreted from the probabilistic guarantee point of view as follows.

With probability ψ, the timing constraints of the real-time tasks are guaranteed to be met if the minimum laxity in the system is larger or equal to the calculated punctual point $T_P(\psi)$. Specifically, with guarantee probability equal to $\psi = 0.999$, for $c \geq 10$, the minimum laxity requirements are on the same order of magnitude as the mean service time (i.e., for $c = 10, 25, 50$ the $T_P(0.999) = 7, 3, 2$, respectively); while for $c \leq 5$ and the same guarantee level, the minimum required laxities are one order of magnitude higher (that is, for $c = 1, 2, 5$ the $T_P(0.999) = 70, 35, 15$). Thus, we conclude that multiprocessors of the order of $c = 10$ are appropriate for real-time design when the timing requirements are tight and the load is moderate to high.

To illustrate the case when the guarantee probability ψ is set to be comparable with hardware reliability probability,[5] Table 2.6 and Table 2.7 are presented. The assumed, and the best that the implemented version of the approximation algorithm for waiting time distribution can reach is a hardware failure probability of 10^{-4}. (This is due to the computation error of the method in use). In spite of the

[5]The current, most stringent, requirement for the probability of hardware failure is in avionics: 10^{-10} for one hour in a ten hour mission.

Table 2.6 Punctual Points for $\rho = 0.7$ Load.

	$M/M/1$	$M/M/2$	$M/M/5$	$M/M/10$	$M/M/25$	$M/M/50$
$T_P(1-10^{-4})$	31	15.5	6.2	3.1	1.23	0.62
$T_P(1)$	54	27	10.7	5.4	2.15	1.07

Table 2.7 Punctual Points for $\rho = 0.9$ Load.

	$M/M/1$	$M/M/2$	$M/M/5$	$M/M/10$	$M/M/25$	$M/M/50$
$T_P(1-10^{-4})$	92	46	18.5	9.2	3.7	1.85
$T_P(1)$	116	58	22.9	11.5	4.58	2.29

computation error of the method in use, we chose to present the results for $\psi = 1$ in order to indicate a general proximity of the minimum laxity requirements for which the system is expected to guarantee "all" timing constraints.

Table 2.6 indicates that the minimum laxity requirements for systems with $c \geq 5$ are on the same order of magnitude as the mean service time (for example, for $c = 5, 10, 25, 50$ the $T_P(1-10^{-4}) = 6.2, 3.1, 1.23, 0.62$, respectively); while in Table 2.7, and for laxities in the same order, the required number of processors is greater than 10 (for $c = 10, 25, 50$ the $T_P(1-10^{-4}) = 9.2, 3.7, 1.85$). Therefore, once again, even for these very stringent requirements, where the probability of missing deadline must equal the hardware reliability, multiprocessor systems with the number of processors on the order of $c = 10$ are guaranteed to perform very satisfactorily.

Due to the fact that the punctual points, and therefore the minimum laxity requirements, are relatively small, even a simple FCFS scheduling algorithm is capable of performing well in $M/G/c$ real-time systems with a moderately large number of processors. For example, for $c = 25$, $\rho = 0.9$ and $\psi = (1-10^{-4})$, the minimum required laxity is just 3.7 times the expected service time. In the other

extreme, a single processor system under the same load and the same guarantee probability requires the minimum laxity of 116 times the expected service time. Such a large laxity requirement is considered unacceptable for most real-time applications.

In conclusion, the best way to design a system with demanding timing requirements is to use multiprocessor systems utilizing a very simple scheduling algorithm. If the laxities are relatively short and the number of processors available is low, more sophisticated real-time scheduling algorithms should be used. However, regardless of the used scheduling algorithm, any dynamic non-deterministic real-time system should anticipate overloads—whether they occur due to processor(s) failure, increased number of requested alternative actions, or simply due to environment changes. Therefore, real-time systems should be designed to operate within the framework that features a methodical approach to overloads in order to keep their performance high at all times.

2.5 Summary

The Well-Timed Scheduling framework presented in this chapter is applicable to non-deterministic real-time systems where the dynamic approach to schedulability analysis is preferred, and where best-effort scheduling algorithms are most suitable. Due to on-line schedulability analysis, it is essential to keep computational costs low without sacrificing the overall performance.

The notion of the punctual point, as a key component of the Well-Timed Scheduling framework, is motivated by these needs. It provides low-cost overhead and high overall performance by determining the most opportune time to schedule scheduling newly arrived tasks. Using this method, only "relevant" tasks, that is, the tasks with laxities smaller than the punctual point are scheduled. The number of scheduled tasks is reduced, and consequently, the computational cost is lowered. The strength of this approach lies in its ability to make trade-offs between given design

parameters and desired performance. Also, it supports the probabilistic guarantees by providing a quantitative justification that the timing requirements of a real-time system can be met.

The punctual point is applicable both during the design phase and during the run-time operation. As a run-time tool, it is especially useful in overloaded situations. A large number of tasks that are eligible for scheduling can be dramatically reduced due to "relevant"/"irrelevant" task separation. On the other hand, as a design tool, it can be used to determine the probability that the tasks are guaranteed to meet their deadlines; or to determine the number of processors that provide a desired level of guarantee; or to determine the laxity values necessary to achieve the desired guarantee level, when the number of processors is fixed.

Summarized, the major contributions of the Well-Timed Scheduling framework are:

- An analytical method is used to determine the probabilistic guarantee in the context of real-time systems. For example, if the guarantee probability is higher than other probabilities of interest, e.g., hardware reliability, software reliability, and code execution time being accurate, then essentially, for all practical purposes, we have an "absolute" guarantee.

- A tool for controlling the number of schedulable tasks is developed. It is especially useful in overloaded situations, thus, providing a scheduling approach that is applicable to the entire range of loads—from low loads to overloads.

- The derivation of the punctual point is based on recent queueing theory results and it is applicable to $M/G/c$ real-time systems.

- As a design tool, it is powerful enough to match the number of processors to given laxity requirements, and vice versa.

Coupling the punctual points and laxity distribution it is observed that: (1) if the minimum laxity is larger than or equal to the punctual point, then all the tasks will meet their deadlines with specified ψ probability, and (2) if the ψ probability is greater than the reliability of the system, an "absolute" guarantee is provided. However, for a given load, ψ probability, and the number of processors, if the punctual point is located within the laxity range, or if it is larger than the maximum laxity, a different approach has to be used. That is, suitable measures must be taken to decrease the punctual point so that it becomes smaller than the minimum laxity. This is accomplished by increasing the number of processors, decreasing ψ, or by moving the load away from load $\rho = 1$.

In the next chapter, the tests of the applicability of Well-Timed-Scheduling to real-time systems with resource constraints and task values is analyzed. The performance measure of interest is the total accrued value, rather then the total number of completed tasks. The results indicate the significant benefits of using the punctual points. As a side effect of scheduling only "relevant" tasks, the probability of tasks having resource conflicts is decreased, and with that, the premature rejection of tasks—due to the resource conflicts—is avoided.

Furthermore, in the next chapter, the effects of the different scheduling costs with and without the use of punctual points are analyzed. Here again, under increasing per task scheduling cost, systems that use Well-Timed-Scheduling exhibit significantly smaller degradation of the overall performance, when compared to systems that perform schedulability analysis on the entire set of eligible tasks.

CHAPTER 3

Integration of Well-Timed Scheduling and Real-Time Schedulers

3.1 Introduction

The scheduling of aperiodic tasks with timing constraints, resource constraints, and contributing values requires more complicated algorithms than the simple real-time algorithms, such as Earliest Deadline First or Minimum Laxity First algorithms. Furthermore, when a system overload is likely to happen, use of a method that controls scheduling overheads is a must. That is, the inability of traditional schedulers to create a feasible schedule, due to the large number of task arrivals, must be overcome in order to avoid a catastrophic system breakdown—a total loss of control. In the previous chapter, we proposed and developed Well-Timed Scheduling as a framework especially designed for system overloads. The control of the scheduling cost is achieved by the early classification of tasks as "relevant" or "irrelevant" based on the analytically derived punctual point. A real time scheduler schedules only "relevant" tasks. The "irrelevant" tasks are delayed until their laxity equals the punctual point.

In this chapter, the benefits of an integration of the Well-Timed Scheduling and real-time scheduling algorithms is examined in detail. Specifically, two real-time scheduling algorithms are developed and integrated with Well-Timed Scheduling. The first algorithm is the DLVD algorithm that schedules the tasks in increasing *deadline* order, and in the case of an infeasible schedule, it rejects a task with *minimum value density*. The second algorithm is the RDS algorithm where a *linear*

combination of task's value density, deadline, and earliest possible start time (given resources needed by a task) is used to decide on the next task to schedule; if an infeasible schedule is reached, the task is rejected.

This chapter is organized as follows. A description of the modeled real-time environment is presented in Section 3.2, followed by the detailed descriptions of DLVD and RDS algorithms in Section 3.2.1 and Section 3.2.2, respectively. The simulation parameters that are used through the analysis of both scheduling algorithms are given in Section 3.3. The integration analysis is given in Section 3.4 which consists of four parts: (1) the analysis of both real-time schedulers without the benefits of the Well-Timed Scheduling and with negligibly small scheduling costs—a traditional application with a minimal scheduling cost, (2) the analysis of the effects of the increasing scheduling costs, and still without the benefits of the Well-Timed Scheduling, (3) the effects of the Well-Timed Scheduling with realistic scheduling costs, and finally, (4) the applicability range and robustness analysis of the proposed integration. The summary of the findings is presented in Section 3.5.

3.2 Model Description

The underlying system for DLVD and RDS algorithms is a shared multiprocessor system with one system processor and several application processors. All scheduling activities are performed on the system processor, while the executions of the aperiodic tasks is performed on the application processors. This functional separation of the processors serves to isolate non-determinism, incurred by unpredictable invocations of a scheduler, from the precisely sequenced (scheduled) executions of aperiodic tasks. As a result, the scheduling algorithm does not disturb the timings of the executions already scheduled on the application processor. That is, the scheduling of new tasks and executions of previously scheduled tasks is performed in parallel. When a scheduler creates a feasible schedule, it combines this schedule with the tasks

left for execution generating a new system task table (STT), that has independent threads of executions for each application processor.

The task arrivals are modeled using the Poisson distribution, and at an arrival time, each task is described with its computation time, initial laxity, contributing value, and its resource requirements. The computation times and laxities are assigned according to two distributions, the exponential and the Erlang-3 distribution. The contributing values are assigned according to the uniform distribution. The requested resources are assigned using a system parameter R_{use} that presents the probability of using a particular resource. A requested resource has an equal probability to be used in shared or exclusive mode. All the task parameters are assigned independent of each other.

The scheduling algorithms are designed to schedule tasks only if their timing constraints are satisfied and if there are no resource conflicts. The tasks are scheduled using the *endorsement*, rather than the *guarantee* policy. The endorsement policy allows a rejection of tasks from the partial feasible schedule, while the guarantee policy does not. In other words, the guarantee policy employs a once-guaranteed-always-guaranteed strategy. Due to this, the guarantee policy is considered to be inferior compared to the endorsement policy, if the objective of a scheduling algorithm is to maximize accrued value. That is, its inability to schedule newly arrived tasks with very high values due to already committed guarantees is considered inappropriate for general dynamic real-time systems with aperiodic tasks that have different contributing values.

In the DLVD algorithm, the endorsement policy always schedules a task with the smallest deadline, and in the case of an infeasible schedule, DLVD rejects a task with minimum value density. On the other hand, the endorsement policy in RDS algorithm uses a heuristic function that is based on the linear combination of value

density, deadline, and resource availability. In the case of an infeasible schedule, the RDS algorithm always rejects the last scheduled task.

In the Well-Timed Scheduling framework, the newly arrived tasks are classified as "relevant" or "irrelevant" tasks. The "relevant" tasks are stored in the S-pool (the scheduling pool), as tasks eligible for immediate scheduling. On the other hand, the "irrelevant" tasks are stored in the D-queue (the delay queue), where they are delayed until their laxity becomes equal to the punctual point, that is, until they become "relevant" and ready to be transferred to the S-pool. Whenever a task is filed in the S-pool the scheduling algorithm is activated. In order to perform the scheduling, the tasks in the old STT must be divided into tasks that will be rescheduled and tasks that will be left for the execution on the application processors. Whether a task is marked as *reschedule* or *leave-for-execution* depends on the estimated scheduling cost. The tasks with scheduled start times prior to the current time plus the estimated scheduling cost (i.e., prior to the *cutoff line* [49]) are considered guaranteed, and they are left for execution. The tasks with scheduled start time larger than the cutoff line are transferred into the S-pool for rescheduling. The estimated scheduling cost, i.e., the cutoff line depends on: (1) the total number of tasks in the STT plus the newly arrived tasks, (2) the computational complexity of the scheduling algorithm, and (3) the scheduling cost to schedule one task, i.e., a scheduling cost proportionality constant, SCF.

During the scheduling, a feasibility check is performed. If the feasibility check is satisfied, a task is transferred into the partial feasible schedule, called the E-chain (the list of endorsed tasks). When the scheduling is completed, the E-chain is appended to the list of tasks left for execution to form a new STT.

It is important to observe that aside from the above identified scheduling cost, the separation of tasks into "relevant" and "irrelevant" tasks also contributes to the scheduling overhead, due to the insertion, maintenance, and removal of tasks

delayed in the D-queue. This cost depends on the computational complexity of the timer algorithm chosen to handle the D-queue. In [61] seven timer algorithms are presented and discussed. Four traditional algorithms, two based on ordered lists, or timer queues of $O(n)$ complexity, and two tree-based algorithms of $O(\log(n))$ complexity, are discussed and analyzed. Additionally, Varghese and Lauck developed and presented three new timer algorithms that have constant computational complexity for setting, stopping, and maintaining a timer. These three timer algorithms are based on the timing wheel mechanism found in logic simulations. They also briefly discuss the potential performance improvements obtained by using a special purpose hardware that is designed to maintain the data structures and to interrupt the host software only when a timer expires. In spite of their $O(1)$ cost, these algorithms can still produce a complexity of $O(n)$. If all n tasks are timed-out at the same time, and if they are transferred into S-pool on a task-by-task basis, the computational complexity becomes $O(n)$. However, if the whole chain of tasks is transferred at once, by simply re-linking the timed-out tasks, the constant time computational complexity is achieved.

In our model, the constant time timer algorithm is assumed, whose cost of handling the D-queue is considered to be a part of the task's computation time. Consequently, the only cost to model is the task scheduling cost. This cost is a function of the number of scheduled tasks and the time needed to schedule a single task—i.e., the scheduling cost proportionality constant, SCF. Varying the values of the SCF, the performance of the near ideal systems (the systems with a negligibly small scheduling cost proportionality constant) and the performance of the more realistic systems (the systems where the scheduling cost has a significant impact on the overall performance) are analyzed.

The main performance metric used is the value loss ratio, the same metric as in the previous chapter. That is, the ratio of the sum of the values contributed by the

tasks that completed within their timing constraints over the total generated value. Similarly, the task loss ratio is defined as the ratio of the number of tasks completed by their deadlines over the total number of generated tasks.

3.2.1 DLVD Algorithm

The basic concept of the DLVD algorithm is: schedule the tasks in *increasing deadline order*, and in case there is an infeasible schedule, reject the task with the *smallest value density*. In [32], a variant of this concept was used in a design of the Best-Effort algorithm—an algorithm for dynamic tasks scheduling of preemptive and restartable aperiodic tasks, with a goal to maximize the accrued system value. Best-Effort runs on a dedicated scheduling processor in a shared memory multiprocessor environment. When invoked, it first orders the tasks in decreasing deadline order. Second, it computes the variance of the total slack time in order to find the probability that the available slack time is less than zero. The calculated probability is used to detect a system overload. If it is less than the user prespecified threshold, the algorithm removes the tasks in increasing value density order. It is important that the tasks are removed, not rejected. The rejection of a task occurs only when its deadline is missed, or when its value drops below the user specified threshold.

Comparison of the Best-Effort algorithm to the other real-time and non-real-time scheduling algorithms showed that the Best-Effort algorithm outperforms other algorithms most of the time, and that it exhibits a highly consistent performance, especially in extremely stressful environments [32]. This result was the motivation to use the basic idea of the Best Effort algorithm as a starting point in the design of the DLVD algorithm.

Figure 3.1 presents the high level description of the DLVD algorithm. In line 1, the tasks eligible for scheduling, i.e, the tasks from the S-pool are sorted in increasing deadline order. This sort is performed once per invocation. The computational

```
1. Order S-pool in minimum deadline order.

2. Repeat While S-pool ≠ empty
3.     Get a task with earliest deadline from S-pool.
4.     Find earliest possible T_sst for the selected task.
5.     If T_sft ≥ T_dl
6.         Perform rejection
7.     else
8.         Transfer the task to E-chain
```

Figure 3.1 DLVD Algorithm.

```
8.a. Find the pointer to the task with minimum value density.
     (The last added task to E-chain has a pointer to this task.)

8.b. Reject the task with minimum value density.

8.c. Transfer the tasks that are chained after the rejected task
     from E-chain to S-pool. (Append transferred tasks to the top
     of S-pool to maintain already established earliest deadline order.)
```

Figure 3.2 DLVD—Rejection Procedure.

complexity of the sort is $O(n \log(n))$. The rest of the algorithm (from line 2 to line 8) is executed once for every task. Specifically, line 3 fetches the top task from the S-pool, that is, the task with the smallest deadline. For this task, the earliest possible scheduled start time (T_{sst}) is determined in line 4 using the computation based on the earliest available time (EAT) vector that describes the availability of the resources and CPUs. (More detailed descriptions of the EAT vector can be found in [64].) In line 5, a simple feasibility check is performed. If the task's deadline is larger than the task's scheduled finish time (T_{sft}), the rejection procedure is invoked (line 6). Otherwise, the task is considered scheduled, and it is appended to the end of the E-chain in line 8.

Figure 3.2 presents the expanded view of the rejection procedure from line 6. For efficiency, whenever a task is included in the E-chain its value density is compared to the current smallest value density, and the pointer to the task with a smaller value is recorded in the task's fields (line 8.a). This information is used in line 8.b for a one step identification of the task with the smallest value density, i.e., of the task to be rejected. Both lines, line 8.a and line 8.b, have a computational complexity of $O(1)$. The higher computational complexity of the rejection procedure comes from the line 8.c. When the rejected task is removed from the E-chain, the tasks below its position cannot be simply moved up because of the potential resource conflicts. They must be transferred to the S-pool for rescheduling. (Note that the tasks in the E-chain are ordered in increasing deadline order, due to the earliest deadline endorsement policy, and the E-chain construction—every endorsed task is simply appended to the chain of previously endorsed tasks. Therefore, the resorting of the S-pool is avoided by transferring tasks in a block fashion.) Due to this, the construction and reconstruction of the E-chain is required. In consequence, the overall computational complexity of the DLVD algorithm is $O(n^2)$.

3.2.2 RDS Algorithm

In contrast to the DLVD algorithm, the RDS algorithm is based on the real-time scheduling algorithm, that is, specifically designed to schedule non-preemptable aperiodic tasks with resource constraints. In [64], Zhao and Ramamritham presented and evaluated the algorithms that used heuristic function based on single task parameters, and their linear combinations. They observed that simple heuristic algorithms do not perform well, while the integrated algorithms perform very close to the performance of an optimal algorithm. Among the tested heuristics, the linear combination $H = (DL + W \times SST)$ is identified as the heuristic with the best cost-performance ratio. Its close to optimal performance and computational

1. *For each task in S-pool*
2. Determine and assign scheduling parameter, and perform feasibility check.
3. Calculate and assign H values to tasks in S-pool.
4. Transfer the task with the smallest H value from S-pool to E-chain.

Figure 3.3 RDS Algorithm.

complexity of $O(n^2)$ made it the best choice. Recently in [44], the improved version with $O(n)$ complexity and no significant loss in performance is presented. This low complexity is achieved by applying the heuristic function to a fixed number of tasks.

Compared to these original algorithms, our RDS algorithm has three major differences: (1) the goal of the RDS algorithm is to maximize the accrued value rather than maximizing the number of completed tasks, (2) tasks have different contributing values rather than a single value, and (3) the scheduler uses an endorsement policy versus the once-guaranteed-always-guaranteed policy. The change in goal of the scheduling algorithm was influenced by the introduction of the individual task values. For the same reason, the once-guaranteed-always-guaranteed policy is replaced with a more flexible endorsement policy that allows the high value tasks to replace the already scheduled ("guaranteed") low value tasks. It would be short sighted to prohibit the high contributing value tasks to replace previously scheduled low value tasks because of previously made commitments.

Figure 3.3 outlines the high level description of the RDS algorithm. In line 2, several state dependent parameters (discussed below) are assigned to the tasks currently present in S-pool; followed by the feasibility check and the rejection of the task that cannot meet its timing constraints. Using the parameters assigned

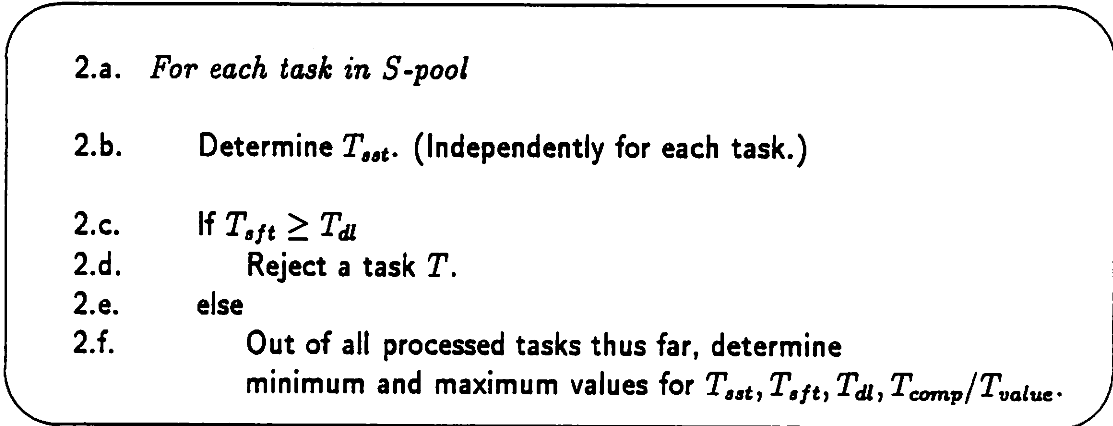

Figure 3.4 RDS—Parameter Assignment and Feasibility Check.

in line 2 and the initially assigned values, the heuristic function H is assigned to the remaining tasks in S-pool in line 3. Finally, the task with the smallest H value is selected and transferred into the E-chain. That is, the task with the smallest H is endorsed. This procedure is repeated until the last task from the S-pool is scheduled. The computational complexity of this algorithm is $O(n^2)$ because the inner steps of the RDS algorithm have $O(n)$ complexity.

To complete the description of the RDS algorithm, we next describe the assignments of the parameters in line 2, followed by the description of the H-function from line 3.

The expansion of line 2 is presented in Figure 3.4. In line 2.a, the earliest scheduling start time (T_{sst}) is determined. This parameter reflects the current availability of the required resources and CPUs. At the same time, the scheduled finish time, T_{sft}, is computed by adding task's computation time to T_{sst}. In line 2.c, T_{sft} is checked against the timing requirements, and if the deadline is passed, the task is rejected. However, if the timing constraints are satisfied, the task is considered for endorsement. This requires the computation of additional parameters. Specifically, the inverse of the value density is computed, and, for each of the

3.a. *For each task in S-pool*

3.b Assign H value to a task as follows:

3.b.1 $R = \frac{T_r - R_{min}}{R_{max} - R_{min}}$, where $T_r = \frac{T_{comp}}{T_{value}}$.

3.b.2 $D = \frac{T_{dl} - DL_{min}}{DL_{max} - DL_{min}}$.

3.b.3 $S = \frac{T_{sst} - SST_{min}}{SFT_{max} - SST_{min}}$.

3.b.4 $H = 2R + D + S$.

3.c Out of all assigned H values thus far, determine what task has the minimum H value.

Figure 3.5 RDS—Heuristic Function.

computed parameters, the minimum and maximum values are determined. These parameters are used for H value calculation.

The expanded view of the H value calculation is given in Figure 3.5. To calculate the H value, the H function factors have to be normalized. Normalizing each factor individually maps the different standards of measure onto the unit scale, and consequently enables a straightforward integration of different parameters.

Factor R which presents a normalized inverse of the value density is calculated in line 3.b.1; factor D which presents a normalized deadline is calculated in line 3.b.2; and factor S which presents a modified version of normalized scheduled start time is calculated in line 3.b.3. (A factor with the min/max subscript denotes the minimum/maximum value of all corresponding factors within the current set of tasks. Therefore, the normalization is performed for every change in a task set.) The linear combination of these three factors in line 3.b.4 determines the H value of a task. In line 3.c, the task with the smallest H value is identified, and the pointer to it is updated.

During the design phase of the RDS algorithm, it was observed that the most important factor is the inverse of value density; next is the deadline, and the least important is the scheduled start time. Furthermore, it was observed that the factor S, when normalized using the SST_{min} and SST_{max}, exhibits a polarization effect at the end values, i.e., at 0 and 1—very often, the tasks can start their execution either at SST_{min}, or, due to the resource conflicts, at SST_{max}. This effect creates a much stronger bias towards the tasks with the smallest start time that have no resource conflicts, regardless of the absolute difference between two SSTs. To avoid this effect and to lower the importance of the scheduled start time, the S factor is normalized using SFT_{max} (maximum schedule finish time) instead of SST_{max} (maximum schedule start time) . This substitution minimizes the polarization effect and it lowers the influence of the S factor. The in-depth analysis of the robustness, sensitivity, and the overall performance of the RDS algorithm is beyond the scope of this dissertation.

3.3 Simulation Description

To test the behavior of the above described real-time scheduling algorithms within the Well-Timed Scheduling framework, we developed a discrete-event simulation model that consists of three parts: (1) a load generator, the module written in SIMSCRIPT II.5 simulation language, (2) the analytically derived punctual points, using the method described in the previous chapter, and (3) the simulator, the module written in C programming language.

For given task and system parameters, the load generator creates a list of events. This list, the analytically derived punctual points, and the simulation parameters are given as input to the simulator. This section describes the input parameters in the following order. First, the descriptions of the system and task parameters used

Table 3.1 System Parameters.

1.	Number of System Processors:	a. One Processor
2.	Number of Application Processors:	a. Two Processors b. Eight Processors
3.	Number of Schedulable Resources:	a. Five Resources

in the load generator are presented. Second, the punctual points used in simulations are tabulated; and finally, the description of the simulation parameters is presented.

3.3.1 System Parameters

The simulator is built to support one system processor and multiple application processors. The tasks are allowed to execute only on the application processors, while the system processor is entirely dedicated to scheduling. Aside from scheduling CPUs, the scheduling algorithm has to avoid conflicts on the resources. The simulator supports a resource model that allows exclusive and shared use of the resources, described in more detail in [64] and [49]. Table 3.1 lists the system parameters used in simulations reported in this chapter.

3.3.2 Task Parameters

Besides the system parameters, the load generator requires detailed descriptions of the tasks. The list of task parameters used in the reported simulations is presented in Table 3.2. Specifically, only aperiodic tasks that are executed in a non-preemptive mode are presented. Each task is characterized with the computation time, laxity, contributing value, and the resource requirements. To model the systems with a

Table 3.2 Task Parameters.

1.	Task Type:	a. aperiodic
2.	Execution Mode:	a. non-preemptive
3.	Computation Time:	a. Exponential Distribution b. Erlang-3 Distribution
4.	Laxity:	a. Exponential Distribution b. Erlang-3 Distribution
5.	Value:	a. Uniform Distribution
6.	Resource Use Probability:	a. Fixed Value per Run
7.	Exclusive Use Probability:	a. Fixed value per run

large standard deviation of computation times and laxities, the exponential distribution is used; and to model the systems with a moderate standard deviation of the computation times and laxities, the Erlang-3 distribution is chosen.

The metric used to validate the Well-Timed Scheduling—the accrued value—required a careful choice of the value assignments. To prevent the schedulers' bias towards short execution times, the task values are assigned independently from the computation times. Next, to prevent the effect that a very small percentage of tasks actually contributes most of the accrued value, for example, 10% of tasks contributes 90% of the accrued value, the uniform distribution is chosen.

The resource requirements are assigned to tasks using two fixed parameters: the probability that a resource is used (R_{use}) and the probability that the resource is used in an exclusive mode (R_{ex}). The values of both parameters are chosen to model a very stressful environment. The probability of using a resource is set to $R_{use} = 0.3$, and the mode of use is set to $R_{ex} = 0.5$, so that the tasks have equal chance of using

the resource in exclusive or shared mode. The choice of both parameters is based on the results from the Ph.D. dissertation that focuses on the scheduling of the tasks with resource constraints [48].

3.3.3 The Punctual Points

The punctual points are used to control the tasks' eligibility for scheduling. When a task's laxity is smaller than the given punctual point, the task is considered "relevant" and, thus, eligible for immediate scheduling. In the previous chapter, the analytical derivation of the punctual points is presented, and validated through simulation. In this chapter, the validation of the proposed punctual points derivations is carried one step further. The applicability of the punctual points is tested in a more complex environment, considering the resource requirements and realistic scheduling costs.

By adding the resource requirements, the effective system load changes. Specifically, the higher the probability of conflicts, the more idle processing time is created. This idle processing time acts as an additional load imposed to the system. The idle tasks require "processing time" but do not contribute any value. The load description and the probability that two tasks will conflict over the same resource is presented in [49]. In a multiprocessor system, more than two tasks can compete for the same resources at the same time. For this reason, we extended the original formula to reflect the conflict probability when n tasks are competing for the resources at the same time.

$$P_c = 1 - \left(\sum_{s=0}^{n} \binom{n}{s} (1 - P_{use})^{n-s} (P_{use} P_{sh})^s + \binom{n}{e} (1 - P_{use})^{n-e} (P_{use} P_{ex})\right)^R. \quad (3.1)$$

In Equation 3.1, P_c is a probability that a task will have a conflict when n tasks are competing for R resources, including the processors. s is the number of

Table 3.3 Punctual Points for $M/M/2$ System

ρ	0.7	0.9	1.2	1.6	2.0
$T_P(0.95)$	5.0	15.0	8.8	4.0	3.0
$T_P(0.999)$	11.5	34.1	20.0	9.2	6.9
$T_P(1-10^{-6})$	22.9	54.5	38.1	18.1	13.7

requests in a shared mode, and e is the number of requests in an exclusive mode. P_{use} is a probability that a resource is used. P_{ex} and P_{sh} are the probabilities that a resource is used in exclusive mode or shared mode, respectively. Equation 3.1 indicates that the conflict probability increases exponentially with the increase in the number of resources. This increase directly affects the quality of the punctual points. That is, the punctual points are calculated for a given number of processors, arrival distribution, service distribution, and the system load. If the system load changes due to resource conflicts, the calculated punctual points become less accurate.

To test the range of applicability of Well-Timed Scheduling, our goal is to test the effectiveness of the punctual points in complex real-time systems with a realistic scheduling cost and potential resource conflicts. In other words, the robustness of the analytically derived punctual points in a very stressful environment is tested.

The values of the punctual points used in these test, expressed in multiples of expected service time (ES), are tabulated in three tables. Table 3.3 lists the

Table 3.4 Punctual Points for $M/E_3/2$ System

ρ	0.7	0.9	1.2	1.6	2.0
$T_P(0.95)$	3.2	9.9	5.6	2.4	1.7
$T_P(0.999)$	7.1	22.7	12.8	5.2	3.7
$T_P(1-10^{-6})$	13.6	32.6	22.5	10.4	7.2

Table 3.5 Punctual Points for $M/E_3/8$ System

ρ	0.7	0.9	1.2	1.6	2.0
$T_P(0.95)$	0.85	2.5	1.4	0.68	0.5
$T_P(0.999)$	2.3	5.7	3.2	1.4	1.1
$T_P(1-10^{-6})$	3.1	8.1	5.5	2.5	1.8

punctual points derived for $M/M/2$ system. In this table, as in the following two tables, three different punctual points are tabulated, namely $T_P(0.95)$, $T_P(0.999)$, and $T_P(1-10^{-6})$. Table 3.4 and Table 3.5 present the analytically derived punctual points for $M/E_3/2$ and $M/E_3/8$ systems, respectively.

3.3.4 Simulation Parameters

The above descriptions of the system and task parameters are given in a very general form. The actual parameter values used in simulations are tabulated in Table 3.6. The tasks arrive according to Poisson distribution. The number of generated arrivals exceeds 3000 arrivals per run. The generated loads range from high loads to heavy overloads, namely form $\rho = 0.7$ to $\rho = 2.0$. Additionally, setting an expected service time (ES) to a fixed value, the laxities range from tight to very loose laxities, that is, l ranges from $1ES$ to $64ES$. Both the computation times and the laxity values are assigned according to exponential and Erlang-3 distributions. The probability that the resource is requested, R_{use}, is 0.3, and the probability that the requested resource is used in an exclusive mode, R_{ex}, is 0.5.

The effects of the Well-Timed Scheduling on the performance of real-time schedulers are tested under two conditions: with and without the punctual points. (When tested without the punctual points, the modeled system corresponds to a traditional real-time system where all arrived tasks are eligible for scheduling.) The punctual points for three tested systems, namely for $M/M/2$, $M/E_3/2$, and $M/E_3/8$ systems

Table 3.6 Simulation Parameters.

1.	Load Generation:	a. Pregenerate Workload
2.	Task Arrival Distribution:	a. Poisson Distribution
3.	Load Range:	a. $\rho = 0.7, 0.9, 1.2, 1.6, 2.0$
4.	Number of Arrivals:	a. Over 3000 per Simulation Run
5.	Laxity Range in $[ES]$	a. $l = 1, 2, 4, 8, 16, 32, 64$
6.	Service and Laxity Distributions:	a. Exponential Distribution b. Erlang-3 Distribution
7.	Resource Use Probability:	a. $R_{use} = 0.3$
8.	Exclusive Use Probability:	a. $R_{ex} = 0.5$
9.	Punctual Points:	a. No Punctual Point b. $T_P(0.95)$, $T_P(0.999)$, $T_P(1^-)$
10.	Number of Application Processors:	a. Two Processors b. Eight Processors
11.	Scheduling Cost Proportionality Constant:	a. $SCF[ES]$ $10^{-6}, 2 \times 10^{-3}$, $4 \times 10^{-3}, 8 \times 10^{-3}, 16 \times 10^{-3}$

are tabulated in Table 3.3, Table 3.4, and Table 3.5, respectively. To observe the behavior of the system with a realistic scheduling overhead, the scheduling cost proportionality constant (SCF) is varied from a negligible cost to a very high cost. The overall scheduling cost (SC) is computed as

$$SC = n^2 \times SCF.$$

That is, the overall scheduling cost is a function of the number of scheduled tasks, the computational complexity of the algorithm used, and the scheduling cost per

task. The actual SCF values, given in terms of the expected service time, are listed in line 11 of Table 3.6.

This concludes the descriptions of the simulation parameters. The in-depth analysis of the DLVD and RDS algorithms in Well-Timed Scheduling framework is discussed next.

3.4 Integration Analysis

The analysis of the system behavior, the quality of the analytically derived punctual points, and the overall benefits of the integration of Well-Time Scheduling with two real-time schedulers are presented in this section. The integration analysis consists of four parts: (1) analysis of DLVD and RDS algorithms performing task scheduling in the traditional manner (the tasks are schedulable at arrival time), and with a near ideal cost (2) analysis of the limitations of the DLVD and RDS algorithms used in more realistic situations is created by the increase in scheduling cost proportionality constant (3) analysis of the effects of the integration of Well-Timed Scheduling and the better performing RDS algorithm, and finally (4) analyses of the range of the applicability of Well-Timed Scheduling and the robustness of the punctual points.

3.4.1 Analysis of DLVD and RDS Algorithms

A comparison and analysis of the performance of the DLVD and RDS algorithms, using a traditional approach of scheduling tasks without the Well-Timed Scheduling framework (i.e., the system without the punctual points) is presented in this section. In the traditional approach, all incoming tasks are eligible for immediate scheduling, regardless of the system load. To obtain the highest possible performance, the scheduling cost proportionality constant is set to a negligible value of $SCF = 10^{-6}ES$, producing a near zero scheduling overhead.

Both algorithms are subjected to two types of load and over the wide range of traffic intensities. In the first type, the tasks' computation times and laxities are generated using the exponential distribution—the distribution with a large standard deviation of 100%. In the second type, the computation times and laxities are generated using the Erlang-3 distribution—the distribution with a moderate standard deviation of 57.7%. The performance metric used is the value loss ratio, the ratio of the total accrued value over the total submitted value.

Let's first compare the algorithms for loads with a high standard deviation of computation times and laxities—the exponential distribution.

The value loss ratios for DLVD and RDS algorithms plotted against the laxities (expressed in multiples of expected service time) are given in Figure 3.6 and Figure 3.7, respectively. The modeled system has $R_{use} = 0.3$, $R_{ex} = 0.5$, a negligible SCF, and without the punctual points. The algorithms are tested from moderate loads to high overloads, namely, from $\rho = 0.7$ to $\rho = 2.0$.

Analyzing the figures, it can be observed that the RDS algorithm outperforms the DLVD algorithm in overloads, and for laxities larger than $l = 16$. For example, for $l = 16$ and $\rho = 2.0$, the value loss ratio for DLVD algorithm is 0.19, whereas the value loss ratio for RDS algorithm is 0.17. The difference is 2%. (Observe that in these figures, the largest confidence interval has the value of 1%.) For the same load but laxity $l = 64$, the value loss ratios are 0.15 and 0.12, for DLVD and RDS, respectively. That is, the RDS algorithm outperforms the DLVD algorithm by 3%. Similarly, for $\rho = 1.2$ and laxity $l = 64$, the RDS algorithm outperforms the DLVD by 2%. On the other hand, for $\rho = 0.7$ and $\rho = 0.9$ both algorithms exhibit no significant difference in performance.

A similar performance trend is observed when the computation times and laxities are generated using the Erlang-3 distribution—the distribution with a moderate standard deviation. The value loss ratios versus laxities (for the same range of loads

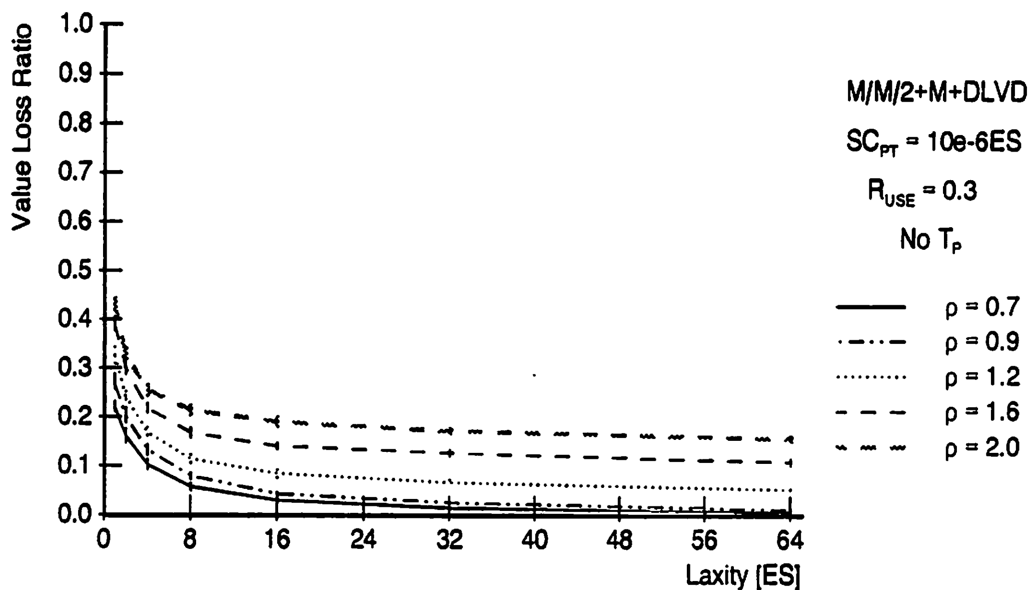

Figure 3.6 DLVD—Value Loss Ratios for Exponential Distributions.

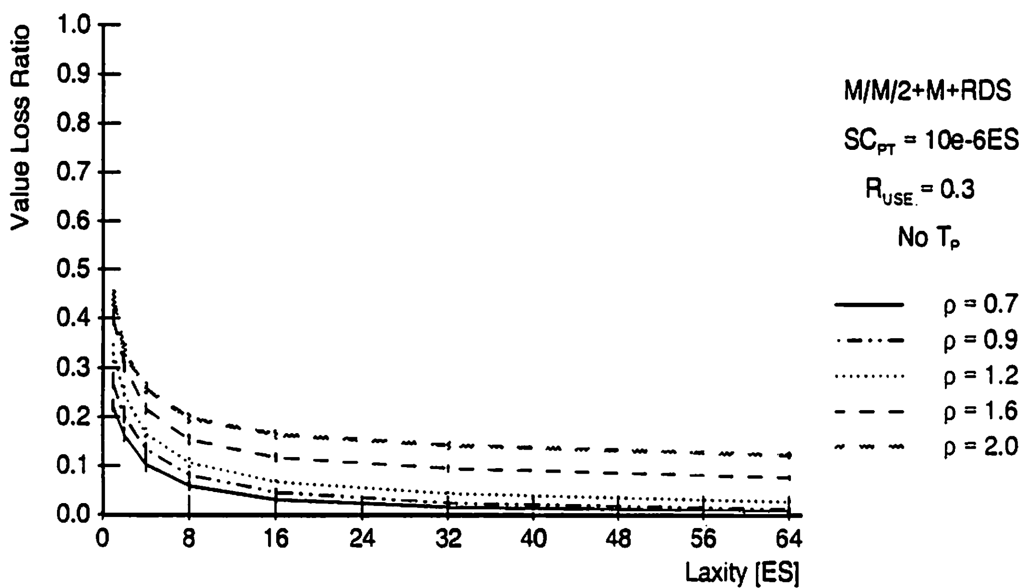

Figure 3.7 RDS—Value Loss Ratios for Exponential Distributions.

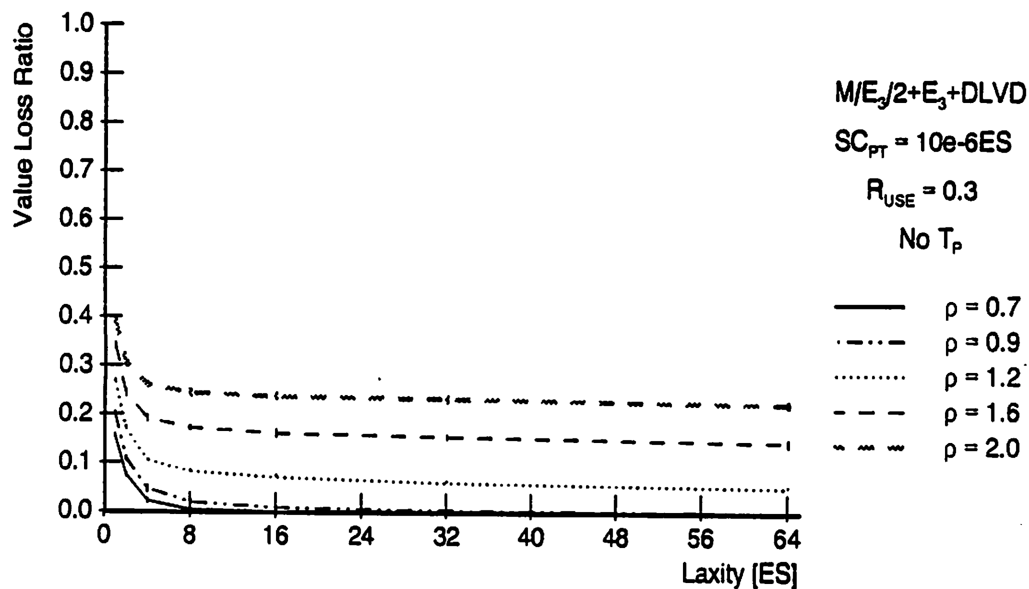

Figure 3.8 DLVD—Value Loss Ratios for Erlang-3 Distributions.

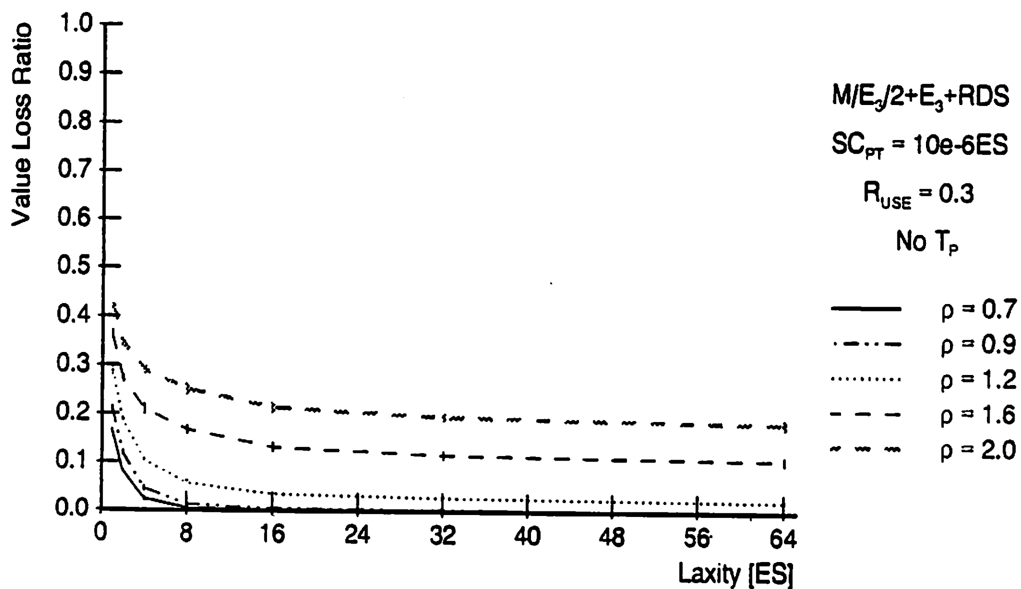

Figure 3.9 RDS—Value Loss Ratios for Erlang-3 Distributions.

as above) are plotted in Figure 3.8 for the DLVD algorithm and in Figure 3.9 for the RDS algorithm.

The better performance of the RDS algorithm is again observed in overloaded systems, and no significant difference is observed in systems with moderate and high loads. The earliest difference in performance is observed at laxity $l = 8$. At this laxity, load $\rho = 1.2$ exhibited the largest difference. Specifically, the value loss ratio for the DLVD algorithm is 0.09 and for the RDS algorithm is 0.06, creating a difference of 3%. The largest difference in both graphs is observed for high overloads and high laxities. Specifically, the value density of the DLVD algorithm for $\rho = 2.0$ and $l = 64$ is 0.23, and for the RDS algorithm is 0.18. The difference is 5%.

In summary, both algorithms perform similarly for the two tested distributions (exponential and Erlang-3 distributions) with a slight edge for the RDS algorithm in high overloads and moderate to large laxities.

The effects of the two distributions on the overall performance exhibits two major differences: (1) the value loss ratios for Erlang-3 distribution saturate faster than the corresponding value loss ratios for exponential distribution, and (2) the value loss ratios in overloads and for large laxities are higher for Erlang-3 than for exponential distribution. For example, for the DLVD algorithm, $\rho = 0.7$ and exponential distribution (Figure 3.6) the value loss ratio approaches zero at laxity $l =$ 64. While for Erlang-3 distribution, the value loss ratio, under the same conditions, approaches zero at laxity $l = 8$ (Figure 3.8). Similar trends can be observed for other loads, including the overloads. Furthermore, due to the very close performance of the two algorithms, the RDS algorithm exhibits the same trends as the DLVD algorithm (see Figure 3.7 and Figure 3.9).

To understand the reasons for the higher value loss ratios for Erlang-3 distribution, we analyzed the laxity over computation time (the L-C ratio) per task. The averaged values are calculated and the results are plotted against the mean

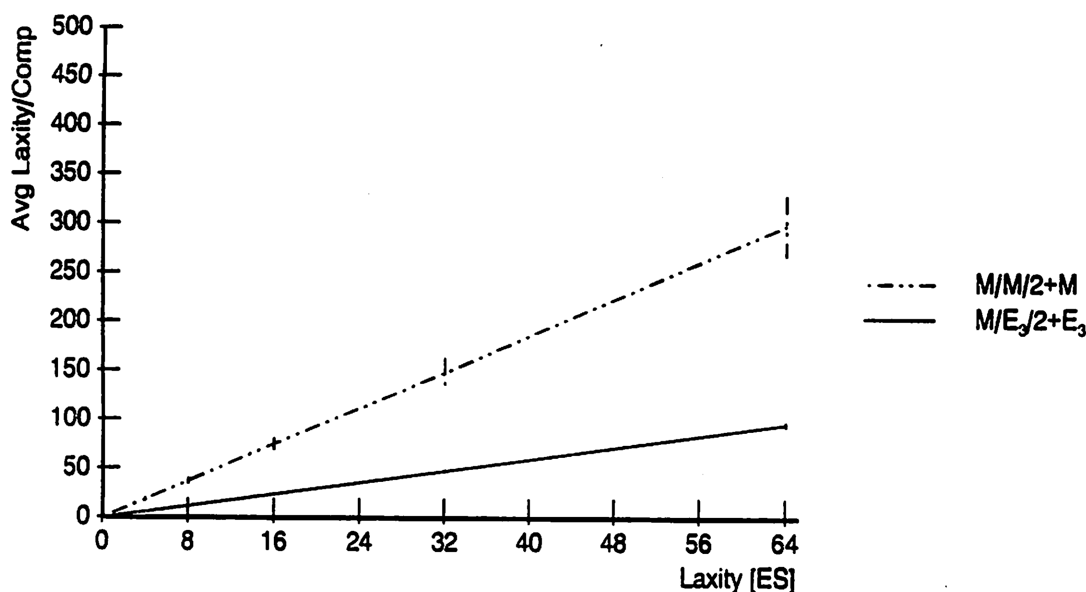

Figure 3.10 Average Laxity/Computation for Exp. and Erlang-3 Distributions.

laxity values in Figure 3.10. From this figure, it is evident that the L-C ratio for exponential distribution has a steeper gradient than the L-C ratio for Erlang-3 distribution. This indicates that for an exponential distribution, large laxities are more often combined with small laxities than for an Erlang-3 distribution. In other words, a larger standard deviation provides a larger number of easy to schedule tasks, the tasks with short computation times and large laxities. This and the uniform distribution of the contributing values indicate that systems with exponential distribution can potentially achieve a higher accrued value within the same time interval, and consequently produce a lower value ratio.

The previously observed differences in the system performance, caused by the two distributions, are explained below.

Difference (1): *Why does the value loss ratio saturate faster for Erlang-3 than for exponential distribution?*

Exponential Distribution: Occasionally, a task with very long computation time hogs the resources (CPU plus other requested resources). This is due to the

large standard deviation of the exponential distribution. Holding the resources for extended periods of time and executing the tasks that might not have high values, results in missed opportunities to execute some more valuable tasks that have shorter deadlines. This, on the other hand, leads to a slower saturation towards the point where the increase in laxities does not make any significant difference.

Erlang-3 Distribution: More uniform computation and laxity values guarantee that no task will block the resources (CPU plus other resources) for an extended period of time. This means that new tasks, with higher value densities, can start their executions faster. Furthermore, more uniform laxities give new tasks higher chance to be scheduled. For these reasons, the saturation point of the value loss ratio is reached relatively fast.

Difference (2): *Why is the value loss ratio for Erlang-3, in extreme cases, higher than for exponential distribution?*

Exponential Distribution: Higher variance in computation times means that more tasks with short computation time and higher contributing values can be executed. That is, overloads create a large choice of tasks, and exponential distribution guarantees that a large number of them are with short computation times. Therefore, very rarely will the system choose to run a task with large computation time and small value.

Erlang-3 Distribution: More uniform computation times and laxities indicate a weak bias towards creating the tasks with short computation times and high contributing values. Therefore, the number of short tasks with large values, when compared to the exponential distribution, is smaller.

To strengthen the above conclusions, let's take a look at the *task loss ratios* (the ratio of executed tasks over the total number of submitted tasks) for DLVD

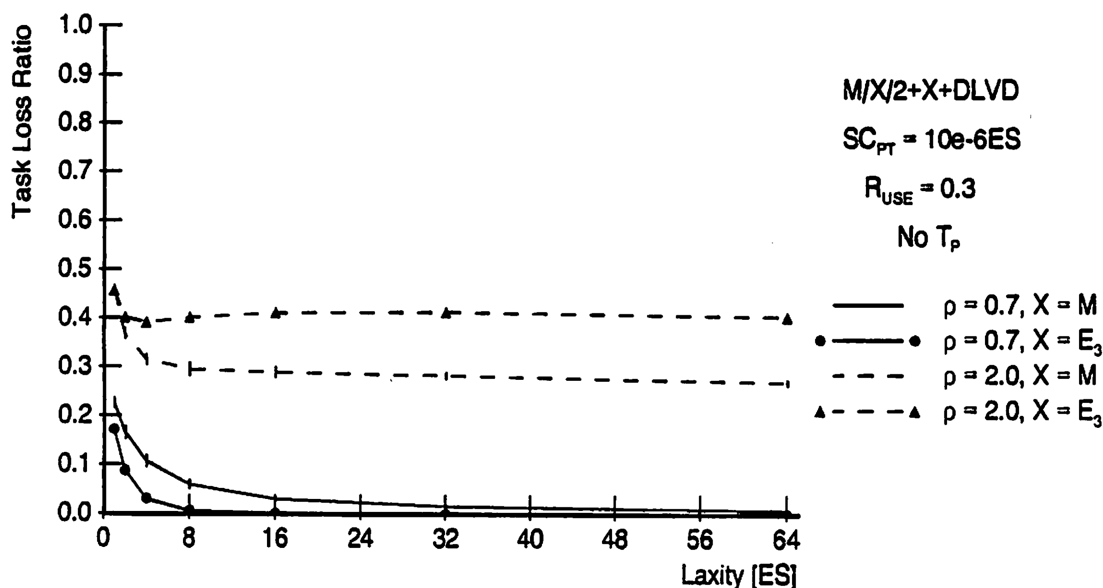

Figure 3.11 DLVD—Task Loss Ratios for Exp. and Erlang-3 Distributions.

algorithm. Figure 3.11 plots the task loss ratios for two loads ($\rho = 0.7$ and $\rho = 2.0$) and for two distributions (exponential and Erlang-3 distributions). The curves with filled curve symbols indicate Erlang-3 distribution, and the curves without the curve symbols indicate the exponential distribution. The same curve type is used to indicate the same system load. It is clear that for $\rho = 2.0$, the system with Erlang-3 distribution rejects more tasks, while for $\rho = 0.7$ the same system rejects less tasks than the one with exponential distribution. This indicates that the system with exponential distributions at $\rho = 2.0$ has stronger bias towards the shorter tasks, while the same system for $\rho = 0.7$ executes less tasks due to the occasional executions of the large tasks with potentially low values.

It is evident, from the above discussion, that Erlang-3 distribution provides larger extremes, that is, it provides lower value loss in low loads and higher value loss in overloads. Additionally, in all experiments performed, the same system behavior, or the same trends were observed for both distributions and for both algorithms. For these reasons, the results for Erlang-3 distribution are presented further on.

3.4.2 The Limitations of DLVD and RDS Algorithms

The negligible cost, used in the previous section, provides the best expected performance, i.e., the least possible value loss for both algorithms. To analyze these algorithms in a more realistic environment, the scheduling cost proportionality constant is increased.

Performing with a negligible cost, both algorithms reach the best expected performance regardless of the number of schedulable tasks. However, in a realistic situation, the scheduling cost is rarely so low that the number of schedulable tasks makes no difference. Very often a scheduling cost presents a significant factor in overall system performance. To model this realistic situation and to find the limitations of both algorithms, this section analyzes the effects of the different scheduling costs proportionality constant (i.e., different scheduling cost factors, SCFs). The SCF is varied from the negligible cost, near ideal cost, to the cost that causes both algorithms to approach total loss of value, i.e., to the breakpoint cost at which the scheduling algorithm looses the control of the system. As with the previous section, both algorithms are used in a traditional manner—tasks are scheduled without the punctual points.

The analysis of the limitations of the algorithms is first performed for a two processor system, and then for an eight processor system. The overall performance is expected to degrade with an increase in SCF and laxities. The increase in laxities is directly reflected in the number of schedulable tasks. That is, the increase in laxities for overloaded systems, essentially, extends the life of the tasks and consequently increases the number of schedulable tasks. This increase is further reflected in the increase of estimated scheduling cost (i.e., the cutoff line), and therefore, a larger number of tasks in STT cannot be rescheduled. The major effects of the increase in SCF and laxities that are especially emphasized in overloads are:

1. The larger the scheduling time, the more tasks will be ready for the next scheduling point, that is, the number of new arrivals increases with the increase of scheduling costs.

2. For systems with given laxity distributions, the constant increase in scheduling overhead increases the number of tasks that have laxities smaller than the scheduling cost.

3. The increase in cutoff line increases the number of tasks left for execution, that is, the number of commitments to the tasks with potentially smaller values than the cutoff line.

3.4.2.1 Two Processor System

Our experiments show that the effects of the increasing SCF are most visible under high overloads. The examples of the value loss ratios for DLVD and RDS algorithms, for $\rho = 2.0$ and over the wide range of loads, are plotted in Figure 3.12 and Figure 3.13, respectively.

The chosen SCF values cover a wide range of costs, from negligibly small scheduling costs proportionality constant ($SCF = 10^{-6}ES$, also called baseline) to a significant cost of $SCF = 0.016ES$. The latter cost produces a scheduling cost of $SCF = 1.6ES$ when $n = 10$ tasks are scheduled ($SC = n^2 \times SCF = 1.6ES$). On the other hand, if $n = 100$, the scheduling cost is $SC = 160ES$. Very loosely speaking, for $n = 10$, the cutoff line is $1.6ES$, and thus, two tasks will be left for execution and eight tasks will be rescheduled. For the case when $n = 100$ and $SC = 160ES$, it appears that no task will be rescheduled. The estimated cutoff line would probably leave all 100 tasks for execution on the application processors.

This observation indicates, first, that both algorithms are susceptible to a loss of schedulability for high SCF and under high overloads, i.e., for a large number of

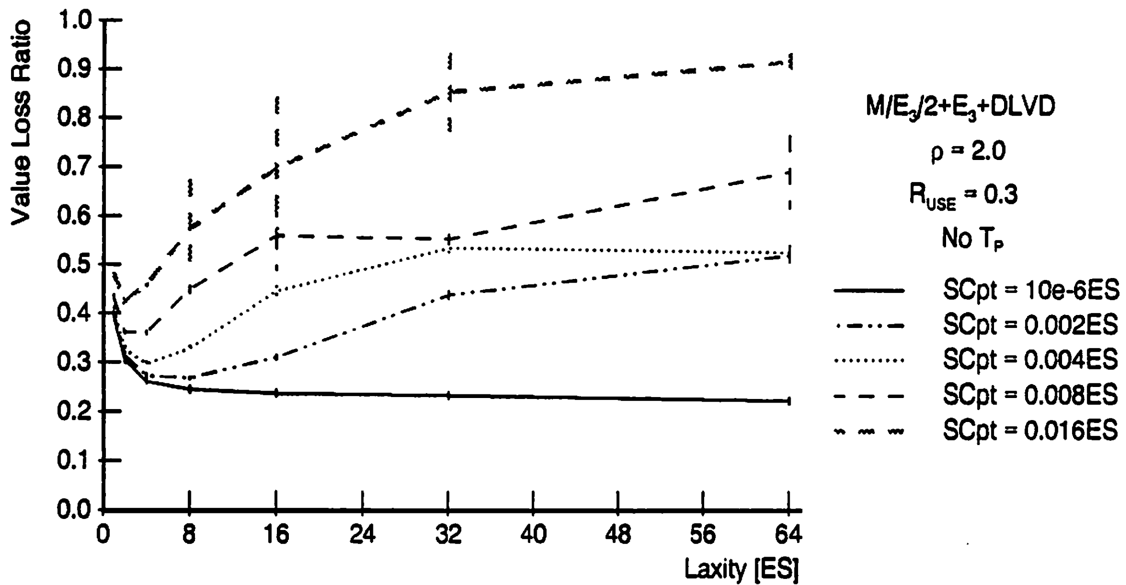

Figure 3.12 DLVD—Value Loss Ratios for $\rho = 2.0$.

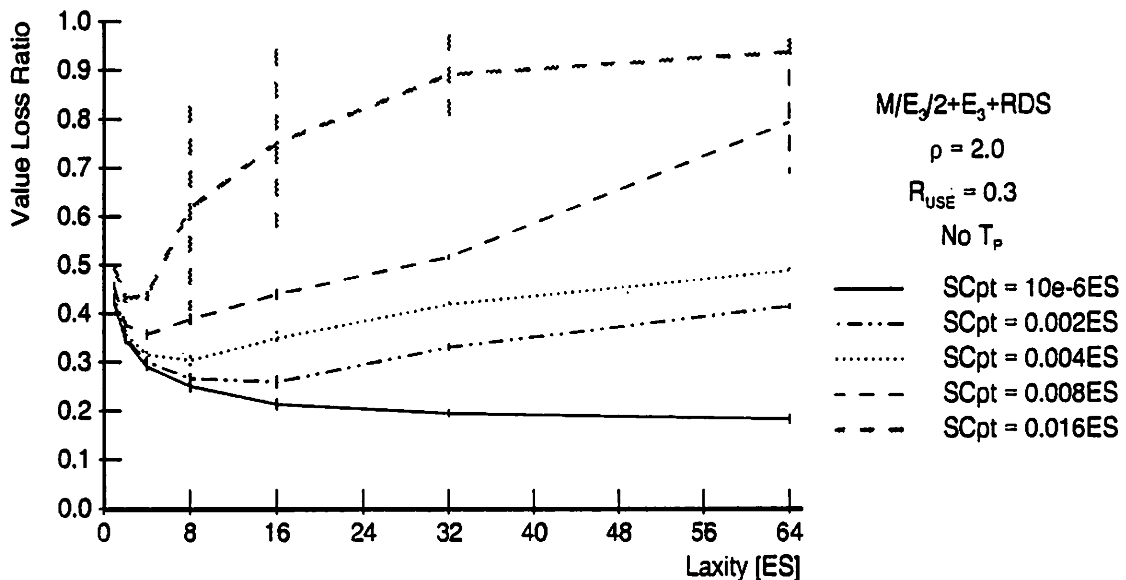

Figure 3.13 RDS—Value Loss Ratios for $\rho = 2.0$.

schedulable tasks. Second, it is easy to observe that under these extreme conditions, a much simpler scheduling algorithm should be used. Likewise, under these extreme conditions, it is more important to perform a simplistic selection that will take less time and thus schedule more tasks. The next choice (the choice later analyzed) is to use the Well-Time Scheduling as a framework that allows a smaller number of tasks to be schedulable at any scheduling point.

As far as the trend in performance is concerned, both algorithms behave the same. In both figures (Figure 3.12 and Figure 3.13), the value loss ratio stabilizes for negligible cost. However, for $SCF = 0.002ES$ and higher, the value loss ratio monotonically increases. It approaches a saturation point at almost 100% loss, for $SCF = 0.016ES$ and laxities of $l = 64$.

In the extreme case, for mean laxities of $l = 64$, the RDS algorithm outperforms the DLVD algorithm most significantly. (The analysis for a baseline, i.e., for a negligible cost, is performed in the previous section, and thus, it will not be repeated in this section.) For $SCF = 0.002ES$, RDS value loss ratio is 0.41 and DLVD's is 0.52 (a difference of 11%). For $SCF = 0.004ES$, it is 0.5 for RDS and 0.51 for DLVD (a difference of only 1%). While, for $SCF = 0.008ES$, it is 0.8 for RDS versus 0.7 for DLVD, but with large confidence intervals that overlap.

The large confidence intervals indicate that the breaking point, the point after which the scheduling cost leads the algorithms into intolerably high value loss ratios, is almost approached. That is, the further increase in scheduling cost proportionality constant ($SCF = 0.016ES$) brings a value loss for RDS to 0.94 and to 0.92 for DLVD—almost a 100% loss with very small confidence intervals. This indicates that SCF of 0.016 presents a breakpoint for these scheduling algorithms at $\rho = 2.0$ and laxities larger than $l = 64$. In fact, the value loss ratios for both algorithms indicate that, for a two processor system, this value loss ratio is intolerably high.

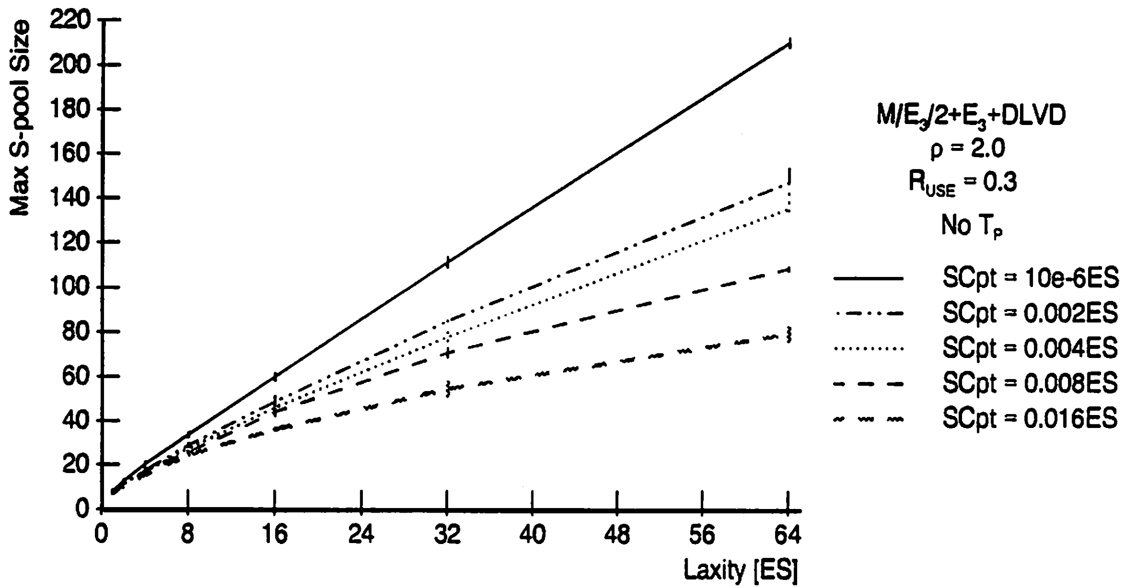

Figure 3.14 DLVD—Maximum S-pool Size for $\rho = 2.0$.

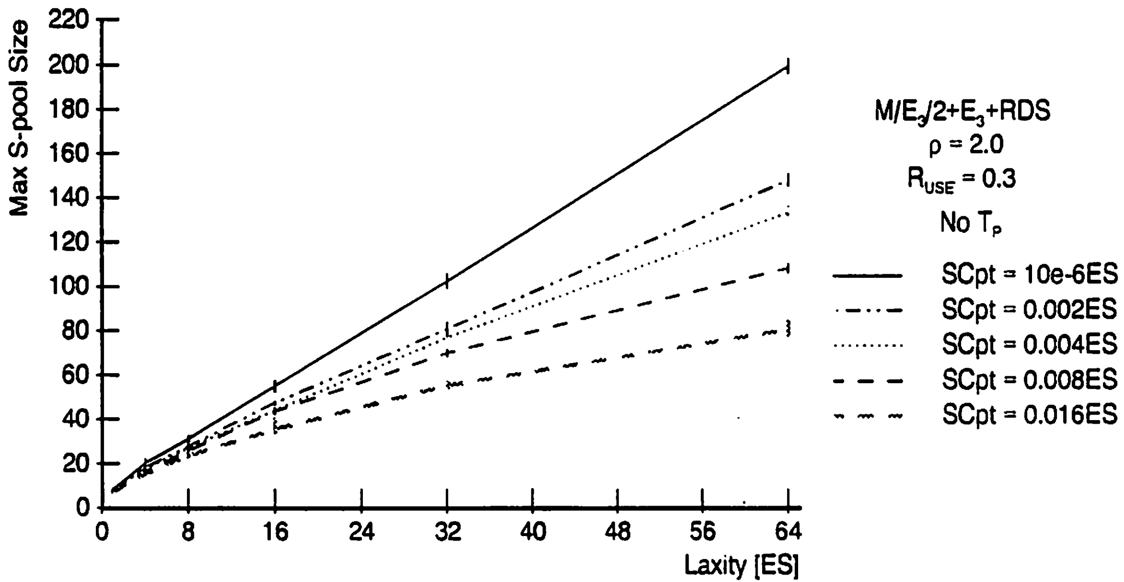

Figure 3.15 RDS—Maximum S-pool Size for $\rho = 2.0$.

As mentioned earlier, the increase in scheduling cost produces the increase in value loss ratios. Figure 3.14 and Figure 3.15 plot the maximum number of tasks found in S-pool (the maximum number of schedulable tasks) for DLVD and RDS algorithms, respectively. It is evident from these figures that the number of tasks in S-pool increases monotonically with the increase in laxities. In spite of slight differences in the maximum S-pool size for the two algorithms, the main point to observe is that the value loss is mainly produced by allowing the scheduling of a large number of tasks.

To better indicate the limitations of the two algorithms, the discussion for an eight processor system is presented next.

3.4.2.2 Eight Processor System

Increasing the number of processors from two to eight increases the value loss ratio for all tested SCF's but for the negligible cost of $SCF = 10^{-6}ES$. Moreover, the eight processor system has worse performance in high overloads. Specifically, for $\rho = 2.0$, the value loss ratios, for all but negligible cost, are saturated and very close to the total loss of accrued value for laxities larger than the expected service time.

The examples of moderate overload for DLVD and RDS algorithms, the load $\rho = 1.2$, are given in Figure 3.16 and Figure 3.17, respectively. A quick glance at the plotted value loss ratios indicates that the RDS algorithm has significantly better performance. For example, for a negligible cost and $l = 64$, RDS has value loss ratio of 0.11 while DLVD has value loss ratio of 0.32—a difference of 21%. Furthermore, for $SCF = 0.002ES$ and at $l = 64$, the RDS algorithm has a value loss ratio of 0.65 while DLVD has a value loss ratio of 0.92. The difference is 27%. Doubling the scheduling cost proportionality constant to $SCF = 0.004ES$ results in a breakpoint value. That is, even though the RDS algorithm outperforms the DLVD algorithm,

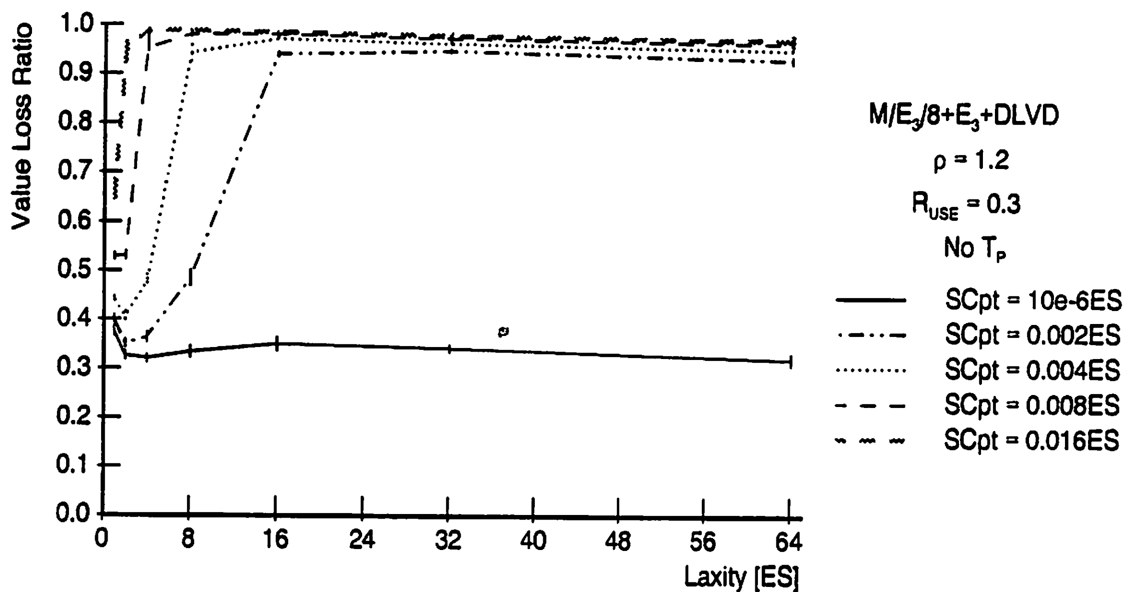

Figure 3.16 DLVD—Value Loss Ratios for 8 CPUs and $\rho = 1.2$.

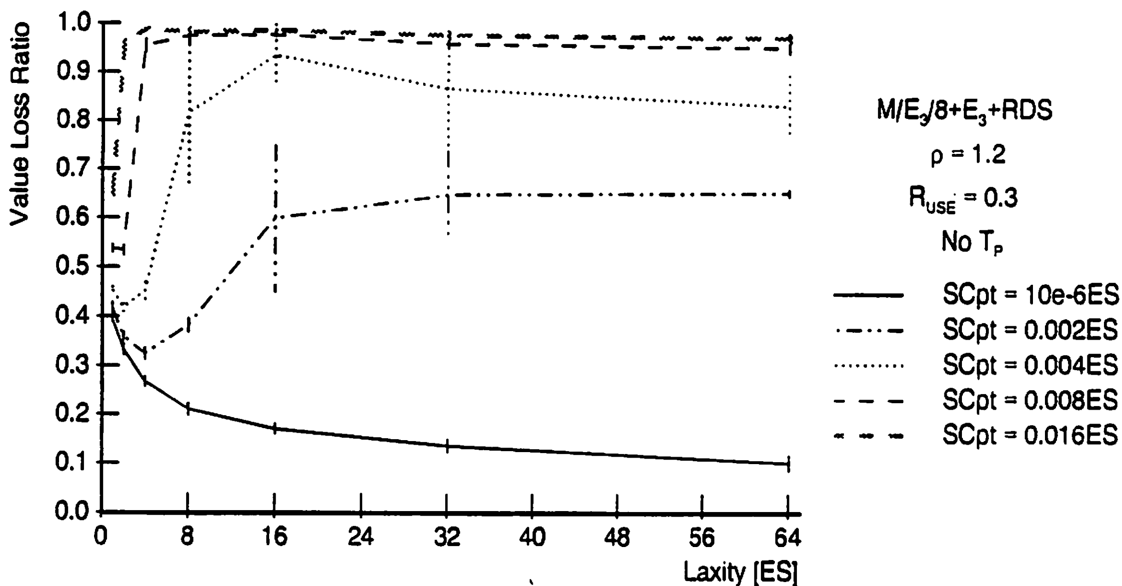

Figure 3.17 RDS—Value Loss Ratios for 8 CPUs and $\rho = 1.2$.

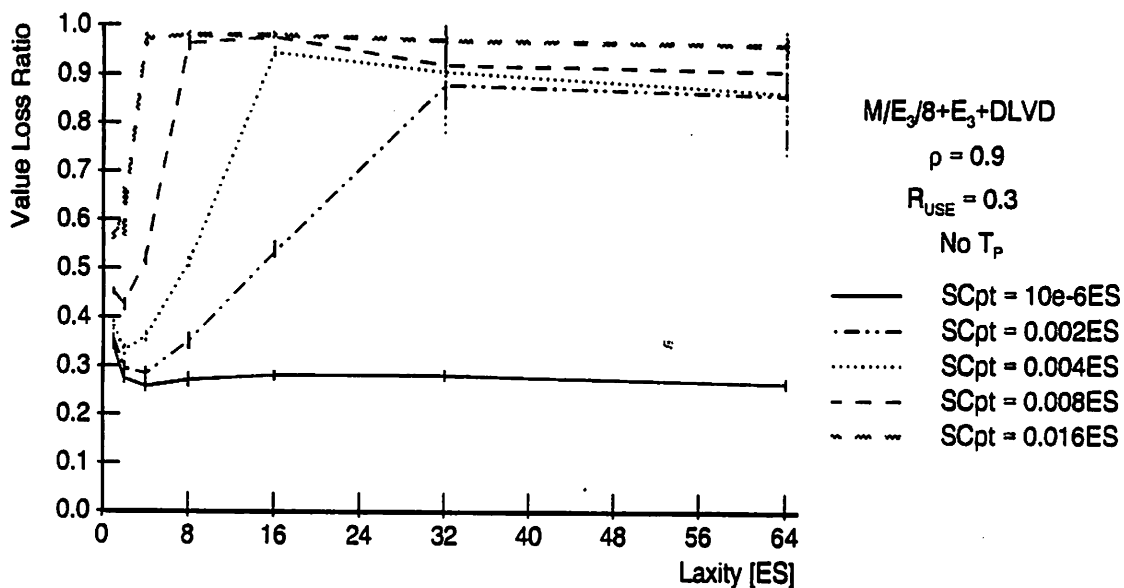

Figure 3.18 DLVD—Value Loss Ratios for 8 CPUs and $\rho = 0.9$.

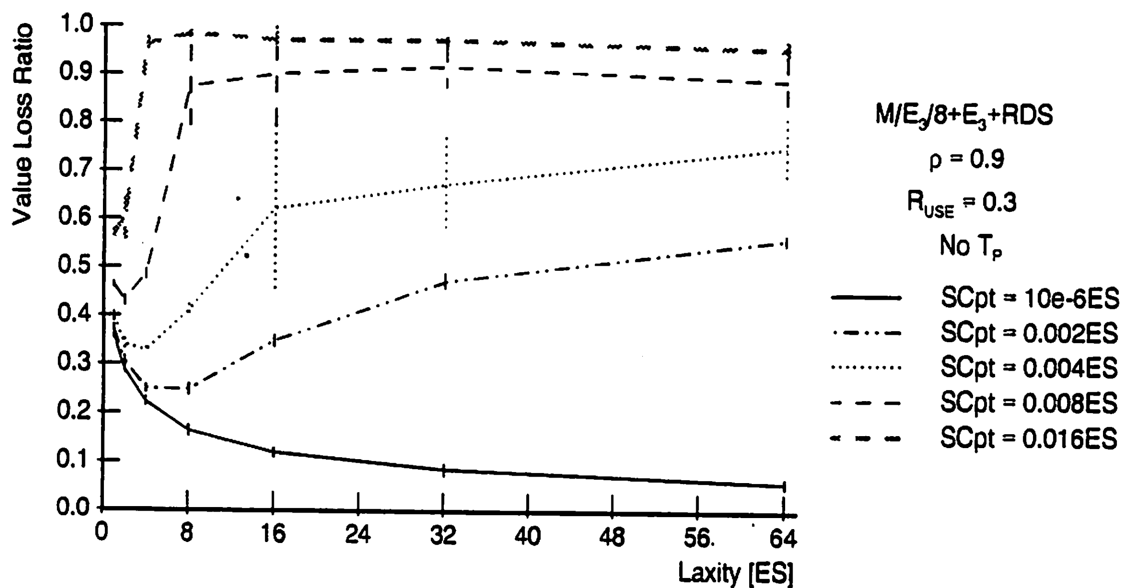

Figure 3.19 RDS—Value Loss Ratios for 8 CPUs and $\rho = 0.9$.

the confidence intervals are very wide indicating the high oscillations in performance. With further increase in scheduling cost proportionality constant both algorithms behave very close. The value loss ratios approach total loss for laxities larger than the expected service time.

To strengthen the conclusion that the RDS outperforms DLVD, the value loss ratios for load of $\rho = 0.9$ are plotted. Figure 3.18 plots the value loss ratios for DLVD algorithm, and Figure 3.19 plots the value loss ratios for RDS algorithm. These figures indicate almost the same observation as above. For a negligible scheduling cost proportionality constant and for $SCF = 0.002ES$, the RDS algorithm significantly outperforms the DLVD algorithm. Somewhat better performance, but again with wide confidence intervals, is observed for $SCF = 0.004ES$. However, this value can still be treated as a breakpoint value; that is, with an additional increase of the scheduling cost proportionality constant, the use of either algorithm leads to an almost total loss of accrued value.

This concludes the comparison and the analysis of limitations of two algorithms for realistic scheduling costs proportionality constant and without the use of the punctual points. From the above, we can conclude that the control of the scheduling cost in overloads is a must. The monotonic increase in value loss ratios that follow the increase in laxities, for all but near ideal scheduling costs, is contrary to expectations. This undesired effect must be avoided.

One way to stop this increase in value loss ratio is to integrate the real-time scheduler and the Well-Timed Scheduling. The analysis of intergated systems, the systems that conform to Well-Time Scheduling framework, is presented in the next section. Due to the better performance of the RDS algorithm, the benefits of using Well-Time Scheduling are less dramatic than if the DLVD algorithm is used. For this reason and because the above results indicate that both algorithms exhibit the same trends, in spite of the differences in performance, only the results for the RDS

algorithm are presented in further sections. Additionally, the analysis that follows does not focus on the exact differences of the two algorithms; it rather discusses the behavior of real-time scheduling algorithms within the Well-Time Scheduling framework.

3.4.3 The Effects of Integration

The analysis in the previous two sections identified that in overloads the value loss ratio increases with the increase in laxities. This, in turn, indicates that the systems of the future—the faster and faster systems that will produce the increase in laxities of tasks whose timing requirements are imposed by the physical constraints—are prone to increase in value loss ratios. In other words, when laxities and loads increase, the number of schedulable tasks increases proportionally leading to intolerably high scheduling costs affecting the overall performance.

Well-Timed Scheduling provides a solution to this problem. Using the punctual points, i.e., scheduling only the "relevant" tasks provides a useful control of the number of schedulable tasks; it lowers the scheduling cost, and extends the applicability of real-time scheduling algorithms.

The benefits of the integration of Well-Timed Scheduling and real-time schedulers, for a wide range of scheduling costs proportionality constant, are discussed and analyzed in this section using a single punctual point $T_P(0.999)$. The analysis is performed on two different multiprocessor systems: a two processor system and an eight processor system. The effects of the different punctual points is discussed in the subsequent section.

3.4.3.1 Two Processor System

The punctual points $T_P(0.999)$, for an $M/E_3/2$ system and loads ranging from $\rho = 0.7$ to $\rho = 2.0$ are tabulated in Table 3.4, Section 3.3. For example, for $\rho = 2.0$ and $\psi = 0.999$, from Table 3.4, the actual value of the punctual point

$T_P(0.999) = 3.7ES$. (The unit, ES, is assumed for all values of the punctual points, but for brevity, it is omitted in the text.) All tabulated punctual points are analytically derived according to the method described in Section 2.2, Chapter 2. The derivation of the punctual points does not depend on the timing or resource constraints of a given system; it rather depends on the system load, number of processors and distribution of the service time, assuming Poisson arrivals. On the other hand, the system discussed in this section is more complex—it is a realistic system where tasks have timing and resource constraints, and where a scheduling cost has a significant influence on the overall performance. In this complex real-time system, the analytically derived punctual points are not used as exact parameters that control the schedulability of incoming tasks as they were in ideal systems presented in Chapter 2. They are rather treated as approximations whose quality and applicability are discussed in this section as well as in the subsequent section. As before, the performance metric used is the value loss ratio.

Through this section, the performance of two systems—the systems with the punctual points and the systems without the punctual points—is compared side by side. Specifically, the pairs of graphs for systems with and without the punctual points are presented for loads of $\rho = 2.0, 1.2$, and 0.9.

Figure 3.20 and Figure 3.21 plot the value loss ratios for the two processor system with load $\rho = 2.0$. In the former figure, the results for the system without the punctual point are presented; while the results for the system with the punctual point are presented in the latter figure. (Figure 3.20, already plotted in the previous section is repeated here for easy comparison of the two approaches).

By analyzing the performance of the two systems, the system with the punctual points shows: (1) the improvement in performance (for realistic scheduling costs per task), and (2) the degradation in performance (for a negligible scheduling cost per task).

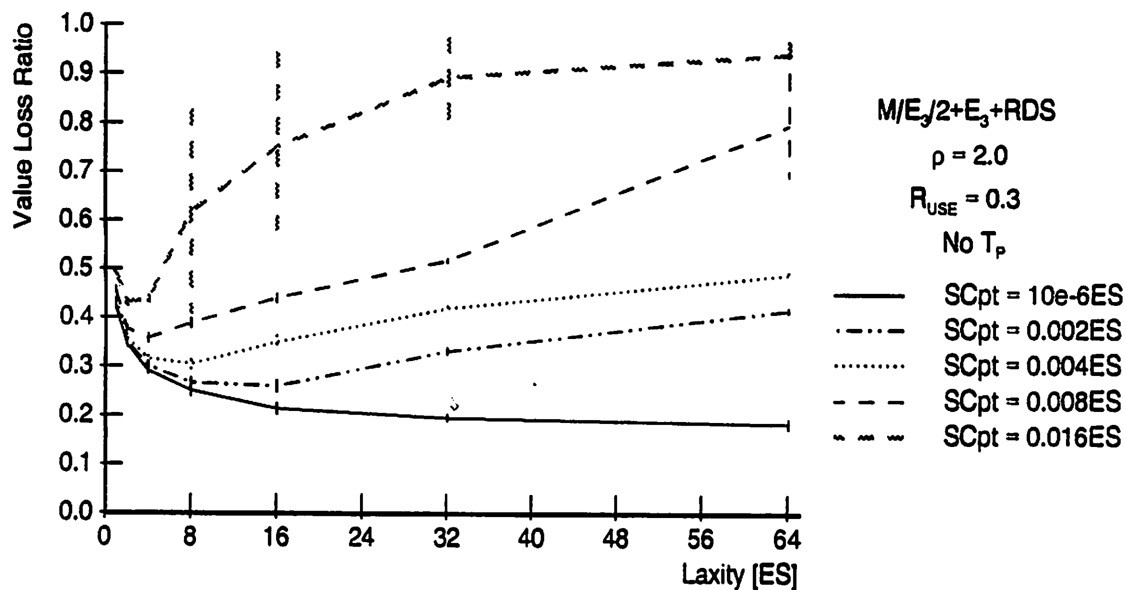

Figure 3.20 RDS—Value Loss Ratios for $\rho = 2.0$.

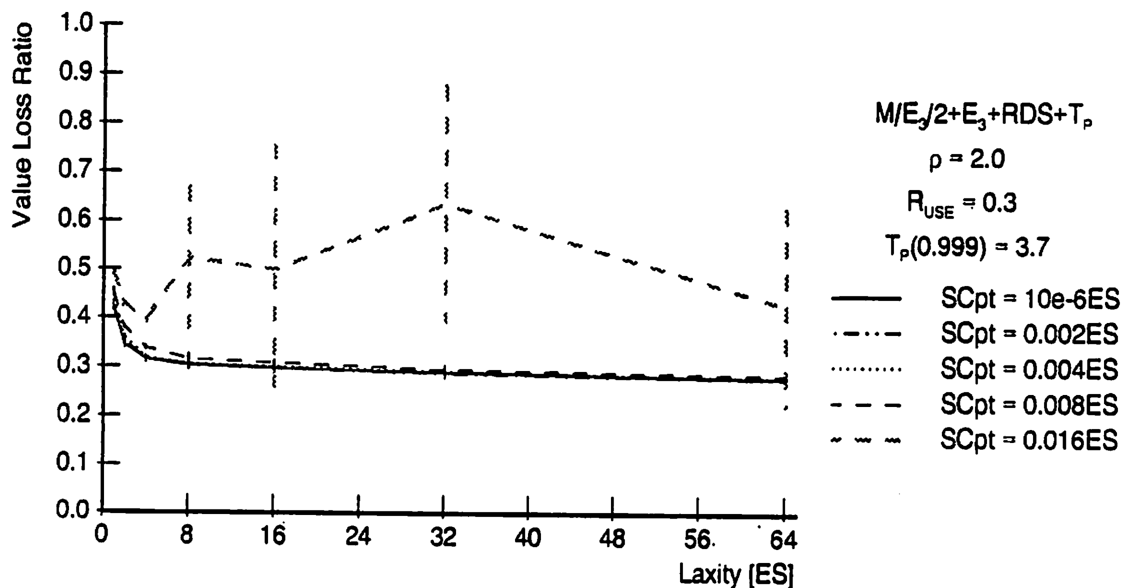

Figure 3.21 RDS—Value Loss Ratio for $\rho = 2.0$ and $T_P(0.999)$.

The improvements in performance are observed for $SCF = 0.002ES, 0.004ES$, and $0.008ES$, with the most pronounced improvement at laxity $l = 64$. For example, the value loss ratio for all three SCF's and in the system that uses the punctual point (Figure 3.21) is 0.28. On the other hand, the system that does not use the punctual points has value loss ratios of 0.42, 0.5, and 0.8 for $SCF = 0.002ES, 0.004ES$, and $0.008ES$, respectively. The largest improvement in performance is observed for $SCF = 0.008ES$. The value loss ratio without the punctual point is 0.8, while the value loss ratio with the punctual point is 0.28. The absolute difference is 51%.

The further increase in the scheduling cost proportionality constant ($SCF = 0.016ES$) results in very wide confidence intervals in both systems. Nevertheless, with the punctual points the results are, on average, much better than without them; the value loss ratio of 0.94 in a system without the punctual points is reduced to 0.43 in a system with the punctual points. The dramatic improvements in performance when using the Well-Timed Scheduling are primarily due to a much smaller number of schedulable tasks.

Aside from the relative improvements in performance, the fast saturation of the value loss ratios characterizes the systems with the punctual points. The value loss ratios saturate at moderately large laxities; while in the systems without the punctual points, the value loss ratios increase monotonically with the increase in laxities.

However, the reduced number of schedulable tasks is not always beneficial. When a scheduling cost proportionality constant is negligibly small, that is, when a very large number of tasks is schedulable in almost zero time, reducing the number of schedulable tasks actually results in an increase of the value loss ratios. Specifically, the value loss ratio of 0.19 (Figure 3.20) increases to 0.28 when the punctual point $T_P(0.999)$ is used, and when the schedulable tasks are reduced to "relevant" tasks

only. This creates the absolute difference of 9%. Thus, in an ideal, zero time scheduling case, the use of the punctual points is not beneficial.

A similar trend in performance is observed for other loads. For example, for $\rho = 1.2$ (a moderate overload), Figure 3.22 and Figure 3.23 present the value loss ratios for traditional systems (the systems without the punctual points) and for the systems with the punctual points, respectively. For this load, both systems exhibit the same behavior as above. The value loss ratios monotonically increase when the punctual point is not used; while the value loss ratios become very stable and with a significant decrease in the value loss ratios when the punctual point is used. Comparing the results for $\rho = 1.2$ versus the results for $\rho = 2.0$ for $SCF = 0.016ES$, it is evident that the value loss ratio for $\rho = 1.2$ has much narrower confidence intervals; that is, for $\rho = 1.2$ the performance is stable, while for $\rho = 2.0$ the performance oscillates heavily. This indicates that for moderate overloads, the cost of $SCF = 0.016ES$ is not a breakpoint in either of two systems (of course, considering the plotted laxities of up to $l = 64$). Nevertheless, even this system is expected to approach the intolerably high loss with the further increase in laxities.

In the final example of the Well-Timed Scheduling benefits, the value loss ratios for load $\rho = 0.9$ are presented. Figure 3.24 plots the value loss ratios for a system without the punctual points; while Figure 3.25 plots the value loss ratios for a system that operates within the Well-Timed Scheduling framework and uses the punctual point $T_P(0.999)$.

Once again, the benefits of the Well-Timed Scheduling are evident, even though on a much smaller scale. Both systems behave the same for all tested SCF's, except for $SCF = 0.016ES$. At this scheduling cost proportionality constant in the system without the punctual point, the value loss ratio shows the slight increase for

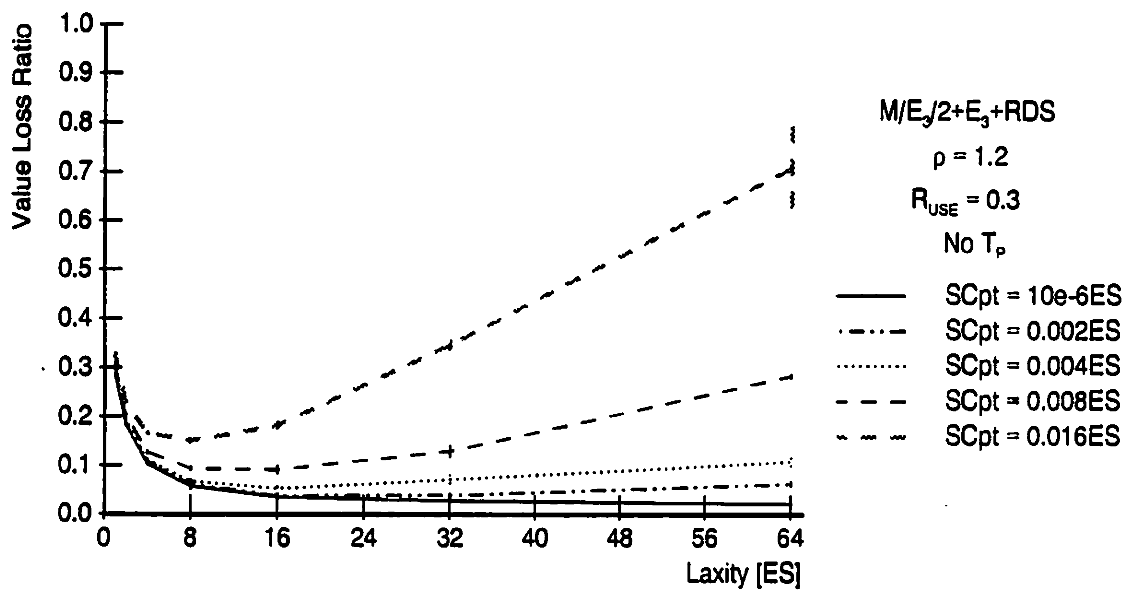

Figure 3.22 RDS—Value Loss Ratios for $\rho = 1.2$.

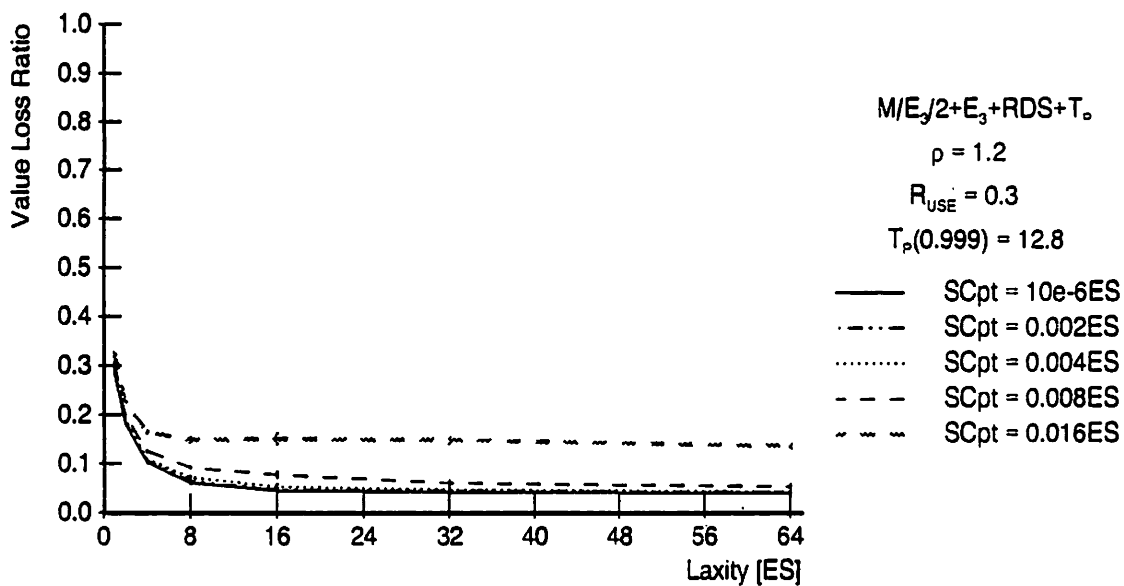

Figure 3.23 RDS—Value Loss Ratio for $\rho = 1.2$ and $T_P(0.999)$.

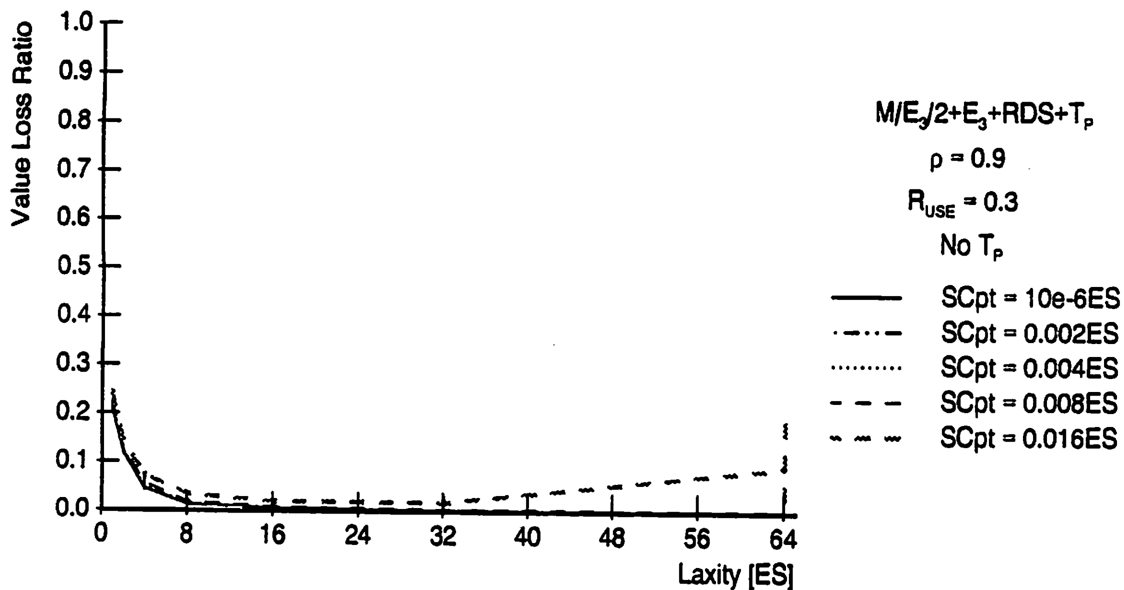

Figure 3.24 RDS—Value Loss Ratios for $\rho = 0.9$.

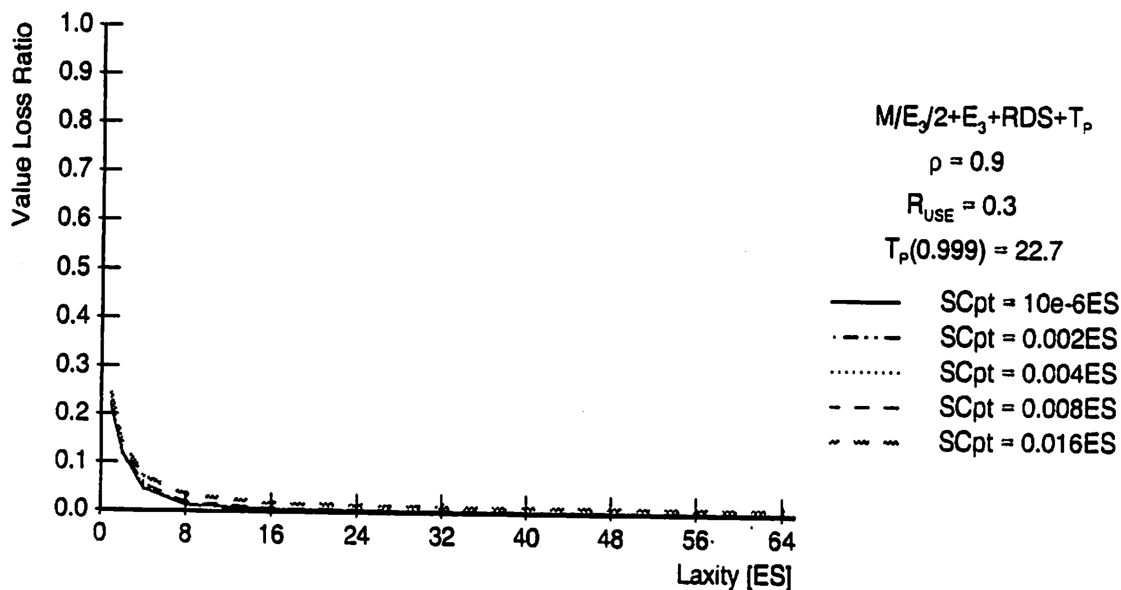

Figure 3.25 RDS—Value Loss Ratio for $\rho = 0.9$ and $T_P(0.999)$.

laxities over $l = 64$. Whereas under the same conditions, the system that utilizes the punctual points has a very stable performance and a low value loss ratio.

This example concludes the analysis of a two processor system with an evident proof of the benefits of using the Well-Timed Scheduling, especially for very large loads and/or laxities.

3.4.3.2 Eight Processor System

To test the usefulness of the Well-Timed Scheduling framework in an even more stressful environment, we increased the number of processors from two to eight. This increases the potential parallelism, but with it the probabilities of resource incurred conflicts, too. That is, it lowers the probability that a newly arrived task will be scheduled. The same load range is used to demonstrate the benefits of the Well-Timed Scheduling in this very stressful environment.

Figure 3.26 plots the value loss ratios for an eight processor system that is subjected to load $\rho = 2.0$, and that does not utilize the punctual points. Figure 3.27, on the other hand, plots the value loss ratio under the same conditions but with the punctual points.

By analyzing these figures, the following is observed:

1. The Well-Timed Scheduling exhibits significant improvements for $SCF = 0.002ES$ and $SCF = 0.004ES$. Specifically, for $SCF = 0.002ES$ and $l = 64$, the value loss ratio without the punctual point is 0.95 (i.e., it is very close to a total loss); while the value loss ratio with the punctual point is 0.43. The difference in performance is 52%. Similarly, for $SCF = 0.004ES$ and $l = 64$, the value loss ratio without the punctual point is 0.97 versus the value loss ratio of 0.49 with the punctual point—the absolute difference of 49%.

2. The degradation in performance is evident if the SCF is approaching zero cost. That is, the value loss ratio of 0.21 (for a system without the punctual

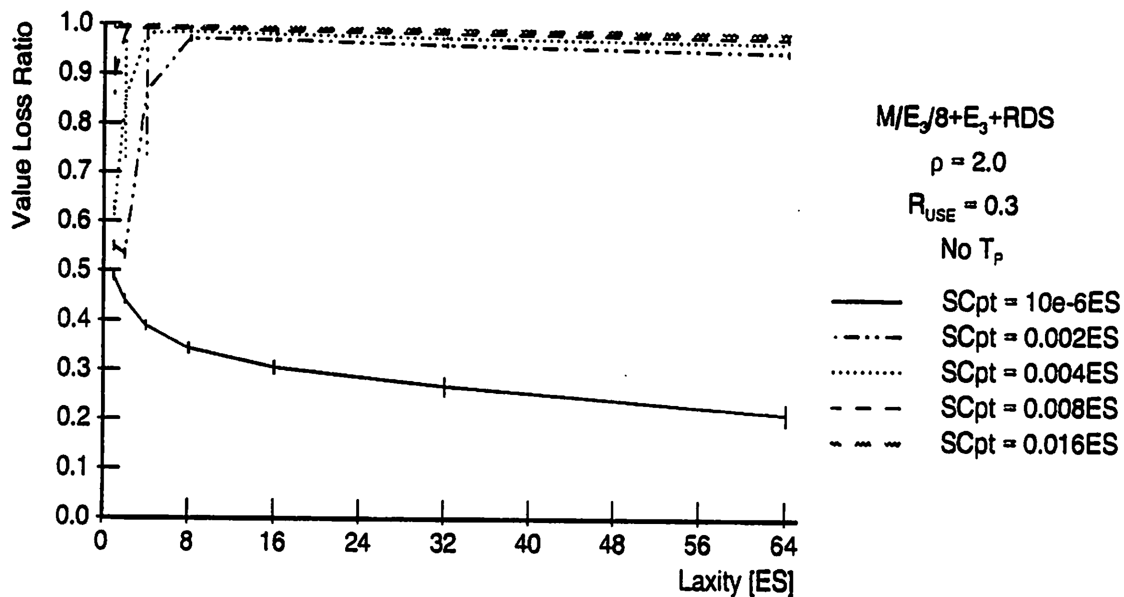

Figure 3.26 RDS—Value Loss Ratios for 8 CPUs and $\rho = 2.0$.

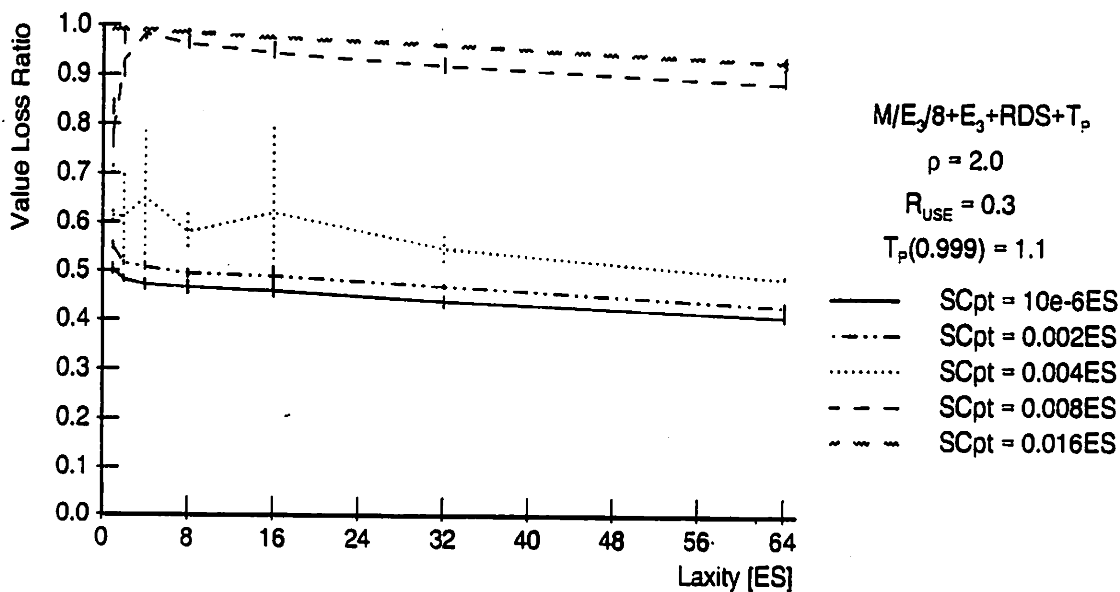

Figure 3.27 RDS—Value Loss Ratio for 8 CPUs, $\rho = 2.0$ and $T_P(0.999)$.

points, negligible cost and $l = 64$) increases to 0.41 when the punctual point is used—creating the absolute difference of 20%. As before, this difference is caused by the reduced selection of schedulable tasks to only "relevant" tasks as opposed to scheduling all arrived tasks.

3. Slight improvement in performance, when Well-Timed Scheduling is used, is observed for $SCF = 0.008ES$. At this scheduling cost per task and laxity $l = 64$, the value loss ratio without the punctual points is 0.98; while with the punctual point the value loss ratio is 0.88—the absolute difference of 10%. On the other hand, an even smaller improvement is achieved for $SCF = 0.016ES$. The value loss ratio without the punctual points is 0.99 versus value loss ratio of 0.94 with the punctual point. In spite of the 5% improvement, both value loss ratios are intolerably high and, therefore, an alternative scheduling approach must be sought.

Two major conclusions can be drawn from the above analysis. First, the punctual points are very useful in stressful environments of complex real-time systems that have realistic scheduling costs. Applying the Well-Timed Scheduling to these systems improves the performance, in some cases, over 50%. Additionally, the integration of the Well-Timed Scheduling and real-time schedulers extends the applicability of the schedulers to a much wider range of scheduling costs proportionality constant when compared to a traditional approach. Second, with the increase in the number of required resources, the increase of the scheduling cost proportionality constant reaches the breakpoint much faster—indicating the limitations imposed by the system characteristics.

Reducing the load to $\rho = 1.2$ and $\rho = 0.9$ results in almost identical trends. Larger improvements are achieved, but the $SCF = 0.008ES$ is still a breakpoint cost.

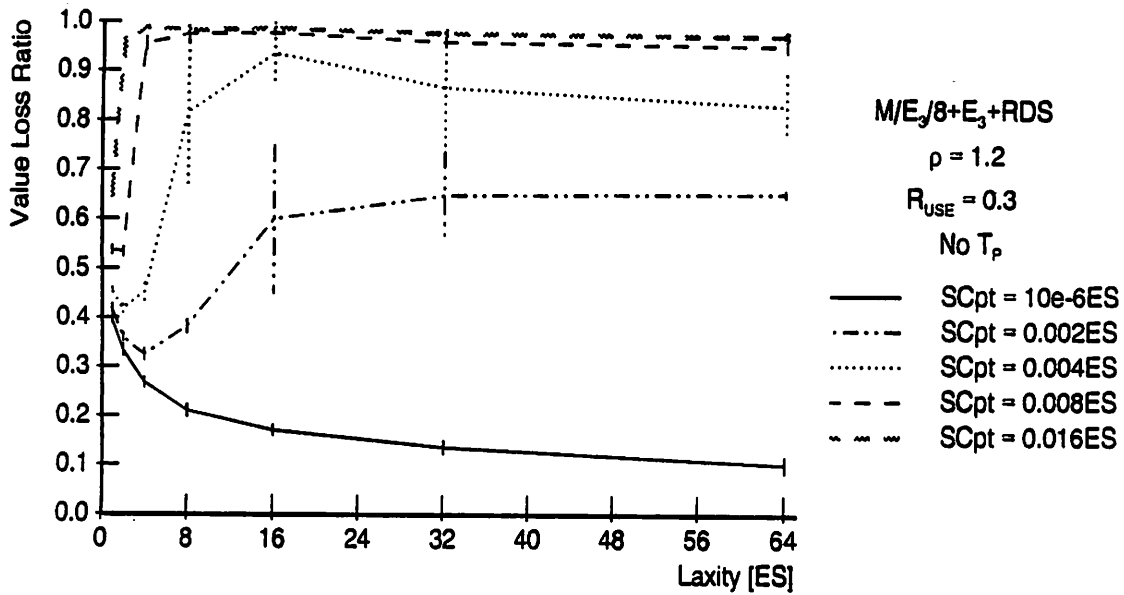

Figure 3.28 RDS—Value Loss Ratios for 8 CPUs and $\rho = 1.2$.

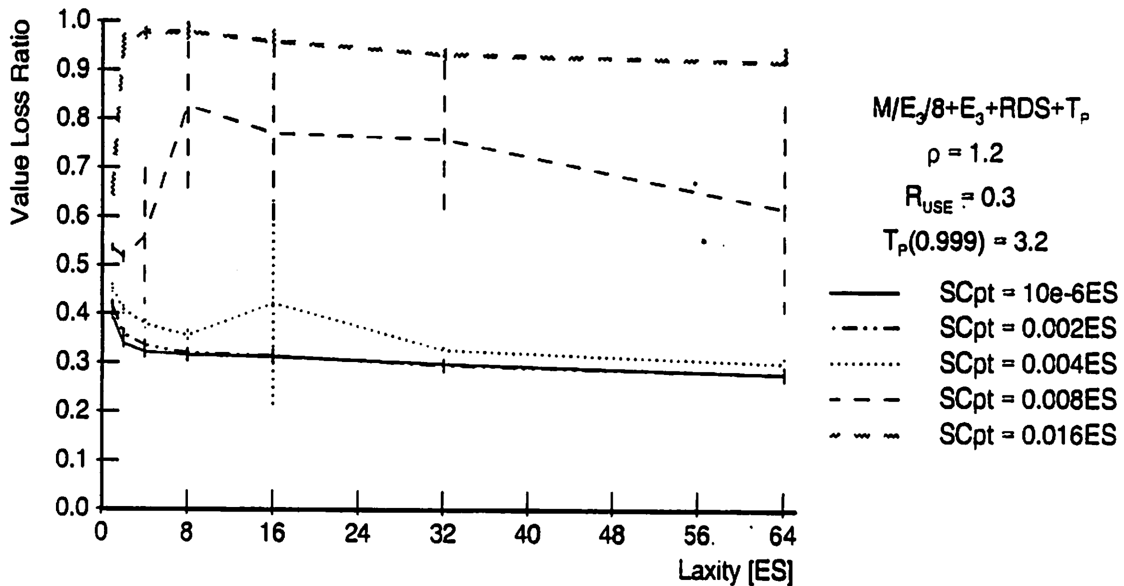

Figure 3.29 RDS—Value Loss Ratio for 8 CPUs, $\rho = 1.2$ and $T_P(0.999)$.

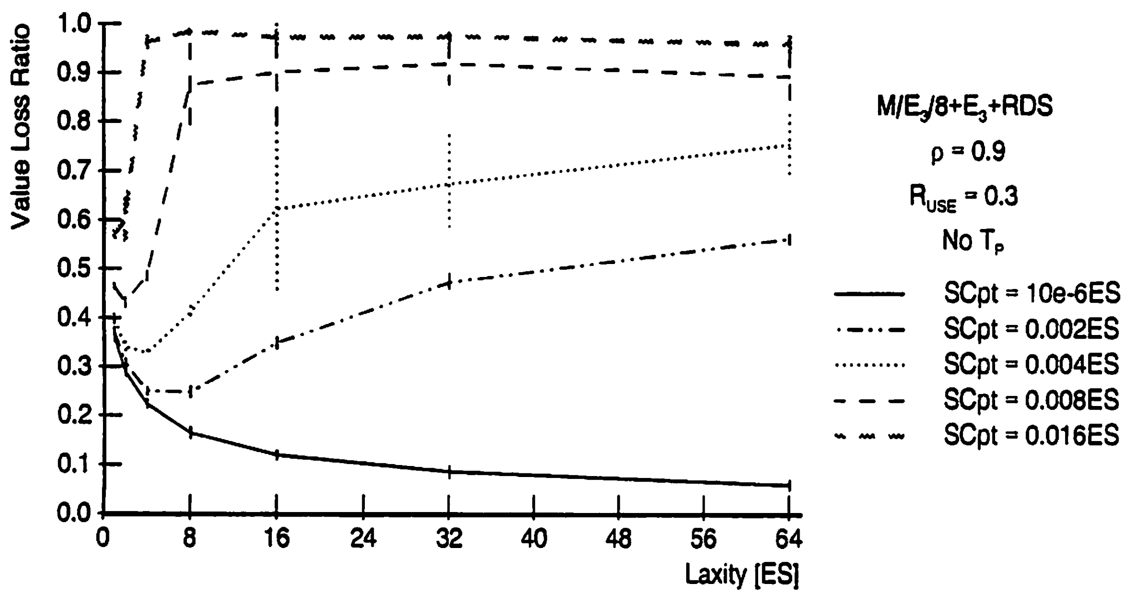

Figure 3.30 RDS—Value Loss Ratios for 8 CPUs and $\rho = 0.9$.

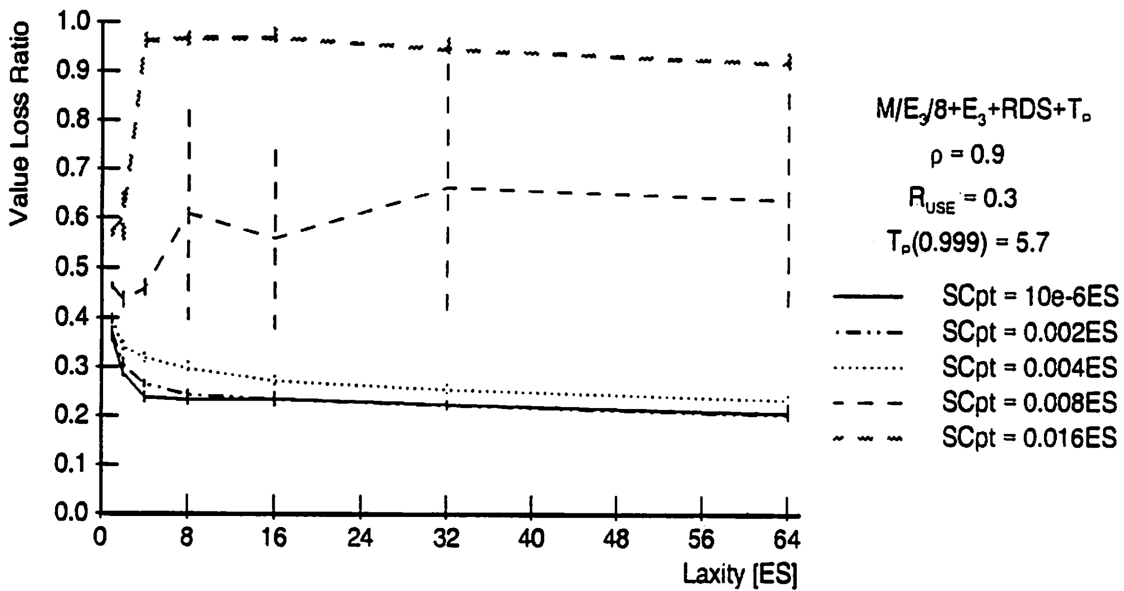

Figure 3.31 RDS—Value Loss Ratio for 8 CPUs, $\rho = 0.9$ and $T_P(0.999)$.

Figure 3.28 and Figure 3.29 plot the value loss ratios for $\rho = 1.2$ and systems without and with the punctual points, respectively. Furthermore, the value loss ratios for $\rho = 0.9$ are plotted in Figure 3.30 (without the punctual points) and Figure 3.31 (with the punctual points).

For the systems with the punctual points (for both loads) the following is observed: (1) there is a significant improvement over the systems without the punctual points for $SCF = 0.002ES$ and $SCF = 0.004ES$, (2) the value loss ratio degrades for a negligible SCF, and (3) slight improvements, but close to a total loss, are noticed when $SCF = 0.008ES$ and $SCF = 0.016ES$. All these trends have already been observed before.

In the above analysis, only one punctual point, $T_P(0.999)$, is used. The effects of the different punctual points on the system performance, the robustness and the overall applicability of the intergration of Well-Timed Scheduling and real-time scheduler is presented next.

3.4.4 The Robustness of the Integration

Considering the results for a single punctual point $T_P(0.999)$, the following question remains to be answered: How good is the performance of the analytically derived punctual point $T_P(0.999)$, when applied to a wide range of real-time multiprocessor systems with high probability of resource conflicts? Does its value have to be adjusted to achieve the best performance, and if so, for how much?

To answer this, the performance analysis for five different punctual points is presented in this section. This analysis consists of two parts: the analysis of a two processor system and the analysis of an eight processor system. In both systems, the punctual point values are varied on both sides of the analytically derived punctual point $T_P(0.999)$. Specifically, in a two processor system the punctual point values are approximately 1/4, 1/2, 1, 2, and 4 times the value of the analytically derived punctual point, $T_P(0.999)$. Similarly, in an eight processor system the punctual

point values are approximately 1/4, 1/2, 1, 2 and 8 times the value of $T_P(0.999)$. Furthermore, the scheduling cost proportionality constant is varied from a negligibly small cost of $SCF = 10^{-6}ES$ to a significant cost of $SCF = 0.016ES$. Due to the large number of results, only the most illustrative examples that describe the behavior and the trends in the performance are presented. As in the previous section, the analysis is conducted for high and moderate overloads, that is, for loads of $\rho = 2.0$ and $\rho = 1.2$.

3.4.4.1 Two Processor System

The values of the analytically derived punctual points, for a given load, $\psi = 0.999$ and a $M/E_3/2$ system, are obtained from Table 3.4. Specifically, for $\rho = 2.0$, the analytically derived punctual point $T_P(0.999)$ is $3.7ES$, and for $\rho = 1.2$, it is $12.8ES$. Varying these reference values as 1/4, 1/2, 1, 2, and 4 times $T_P(0.999)$, the following punctual points are obtained. For a $\rho = 2.0$ system, the punctual point values are: T_P = 1, 2, 3.7, 8, and 16 times ES; and for a $\rho = 1.2$ system, they are: T_P = 3, 6, 12.8, 24, and 48 times ES. Additionally, these T_P systems are compared to a traditional system—the system without the punctual points. The traditional approach, in essence, corresponds to the T_P system with infinitely large punctual point, that is, the punctual point at least as large as the largest laxity in the system.

The main effect of Well-Timed Scheduling is the reduction in the number of schedulable tasks. To illustrate the dramatic reduction of schedulable tasks, caused by the use of the punctual points, the maximum S-pool size (i.e., the maximum number of schedulable tasks at a scheduling time) is plotted against the laxities. The smaller the punctual point, the smaller the number of schedulable tasks. Figure 3.32 and Figure 3.33 plot the maximum S-pool sizes versus laxities for $\rho = 2.0$ and $\rho = 1.2$, respectively, for a negligible scheduling cost proportionality constant.

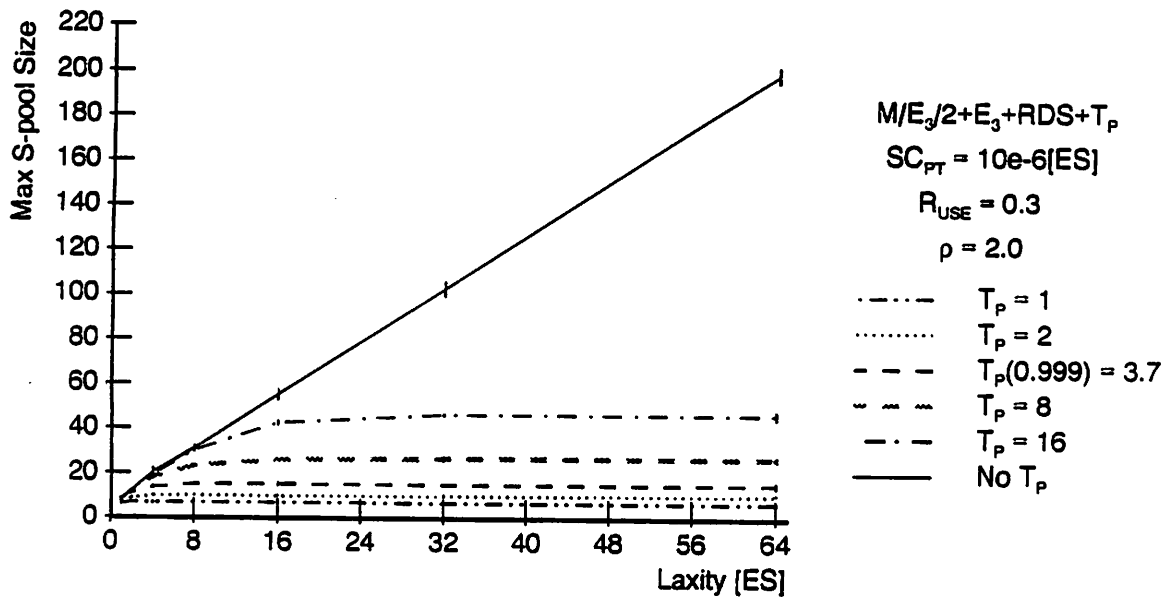

Figure 3.32 RDS—Maximum S-pool Size for $\rho = 2.0$ and $SCF = 10^{-6}ES$.

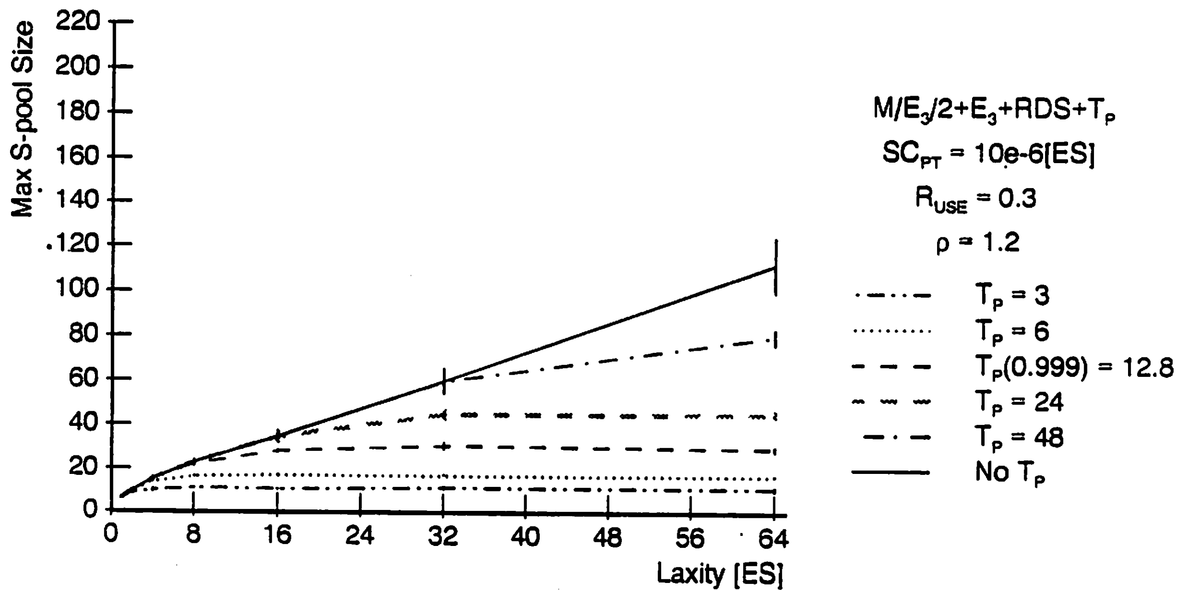

Figure 3.33 RDS—Maximum S-pool Size for $\rho = 1.2$ and $SCF = 10^{-6}ES$.

Let's, first, discuss the results without the punctual points, the baseline. It is obvious that the number of schedulable tasks increases linearly with the increase in laxities. The larger the laxities, the longer the lifetimes of the tasks, and thus, the larger the number of schedulable tasks. For example, for $\rho = 2.0$ in Figure 3.32, about 8 tasks are schedulable at laxity $l = 1$, while at laxity $l = 64$, the number of schedulable tasks increases to a significant 200. A somewhat lower, but still a very large number of schedulable tasks is observed for $\rho = 1.2$ and $l = 64$—about 115 tasks (Figure 3.33).

The dramatic decrease in the number of schedulable tasks, in both figures, is achieved by using the punctual points. Not only is the number of tasks reduced, but their linear increase with laxities is also stopped. That is, the maximum S-pool sizes for systems with the punctual points saturate. How fast it saturates and for what laxities depends on the punctual point value. For example, for $\rho = 2.0$ and three punctual points, $T_P = 1$, $T_P(0.999) = 3.7$, and $T_P = 16$, the maximum S-pool sizes saturate, respectively, at laxities $l = 1, 4$, and 16, reaching the corresponding S-pool sizes of 6, 14, and 48 tasks. For load $\rho = 1.2$ (Figure 3.33) and the punctual points $T_P = 3$, $T_P(0.999) = 12.8$, and $T_P = 24$, the maximum S-pool sizes saturate at respective laxities of $l = 8, 16$, and 32, reaching the S-pool sizes of 18, 30, and 42 tasks. For the $\rho = 1.2$ system and $T_P = 48$, the saturation occurs for laxities beyond the tested range, i.e., for laxities larger than 64.

The observed trend is very intuitive. The smaller the punctual point value, the smaller the number of schedulable tasks, and the smaller the laxities at which it saturates. The saturation at the lower number of schedulable tasks produces two opposing side effects. First, the reduction in the number of tasks lowers the chance that the scheduler will select and schedule the most important task. This, of course, is reflected in the value loss ratios—the smaller the selection set, the worse the performance. Second, with an increasing scheduling cost proportionality constant,

the overall scheduling cost increases proportionally. The analysis of these effects for different punctual point values is presented next.

The performance graphs for two analyzed overloads and a given SCF are paired for clearer illustration of the trends in performance. Each graph plots the value loss ratios: (1) for a baseline, i.e., the system without the punctual points, and (2) for five different punctual points. The baseline curve is labeled "No T_P," while the T_P curves are labeled with the actual punctual point values. Additionally, the analytically derived punctual point is labeled with its $\psi = 0.999$ probability.

Figure 3.34 and Figure 3.35 plot the value loss ratios for $\rho = 2.0$ and $\rho = 1.2$, respectively, for a negligibly small scheduling cost proportionality constant of $10^{-6}ES$. As observed before, the performance, for this cost, degrades with the use of the punctual points. The degradation in performance follows the decrease in punctual point values; the smaller the punctual point value, the smaller the S-pool, and thus, the worse the performance. In other words, for the near ideal cost, the larger the S-pool, the better the performance. This is just as expected.

Observing the same figures, it is evident that $\rho = 2.0$ produces the higher degradation in performance than $\rho = 1.2$, for the same relative range of the punctual points. Specifically, for $\rho = 2.0$ and $l = 64$, the value loss ratio of the baseline ("No T_P") curve is 0.19. Under the same conditions, the $T_P = 16$ system achieves the value loss ratio of 0.22. That is, its performance degrades by 3%. Furthermore, the $T_P(0.999)$ system achieves the value loss ratio of 0.28, producing the performance degradation of 9%. While the largest performance degradation is observed for a $T_P = 1$ system—the value loss ratio of 0.37 and the performance degradation of 18%. This gives a spread of 15% for punctual points ranging from $1/4 \times T_P(0.999)$ to $4 \times T_P(0.999)$, i.e., from $T_P = 1$ to $T_P = 16$.

On the other hand, a much smaller degradation in performance is observed for $\rho = 1.2$ system (Figure 3.35). In this system, the value loss for the $T_P = 48$, or

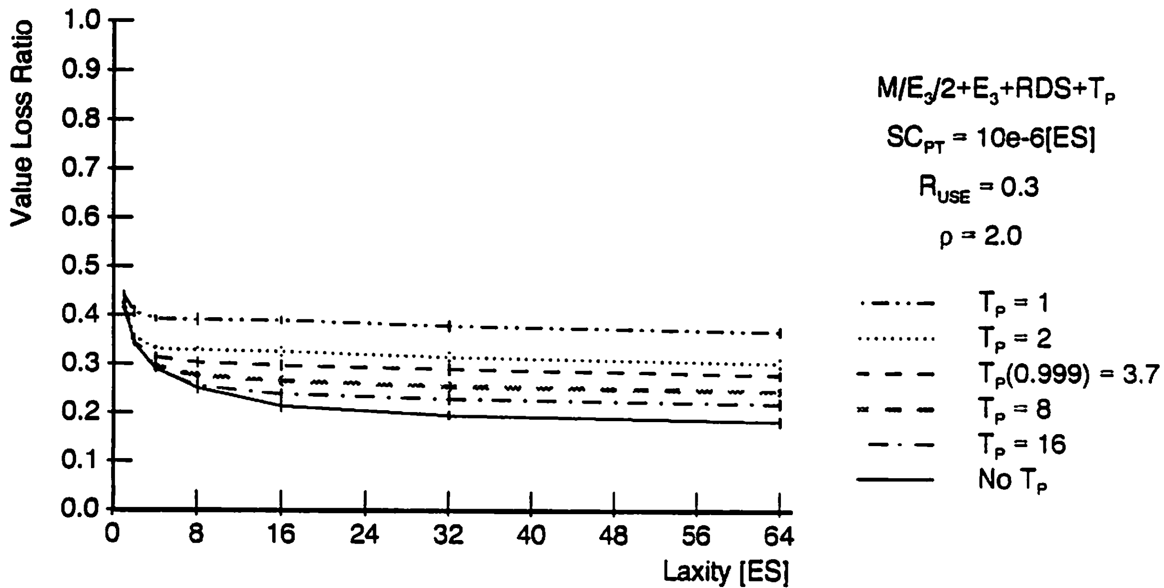

Figure 3.34 RDS—Value Loss Ratios for $\rho = 2.0$ and $SCF = 10^{-6}ES$.

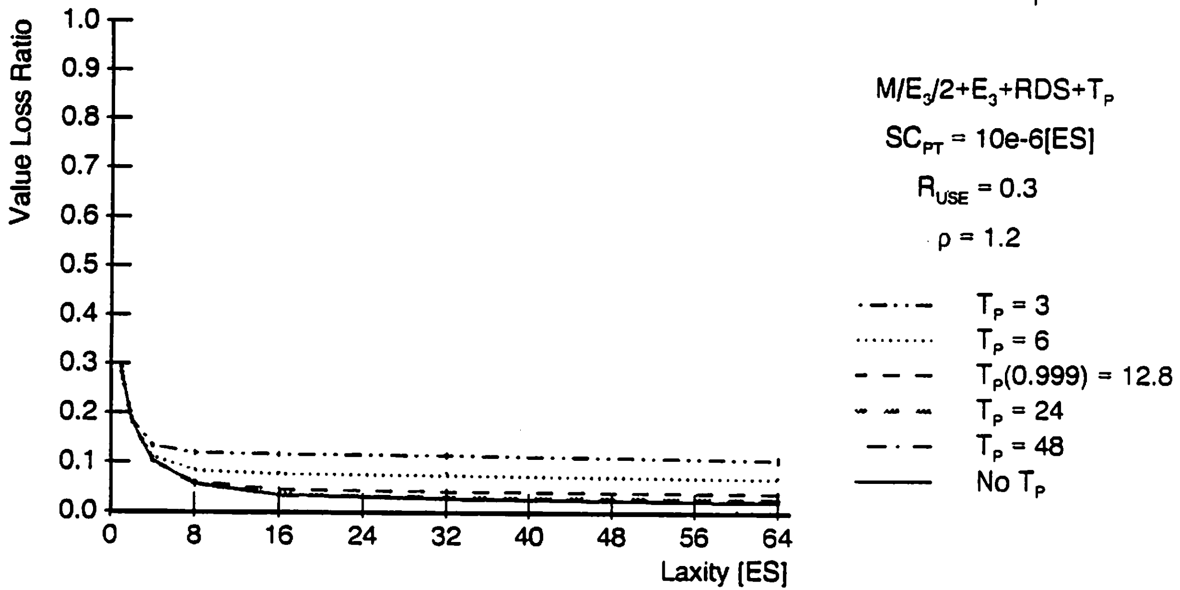

Figure 3.35 RDS—Value Loss Ratios for $\rho = 1.2$ and $SCF = 10^{-6}ES$.

$4 \times T_P(0.999)$, system is almost identical to a baseline at laxity $l = 64$. The $T_P(0.999)$ system has the value loss ratio of 0.04, a 2% in performance degradation; and the $T_P = 3$, or $1/4 \times T_P(0.999)$, system has the value loss ratio of 0.11, a 9% loss in performance when compared to a baseline.

This analysis indicates that $4 \times T_P(0.999)$ punctual point value gives the best possible performance in moderate overloads and near ideal scheduling cost. For higher overloads, the punctual point value has to be increased more than $4 \times T_P(0.999)$.

To analyze the effects of the realistic SCF's, the value loss ratios for increasing scheduling cost proportionality constant are presented next. Here again, the graphs with the same scheduling cost proportionality constant and two loads, $\rho = 2.0$ and $\rho = 1.2$, are paired.

Figure 3.36 and Figure 3.37 plot the value loss ratios versus laxities for $\rho = 2.0$ and $\rho = 1.2$, respectively, for $SCF = 0.004ES$. At this SCF, in both graphs, the baseline value loss ratios exhibit the increasing degradation in performance with the increase in laxities. Comparing the baselines of the two graphs, a much earlier and steeper degradation in performance is observed for the $\rho = 2.0$ system. This baseline departs from the value loss ratio of the $T_P = 8$ system at laxity $l = 4$; creating a difference of 23% at the laxity $l = 64$.

Further analysis of the $\rho = 2.0$ system indicates that for the largest punctual point $T_P = 16$, the performance degrades at modest laxities, while it improves at large laxities. For this scheduling cost this degradation is largest at laxity $l = 16$, with a small degradation of 4%. This behavior is an early indication that the $T_P = 8$ system will degrade even further with further increase in the scheduling cost proportionality constant. In contrast, the system performance for the remaining punctual points remains unchanged—the same as for a negligibly small cost.

Similarly, for $\rho = 1.2$ in Figure 3.37, the performance degradation for the baseline increases with the laxity increase. The baseline departs from the $T_P = 24$

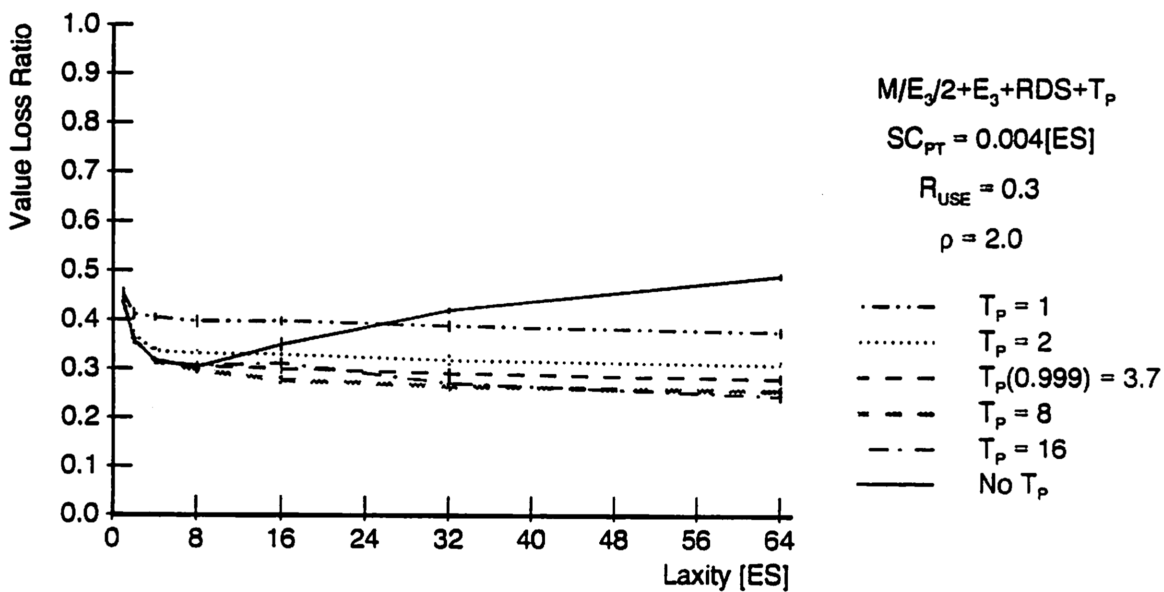

Figure 3.36 RDS—Value Loss Ratios for $\rho = 2.0$ and $SCF = 0.004ES$.

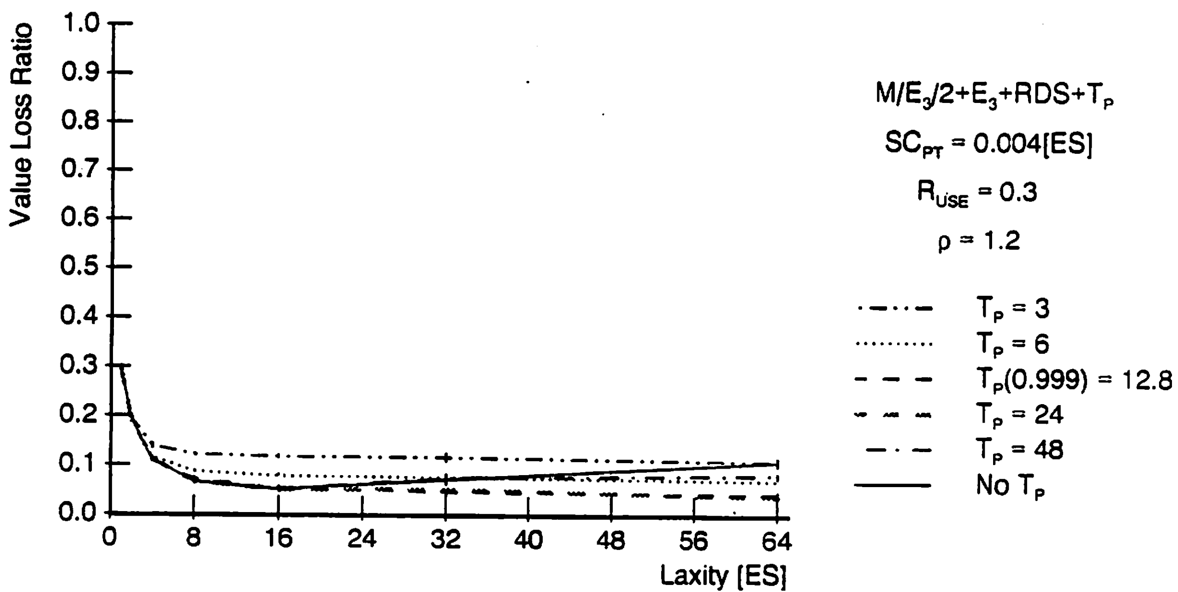

Figure 3.37 RDS—Value Loss Ratios for $\rho = 1.2$ and $SCF = 0.004ES$.

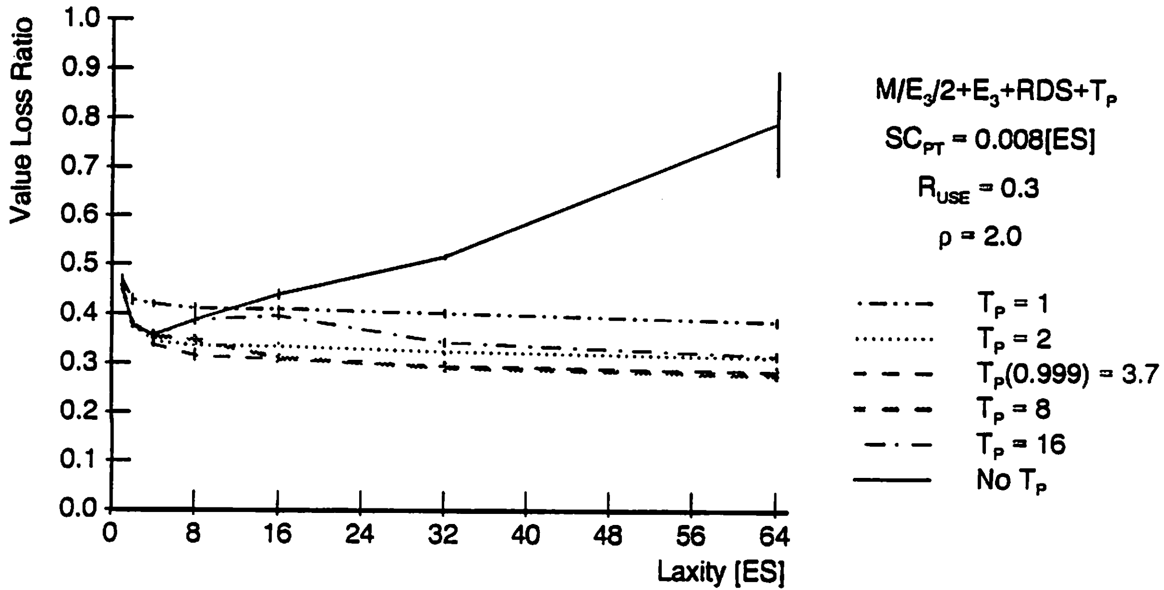

Figure 3.38 RDS—Value Loss Ratios for $\rho = 2.0$ and $SCF = 0.008ES$.

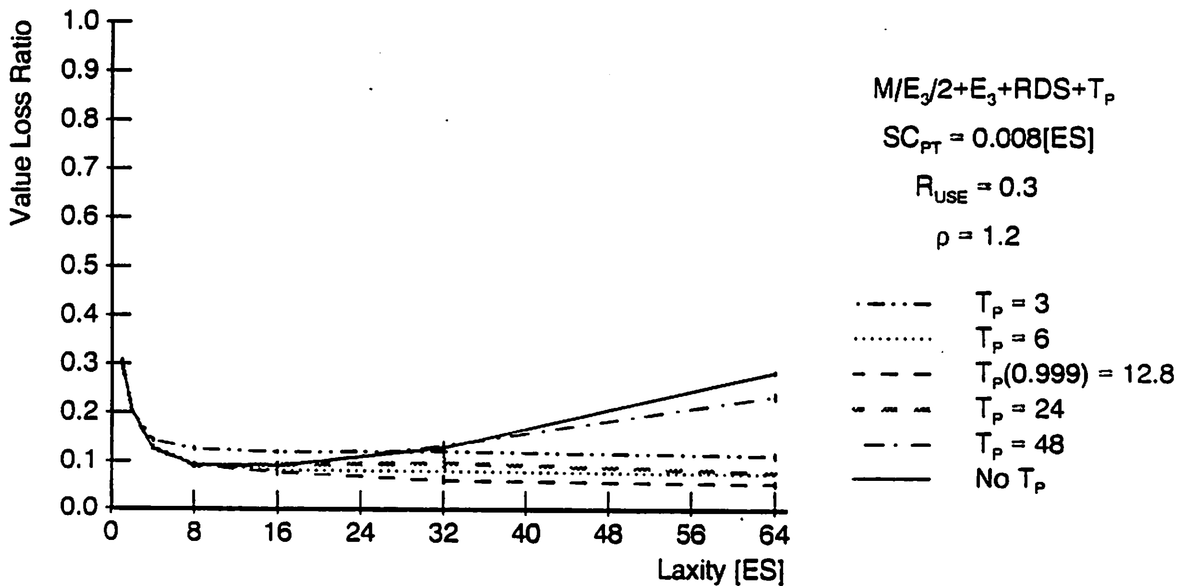

Figure 3.39 RDS—Value Loss Ratios for $\rho = 1.2$ and $SCF = 0.008ES$.

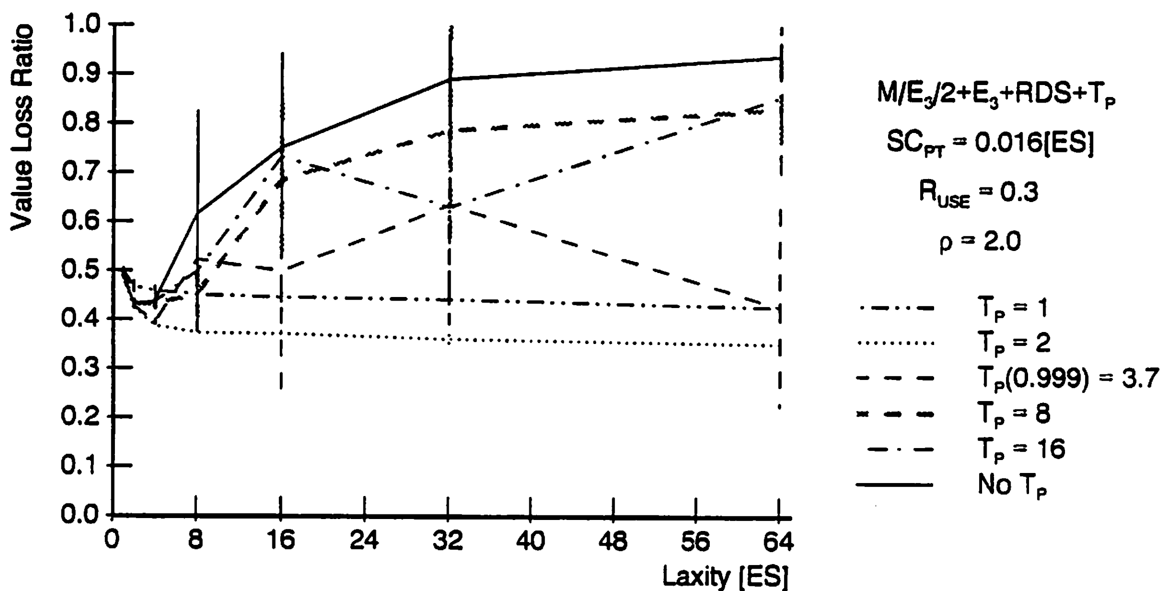

Figure 3.40 RDS—Value Loss Ratios for $\rho = 2.0$ and $SCF = 0.016ES$.

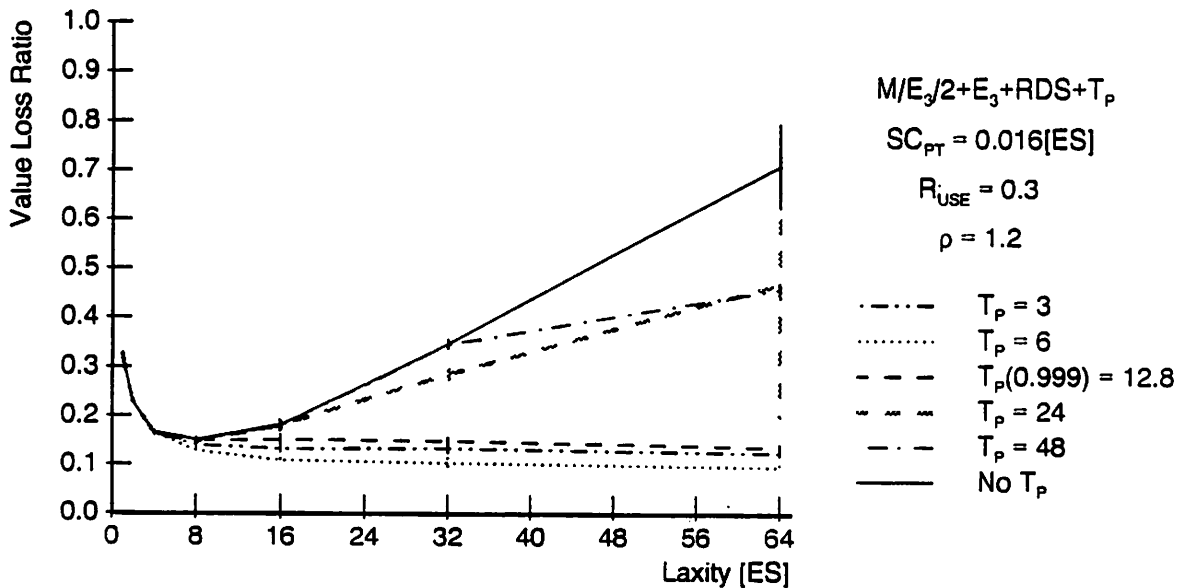

Figure 3.41 RDS—Value Loss Ratios for $\rho = 1.2$ and $SCF = 0.016ES$.

curve at laxity $l = 16$, reaching the difference of 8% at laxity $l = 64$. At the same laxity, the $T_P = 48$ system departs from the $T_P = 24$ creating a difference of 6% at laxity $l = 64$. On the other hand, the performance of the remaining four T_P systems stays unchanged, i.e., the same as for a negligibly small scheduling cost.

Let's double the scheduling cost proportionality constant and observe the performance trend for $SCF = 0.008ES$. The value loss ratios for $\rho = 2.0$ are plotted in Figure 3.38, and for $\rho = 1.2$ in Figure 3.39. For $\rho = 2.0$, the baseline and $T_P = 16$ degrade significantly worse than for the previous scheduling cost. Specifically, starting from $l = 4$, the value loss ratio of the baseline increases steeply, and at laxity $l = 64$, its performance when compared to $T_P = 8$ system is 50% worse. Furthermore, the $T_P = 8$ exhibits very small degradation in performance for moderate laxities, just like $T_P = 16$ for the previous scheduling cost proportionality constant of $0.004ES$. The performance of all other systems remains unchanged.

The most important observation, at this scheduling cost and load, is that with the further increase in laxities, the baseline would reach the total loss with an additional increase in laxities. That is, the scheduling algorithm without the punctual points would lead the system with very large laxities into a catastrophe.

On the other hand, the system with $\rho = 1.2$ in Figure 3.39 exhibits somewhat less steep increase in performance for the baseline. The increase starts from laxity $l = 16$, and at laxity $l = 64$, it reaches the performance of 22% worse than the one of the $T_P(0.999)$ system. The degradation in performance is also observed for $T_P = 48$ and $T_P = 24$ systems. Specifically, the $T_P = 48$ closely follows the performance of the baseline. Its performance degradation monotonically increases, and it is 19% worse than the performance of the $T_P(0.999)$ system, at laxity $l = 64$. Much smaller performance degradation is observed for $T_P = 24$ system—only 2% worse than for the $T_P(0.999)$. This small difference translates into the first signal that $T_P = 24$ will degrade significantly more with further increase in scheduling cost. The remaining

systems, namely, $T_P(0.999)$, $T_P = 6$, and $T_P = 3$ have still the same performance as in the case of the near ideal cost.

To illustrate that even the system that use the punctual points can approach the total loss, the scheduling cost proportionality constant is doubled again ($SCF = 0.016ES$). The value loss ratios for $\rho = 2.0$ and $\rho = 1.2$ systems are plotted in Figure 3.40 and Figure 3.41, respectively. The baseline system for $\rho = 2.0$ reaches an intolerably high value loss ratio quickly. Specifically, the high oscillations in performance are observed for moderately large laxities of $l = 8$ and $l = 16$; starting at $l = 32$, the baseline approaches a total value loss. Aside from the baseline, the same trend is observed for the $T_P = 16$ and $T_P = 8$ systems. This trend is, clearly, the result of the large number of schedulable tasks.

Furthermore, the $T_P(0.999)$ system starts to oscillate for moderate laxities. As before, the degradation in performance at moderate laxities is next followed by the degradation in performance at large laxities, for the further increase in scheduling cost proportionality constant. In spite of its high oscillations for moderate laxities, the average performance of the $T_P(0.999)$ system is still much better than of the baseline.

Even better is the performance of the $T_P = 2$ and $T_P = 1$ systems. The $T_P = 2$ system performs only 4% worse, while the $T_P = 1$ system performs 6% worse than their corresponding performances for a negligible scheduling cost. Their performance remain very stable and saturated. This is obviously due to the very low punctual points, that is, to a very small number of schedulable tasks.

The $\rho = 1.2$ system follows the same trend as the highly overloaded system. In Figure 3.41, the larger number of schedulable tasks, for baseline $T_P = 48$ and $T_P = 24$ systems, leads to a significant performance loss at high laxities. The baseline curve approaches intolerably high value loss ratio of 0.72, at $l = 64$. Similarly, the $T_P = 48$ and $T_P = 24$ curves decrease in performance by 46% at $l = 64$. On the other

hand, the remaining T_P systems are stable and saturated over all laxities, and with a very small degradation in performance. Among these, the $T_P(0.999)$ degrades in performance the most, 8% when compared to its performance for a negligible cost. The other two systems, $T_P = 6$ and $T_P = 3$, degrade by about 2% from their best performance. Following the previously observed trends, it is to be expected that with a further increase in SCF, the performance for all curves, starting first with large punctual points, would continue to degrade until total loss is reached.

In summary, if the scheduling cost is negligibly small, a scheduling algorithm should avoid the use of the punctual points. However, in overloads and for $SCF \leq 0.008ES$, the clear winner in performance is the system that uses the punctual points with $2 \times T_P(0.999)$. The $T_P(0.999)$ system, over the same range of scheduling costs, exhibits slightly smaller performance degradation, but is applicable to a wider range of SCF's. The need for using the punctual points smaller than the $T_P(0.999)$ is observed in systems where scheduling cost proportionality constant is very close to the breakpoint ($SCF > 0.008ES$).

3.4.4.2 *Eight Processor System*

The analytically derived punctual points for an $M/E_3/8$ system, $\psi = 0.999$ and loads $\rho = 2.0$ and $\rho = 1.2$ are tabulated in Table 3.5. By matching given ψ probability with a given load, the value of the punctual point $T_P(0.999)$ is obtained. From this table, for a load of $\rho = 2.0$, $T_P(0.999) = 1.1$; and for a load of $\rho = 1.2$, $T_P(0.999) = 3.2$. These punctual points are used as the basis to determine the other punctual points used in the analysis that follows. Expressed in terms of $T_P(0.999)$ values, the punctual points used in the analysis of an eight processor system are: $T_P = 1/4$, $1/2$, 1, 2, and 8 times the $T_P(0.999)$. Specifically, the actual values of the punctual points for a $\rho = 2.0$ system are: $T_P = 0.25$, 0.5, 1.1, 2, and 8 times ES;

and for a $\rho = 1.2$ system the actual values are: $T_P = 0.75$, 1.5, 3.2, 6, and 24 times ES.

The goal of the eight processor system analysis is to show that the centralized scheduling algorithm (the algorithm executed on a dedicated system processor) has its obvious limitations in systems with the large number of processors. Specifically, due to the large number of processors, the overload of $\rho = 2.0$ creates a tremendous number of schedulable tasks for a negligibly small scheduling cost proportionality constant. The maximum S-pool sizes for this system is plotted in Figure 3.42. For the baseline, the increase in laxities form $l = 1$ to $l = 64$ increases the maximum S-pool size from 20 to 520 tasks. Remember, for a two processor system the maximum S-pool size at $l = 64$ was only 200 tasks, a large difference for an algorithm with a high computational complexity. .

As before, the use of the punctual points reduces the number of schedulable tasks significantly, and furthermore, they saturate at low laxity values. For the same punctual points—expressed relative to the corresponding $T_P(0.999)$ values—the number of schedulable tasks reaches almost the same saturation levels as for a two processor system. The difference is that the punctual points for an eight processor system saturate much earlier, at laxity $l = 1$.

From Figure 3.43, the same behavior is observed for $\rho = 1.2$ system. In this case, the number of tasks for a baseline increases from 15 to 275 tasks (significantly larger when compared to 115 tasks of a two processor system at $l = 64$). Once more, the S-pool size saturate at the same levels as in a two processor system, but much earlier. Clearly, the punctual points, even for the large number of processors, successfully limit the number of schedulable tasks.

To observe the effects of the reduced number of schedulable tasks, Figure 3.44 and Figure 3.45 plot the value loss ratios for $\rho = 2.0$ and $\rho = 1.2$ systems, respectively, for a negligibly small scheduling cost proportionality constant of $SCF =$

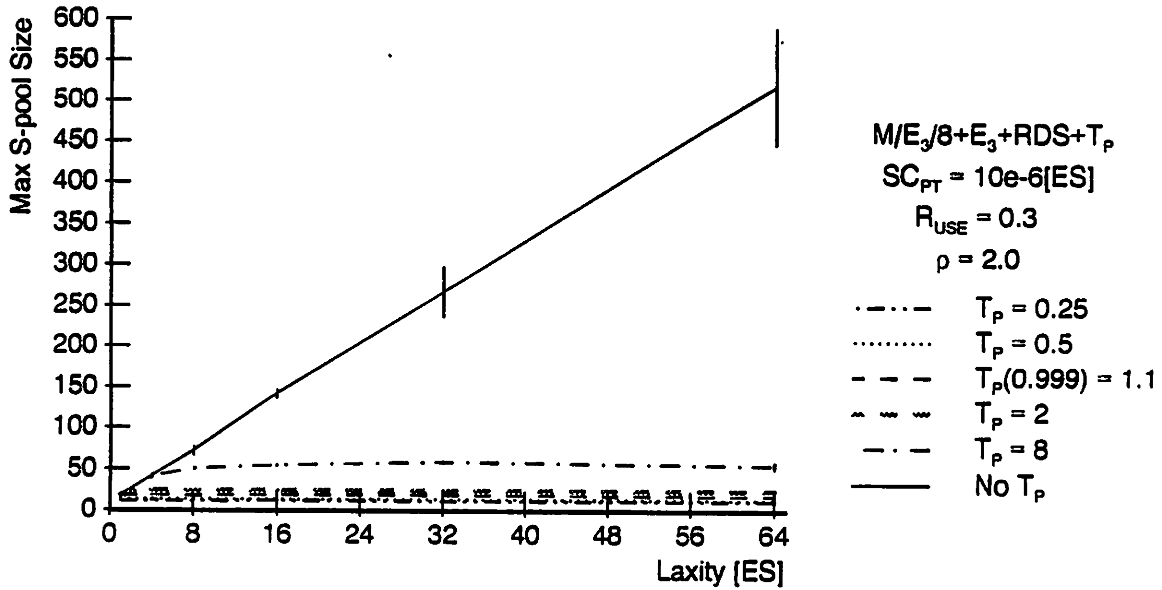

Figure 3.42 RDS—Maximum S-pool Size for 8 CPUs, $\rho = 2.0$ and $SCF = 10^{-6}ES$.

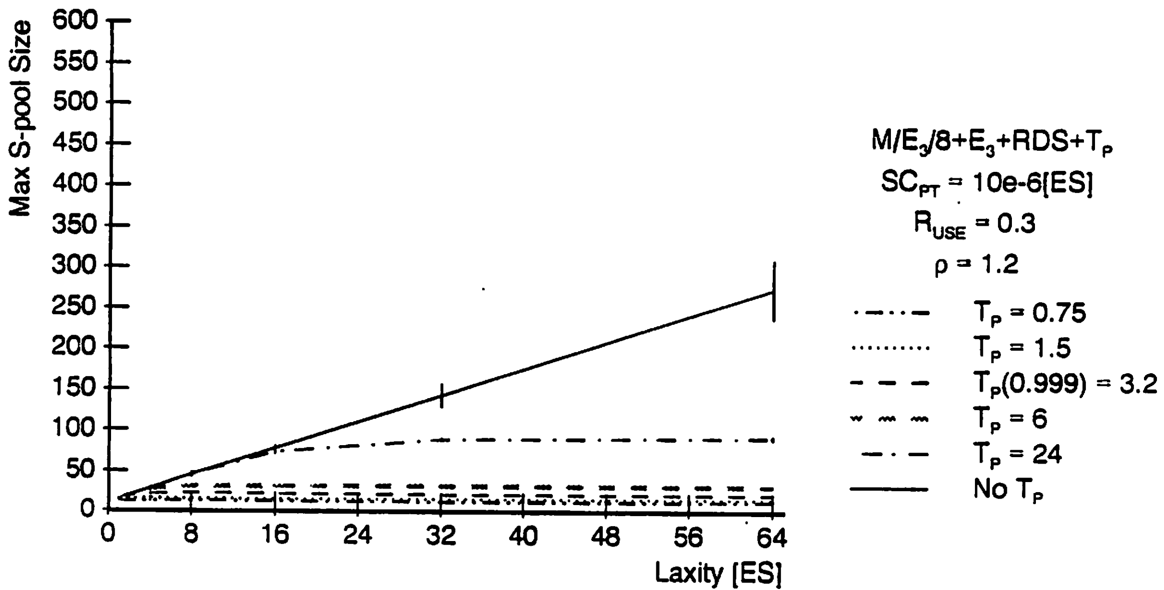

Figure 3.43 RDS—Maximum S-pool Size for 8 CPUs, $\rho = 1.2$ and $SCF = 10^{-6}ES$.

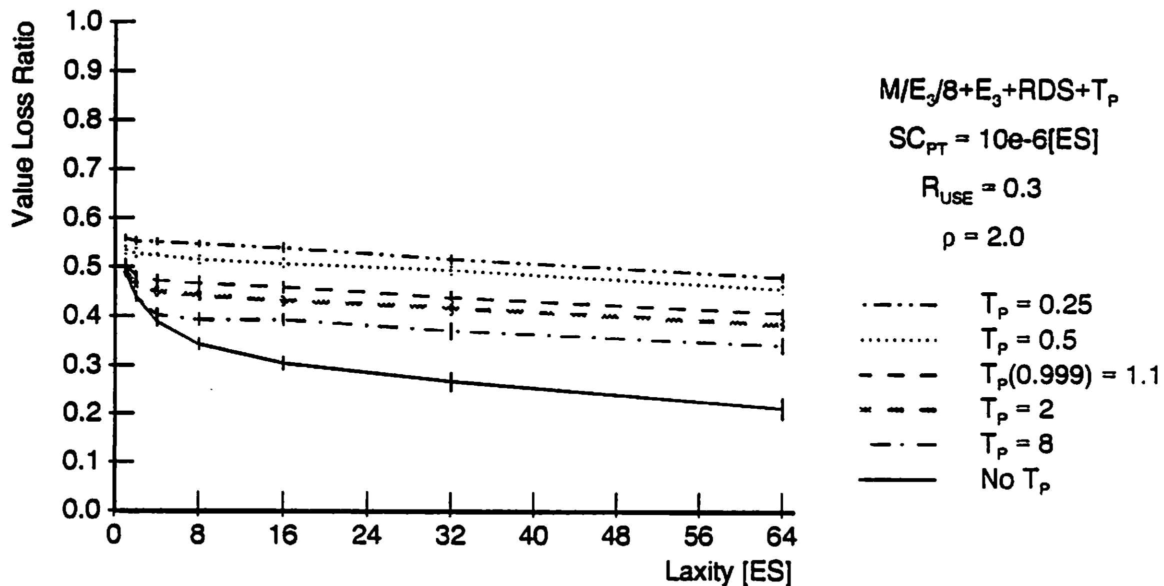

Figure 3.44 RDS—Value Loss Ratio for 8 CPUs, $\rho = 2.0$ and $SCF = 10^{-6}ES$.

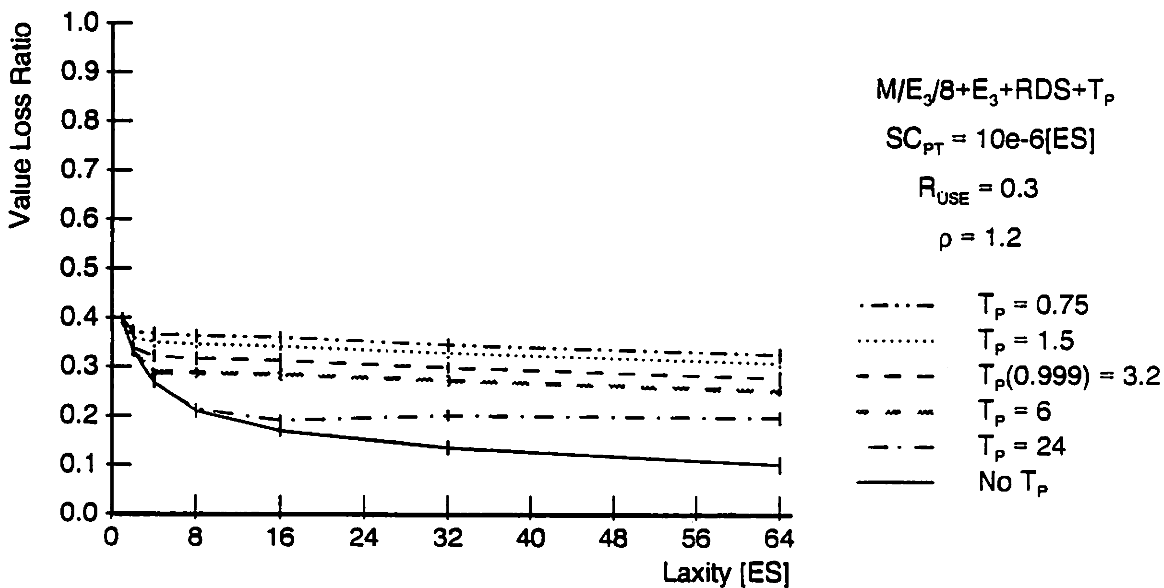

Figure 3.45 RDS—Value Loss Ratio for 8 CPUs, $\rho = 1.2$ and $SCF = 10^{-6}ES$.

$10^{-6}ES$. In both figures, the systems with punctual points exhibit an increase in the value loss ratio that is approximately two times larger than in a two processor system. In spite of the larger loss in performance, the relative performance for the different punctual points remains almost the same. The largest difference in value loss ratio between the T_P systems is about 14%.

A strong degradation in performance is exhibited already at the scheduling cost of $SCF = 0.002ES$. For this scheduling cost proportionality constant, the baseline for $\rho = 2.0$ (Figure 3.46) approaches almost a total loss for laxities $l > 8$. (Remember, in a two processor system, the baseline approaches this high loss for $SCF > 0.016ES$ and at laxities $l > 64$.) This is a significant degradation in performance for an eight processor system. Additionally, $T_P = 8$ system exhibits high oscillations and very large loss in performance at moderate laxities, about 40 % at $l = 8$. Its performance improves and stabilizes for laxities $l > 64$. As observed in a two processor analysis, the high oscillations at moderate laxities are the first sign that the T_P system is approaching the breakpoint scheduling cost.

On the other hand, the value loss ratios for the remaining four T_P systems are very stable, and only 1% to 2% worse than the for a negligibly small SCF.

For $\rho = 1.2$ system, Figure 3.47, the baseline has a high but not a total loss. Specifically, the baseline saturates for laxities $l > 16$ approaching the value loss ratio of 66%, a performance degradation of 54% when compared to the value loss ratio for a negligible cost (Figure 3.45). Here again, the largest T_P system, the $T_P = 24$ system, shows the first sign that the breakpoint scheduling cost is very near. While the value loss ratios for the other T_P systems remain unchanged, i.e., the same as for a negligible cost.

Figure 3.48 and Figure 3.49 plot value loss ratios for $\rho = 2.0$ and $\rho = 1.2$ systems, respectively, for a cost of $SCF = 0.004ES$—a two fold increase. As in the previous case, the value loss ratio for the baseline in a $\rho = 2.0$ system approaches the total

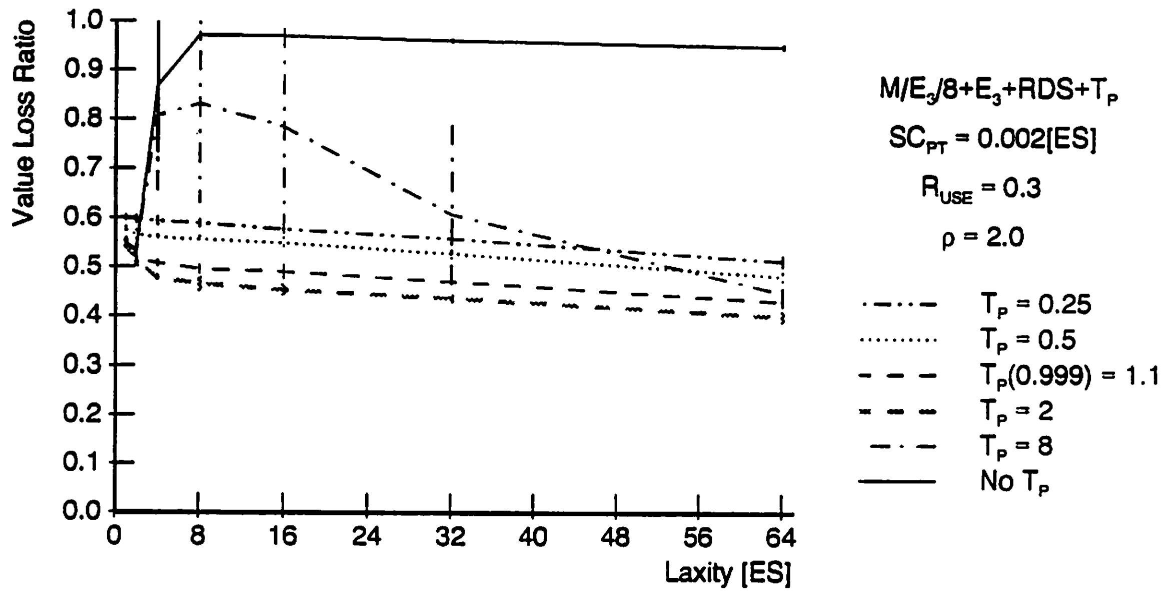

Figure 3.46 RDS—Value Loss Ratio for 8 CPUs, $\rho = 2.0$ and $SCF = 0.002ES$.

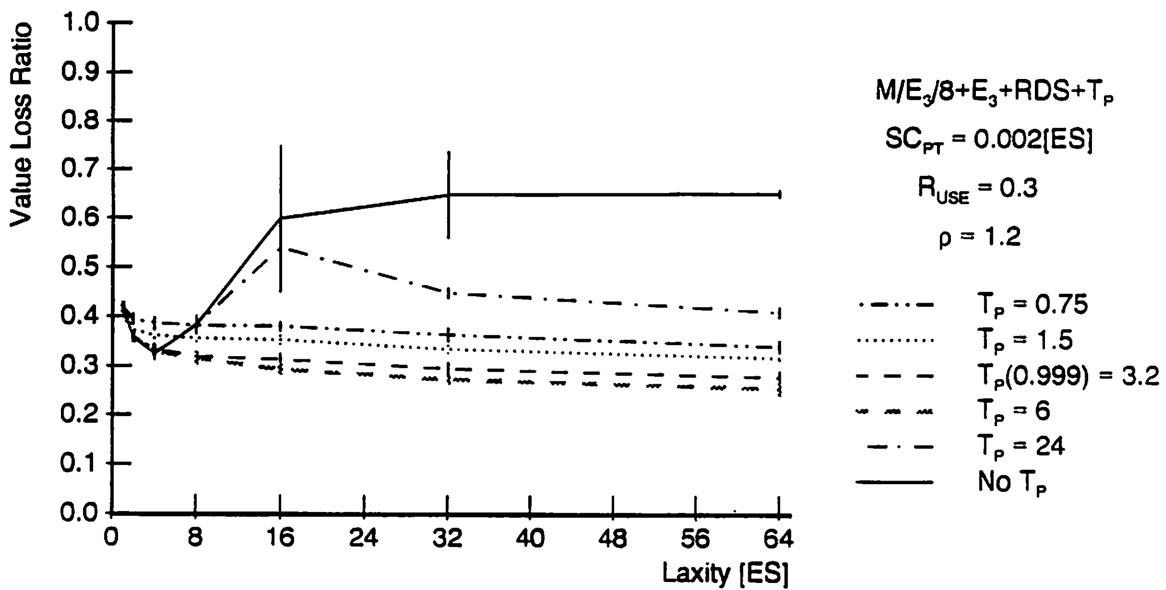

Figure 3.47 RDS—Value Loss Ratio for 8 CPUs, $\rho = 1.2$ and $SCF = 0.002ES$.

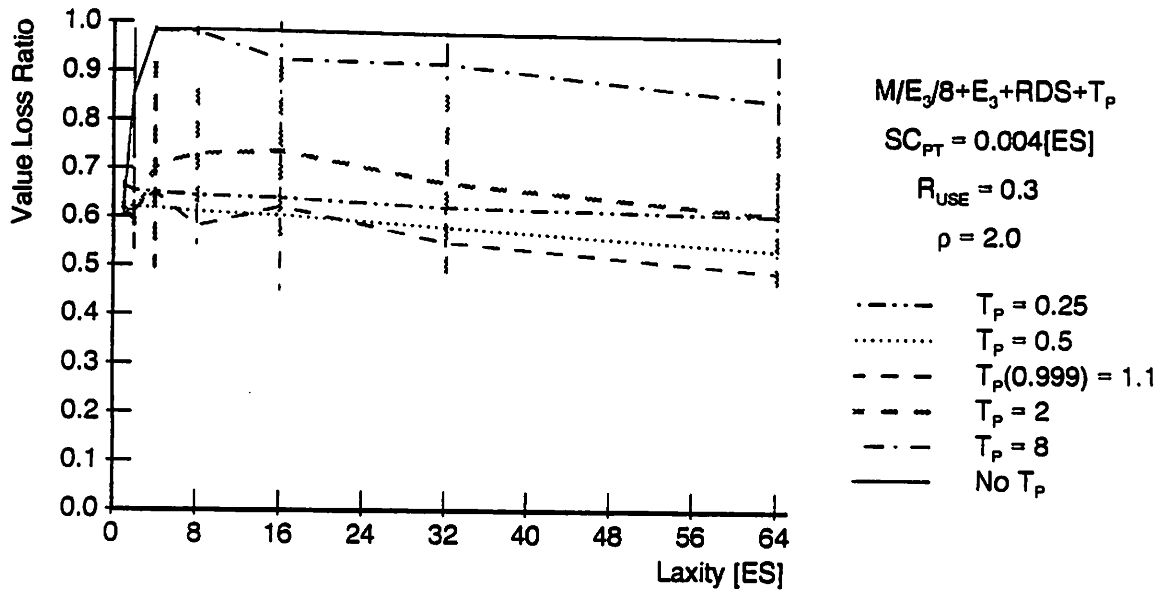

Figure 3.48 RDS—Value Loss Ratio for 8 CPUs, $\rho = 2.0$ and $SCF = 0.004ES$.

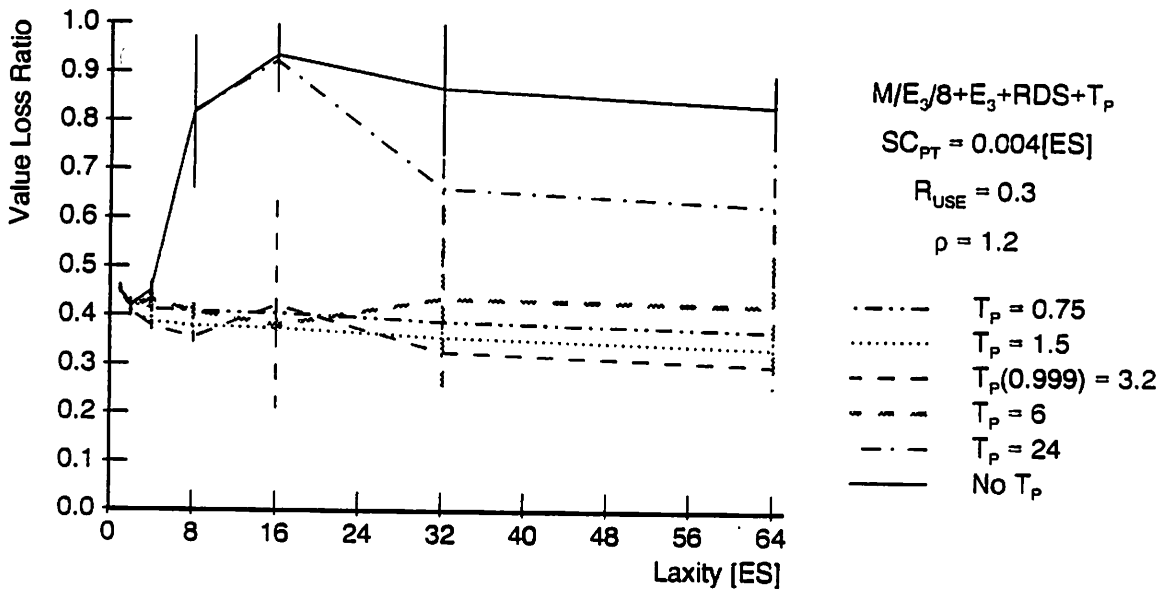

Figure 3.49 RDS—Value Loss Ratio for 8 CPUs, $\rho = 1.2$ and $SCF = 0.004ES$.

loss. This time the baseline saturates even earlier, at laxity $l = 4$. For $\rho = 1.2$ system, the baseline exhibits almost total loss at laxities $l > 16$. This indicates the inability of the traditional scheduling approach to deal with overloaded situation at this high scheduling costs proportionality constant.

At the same time, for both loads, the value loss ratios for all T_P systems, but the $8 \times T_P(0.999)$, are still low, on average. They are 15% to 20% worse than for the negligibly small scheduling cost, and still relatively stable in performance. The exception is the $8 \times T_P(0.999)$ system that exhibits a significant increase in value loss ratios, an intolerably high loss of over 60% (Figure 3.48 and Figure 3.49). This is again due to the larger set of schedulable tasks.

The noticeable oscillation, i.e., the wider confidence intervals at this cost are the first sign that the further increases in scheduling cost proportionality constant would lead to a significant degradation in performance. To verify this the scheduling cost proportionality constant is doubled again. The value loss ratios for $\rho = 2.0$ and $\rho = 1.2$ systems for $SCF = 0.008ES$ are plotted in Figure 3.50 and Figure 3.51, respectively.

It is obvious, from Figure 3.50, that the $\rho = 2.0$ system has intolerably high loss ratio with or without the punctual points. The total loss is reached, and for this SCF, the used scheduling algorithm is clearly inadequate. On the other hand, in $\rho = 1.2$ system, Figure 3.51, the value loss ratios for all but $T_P = 1.5$ and $T_P = 0.75$ systems are intolerably high. These two T_P systems allow the smallest number of schedulable tasks, and due to this their value loss ratios average about 0.45 for $l > 16$. However, the wide confidence intervals for low laxities as well as for $l = 64$ indicate that even these T_P systems are approaching the breakpoint, and with it, a total loss of control.

In conclusion, the eight processor system, with real-time tasks and a high probability of resource conflicts, exhibits the performance degradation at much

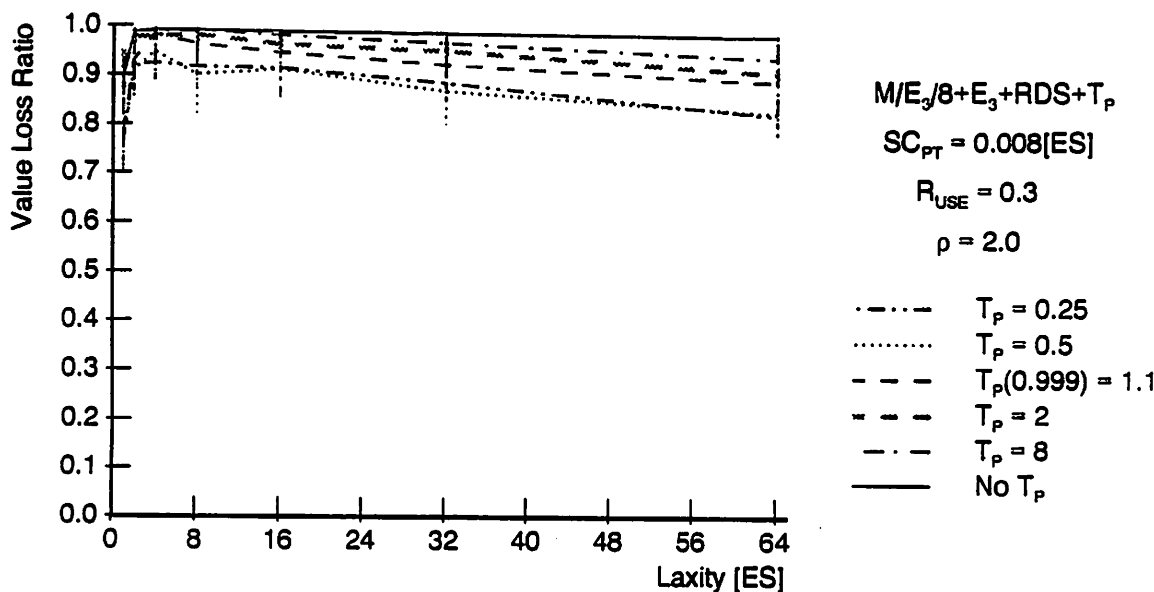

Figure 3.50 RDS—Value Loss Ratio for 8 CPUs, $\rho = 2.0$ and $SCF = 0.008ES$.

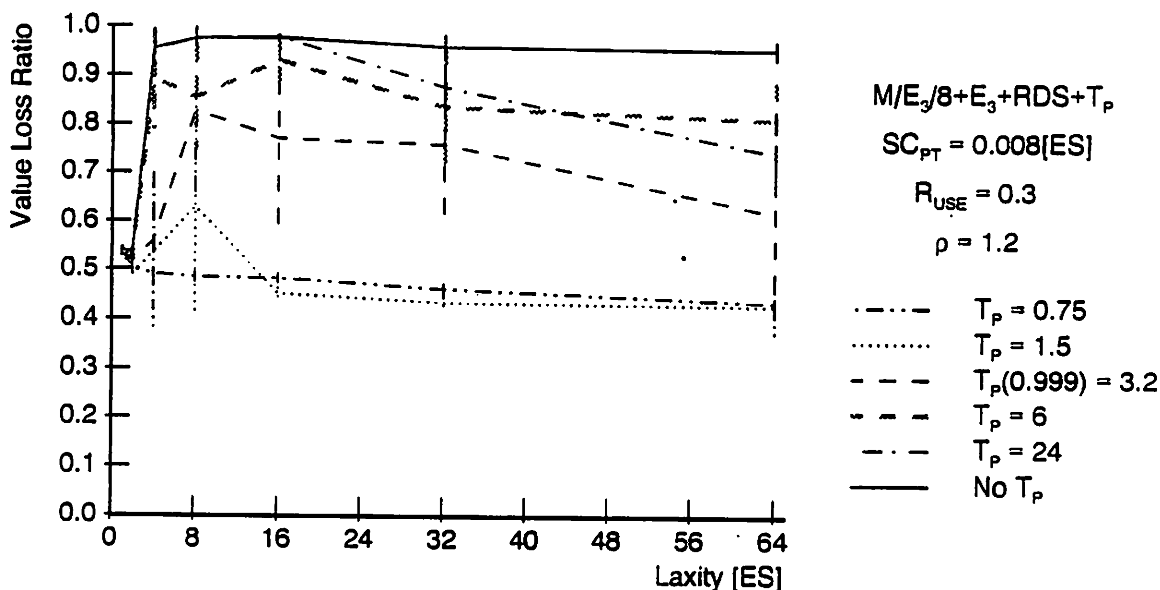

Figure 3.51 RDS—Value Loss Ratio for 8 CPUs, $\rho = 1.2$ and $SCF = 0.008ES$.

smaller SCF's than a two processor system. That is, in a two processor system the punctual points are usable up to $SCF = 0.016ES$, while in an eight processor system they are usable up to $SCF = 0.004ES$—a four times smaller cost. Aside from the smaller applicability range, the general observations are the same in eight and two processor systems. That is, the larger punctual point values are preferred for lower costs, and the smaller punctual point values are preferred for the cost close to the breakpoint. The applicability range of the adjusted punctual points as well as the other findings are summarized and tabulated in the next section.

3.5 Summary

The issue of system overloads in current real-time systems is very often overlooked. If this issue is ignored, the scheduling algorithm can lead a system to total loss of value, due to its inability to deal with a large number of tasks. The number of schedulable tasks is of utmost importance when the scheduler is designed to create feasible schedules for each of the application processors, especially, in presence of the resource constraints and different task values. In this chapter, we presented the analysis of the effects of overloads for a wide range of real-time systems using two different scheduling approaches (1) the traditional approach, where the tasks are considered for scheduling at their arrival times, and (2) the Well-Timed Scheduling approach, where only the "relevant" tasks—the tasks with laxities smaller or equal to the current value of the punctual point—are considered for scheduling. In traditional systems, the number of tasks considered for schedulability increases monotonically with the increase in laxities and/or loads. This increase is the main reason for their poor performance during overloads and with scheduling costs that occur in practice (the costs that present a significant impact on the overall performance). On the other hand, Well-Timed Scheduling reduces the number of schedulable tasks

through the use of punctual points; and consequently stops the increase in the number of schedulable tasks, approaching saturation at some laxity level.

Both approaches were analyzed by using two newly developed scheduling algorithms, the DLVD and RDS algorithms. The DLVD algorithm scheduled tasks in the increasing deadline order, and when an infeasible schedule was reached, it rejected the task with the minimum value density. The RDS algorithm scheduled the tasks according to their deadlines, earliest available start times, and value densities, combined in the heuristic function. If an infeasible schedule is reached, it rejected the tasks that created infeasibility.

The analysis of these algorithms, for a negligibly small scheduling cost proportionality constant, i.e., at their best performance, showed that both algorithms follow the same trend and produce very similar value loss ratios. Nevertheless, in high overloads, the RDS algorithm performed slightly better than the DLVD algorithm.

To determine the limitations of these scheduling algorithms, i.e., their applicability range in more realistic environments, the scheduling cost proportionality constant was increased until the breakpoints for both approaches were reached. It was increased until the point at which the algorithms began to lead the system to total value loss.

In the traditional approach, i.e., one without punctual points, the value loss ratio increases monotonically with the increase in laxities, when compared to the value loss ratio for a negligibly small scheduling cost proportionality constant. Furthermore, the value loss ratio increases even faster with the increase in scheduling cost proportionality constant. This is illustrated in Figure 3.52, where the value loss ratios, for a two processor system at load $\rho = 2.0$, are plotted for wide range of laxities and scheduling costs per task. In contrast, the integrated system, under the same conditions, shows much better overall performance. The value loss ratio

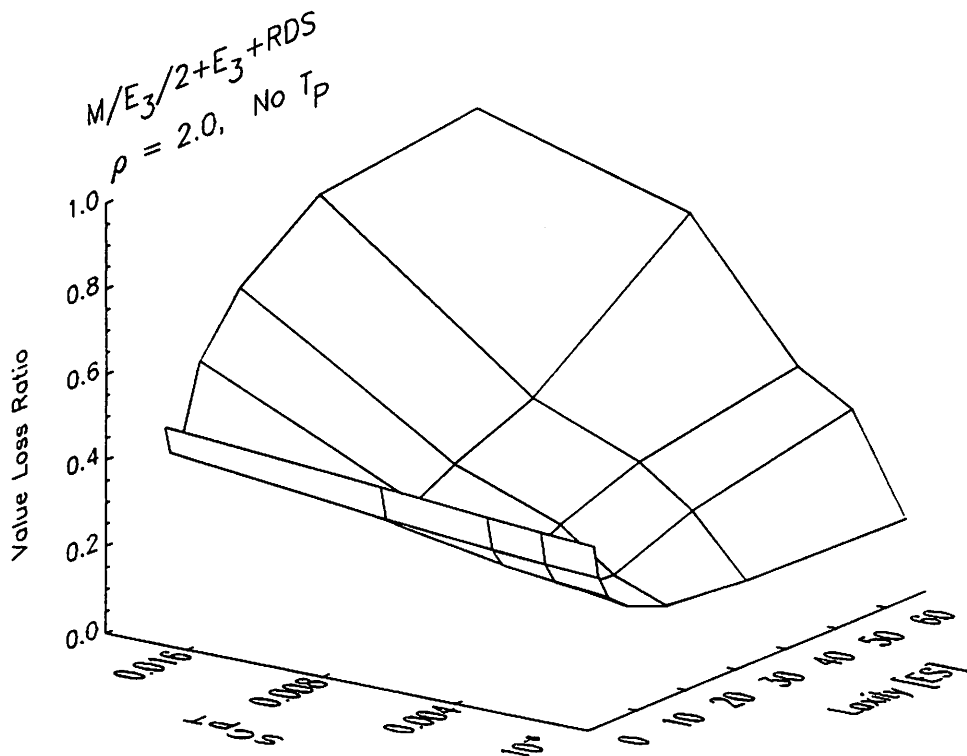

Figure 3.52 RDS—Value Loss Ratios for a Traditional Approach (2 CPUs)

does not increase with the increase in laxities, and the system performance is stable over a wider range of scheduling costs proportionality constant. Using the system parameters from the previous example, Figure 3.53 plots the value loss ratios for an integrated system that uses the analytically derived punctual point $T_P(0.999) = 3.7$.

The improvement in performance is observed for systems with realistic costs. In a near ideal system with a negligibly small scheduling costs, the reduced number of schedulable tasks, actually, produces a degradation in performance. For example, for a two processor system and $\rho = 2.0$, the system with the punctual points performs about 10% worse than the traditional system, for laxities $l > 4$ and $SCF = 10^{-6}ES$.

The relative difference in performance of the two approaches decreases with the decrease in load. Additionally, the number of processors affected the relative performance: the larger the number of processors, the larger the difference in

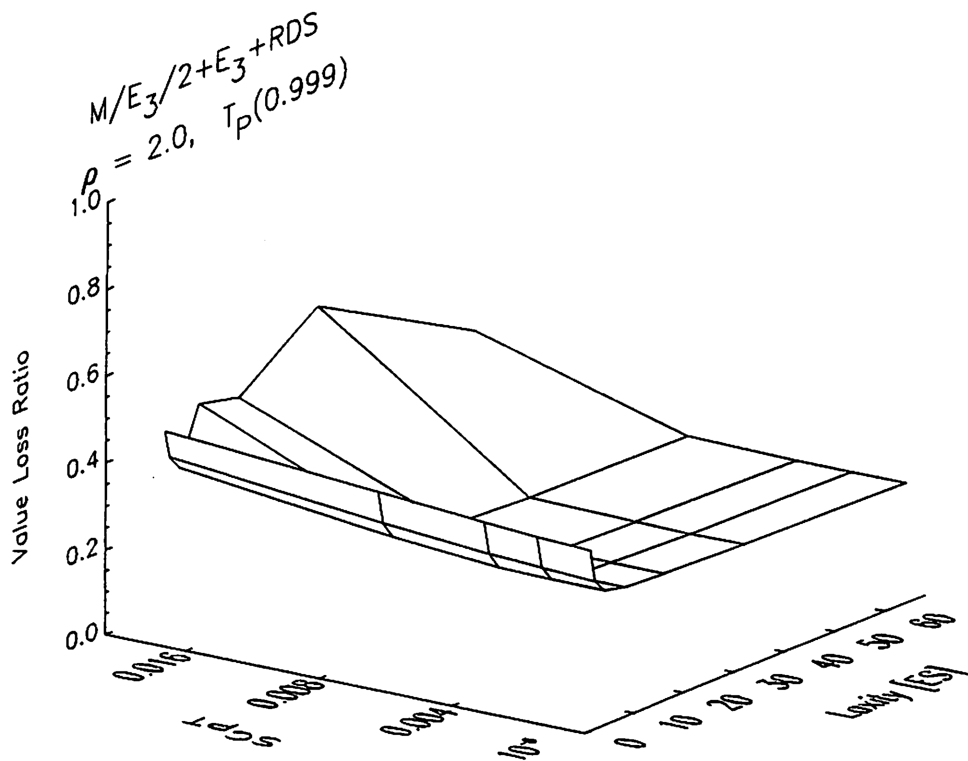

Figure 3.53 RDS—Value Loss Ratios for an Integrated Approach (2 CPUs)

performance between the traditional and Well-Timed Scheduling approaches. For example, the increase from a two to an eight processor system, the difference in relative preformance increaseed about 10% to 20%. In other words, in an eight processor system with load $\rho = 2.0$ and at high laxities, the traditional system outperformed the Well-Timed Scheduling by about 20%.

The applicability range of the traditional approach for a two processor system and load $\rho = 2.0$ is tabulated in Table 3.7. The first row presents the unitless values of the scheduling cost proportionality constant. In the second row, the scheduling costs are calculated for an expected service time of $1sec$. The value loss ratios for corresponding SCF's are tabulated for a laxity of 64 times the expected service time. The last row, labeled "Stable", indicates the stability of performance, i.e., it indicates the applicability range in terms of laxities. For example, for a negligibly

Table 3.7 Applicability Range for a Traditional Approach (2 CPUs).

$M/E_3/2 + E_3 + RDS$ for $\rho = 2.0$

SCF in $[ES]$	10^{-6}	0.002	0.004	0.008	0.016
SCF for $ES = 1sec$	$1\mu sec$	$2msec$	$4msec$	$8msec$	$16msec$
Loss at $l = 64sec$	0.19	0.41	0.5	0.7	0.94
Stable	$l > 0$	$l < 4sec$	$l < 2sec$	$l < 1sec$	unstable

Table 3.8 Applicability Range for an Integrated Approach (2 CPUs).

$M/E_3/2 + E_3 + RDS + T_P$ for $\rho = 2.0$

SCF in $[ES]$	10^{-6}	0.002	0.004	0.008	0.016
SCF for $ES = 1sec$	$1\mu sec$	$2msec$	$4msec$	$8msec$	$16msec$
$4 \times T_P(0.999)$					
Loss at $l = 64sec$	0.22	0.23	0.24	0.31	0.94
Stable	$l > 0$	$l > 0$	$l > 0$	$l > 32sec$	unstable
$T_P(0.999)$					
Loss at $l = 64sec$	0.28	0.28	0.28	0.28	0.42
Stable	$l > 0$	$l > 0$	$l > 0$	$l > 0$	unstable
$1/4 \times T_P(0.999)$					
Loss at $l = 64sec$	0.37	0.37	0.37	0.37	0.42
Stable	$l > 0$	$l > 0$	$l > 0$	$l > 0$	$l > 0$

small cost of $1\mu sec$, the approach is stable—and thus, applicable—for all laxities; for a cost of $2msec$, the monotonic increase in performance degradation is noticed for laxities larger than $4sec$. Therefore, the departure from the best performance starts at smaller laxity values for larger scheduling costs. The unstable state is reached at cost of $16msec$. At this cost, the system approaches total value loss very fast.

Similarly, Table 3.8 presents the applicability range for an integrated approach. Three different punctual point values are used: (1) a value that is 4 times the value

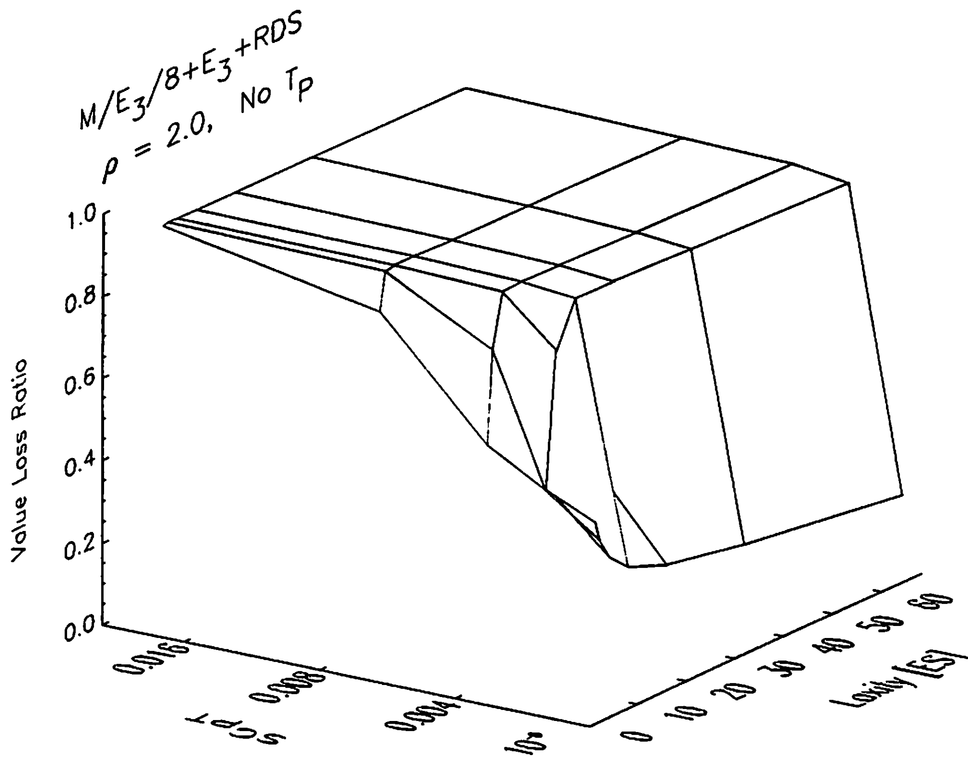

Figure 3.54 RDS—Value Loss Ratios for a Traditional Approach (8 CPUs)

of the analytically derived punctual point $T_P(0.999)$, (2) the analytically derived punctual point, $T_P(0.999)$, and (3) the four times smaller punctual point value. From this table, all three punctual points indicate the increase in applicability range. The $4 \times T_P(0.999)$ performs stably and very well (see the value loss ratios) for the laxities up to $l = 32sec$. At this laxity the performance starts to oscillate, but only for moderately large laxities, leading the system to an unstable performance with further increases in scheduling cost proportionality constant. The $T_P(0.999)$ punctual point is applicable for scheduling costs up to $16msec$. In this range, the performance is excellent for all laxity values. However, the system with four times smaller punctual points outperforms the other T_P systems at very large laxities.

Therefore, the choice of the punctual point value depends primarily on the scheduling cost. For small scheduling cost proportionality constant, the punctual

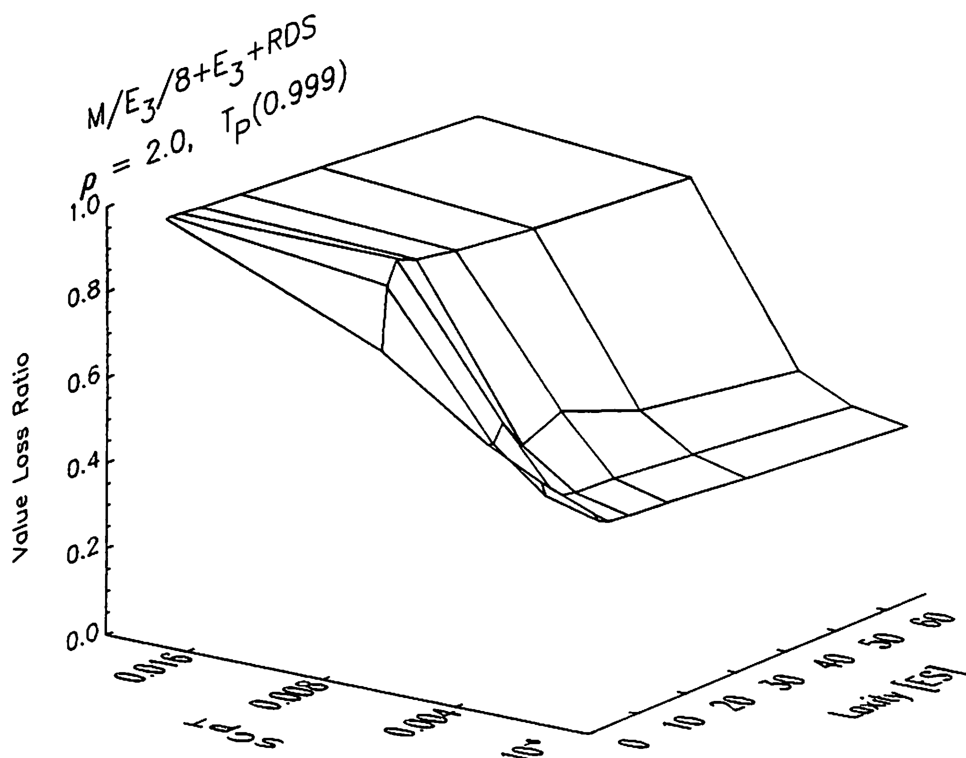

Figure 3.55 RDS—Value Loss Ratios for an Integrated Approach (8 CPUs)

points of $T_P(0.999)$ and larger perform the best; while for the scheduling costs close to the breakpoint, the punctual points smaller than $T_P(0.999)$ are required for the best performance.

The value loss ratios plotted over the wide range of laxities and scheduling costs for the traditional and integrated approach are given in Figure 3.54 and Figure 3.55, respectively; while the summary of the effects of the increased number of processors is tabulated in Table 3.9 and Table 3.10, respectively. In this eight processor system, the traditional approach is usable only for very small scheduling costs proportionality constant. Its performance becomes unstable for scheduling costs of $2msec$ and larger. On the other hand, the integrated approach performs very stably, even though with

Table 3.9 Applicability Range for a Traditional Approach (8 CPUs).

$M/E_3/8 + E_3 + RDS$ for $\rho = 2.0$

SCF in $[ES]$	10^{-6}	0.002	0.004	0.008
SCF for $ES = 1sec$	$1\mu sec$	$2msec$	$4msec$	$8msec$
Loss at $l = 64sec$	0.22	0.94	0.96	0.97
Stable	$l > 0$	unstable	unstable	unstable

Table 3.10 Applicability Range for an Integrated Approach (8 CPUs).

$M/E_3/8 + E_3 + RDS + T_P$ for $\rho = 2.0$

SCF in $[ES]$	10^{-6}	0.002	0.004	0.008
SCF for $ES = 1sec$	$1\mu sec$	$2msec$	$4msec$	$8msec$
$8 \times T_P(0.999)$				
Loss at $l = 64sec$	0.34	0.44	0.83	0.92
Stable	$l > 0$	$l > 0$	$l > 64$	unstable
$T_P(0.999)$				
Loss at $l = 64sec$	0.40	0.42	0.49	0.88
Stable	$l > 0$	$l > 0$	$l > 0$	unstable
$1/4 \times T_P(0.999)$				
Loss at $l = 64sec$	0.48	0.51	0.60	0.81
Stable	$l > 0$	$l > 0$	$l > 0$	unstable

relatively large value loss ratios, up to the scheduling cost of $8msec$. At this cost even the integrated approach becomes unstable and approaches total value loss.

As a final remark, the centralized scheduling approach, with and without the punctual points, faces the obvious limitations when used in multiprocessor systems with the large number of processors. In overloads, the applicability range of the centralized real-time schedulers dramatically decreases with the increase in the number of processors. Therefore, these systems should consider the alternative types

of scheduling concepts, such as, the scheduling where the application processors are clustered into smaller groups having their own scheduling processor. That is, a multilevel scheduling approach might be a better choice.

CHAPTER 4

PRECEDENCE CONSTRAINS SCHEDULING

4.1 Introduction

Tasks interrelated by precedence constraints are commonly found in complex real-time systems. The interrelations between the tasks arises from external constraints imposed by the physical characteristics of the process under control and from the application's structural characteristics imposed by the modular software design. Such imposed precedence constraints are usually presented in a form of directed acyclic graphs, where nodes represent individual tasks and arcs represent interrelational constraints between two nodes.

In a modular software design approach, a tendency is to create the task groups with well-defined, pseudo-independent tasks, interrelated to other tasks only through the pre-specified precedence constraints. These pseudo-independent tasks execute within their own context and their own address space, and are scheduled as part of a task group. The only requirement is to satisfy the given precedence constraints. For example, given a group of three software modules, namely, Data Acquisition Module, Display Module, and Data Logging Module, with the scheduling restriction that the Data Acquisition Module must complete before the other two modules are allowed to execute. The modules can execute on any available application processor as long as the given precedence constraints are obeyed.

From the scheduling point of view, these task groups that have no additional constraints aside from the precedence constraints are well suited for scheduling on

multiprocessor systems. The inherent parallelism within a group can potentially be exploited to its fullest extent, providing that there is a sufficient number of available application processors. In addition to the inherent parallelism, the modular design supports some other very important features. For instance, in some cases a task group can partially be executed, and the entire task group does not have to be rolled back if its execution gets canceled. Therefore, the contributing value of such task groups corresponds strictly to the tasks completed, rather than only to the contributing values of the entire task group. The task groups that allow partial executions are called *non-atomic task groups.*

On the other hand, *atomic task groups* demand the completion of the entire group, or no value would be contributed to the system. The atomic task groups must be scheduled end-to-end, they can not be unscheduled without "unscheduling" all tasks within a group; and if aborted, the already executed tasks must be rolled back. Examples of atomic task groups are found in systems where the monolithic processes are decomposed into executional episodes, sometimes called tasks, to provide more efficient resource utilization. Also, these task groups are found in critical real-time systems, and in mandatory parts of the imprecise computations.

In contrast to non-atomic task groups, the atomic task groups are not easily scheduled on separate application processors, mainly, because of the side effects of the process decomposition. Specifically, all executional episodes (tasks) of an original monolithic process must be executed within the same address space and the same context. That is, within the address space and the context of a given atomic group. This restriction is somewhat relaxed if the atomic task groups are made up of multiple processes. In this case, the different processes, but not the tasks within a single process, can be scheduled and executed on different application processors.

However, the differences between atomic and non-atomic task groups, from the scheduling point of view, clearly indicate that the non-atomic task groups are more

flexible to schedule. Specifically, non-atomic task groups do not impose any other interrelation constraints but the precedence constraints. They can be partially executed, i.e., severed if a more valuable task group arrives, and their contributing values depend on the tasks executed, not solely on the groups executed. Finally, the non-atomic task groups poses an inherent parallelism, with tasks well defined and suited for multiprocessor scheduling.

These characteristics of non-atomic task groups contribute to the greater flexibility and larger number of potential schedules. However, the greater flexibility and the larger scheduling possibilities can cause an increase in scheduling complexity. Specifically, aside from focusing on the groups' characteristics (such as precedence, resource and timing constraints), a scheduler must anticipate the arrivals of more valuable task groups. Therefore, it has to select not only the best task group to schedule next but also the best branch within a selected task group. That is, the scheduler must pay attention to the order of tasks within each task group as well as to the order of given task groups.

To illustrate this, the need for the careful ordering of tasks within a group, let's consider the following example. Assume that in a final feasible schedule, the low value branches are placed in front of the more valuable branches, of a given group. To prove that this ordering, even though it already is a feasible schedule, can create unnecessary value loss, let's assume that during the group's execution, a highly valuable group arrives. The timing constraints of both groups are such that the first group must be severed to accommodate a newly arrived, more valuable group. Clearly, the loss is minimized if the more valuable branches are positioned in front of low valuable branches. A newly arrived task would then sever the least valuable part of the partially executed task group. Therefore, to avoid unnecessary value loss upon the unpredictable arrivals of very valuable tasks, more valuable branches of each task group should be scheduled as early as possible. In other words, to maximize the

system's value, the scheduler should create the schedule based on the characteristics of given task groups, their individual tasks, and also on the characteristics of all successors of tasks eligible to be scheduled next.

This requirement, unfortunately, increases the computational complexity of on-line schedulers due to the frequent scanning of the remaining part of eligible task groups. By recognizing that the computational complexity of the schedulers for non-atomic task groups is increased, we designed a method that reduces scheduler's computational complexity. This feature is especially important in real-time systems with on-line schedulers. Specifically, to reduce the complexity of on-line schedulers, we develop an off-line preprocessing method for arbitrary task groups that lends itself to the use of already existing, and demonstrated as effective, independent task scheduling algorithms. The focus of this chapter is on the scheduling of non-atomic task groups with arbitrary precedence constraints, the task groups with a common deadline, where resource constraints along with the contributing values assigned to their individual tasks. The goal is to provide a method that enhances the performance of on-line schedulers. That is, we desire to provide a method that efficiently and effectively schedules task groups in dynamic non-deterministic real-time systems.

The remainder of this chapter is organized as follows. The related work for both single and multiprocessor systems is presented in Section 4.2. The description of the underlying model is outlined in Section 4.3, and the overall strategy is presented in Section 4.4. Two off-line preprocessing algorithms: (1) the VDP-R algorithm, developed for the rooted tree precedence constraints, and (2) the VDP-G algorithm, developed to handle the arbitrary precedence constraints are presented in Section 4.5 and Section 4.6, respectively. The summary of the main contributions of the developed scheduling method is given in Section 4.7.

4.2 Related Work

Current on-line scheduling algorithms for dynamic non-deterministic real-time systems do not attempt to maximize the system value at all points in time. In these algorithms, as for example in [7] [25], as long as a feasible schedule is reached and the minimum number of low value tasks are rejected, the internal order of the scheduled tasks is not important. That is, the order in which the low value and the high value are scheduled within a feasible schedule is ignored. This approach, however, does not produce the best results in systems with the objective to maximize the accrued system value, especially in the presence of temporary or permanent overloads.

To maximize the accrued value in presence of overloads, a scheduler must adjust the design of a feasible schedule by anticipating overloads. Due to non-deterministic arrivals, a created schedule must maximizes the system value at all times.

Regarding this objective, the study by Huang *et al.* [25] in the area of distributed real-time databases, shows that criticalness (or in our case, the task's contributing value) presents the most dominant factor. That is, criticalness is more important than the deadline, if the metric is a total accrued value. This conclusion is reached through the comparison of different scheduling algorithms in a real-time transaction model. They also showed that the conflict resolution protocols that combine both deadline and criticalness have better performance than the resolution protocols based on a single criterion.

These findings correspond to the observations we made during the development of the RDS scheduling algorithm—the independent task scheduling algorithm, presented in Chapter 3, Section 3.2. By combining several task and system parameters—namely, by combining the earliest possible start time, the deadline and the value density—we obtained a very robust, effective and efficient on-line scheduling algorithm for dynamic and very complex real-time system. To maximize the accrued system value, the heuristic function in RDS algorithm assigns the largest weight

to the value density, a parameter that measures the contributed value per unit computation time.

To understand the broader issues concerning the task group scheduling, let's look at the current Scheduling Theory results. The results that address the problem of maximizing the accrued system value in single and multiprocessor systems. Before presenting the state-of-the-art, a short introduction to the Scheduling Theory notation commonly used to describe a wide class of scheduling problems is presented; followed by a brief discussion of the choice of the scheduling metric.

4.2.1 Notation and Problem Classification

The terminology used in Computer Science can be easily mapped to the terminology commonly found in Scheduling Theory. For example, a "task" corresponds to "job," "processor" to "machine," "processing time" to "computation time" and "importance" or "value" to "weight." Furthermore, when addressing the precedence constraints the "task" corresponds to a "node," the term used in both Scheduling Theory and Graph Theory.

To describe different scheduling problems, we adopt the 3-field notation proposed in [13]. Using this notation, a scheduling problem is specified as a triplet $\alpha|\beta|\gamma$, where α indicates the *processor environment* (for example, single processor, or parallel processors), β indicates the *task characteristics* (for example, independent tasks vs. precedence constrained tasks), and γ indicates the *optimality criterion* (such as makespan, or maximum lateness). This notation provides a very concise description of a variety of scheduling problems[1]. For example, $1||\sum w_jC_j$ specifies the scheduling problem for a single processor system with objective to maximize the total weighted completion time, where w_j is task's contributing value (or weight), and C_j is task's completion time); or in the next example, $1|prec, res, d_j|\sum w_jC_j$ specifies the same

[1]For detailed description, and for a description of an extended 3-field notation see [45].

scheduling problem, but subject to arbitrary precedence constraints (*prec*), resource constraints (*res*) and timing constraints (deadlines, d_j).

In this notation, the problem addressed in this chapter is specified as the $m|prec, res, d_j|\sum w_j C_j$ scheduling problem. That is, our scheduling problem is the multiprocessor scheduling problem with the objective of minimizing the total weighted completion time subject to deadline, resource, and precedence constraints. Note that the deadline in our case is assigned to task groups, while the specified scheduling problem assumes that the deadlines are assigned to individual tasks. To resolve this discrepancy, a task group deadline is preprocessed and the individual pseudo deadlines are assigned to each task according to the ALAP (as late as possible) algorithm [38], also known as the latest start time first algorithm [42].

4.2.2 Metric

Among the many different metrics found in Scheduling Theory, the metric that maximizes the accrued value at all times is the one that minimizes the weighted completion time. This metric, generally, favors the algorithms that schedule tasks in the largest contributing value per unit time first fashion, i.e., that schedule tasks in the decreasing order of their value densities. The method is especially important in dynamic real-time systems, where temporary and/or permanent overloads are likely to happen, and where a scheduler must anticipate for these overloads to maximize the accrued system value.

To illustrate this, let's consider a single processor scheduling problem with three independent tasks, namely T_1, T_2, T_3. The task parameters are given in Table 4.2.2. Observe that the T_1 and T_2 are schedulable at time $t = 0$. Task T_3, the most valuable task, is scheduled to arrive during the execution of the first scheduled task, either task T_1 or T_2. Specifically, T_3 is schedulable at time $t = 5$. Furthermore, observe that due to the given timing constraints at the arrival of task T_3 an overload occurs, implying that at time $t = 5$, one of the tasks must be rejected.

Table 4.1 Parameters for Three Independent Tasks.

Task	Arrival	Processing Time	Contributing value	Deadline
T_1	$a_1 = 0$	$p_1 = 10$	$w_1 = 1$	$d_1 = 27$
T_2	$a_2 = 0$	$p_2 = 10$	$w_2 = 10$	$d_2 = 28$
T_3	$a_3 = 5$	$p_3 = 10$	$w_3 = 100$	$d_3 = 29$

If, for example, a scheduler orders tasks using the earliest deadline first order, the schedule at time $t = 0$ is $Q_1 = \{T_1, T_2\}$. Consequently, the task T_2, even though more valuable than the task T_1, is going to be rejected upon the arrival of the task T_3. The total weighted completion time, σ_1, for Q_1 sequence at time $t = 0$ is:

$$\sigma_1 = \sum_{j \in Q_1} w_j C_j = w_1 \times C_1 + w_2 \times C_2 = 1 \times 10 + 10 \times 20 = 210.$$

On the other hand, if the scheduler sequenced the tasks in the decreasing order of their value densities, a sequence $Q_2 = \{T_2, T_1\}$ would be obtained. Therefore, in this sequence the task T_1 (the least valuable task) is rejected at time $t = 5$, thus, producing a higher system value. Furthermore, this sequence produces a smaller weighted completion time, too. Specifically, the total weighted completion time, σ_2, for sequence Q_2 at time $t = 0$ is:

$$\sigma_2 = \sum_{j \in Q_2} w_j C_j = w_2 \times C_2 + w_1 \times C_1 = 10 \times 10 + 1 \times 20 = 120.$$

In conclusion, this example shows that by minimizing the weighted completion time, the total accrued value in dynamic real-time systems is maximized, too. That is, the scheduler that schedules more valuable tasks up-front is more effective in overloads. For this reason, let's focus on the classical and the most recent scheduling results with respect to the above discussed metric: the minimization of the weighted completion time.

4.2.3 Single Processor Scheduling

The fundamental result for the $1||\sum w_j C_j$ scheduling problem, presented by Smith in [53], says that any sequence that places the tasks in the order of non-decreasing ratios, $\rho_j = p_j / w_j$, is an optimal sequence. This rule is well known as the Smith's rule, or simply as the "ratio rule," and it is applicable only to scheduling of independent tasks without any constraints. It requires only a sorting procedure, resulting in a computational complexity of $O(n \log n)$. Note that in its simpler form, i.e., when applied to a set of tasks with unit weights, the ratio rule produces an optimal sequence by placing the tasks in a non-decreasing order of their processing times—same as like the SPT (shortest processing time) rule.

Regarding the problems with precedence constraints, the $1|chain|\sum C_j$ problem was solved first in [13]; while more complicated problems with tree-like precedence constraints and arbitrary weights are solved in [23, 1, 50]. All these algorithms have the computational complexity of $O(n \log n)$.

One algorithm that stands out with its very elegant solution for rooted tree problems is presented by Adolphson [2], where the algorithm for arbitrary precedence constraints is presented, too. Unfortunately, the algorithm for arbitrary precedence constraints provides only a partial solution in some instances, and it has a computational complexity of $O(n^3)$. The limited applicability is due to the reduction theorems that are not applicable to some special, but very common task group structures.

Further progress in task group scheduling was made by Lawler in [27], where he presented an algorithm for series-parallel digraphs based on Sidney's theory [51]—the theory developed for the decomposition of digraphs into modules. Sidney's theory defines a very important notion of so-called "job modules". It states that there exists an optimal sequence of a given digraph that is *consistent* with the optimal sequence found in a "job module". Note that the tasks in a "job module" are

related in the same way to tasks not found in the "job module." The consequence of this property is that an optimal sequence within a "job module" is always a subsequence of a global optimal solution. It is necessary to mention that Lawler's solution requires the rooted binary tree (called a decomposition tree) to be given for any series-parallel digraph. To identify that a given digraph is a series parallel graph and to obtain its decomposition tree, an algorithm presented in [58] is suggested. This algorithm has a computational complexity of $O(|N| + |A|)$, where N is the number of tasks and A is the number of arcs in a given digraph.

The slow progress in providing the scheduling algorithms for problems with precedence constraints is mainly due to the NP-completeness of even "simple" scheduling problems. For example, it has been shown that the digraph with general precedence constraints results in NP-hardness, even if all $p_j = 1$ or all $w_j = 1$. That is, either $1|prec, p_j = 1|\sum w_jC_j$ or $1|prec|\sum C_j$ reduces to an NP-hard problem [27, 29]. Furthermore, what makes the progress even more difficult is that the $1|r_j|\sum w_jC_j$ problem, the problem without any precedence constraints but with release dates, is, also, an NP-hard problem [30].

The above results do not address the timing constraints. So, how about the results that deal with this most prominent real-time scheduling criterion: meeting the given deadlines?

The solution for a very simple single processor problem without the precedence constraints, the $1|d_j|\sum C_j$ problem, was originally proposed in [53]. In this paper, Smith suggested the following procedure: (1) from among all tasks that are eligible to be sequenced last ($d_i \geq p_1 + ... + p_n$), put the task with the largest possible time last, and (2) repeat the procedure on the set of $(n-1)$ remaining tasks. This algorithm has an $O(n \log n)$ computational complexity.

In the same paper, Smith suggested an extension to his algorithm by proposing a solution for the $1|d_j|\sum w_jC_j$ problem. However, Emmons showed that this extension

is incorrect by providing a counter example in [16]. Independently of Emmons, Burns showed the same in [9], where he proves that the Smith's algorithm converges to the constrained local optimum sequence Q with the property that $\sum_{j \in Q} w_j C_j$ is less than or equal to the sum of weighted completion times, for a subset of feasible adjacent sequences formed by considering only adjacent pair-wise interchange. Burns, also, suggested his improvement to the Smith's algorithm, which still provides a local optimum that may not always turn equal to a global optimum.

Finally, Lenstra *et al.* proved that $1|d_j|\sum w_j C_j$ problem is an NP-hard problem in [30]. Therefore, an optimal solution that minimizes the total weighted completion time subject to deadline constraints of independent tasks, i.e., the optimal solution for a basic real-time scheduling problem does not exist.

Considering the restrictions of the constructive algorithms, Lawler suggested the use of the "hybrid" algorithms [28]. Specifically, he suggested that the algorithms with a real potential should combine the best features of branch-and-bound technique and dynamic programming [28].

One example of a such an attempt is found in [6]. In this paper, Bansal uses the branch-and-bound technique to schedule independent tasks. Both the branching step and the pruning (the node elimination) step are based on the lower bound of the weighted sum of completion times. This algorithm provides a global optimum solution. However, in extreme cases it tends to have a very long execution time that approaches the exhaustive search bound.

On the other hand, Sidney and Steiner [52] used the dynamic programming to modify the scheduling algorithm proposed by Möhring and Radermacher in [34]. Möhring and Radermacher have shown that the complexity of the $1|prec|\sum w_j C_j$ problem remains polynomial over the class of precedence constraints that can be decomposed into "building blocks" with size limited by the constant $m \geq 4$. Their solution has the worst case complexity of $O(n^{(m^2)})$. While Sidney and Steiner use

dynamic programming to cut the complexity down to $O(n^{(m+1)})$, where m is the greatest width of the "building blocks."

Finally, let's mention that the most recent progress in solving the deadline constrained scheduling problems with the objective to minimize the weighted completion time is made by a number of heuristic algorithms. A survey of more important heuristics can be found in [40]. Additionally, in this paper, Posner derives optimization conditions and shows that the worst case bound of any algorithm cannot be found independently of the problem parameters, namely, the number of tasks and the width of a graph.

4.2.4 Multiple Processor Scheduling

Minimizing the sum of completion times subject to deadline and precedence constraints for multiprocessor systems is far more difficult than for single processor systems.

Even the simplest problem of scheduling the non-preemptive independent tasks on identical parallel processors, the $m||\sum w_j C_j$ problem, is NP-hard, and it holds even for a two processor system, $2||\sum w_j C_j$ problem.

In contrast to the problems with arbitrary weights, the problem with unit weights, the $m||\sum C_j$ problem is very simple. A greedy scheduling algorithm that resembles the SPT rule, proposed first in [13], provides an optimal solution. Furthermore, a solution for the same problem, but on the uniform processors—the processors with constant but potentially different processing speeds—is presented in [24]. The computational complexity of these algorithms is in $O(n \log n)$.

On the other hand, an interesting heuristic scheduling approach was taken by Bruno *et al.* [8], where they analyzed the algorithm called RPT (reversed processing time) algorithm. This algorithm consists of three phases: (1) use the list scheduling algorithm based on the largest processing time first (LPT) algorithm, (2) reverse the

order of the job schedules of individual processors, and (3) left justify each schedule. A similar heuristic to this is presented and empirically tested in [5].

In spite of these results, the problems that include the precedence constraints, even in the simplest form are NP-hard. Sethi showed that minimization of the sum of unit weight completion times for a two processor system with tree like precedence constraints, the $2|tree|\sum C_j$ problem, is already NP-hard. This holds both for in-tree and out-tree task group structures [46].

It is important to mention that allowing the preemption of tasks (nodes) does not make the problem any easier. Following the McNaughton's theorem [33], it can be shown that by mapping the preemptive case to the non-preemptive case, the above two processor problem still remains NP-hard. Interestingly enough, for every non-preemptive algorithm there exists a preemptive counterpart for almost all unit-time scheduling problems. These preemptive algorithms use the same techniques for dealing with precedence constraints as the corresponding non-preemptive algorithms, with one caveat: the preemptive algorithms are considerably more complex.

Finally, besides the work for uniprocessor systems [35] [12] and for distributed systems [11], the work most closely related to the scheduling problem considered in this and the next chapter is found in [63] [41] and [39]. Xu and Parnas [63] developed a pre-run-time scheduler that starts by using some initial schedule (for example, a schedule provided by the earliest-deadline-first algorithm) which is then iteratively improved until an optimal (or the best feasible schedule, after a limited number of iterations) is found. Their approach is designed for static "hard" real-time systems, where tasks are give deadlines, release times, arbitrary precedence constraints, and *exclusion* constraints (meaning that a task cannot be preempted by any task from the set it excludes).

On the other hand, Ramamritham [41] and Peng and Shin [39] present the allocation and scheduling, branch and bound, algorithms for complex periodic task groups with arbitrary precedence constraints and communication costs for distributed real-time systems. The former, Ramamritham's algorithm that uses a heuristic approach to derive the bounding rules is, in addition, designed to handle allocation and scheduling problems of distributed multiprocessor systems with fault tolerance requirements.

In spite of the very wide range of covered problems, the above algorithms do not support task groups with different levels of contributing values, and they can not be used for on-line scheduling of non-deterministic dynamic real-time systems.

In this chapter, we present our solution to on-line multiprocessor scheduling of task groups with arbitrary precedence constraints, deadlines, different contributing values, and resource constraints.

4.3 Model Description

Similar to the model described in Chapter 3, Section 3.2, the underlying model for scheduling of tasks with precedence constraints assumes a tightly coupled multiprocessor system, consisting of one system processor and one or more application processors. The system processor, dedicated to scheduling only, off-loads the scheduling overhead from the application processors. On the other hand, application processors are guaranteed to have predictable executions. That is, a system processor is used as a buffer between the non-deterministic real-time system on one side, and the deterministic task executions on the application processors on the other side.

The task groups considered in our model are non-atomic task groups with *a priori* described timing and precedence constraints. The arbitrary precedence constraints are specified by a directed acyclic graphs (DAGs) with a *transitive closure* property—the property that does not allow redundant arcs. An example of

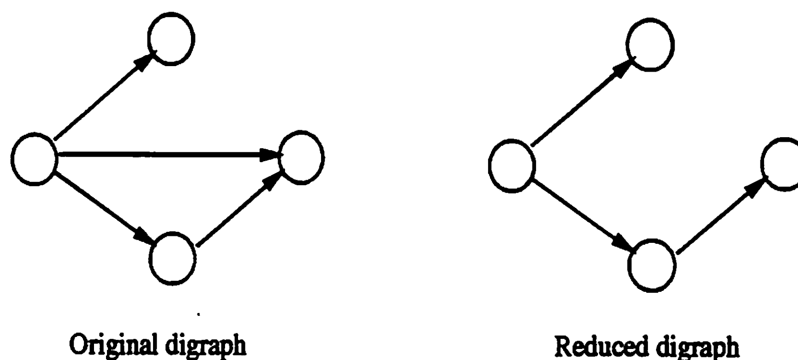

Figure 4.1 Reduction to a DAG with a Transitive Closure.

an arbitrary group and a reduced DAG with transitive closure is given in Figure 4.1. Additionally, the individual tasks (the tasks within a task group) are well-defined non-preemptive tasks that can be executed on any available application processor. The individual tasks are characterized with their worst case computation time, p_j, resource requirements, $\vec{r_j}$, and contributing value, w_j.

A task T_i is called a *terminal* task if and only if there are no arcs directed *out* of T_i. Consequently, a task T_j is called a *source* task if and only if there are no arcs directed *into* T_j. Furthermore, a task T_i is a *predecessor* of a task T_j if and only if there is an arc from T_i to T_j, and T_j is T_i's *successor*. A task T_i is an *immediate predecessor* of T_j if and only if T_j is its successor, and no other predecessor of T_j is a successor of T_i, implying that T_j is an *immediate successor* of T_i.

It should be observed that in spite of the assumption that the task groups are eligible for execution at the arrival time, the task groups with future release times are intrinsically supported, as well. This support is a side-effect of the task group scheduling on multiprocessor systems, where the individual tasks are indirectly assigned release times due to the given precedence constraints. Therefore, any scheduler that handles task groups must also handle the tasks with future release times, whenever the tasks of a given group are allowed to be scheduled on more than one application processor.

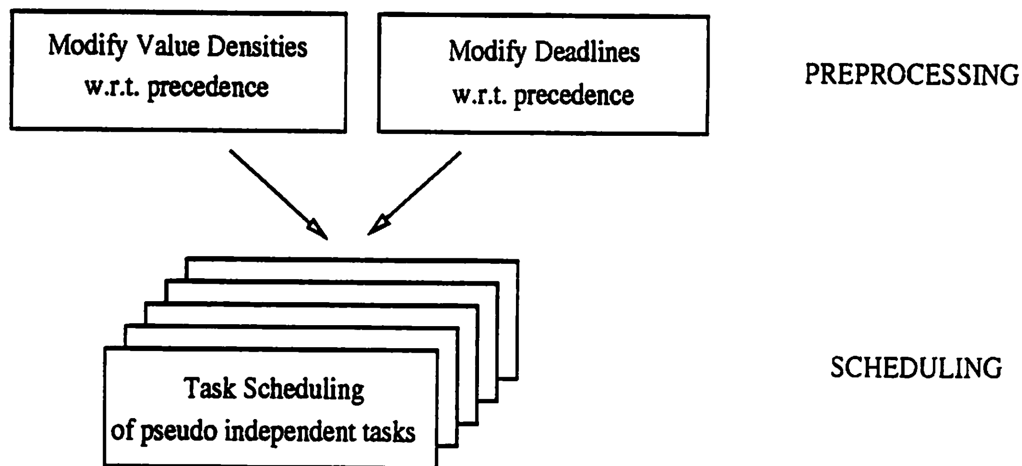

Figure 4.2 The Task Group Scheduling Approach.

4.4 Design Strategy

Our approach to the problem of on-line scheduling of tasks with precedence constraints uses a divide and conquer strategy. Each task group is off-line preprocessed with respect to deadlines and value densities. The obtained parameters can then be used by a wide range of on-line schedulers, designed for independent task scheduling (Figure 4.2).

This method supports efficient scheduling of arbitrary task groups. Specifically, its high efficiency is achieved by reducing a very complex problems of task group scheduling to much simpler problems of pseudo independent task scheduling— a benefit of utilizing off-line preprocessing. Furthermore, utilizing the parameters obtained by off-line preprocessing, the effectiveness of on-line scheduling algorithms is improved.

In this thesis, we focus only on the value density preprocessing algorithms; the deadline preprocessing is the subject of [62]. Value density preprocessing involves assigning of so-called "reflective" value densities to individual tasks, based on given precedence constraints and initial task parameters. The reflective value densities should contain the information about tasks' initial value densities and about the

value densities of their successors. By utilizing the reflective value densities, the on-line scheduler does not need to traverse the entire task group to select the best task to schedule next.

Specifically, the reflective value densities indicate whether some low value task is followed by more valuable task(s) or not. If a task is followed by more valuable tasks, its reflective value density is larger than its initial value density; and if not, its reflective value density remains equal to its initial value density. The reflective value density can be used to indicate which branch of a given task group contains more valuable tasks, and thus should be scheduled as early as possible. Utilizing this information, the higher total accrued value can be achieved in temporary and permanent overloads.

The advantages of our design strategy are summarized as follows:

- The domain of already developed scheduling algorithms for independent tasks is extended to problems of scheduling the task groups with arbitrary precedence constraints. This eliminates the need to develop new scheduling algorithms that support task groups when the individual tasks have different contributing values.

- Off-line preprocessing enables more effective and efficient scheduling of task groups with arbitrary precedence constraints through the use of reflective parameter.

Observe that further benefits are obtained by separating the preprocessing of timing constraints from the preprocessing of value densities. Due to this separation, real-time and non real-time schedulers can easily improve their efficiency and effectiveness. The difference between these two systems is that non real-time systems do not require the deadline preprocessing, while the value density preprocessing is commonly found in both systems. Clearly, the improvement in performance is obtained only if the scheduling objective is to maximize the accrued system value.

In the next two sections, a detailed description of two different value density preprocessing algorithms is presented. First, the VDP-R algorithm for tree-like task groups is described; and second, the VDP-G algorithm for task groups with arbitrary precedence constraints is presented. The analysis and verification through simulation of the more general VDP-G algorithm is discussed in the next chapter, Chapter 5.

4.5 VDP-R: An Algorithm for Rooted Tree Task Groups

The VDP-R algorithm preprocesses value densities for task groups with the chain or rooted tree precedence constraints. This simple algorithm for $1|tree|\sum w_jC_j$ scheduling problem is primarily presented to illustrate the principles of the method later used to develop an algorithm for general precedence constraints, the VDP-G algorithm.

By definition, the value density of an individual task is the ratio of the task's contributing value over the task's computation time. According to common Scheduling Theory practice, instead of using the value density when describing the scheduling algorithm, the inverse of the value density (referred to as the ratio, ρ) is used. Thus, the individual ratio, ρ_j of a task T_j, is defined as the task's computation time over the task's contributing value, i.e., $\rho_j = p_j/w_j$. On the other hand, the task's reflective parameters are defined similarly, and denoted as ρ'_j, p'_j, and w'_j.

All illustrative examples in this chapter, use ordered pairs (p_j, w_j) and (p'_j, w'_j) to specify the initial and reflective parameters of individual tasks, respectively. Note that the ratios are not explicitly indicated because they can easily be obtained from the given parameters.

Before presenting the VDP-R algorithm, let's look at two very simple examples, and also, discuss the issues involved in the preprocessing of group's value densities. In both examples a task group is made up of only three tasks with chain-like precedence constraints. Figure 4.3 illustrates the first example with the initial and

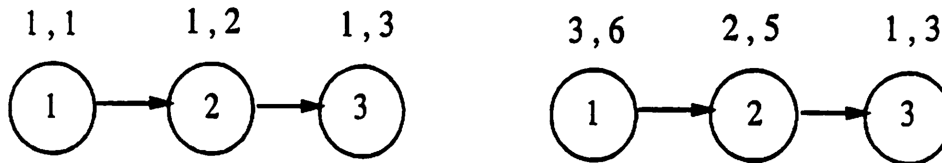

Figure 4.3 Initial and Reflective Parameters with Decreasing Ratios

reflective parameters indicated above the corresponding individual tasks, T_1, T_2, and T_3. The initial parameters, the (p_j, w_j) pairs, are specified in the task group on the left; and the reflective parameters are specified in the task group on the right. Note that the initial ratios are assigned in a decreasing ratio order; with the source task, T_1, as the *least* valuable task, and the terminal task, T_3, as the *most* valuable task[2].

Let's assume that task T_1 (the first task eligible to be scheduled) is competing for the processor time with some other tasks in the system, not indicated in this figure. Being the least valuable task in its group, it looks counter intuitive to use T_1's initial parameters when deciding which task to schedule next. T_1's initial parameters do not fully represent the importance of selecting T_1 over the other tasks, they do not reflect the presence of its more valuable successors, T_2 and T_3.

The parameters that indicate the presence of task's more valuable successors are the reflective parameters given in figure on the right. Observe that the reflective ratios for tasks T_1 and T_2 are smaller that their initial ratios, i.e., $\rho_1' = 3/6 < \rho_1 = 1$ and $\rho_2' = 2/5 < \rho_2 = 1/2$. This indicates that tasks T_1 and T_2 are more competitive, and that they represent the existence of more valuable successors. The intuition behind the use of reflective parameters is that by using the more competitive ratios, the more valuable tasks will be scheduled sooner, and thus, they will contribute their value to the system sooner. In other words, by using more competitive, i.e.

[2]The distinction as to which task is more valuable is based on the Smith's ratio rule [53], where the task with the lower ratio is more valuable than the task with the higher ratio.

Figure 4.4 Initial and Reflective Parameters with Increasing Ratios

reflective, ratios more valuable tasks have a better chance to be scheduled earlier, even if their predecessors are less valuable.

In the second example, Figure 4.4, the individual tasks are assigned the parameters in the reverse ratio order, the most valuable task first and the least valuable task last. In this case, the reflective parameters in figure on the right are the same as the initial parameters given in figure on the left. The intuition is that to give proper scheduling treatment to individual tasks, the individual tasks should not lower their value densities to indicate their less valuable successors. The individual tasks should always use the higher of two value densities as the one to advertize, and thus, successfully compete with other tasks eligible for scheduling.

To describe the method used to assign the reflective parameters given in these examples, let's return to the first example given in Figure 4.3—the group with the increasing value densities. To assign the reflective parameters, the task group is processed from the terminal task towards the source task. By definition, a terminal task has no successors, and thus, its reflective parameters are equal to its initial parameters, $(p_3, w_3) = (p'_3, w'_3) = (1, 3)$. After processing the terminal task, the task from the next level, T_2, can be processed. This task has higher ratio than the terminal task, T_3. To determine its reflective parameters, we use the concept first presented in [17]. It states that the reflective computation time is equal to the sum of its initial computation time and the reflective computation times of all, more valuable immediate successors. The same holds for the weights. Therefore, in this

example, the reflective computation time and value for T_2 are: $p'_2 = p_2 + p'_3$, and $w'_2 = w_2 + w'_3$, respectively.

Next, the task T_1 is ready to be processed. Its initial ratio is smaller than T_2's reflective ratio ($\rho_1 = 1/1 > \rho_2 = 2/5$). Therefore, T_2 is the more valuable task, so T_1's reflective ratio should indicate that. Following the same procedure, the reflective parameters for T_1 are: $p'_1 = p_1 + p'_2$, and $w'_1 = w_1 + w'_2$. The obtained reflective ratio indicates that T_1 precedes some more valuable tasks, and that to give them a proper chance to get scheduled accordingly, T_1 should use its reflective rather than its initial ratio to compete with other tasks in the system.

On the other hand, in the example given in Figure 4.4—the group with the decreasing value densities—the reflective pairs assigned are the same as their corresponding initial pairs. To determine the values of individual reflective pairs, the task group is processed from its terminal task towards its source task. In contrast to the previous example, this task group does not contain any task whose successor is more valuable than the task by itself. Therefore, each task retains its initial values, $p'_i = p_i$ and $w'_i = w_i$. That is, the task does not decrease its value because of its less valuable successors. The task always uses the higher value to compete with other tasks in the system.

The simplified description above of the assignment of the reflective parameters is a part of a more general algorithm, the VDP-R algorithm, that handles the task groups with rooted tree-like precedence constraints. The value density preprocessing for the VDP-R algorithm is given in Figure 4.5.

To describe this algorithm, let's use a rooted tree task group with the initial parameters assigned as in the example from [2]. This task group with specified initial parameters is given on the left, Figure 4.6. The only modification made to the original example is that the $-\infty$ contributing value to the root task is substituted with a finite contributing value, i.e., $w_0 = 1$.

The procedure described in Figure 4.5 is similar to the procedure presented in [17]. The difference is that the Garey's algorithm produces a schedule sequence only for a particular $1|tree|\sum w_jC_j$, non-real time problem, while our algorithm computes the reflective parameters that can be later utilized by a number of different, real-time and non real-time scheduling algorithms. Furthermore, our approach provides the absolute measure of how valuable each task is rather then simply producing a relative order. It processes a single task group at a time, rather than all task groups eligible for scheduling. Due to the absolute measure of tasks' values, the reflective parameters can be utilized in a much wider range of scheduling problems—ranging, for example, from single to multiprocessor systems, or from real-time to non real-time systems.

Both approaches process a group starting from the tasks on the highest unprocessed level (with terminal nodes being on the highest level) towards the root of the tree. At the same unprocessed level, the most valuable task is processed first. Its reflective parameters, initially being the same as its initial parameters, are gradually improved by adding the reflective parameters of the more valuable immediate successors—starting with the most valuable immediate successor first.

To illustrate the VDP-R algorithm, let's trace the assignment of the reflective parameters step by step. For example, the label STEP 1 corresponds to the first step of the algorithm in Figure 4.5, and so on. The example is concluded by performing a non real-time sequencing procedure. The results obtained are compared with the results given in [2].

STEP 1.—Initialize the reflective parameters

$$w'_j = w_j, \text{ and } p'_j = p_j \text{ for } j = 0, 1, \ldots, 8.$$

STEP 2.—Sort a ready-to-process list in an ascending order of the reflective ratios.

$$R_L = \{T_8, T_6, T_7\}.$$

1. Set all reflective parameters equal to corresponding initial parameters.
2. Include all tasks from the highest unprocessed level into ready-to-process list, R_L, in ascending order of reflective ratios.
3. From R_L, get the first task, T_j, i.e., the most valuable task with the smallest ratio ρ'_j.
4. Find the most valuable immediate successor T_i of task T_j. If $\rho'_j < \rho'_i$ then $w'_j = w'_j + w'_i$ and $p'_j = p'_j + p'_i$.
5. If the last node from R_L is processed, include all tasks from the next, unprocessed level into R_L, in ascending order of reflective ratios.
6. Go to step 3 until all tasks in the given task group are processed.

Figure 4.5 VDP-R: The Value Density Propagation Algorithm for Rooted Trees.

STEP 3.—Get the first task from R_L, the task T_8.

STEP 4.—T_8 is more valuable than its predecessor T_5, $\rho'_8 < \rho'_5$ $(1/7 < 2/2)$.

Therefore, $w'_5 = w'_5 + w'_8 = 9$ and $p'_5 = p'_5 + p'_8 = 3$.

STEP 3.—Get task T_6.

STEP 4.—$\rho'_6 > \rho'_4$ $(5/2 > 1/4)$; no change—T_6's predecessor is more valuable.

STEP 5.—Update the ready-to-process list. $R_L = \{T_4, T_3, T_5\}$.

STEP 3.—Get task T_4.

STEP 4.—$\rho'_3 > \rho'_1$ $(2/6 > 2/7)$; do nothing.

STEP 3.—Get task T_3.

STEP 4.—$\rho'_3 > \rho'_1$ $(2/6 > 2/7)$; do nothing.

STEP 3.—Get task T_5.

STEP 4.—$\rho'_5 > \rho'_1$ $(3/9 > 2/7)$; do nothing.

STEP 5.—Update the ready-to-process list. $R_L = \{T_1, T_2\}$.

STEP 3.—Get task T_1.

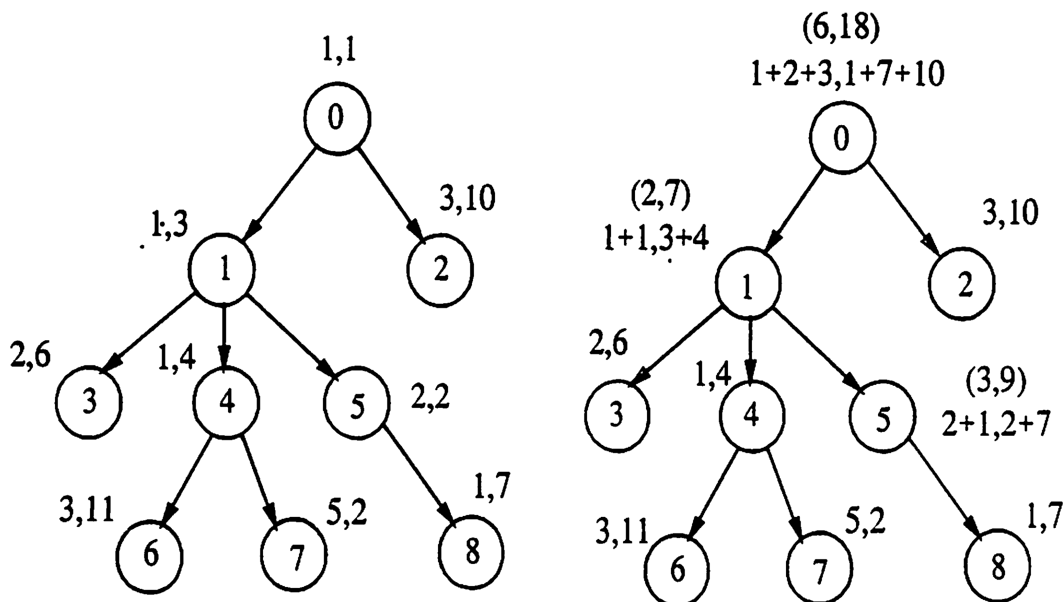

Figure 4.6 VDP-R Applied to an Example from [2].

STEP 4.—$\rho_1' < \rho_0'$ $(2/7 < 1/1)$, therefore, $w_0' = 8$ and $p_0' = 3$.

STEP 3.—Get task T_2.

STEP 4.—$\rho_2' < \rho_0'$ $(3/10 < 3/8)$, therefore, $w_0' = 18$ and $p_0' = 6$.

STEP 5.—Update the ready-to-process list, $R_L = \{T_0\}$.

DONE.

The above obtained reflective parameters are illustrated in the figure on the right, Figure 4.6.

This completes the description of the value density preprocessing algorithm for rooted trees. To compare our results with the original results, let's sequence the tasks according to the objective of the given problem—the minimization of the weighted completion time.

By utilizing the results of the preprocessing algorithm, the $1|tree|\sum w_jC_j$ problem is reduced to a simpler $1||\sum w_jC_j$ problem. A sequencing algorithm for the reduced problem that provides an optimal solution is the algorithm based on the Smith's rule. For the illustration purpose, the sequencing of preprocessed task group is presented next.

- Form an enabled set. $E = \{T_0\}$.
- Schedule task T_0. $Q = \{T_0\}$.
- Form an enabled set in non-decreasing order of ρ_j. $E = \{T_1, T_2\}$.
- Schedule task T_1, the most valuable task from the enabled set. $Q = \{T_0, T_1\}$.
- Insert newly enabled tasks to the enabled set, maintaining the non-decreasing order of ρ_j. $E = \{T_4, T_2, T_3, T_5\}$.
- Schedule task T_4, and enable tasks T_6 and T_7.
  $Q = \{T_0, T_1, T_4\}$ and $E = \{T_6, T_2, T_3, T_5, T_7\}$.
- Schedule task T_6—There are no new enabled tasks.
- Schedule task T_2—There are no new enabled tasks.
- Schedule task T_3—There are no new enabled tasks.
- Schedule task T_5, and enable task T_8.
  $Q = \{T_0, T_1, T_4, T_6, T_2, T_3, T_5\}$ and $E = \{T_8, T_7\}$.
- Schedule task T_8, and then task T_7.
- The final schedule is $Q = \{T_0, T_1, T_4, T_6, T_2, T_3, T_5, T_8, T_7\}$.

Note that while scheduling task T_3, there was a tie with task T_5 ($\rho_3 = 2/6 = \rho_5 = 3/9$). Due to the characteristic of trees, this tie can be solved arbitrarily. For example, if T_5 were chosen instead of T_3, the final schedule would be optimal, too. The obtained sequence would be the same as the one in [2]. That is, the resulting sequence would be $Q' = \{T_0, T_1, T_4, T_6, T_2, T_5, T_8, T_3, T_7\}$.

4.6 VDP-G: An Algorithm for Arbitrary Task Groups

Even in a very simple form, the problem with arbitrary precedence constraint for a single processors, the $1|prec|\sum w_jC_j$ problem, is an NP-hard scheduling problem [27, 29]. The NP-hardness holds even for unit processing times ($p_j = 1$) and unit weights ($w_i = 1$). In other words, the simplest scheduling problem with arbitrary precedence constraints, the $1|prec, p_j = 1|\sum C_j$ problem is NP-hard.

How do these results relate to our approach made up of two parts: the off-line assignment of the reflective parameters and the on-line scheduling of the pseudo independent tasks?

For the sake of argument, assuming a given optimal assignment of the reflective parameters, let's determine if the on-line scheduling procedure can produce an optimal solution.

In the previous section, we observed that the optimal schedule for independent tasks on a single processor system, the $1||\sum w_jC_j$ problem, is obtained by using the Smith's rule. Unfortunately, this rule does not scale up to the multiprocessor case. Even worse, the preprocessing reduces the $m|prec|\sum w_jC_j$ problem only to a $m|r_j|\sum w_jC_j$ problem—a problem with future release times, r_j. That is, the problem of scheduling the arbitrary task groups is reduced to the problem of scheduling pseudo independent tasks with future release times. The introduction of the release times is a result of the imposed precedence constraints and the potential parallelism when scheduling a group on the multiprocessor system. The reduced problem is, again, an NP-hard problem [30]. That is, there is no polynomial, or for that matter not even a pseudo polynomial algorithm that produces an optimal solution.

In spite of the inherent NP-hardness, the multiprocessor scheduling of the tasks with arbitrary precedence constraints should not be neglected. In this section, we develop the polynomial value density preprocessing algorithm, VDP-G, that generates reflective parameters based on a heuristic approach. The effectiveness of

this algorithm is analyzed and verified through simulation in Chapter 5. Specifically, the quality of the produced reflective parameters is tested using an on-line real-time scheduling algorithm designed for complex real-time systems—namely, the RDS algorithm, developed in Chapter 3, Section 3.2.

The remainder of this section is organized as follows. First, the notation and the definitions of the terms used in the development of the VDP-G algorithm is presented. Second, the actual algorithm is outlined. And third, three representative examples that cover a wide range of task groups are presented.

4.6.1 The Notation

The following notation defines the terms used in the development of the VDP-G algorithm.

- *I-params* are the initial parameters of the individual tasks within a given task group. They include:
  - w_j as an initial weight of a task T_j,
  - p_j as an initial worst case processing time of a task T_j, and
  - ρ_j as an initial ratio of a task T_j ($\rho_j = p_j/w_j$).
- *R-params* are the reflective parameters produced by the algorithm. These parameters include:
  - w'_j as the reflective contributing value of a task T_j,
  - p'_j as the reflective computation time of a task T_j,
  - ρ'_j as the reflective ratio of a task T_j ($\rho'_j = p'_j/w'_j$).

Additionally, a task that has the R-params equal to its I-params, i.e., the task that represents only itself and has no successors that are more valuable, is called a "supreme" task, labeled as *S-task*.

The goal of the VDP-G algorithm is to identify all S-tasks in a given group. Furthermore, it has to provide a method to measure the influence region and the intensity with which the S-tasks improve the reflective parameters of other, S-tasks in a group. This is achieved by maintaining a two dimensional matrix that records all S-tasks that contribute to the reflective parameters of individual tasks. This matrix is called $\vec{S} \times \vec{V}$ space. The vector $\vec{S_j} = \{s_j^0, s_j^1, \ldots, s_j^{n-1}\}$, where s_j^i is a Boolean variable, identifies all tasks that contribute to the values of R-params of a task T_j. If a task is a contributor to T_j's R-params, a corresponding s_j^i is set to 1. If however, T_j has no other contributors but itself, only s_j^j is set to 1 and the task is called an S-task. On the other hand, a vector $\vec{V_j} = \{v_j^0, v_j^1, \ldots, v_j^{n-1}\}$, where v_j^i is a Boolean variable, points only to T_j's successors that contribute to its R-params. Clearly, $v_j^j = 1$.

The information in $\vec{S} \times \vec{V}$ space is updated while preprocessing a single task, and it is used to compute current R-params, during the preprocessing phase as well as during the sequencing phase. Specifically, to determine the R-params for a task T_j, all contributing tasks must be identified first (the tasks with s_j^i set to 1). Next, for each identified task, a set of its successors that are S-tasks is found. A union of all $\vec{V}$ vectors of all identified tasks is determined, and labeled as $\vec{\Gamma_j} = \{\gamma_j^0, \gamma_j^1, \ldots, \gamma_j^{n-1}\}$, where γ_j^i is a Boolean variable. For each γ_j^i set to 1, the corresponding I-params of each task T_i are summed up to form the R-params of a task T_j.

The other notation used includes:

- *Enabled set* is a subset of tasks whose predecessors have already been scheduled or executed.
- *Enabled task* is a task from the enabled set.
- R_L is a list of ready-to-process tasks used during the preprocessing phase.
- R'_L is an auxiliary ready-to-process list.

4.6.2 The Algorithm Description

Using the above specified notation, this section describes the VDP-G algorithm in a procedural fashion.

The main body of the algorithm is given in Figure 4.7. Notice that at each processing level, the most valuable task from the ready-to-process list is processed next, Figure 4.7, line 7. After selecting the most valuable task, the $\vec{S} \times \vec{V}$ space is updated in line 8. The procedure for $\vec{S} \times \vec{V}$ space update is presented in Figure 4.8. The basic idea of this update is to let the task being processed to inherit the pointers to the S-tasks from its, more valuable immediate successors. For example, in Figure 4.8, line 8.4, a task T_j inherits T_i's S-tasks by performing a *logical or* operation on the corresponding $\vec{S}$ vectors.

To determine whether the immediate successor is more valuable than task T_j, the fresh reflective parameters must be computed from the current $\vec{S} \times \vec{V}$ space. To do this, first, the union of all task that contribute to the R-params of task T_j is obtained using the following formula:

$$\vec{\Gamma_j} = \sum_{i=0}^{n-1} s_j^i \vec{V_i} \tag{4.1}$$

Second, the initial parameters of each contributing task are added up to get the fresh R-params. Specifically, to determine the reflective values and reflective processing times the following formulas are used:

$$w'_j = \sum_{i=0}^{n-1} \gamma_j^i w_i \tag{4.2}$$

$$p'_j = \sum_{i=0}^{n-1} \gamma_j^i p_i \tag{4.3}$$

When all the tasks in a graph are processed, the $\vec{S} \times \vec{V}$ space is used to determine the reflective parameters of source tasks only, the tasks initially eligible for scheduling. As the tasks are scheduled, the $\vec{S} \times \vec{V}$ space is updated by simply

removing the $\vec{S}_j$ vector of the newly scheduled task T_j. This update enables the accurate calculation of the reflective parameters for the remaining tasks, the already scheduled task must not contribute its I-params to the remained unscheduled tasks.

At this point, it is important to notice that the best results and the most accurate reflective values are obtained if the $\vec{S} \times \vec{V}$ space is continuously updated during the on-line scheduling procedure. However, the continuous updates and recalculations tend to increase the complexity of the pseudo independent task schedulers. In order to maintain their low computational complexity, we suggest the use of a simple off-line sequencing procedure, such as the Smith's rule, to provide R params that are ready to use by the on-line schedulers. Such obtained R-params are just approximations of the actual values. In Chapter 5, we analyze the effectiveness of this

1. Set diagonal elements of the $\vec{S} \times \vec{V}$ space to 1, and non-diagonal to 0.
2. Include all terminal tasks into R'_L list in a non-decreasing R-ratio order.
3. While R'_L is not an empty set do
4. $\quad R_L = R'_L$ (* copy the auxiliary list *)
5. $\quad R'_L = \{\emptyset\}$ (* get ready to process a next set *)
6. $\quad$ While R_L is not an empty set do
7. $\quad\quad$ Get task T_j such that $\rho'_j = \min_{T_i \in R_L}\{\rho'_i\}$ (* get the most valuable task from R_L *)
8. $\quad\quad$ Update $\vec{S} \times \vec{V}$ space
9. $\quad\quad$ Insert all enabled T_j's immediate predecessors into R'_L
10. $\quad\quad$ Remove T_j from the R_L

Figure 4.7 VDP-G: The Main Body.

8.1. If T_j is a terminal task then RETURN, else continue.

8.2. Until all immediate successors of T_j are processed do:

8.3. Get T_j's unprocessed immediate successor T_i with the smallest ρ_i'.

8.4. If $\rho_j' > \rho_i'$ set $\vec{S}_j = \vec{S}_j + \vec{S}_i$. (* improve T_j's R-params *)

Figure 4.8 VDP-G: Update $\vec{S} \times \vec{V}$ Space Procedure.

simpler approach that does not increase the computational complexity. This is a very important feature for complex real time systems, where the imposed constraints, such as timing, resource, and other constraints, already force a scheduling algorithm to have a high computational complexity.

In conclusion, the main advantage of providing the $\vec{S} \times \vec{V}$ space is that not only the sequence of tasks within a task group is obtainable but also the measures of how valuable individual tasks are—a feature very important when scheduling dynamically arrived tasks and task groups. Utilizing this information, the on-line schedulers can make more intelligent decisions when performing multicriteria scheduling subject to multiple constraints, such as timing, resource, and precedence constraints in a dynamic non-deterministic multiprocessor environment.

4.6.3 Three Illustrative Examples

To illustrate the above presented algorithm and to demonstrate its applicability, three examples that cover a very wide spectrum of possible task groups are presented next. Specifically, the VDP-G algorithm is applied to out-tree, in-tree, and to a task group with arbitrary precedence constraints.

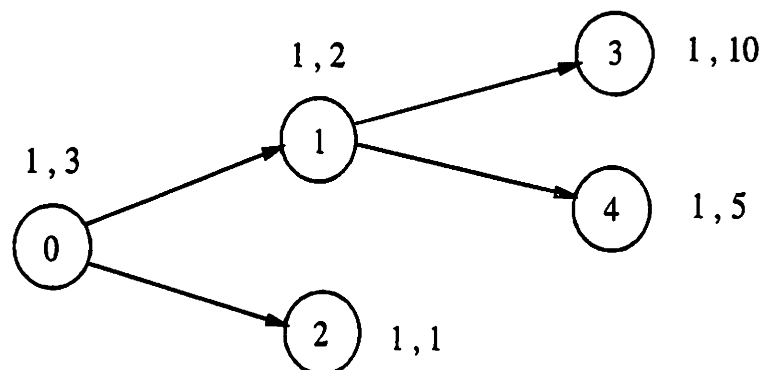

Figure 4.9 A Task Group with Out-Tree Precedence Constraints.

4.6.3.1 The Out-Tree Example

In this example, a task group is described using an out-tree precedence constraints. Figure 4.9 presents the task group structure and specifies the I-params as ordered (p_j, w_j) pairs.

The following is a derivation of $\vec{S} \times \vec{V}$ space for this example.

- Initially all diagonal elements of the $\vec{S} \times \vec{V}$ space are set to 1, and all non-diagonal elements are set to 0.

- All terminal tasks are inserted in R'_L in a non-decreasing order of their I-ratios. $R'_L = \{T_3, T_4, T_2\}$.

- $R'_L \neq \{\emptyset\}$, thus, proceed with the algorithm.

- Create a working list, $R_L = R'_L = \{T_3, T_4, T_2\}$.

- Empty the auxiliary list, that is, prepare it for the processing of tasks that will be enabled after processing the tasks in a current working list.

- Since all the tasks in a working list are the terminal tasks, their R-params are equal to their I-params; and since all diagonal elements of the $\vec{S} \times \vec{V}$ space are already set to 1, the processing of a current working list will make no changes to the $\vec{S} \times \vec{V}$ space. All enabled tasks are simply added to R'_L list. In this case, $R'_L = \{T_1\}$.

- Since R'_L is not empty, continue.
- Create a working list: $R_L = R'_L = \{T_1\}$. Empty the auxiliary list.
- Get the most valuable task from the working list, the task T_1.
- T_1 has two immediate successors, T_3 and T_4. Check whether T_1's R-ratio, initially equal to its I-ratio, should be improved by its immediate successors. To check this, compute fresh ρ'_3 and ρ'_4. Since only the terminal tasks are processed up to now, the only elements set to 1 in the $\vec{S} \times \vec{V}$ space are the diagonal elements. Therefore, $w'_3 = w_3 = 10$, and $p'_3 = p_3 = 1$, resulting in $\rho'_3 = 1/10$. Similarly, for task T_4, $\rho'_4 = 1/5$.
- Get the most valuable immediate successor, task T_3, and compare its R-ratio with ρ'_1.
- Since $\rho'_1 = 1/3 > \rho'_3 = 1/10$, T_1 must inherit T_3's S-tasks. That is, $\vec{S_3}$ vector is logically or'ed with vector $\vec{S_1}$, $\vec{S_1} = \vec{S_1} + \vec{S_3}$.
- To compare T_1's R-ratio with the next immediate successor, the current R-params must be computed. A new ratio for $\rho'_1 = 2/12$, while ratio ρ'_4 remains the same (T_4 is a terminal task).
- Since $\rho'_1 < \rho'_4$, i.e., since T_4 is less valuable than T_1, no update is required.
- Table 4.2.1 presents the $\vec{S} \times \vec{V}$ space after processing the task T_1.
- After processing T_1, the last immediate successor of task T_0 is processed, and therefore, the R'_L is extended by T_0. $R'_L = \{T_0\}$.
- Next, create the working list R_L. $R_L = R'_L = \{T_0\}$.
- Prepare R'_L for the next set of enabled tasks, empty it.

Table 4.2 Derivation of $\vec{S} \times \vec{V}$ Space for Figure 4.9

Table 4.2.1: After processing $R_L = \{T_1\}$

	$\vec{V_0}$	$\vec{V_1}$	$\vec{V_2}$	$\vec{V_3}$	$\vec{V_4}$
$\vec{S_0}$	1	0	0	0	0
$\vec{S_1}$	0	1	0	1	0
$\vec{S_2}$	0	0	1	0	0
$\vec{S_3}$	0	0	0	1	0
$\vec{S_4}$	0	0	0	0	1

Table 4.2.2: After processing $R_L = \{T_0\}$

	$\vec{V_0}$	$\vec{V_1}$	$\vec{V_2}$	$\vec{V_3}$	$\vec{V_4}$
$\vec{S_0}$	1	0	0	1	0
$\vec{S_1}$	0	1	0	1	0
$\vec{S_2}$	0	0	1	0	0
$\vec{S_3}$	0	0	0	1	0
$\vec{S_4}$	0	0	0	0	1

- Get the most valuable task from the working list, the task T_0.
- As in T_1's case, compute R-params of all immediate successors for task T_0, and sort them in a non-decreasing order of their R-ratios. Use the $\vec{S} \times \vec{V}$ space from the Table 4.2.1 and obtain the R-ratios for T_1 and T_2.
  - Since only s_1^3 is set to 1, $\Gamma_1 = \vec{V_3} = \{0, 1, 0, 1, 0\}$, and thus, the R-params for T_1 are: $w_1' = w_1 + w_3 = 12$, $p_1' = p_1 + p_3 = 2$, and $\rho_1' = 2/12$.
  - For a terminal task T_2, the R-ratio is: $\rho_2' = 1/1$.
- Compare T_0's R-ratio with the R-ratio of its most valuable successor, task T_1.
- Since $\rho_0' > \rho_1'$ (i.e., $1/3 > 2/12$), T_0 should inherit all T_1's S-tasks, that is, $\vec{S_0} = \vec{S_0} + \vec{S_1}$.
- Next, since $\rho_0' = 3/15 < \rho_2' = 1/1$ no update is required.
- The final look of $\vec{S} \times \vec{V}$ space is given in Table 4.2.2.

To assign the reflective parameters to individual tasks, a sequencing procedure similar to the one used for VDP-R algorithm, described in section 4.5, is used. This

procedure is based on the Smith's rule that sequences enabled tasks in the increasing order of their ratios—or in our case, in the increasing order of reflective ratios. If two enabled tasks have the same reflective ratios, the task with a smaller initial ratio is selected.

In contrast to the sequencing procedure for VDP-R algorithm, where the reflective parameters are assigned by the VDP-R algorithm, the sequencing procedure for the VDP-G algorithm uses the $\vec{S} \times \vec{V}$ space and the initial parameters to derive corresponding reflective parameters[3].

Specifically, this sequencing procedure calculates new reflective parameters for all enabled tasks at each sequencing step. The task with the smallest ratio is selected and sequenced; its reflective parameters are recorded and all its S-task entries are removed from the $\vec{S} \times \vec{V}$ space, that is, they are set to zero. By setting the S-tasks to zero, it is ensured that the already sequenced tasks are not contributing to the reflective parameters of tasks yet to be scheduled.

Using this sequencing procedure, the final task sequence, $\{T_0, T_1, T_3, T_4, T_2\}$, and the respective (p_i', w_i') pairs, {(3,15), (2,12), (1,10), (1,5), (1,1)}, are obtained for the out-tree example presented above.

4.6.3.2 The In-Tree Example

The example of a task group with the precedence constraints given in a form of an in-tree with corresponding I-params is presented in Figure 4.10. An interested reader can follow the algorithm, create the $\vec{S} \times \vec{V}$ space, and check the final results against the results presented in Table 4.3.2.

[3]For the calculation of reflective parameters see section 4.6.2.

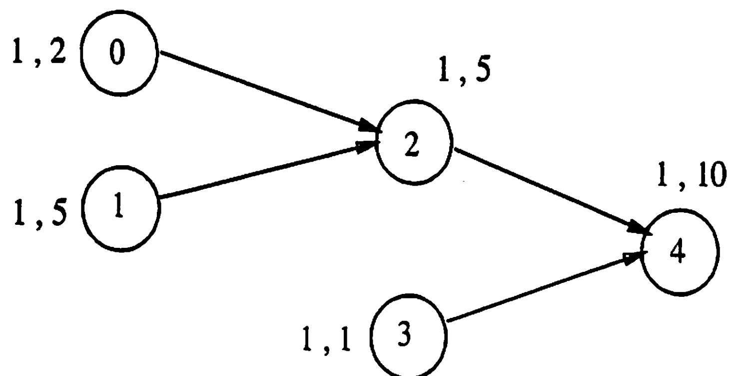

Figure 4.10 A Task Group with In-Tree Precedence Constraints.

Table 4.3 Derivation of $\vec{S} \times \vec{V}$ Space for Figure 4.10

Table 4.3.1: After processing $R_L = \{T_2, T_3\}$

	$\vec{V_0}$	$\vec{V_1}$	$\vec{V_2}$	$\vec{V_3}$	$\vec{V_4}$
$\vec{S_0}$	1	0	0	0	0
$\vec{S_1}$	0	1	0	0	0
$\vec{S_2}$	0	0	1	0	1
$\vec{S_3}$	0	0	0	1	1
$\vec{S_4}$	0	0	0	0	1

Table 4.3.2: After processing $R_L = \{T_0, T_1\}$

	$\vec{V_0}$	$\vec{V_1}$	$\vec{V_2}$	$\vec{V_3}$	$\vec{V_4}$
$\vec{S_0}$	1	0	0	0	1
$\vec{S_1}$	0	1	0	0	0
$\vec{S_2}$	0	0	1	0	1
$\vec{S_3}$	0	0	0	1	1
$\vec{S_4}$	0	0	0	0	1

Following the sequencing procedure described in the out-tree example, the task sequence $\{T_1, T_0, T_2, T_3, T_4\}$ and the respective (p_i', w_i') pairs, {(1,5), (4,18), (3,16), (2,11), (1,10)}, are obtained for this in-tree example.

4.6.3.3 The Example with Arbitrary Precedence Constraints

The example of the task group with arbitrary precedence constraints with corresponding I-params is presented in Figure 4.11. As in Section 4.6.3.2, we will not present the derivation procedure, instead, the final results of the preprocessing are given in Table 4.4.2.

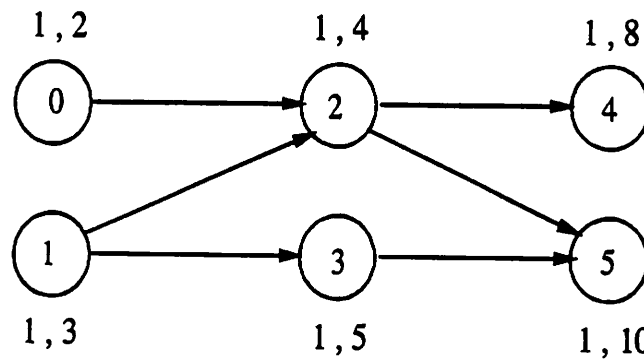

Figure 4.11 A Task Group with Arbitrary Precedence Constraints.

Table 4.4 Derivation of $\vec{S} \times \vec{V}$ Space for Figure 4.11

Table 4.4.1: After processing $R_L = \{T_2, T_3\}$

	$\vec{V_0}$	$\vec{V_1}$	$\vec{V_2}$	$\vec{V_3}$	$\vec{V_4}$	$\vec{V_5}$
$\vec{S_0}$	1	0	0	0	0	0
$\vec{S_1}$	0	1	0	0	0	0
$\vec{S_2}$	0	0	1	0	1	1
$\vec{S_3}$	0	0	0	1	0	1
$\vec{S_4}$	0	0	0	0	1	0
$\vec{S_5}$	0	0	0	0	0	1

Table 4.4.2: After processing $R_L = \{T_0, T_1\}$

	$\vec{V_0}$	$\vec{V_1}$	$\vec{V_2}$	$\vec{V_3}$	$\vec{V_4}$	$\vec{V_5}$
$\vec{S_0}$	1	0	0	0	1	1
$\vec{S_1}$	0	1	0	0	1	1
$\vec{S_2}$	0	0	1	0	1	1
$\vec{S_3}$	0	0	0	1	0	1
$\vec{S_4}$	0	0	0	0	1	0
$\vec{S_5}$	0	0	0	0	0	1

Finally, by using the same sequencing procedure as in the two previous examples, the sequence $\{T_1, T_3, T_0, T_2, T_5, T_4\}$ and the respective (p'_i, w'_i) pairs, {(6,32), (5,29), (4,24), (3,22), (1,10), (1,8)}, are obtained.

4.7 Summary

This chapter presents two off-line algorithms designed to assist on-line scheduling of task groups in dynamic non-deterministic multiprocessor real-time systems. Specifically, these algorithms preprocess task groups with rooted tree and arbitrary precedence constraints, where individual tasks are assigned different contributing values.

The issue of scheduling tasks according to their values is essential to schedulers whose objective is to maximize total accrued system value. In complex real-time systems, these schedulers are required to schedule tasks according to given timing constraints and resource constraints beside the precedence constraints and different contributing values. Due to the complexity of the problem and the requirements of the dynamic environment, one of the most important issues of on-line schedulers is efficiency. In these systems, if an overload is likely to happen, it is very important to schedule not only the right, very valuable task groups but also the right branches within each task group.

Our two preprocessing algorithms are designed to derive the parameters that, if utilized, can increase the effectiveness and the efficiency of on-line scheduling algorithms. The basic idea is to reduce the complexity of a very difficult problem of scheduling task groups to a much simpler problem of scheduling the pseudo independent tasks without loss in performance.

The first preprocessing algorithm, the VDP-R algorithm, preprocesses value densities of only rooted tree task groups. On the other hand, the second algorithm, the VDP-G algorithm, preprocesses task groups with arbitrary precedence constraints. Both algorithms are designed to derive so-called reflective parameters. These parameters integrate the information about the individual tasks and about their successors. Each task in a group is assigned an absolute measure of how valuable it is with respect to other tasks in its group. By utilizing the reflective parameters during on-line scheduling, more valuable successors are given better chances to get scheduled and executed as early as possible—thus, dealing with overloads in a more meaningful way.

In summary, the advantages of separating the task group preprocessing from on-line scheduling are:

- The complex problem of scheduling arbitrary task groups is reduced to a problem of scheduling pseudo independent tasks.
- The domain of existing independent task scheduling algorithms is extended. That is, the existing independent task scheduling algorithms can be used to schedule task groups.
- Task groups with arbitrary precedence constraints can be preprocessed.
- The use of reflective parameters provides a very effective on-line scheduling with low scheduling cost.

In general, the algorithms for scheduling of complex real-time systems can be enhanced by fully utilizing off-line preprocessing; not only to off-load the constant computation cost of on-line schedulers but to potentially reduce the problem to a much simpler one. That is, the synergism of off-line preprocessing and on-line scheduling can result in significant improvements of real-time scheduling algorithms.

In the next chapter, we analyze the effects of the integration of off-line preprocessing and on-line scheduling using the RDS algorithm from Chapter 3, Section 3.2.

CHAPTER 5

VDP-G ALGORITHM: APPLICABILITY ANALYSIS

5.1 Introduction

The value density propagation algorithm for arbitrary task groups, the VDP-G algorithm described in the previous chapter, Chapter 4, is tailored to assist on-line schedulers of complex real-time and non real-time systems. Specifically, the algorithm is aimed at systems that maximize total system value in the presence of arbitrary task groups, groups where the individual tasks contribute different values upon their completion.

The output of the VDP-G algorithm is the $\vec{S} \times \vec{V}$ space that is used to assign reflective parameters (R-params) to individual tasks within each task group. The R-params describe how valuable individual tasks are by themselves as well as relative to their successors. Most importantly, the R-params enable a system wide comparison of all tasks eligible for scheduling.

Another important characteristic is that the $\vec{S} \times \vec{V}$ space can be used to calculate R-params either off-line or on-line. Off-line assignment, such as the one presented at the end of the previous chapter, requires no increase in the computational complexity of on-line schedulers. This is a very important feature for dynamic systems with non-deterministic task and task group arrivals, where the scheduling overhead must be kept as low as possible. On the other hand, the on-line assignment of R-params, even though more accurate, increases the computational complexity, and as such, it is less adequate for scheduling complex real-time systems.

In this chapter, we analyze the effects of the off-line derived R-params on the overall system performance, under different scheduling heuristics and multiprocessor systems.

To experimentally evaluate the effects of R-params applied to a complex scheduling problem—such as the problem of scheduling arbitrary task groups with different contributing values in a multiprocessor environment—a large number of tests are required due to the large number of relevant system parameters. We tested:

- a wide range of task group laxities,
- a number of system loads (ranging from moderate and high loads to heavy overloads),
- different task group sizes,
- different number of application processors,
- three types of task groups: in-tree, out-tree, and task groups with arbitrary precedence constraints,
- three types of value assignments: (1) a top-heavy assignment, where the source tasks are the most valuable tasks and the terminal tasks are the least valuable tasks, (2) a bottom-heavy assignment, a reverse of the top-heavy assignment, and (3) a random value assignment,
- two types of time-value functions: a step value function, and a linearly diminishing value function, and finally,
- tasks are scheduled using: (1) initially assigned parameters, (2) the group parameters, and (3) off-line derived reflective parameters.

The details of the simulation model assumed in these experiments are described in Section 5.2. The simulation setup and the actual values of variables used in

simulations are presented in Section 5.3. The simulation results are analyzed in a three part section consisting of: (1) the analysis of systems with out-tree task groups, (2) the analysis of systems with in-tree task groups, and (3) the analysis of systems with arbitrary task groups. This three part analysis is presented in Section 5.4. The summary of general learnings and observed trends are presented in Section 5.5.

5.2 Model Description

The underlying system used in this chapter is very similar to the system described in Chapter 3, Section 3.2. It is a multiprocessor system with dedicated scheduling processor and multiple application processors. The dynamically arriving events are defined by their timing constraints, resource requirements, and contributing values. The arrived events are scheduled using a centralized on-line scheduler that creates feasible schedules per application processor.

The major difference between the model in this chapter and the previous model is that the model in this chapter allows the arrivals of arbitrary task groups besides the arrivals of independent tasks.

One feature of the scheduling approach presented in the previous section is that the existing scheduling algorithms can be used to schedule arbitrary task groups. To demonstrate this, we use the RDS algorithm, the algorithm that has already been proven to be a very efficient and effective algorithm, as the basic on-line scheduling algorithm. Starting from this algorithm, a more general variant, the RDS-G algorithm is developed. The major enhancements needed to be included in RDS-G algorithm in order to successfully schedule arbitrary task groups are:

- The RDS-G algorithm handles arbitrary precedence constraints by performing scheduling on the enabled set of tasks (the tasks whose predecessors have already been executed or scheduled). In a case of an infeasible schedule, the

RDS-G algorithm rejects not only the tasks that can not be scheduled but, also, all their successors.

- The heuristic function used to select a task next to scheduled, in the RDS-G algorithm, is enhanced to accommodate different types of value density parameters. Specifically, the heuristic function is designed to use either initial value densities, group value densities, or reflective value densities.

- Due to the imposed precedence constraints, the RDS-G algorithm is designed to handle tasks with future release times. This feature is inherent to all multiprocessor schedulers that deal with task groups. The earliest time a pseudo independent task can start its execution depends on the arrival time of a group, processor and resource availability, and the scheduled finish times of its immediate predecessors.

Besides the above differences, the RDS and RDS-G algorithms are identical. Both algorithms belong to the class of best effort algorithms with the objective to maximize the accrued system value.

The, in detail, description of the parameters used in simulations based on this model are presented in the next section.

5.3 Simulation Testbed Description

The simulation testbed has three major parts: (1) the task group generator, (2) the load generator, and (3) the simulator. The task group generator creates a list of off-line preprocessed task groups. The load generator creates a list of dynamic arrivals of preprocessed task groups. The output from both generators are used by the simulator that preforms on-line scheduling of dynamically arrived task groups.

5.3.1 The Task Group Generator

The off-line preprocessed task groups are required by the scheduling approach presented in the previous chapter, Chapter 4. That is, for each task group, the precedence constraints, the individual contributing values, and the computation times of its individual tasks must be known *a priori*. These parameters are required if the value density propagation algorithm, the VDP-G algorithm, is to be used.

Besides the off-line assignments of R-params, the individual tasks must be preprocessed with respect to the group timing constraints, too. This requires only that task group precedence constraints and the computation times of individual tasks are given at design time—which is just a subset of the specifications required for the derivation of R-params.

The deadline preprocessing algorithm used in this chapter is designed to handle a broad range of deadline assignments. Instead of using the fixed, off-line specified timing constraints for each task group and then preprocessing them, as in a traditional ALAP algorithm, we developed a new, *relative* deadline preprocessing algorithm that does not require deadline assignment at design time.

Our algorithm assumes that all terminal tasks have zero deadline. With this assumption, the relative deadline propagation algorithm derives the corresponding, negative deadlines for all other, non-terminal tasks. At arrival time, such obtained relative deadlines are simply added to the absolute task group deadline to obtain the absolute deadlines for individual tasks.

The advantage of assigning the relative deadlines at the preprocessing stage is that the absolute deadlines can be assigned based of the system state and other requirements known only at the invocation time. In addition to this, if the same group is allowed to have different deadlines at different invocations—a likely scenario—the basic group needs to be processed only once. In general, the applicability range of relative deadline preprocessing is wider and more practical.

Table 5.1 lists the parameters required and parameters generated by the pre-processing procedure.

Table 5.1 Task Group Generator Parameters.

1.	Precedence Constraints	a. In-Tree b. Out-Tree c. Arbitrary
2.	Computation Times	a. Exponentially Distributed
3.	Contributing Values	a. Uniformly Distributed b. Exponentially Increasing c. Exponentially Decreasing
4.	Derived Parameters	a. Reflective Contributing Values b. Reflective Computation Times c. Relative Deadlines

5.3.2 The Load Generator

The load generator creates an event list of task group arrivals. Each event is assigned its arrival time, identification number of the group it represents, and laxity. An absolute group deadline is derived as a sum of group's arrival time, group's longest path, and its laxity. (Note that the task computation times and the resource requirements are assigned by the task group generator.) The list of the parameters used by the load generator, specific for the simulations discussed in this chapter are given in Table 5.2.

5.3.3 The Simulator

The simulator uses the fixed list of preprocessed task groups and the generated event list to simulate the dynamic multiprocessor system with on-line scheduling

Table 5.2 Load Generator Parameters.

1.	System Processors	a. One
2.	Application Processors	a. Two b. Five
3.	Group Arrival Times	a. Exponentially Distributed
4.	Group Invocations	a. Uniformly Distributed
5.	Group Laxities	a. Exponentially Distributed

of arbitrary task groups. Specifically, the simulator handles scheduling of non-atomic task groups made up of non-preemptive tasks—the tasks defined with their worst case computation times, contributing values, and resource requirements. The scheduling is performed in a multiprocessor environment with two and five application processors. For each set of parameters listed in Table 5.3, one of the three heuristic functions is used to create a feasible schedule. Specifically, the heuristic that combines either task's initial value density, group value density or reflective value density with task's deadline and processors' availability is used to select the best task to schedule next.

The analysis of the results obtained from the simulations performed using the parameters from Table 5.3 are presented in the next section.

5.4 The Analysis

The analysis of overall system performance of multiprocessor real-time systems with dynamic arrivals of three types of task groups is presented in this section. Specifically, this section covers: (1) the analysis of systems with out-tree task groups,

Table 5.3 Simulator Parameters.

No.	Parameter	Value
1.	Group Type	a. Non-atomic
2.	Task Type	a. Non-preemptive
3.	On-line Scheduling Algorithm	a. RDS-G
4.	Preprocessing Algorithms	a. VDP-G, and ALAP
5.	Number of Preprocessed Groups	a. 20
6.	Number of Group Arrivals	a. 900
7.	Group Size	a. 5 to 15 b. 20 to 30
8.	System Load	a. 0.7, 0.9, 1.2, 1.5, 2.0
9.	Laxity in ES of a Single Task	a. 1/8, 1/4, 1/2, 1, 2, 4
10.	Linearly Diminishing Values	a. From 100% to 50%
11.	Step Values	a. 100% of Assigned Value
12.	Uniformly Distributed Values	a. 10 to 100
13.	Exponentially Increasing Values	a. 1.8% to 100% of Max. Value
14.	Exponentially Decreasing Values	a. 100% to 1.8% of Max. Value
15.	Schedulable Resources	a. Five
16.	Resource Use Probability	a. Fixed to 0.3
17.	Exclusive Use Probability	a. Fixed to 0.5

(2) the analysis of of systems with in-tree task groups, and (3) the analysis of systems with arbitrary task groups.

5.4.1 Analysis of Out-Tree Task Groups

The overall system performance of complex multiprocessor real-time systems, where task groups with out-tree precedence constraints are scheduled dynamically is discussed in this section. This, the most comprehensive analysis presented in this chapter, covers a wide range of different systems. The covered case studies are:

1. The analysis of five processor systems with task groups composed of 5 to 15 tasks, where individual tasks are assigned random contributing values according to a uniform distribution.

2. The analysis of systems that differ from systems in case study (1) in the assignment of contributing values only. That is, this study deals with systems where contributing values are assigned in:

   (a) linearly decreasing order, producing so-called *top-heavy* task groups, and

   (b) linearly increasing order, producing so-called *bottom-heavy* task groups.

3. The analysis of systems with large task groups that are made up of 20 to 30 tasks. In this case all other parameters are the same as in case study (1).

4. The analysis of systems where the randomly assigned contributing values diminish with time. In this system, a task value linearly decreases with time, from 100% to 50% of the initially assigned value. The 100% value is contributed to the system if a task finishes at its earliest possible time, assuming that there are no resource and no processor conflicts; while the contribution of the 50% of its initially assigned value is given to the system if a task finishes at its latest possible time, assuming given timing constraints. All other parameters are kept the same as in case study (1).

5. The analysis of two processor systems, where all other parameters are kept the same as in case study (1) that analyzes a 5 processor systems.

The following sections present the analysis of the these case studies in the order they are listed above.

5.4.1.1 Groups with Random Values

The analyzed systems are five processor systems with task groups made up of 5 to 15 tasks. The computation times of tasks within a given group are exponentially distributed; their contributing values are uniformly distributed; and the resources needed are assigned according to the given use and exclusive/shared probabilities. Furthermore, the initially assigned contributing values remain constant over the time. That is, as long as they complete within their timing constraints, the contributed values are the same as the initially assigned ones.

The on-line scheduling algorithm is the RDS-G algorithm that supports three different heuristic functions. All three functions integrate task's deadline, its earliest possible start time, and one of three forms of its value density:

- the value density obtained using the *initial* parameters,
- the value density obtained using the *group* parameters, or
- the value density obtained using the *reflective* parameters.

Depending on the choice of value density parameters, the systems are labeled as the systems that use either I-heuristic, G-heuristic, or R-heuristic, respectively.

In this system, as well as in all systems studied in this chapter, the overall system performance is presented as the value loss ratio plotted against the laxities. The tested laxity range is from 1/8 to 4 times the expected service time of the individual tasks. To obtain an absolute task group deadline, the laxity, which is

assigned using the exponential distribution, is added to group's arrival time and to group's longest execution path. Furthermore, all analyzed systems are tested under five different system loads, ranging from moderate loads of $\rho = 0.7$ through heavy loads of $\rho = 0.9$ to heavy overloads of $\rho = 2.0$. The plotted simulated points are calculated with 95% confidence, and the obtained confidence intervals are indicated with a vertical bar.

For the this case study, the performance graphs for systems with I-heuristic, G-heuristic, and R-heuristic are plotted in Figure 5.1, Figure 5.2, and Figure 5.3, respectively. It is evident from these graphs that systems with reflective parameters provide the best performance of all. In contrast, systems with group parameters have the highest value loss, especially pronounced under overloads.

The specific differences in performance between the tested systems are tabulated as rounded values in Table 5.4 and Table 5.5. Table 5.4 lists the relative differences between systems that use I-heuristic and systems that use G-heuristic. Table 5.5 lists the relative differences between systems that use R-heuristic and systems that use G-heuristic. Besides indicating the difference in performance, both tables indicate the overall performance trend, as well as the laxity range for which the given differences and trends are valid. (Note that the difference in performance is given in percentages obtained by subtracting the value loss of the system with G-heuristic from the value loss of the system with I-heuristic. Therefore, if the first system outperforms the second system, the difference is positive; and if the first system has higher value loss than the second system, the difference is negative.)

It can be observed from Table 5.4 that under overloads, systems with I-heuristic outperform systems with G-heuristic. Specifically, the I-heuristic outperforms the G-heuristic by up to 6%, 6%, and 9%, for loads of 1.2, 1.5, and 2.0, respectively. Under lower loads, systems with I-heuristic perform the same, or just slightly better than systems with G-heuristic.

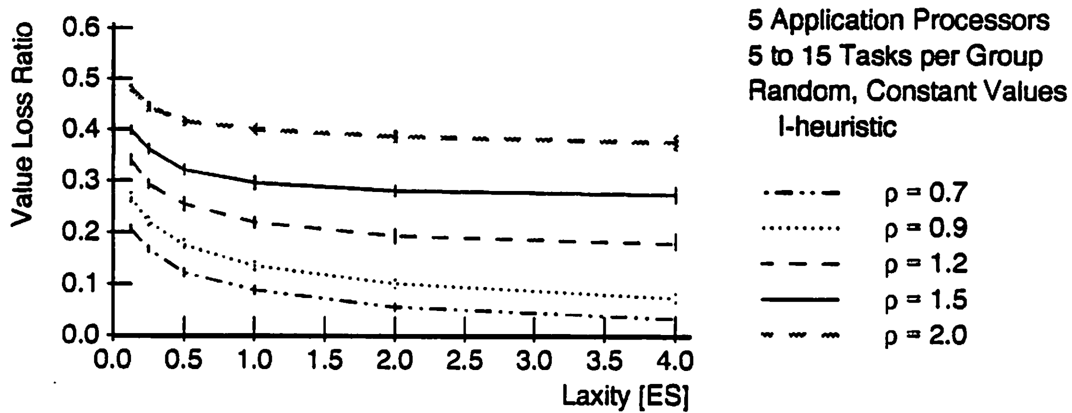

Figure 5.1 Value Loss Ratios for Groups with Random Values, I-heuristic.

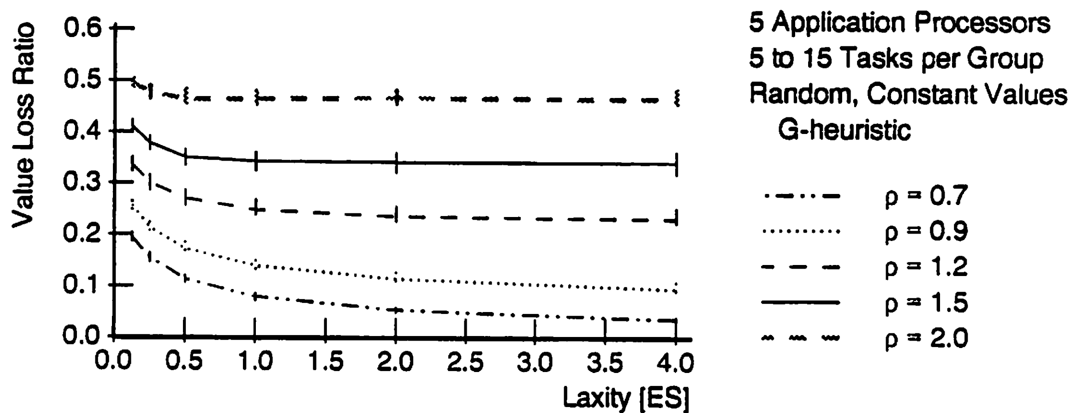

Figure 5.2 Value Loss Ratios for Groups with Random Values, G-heuristic.

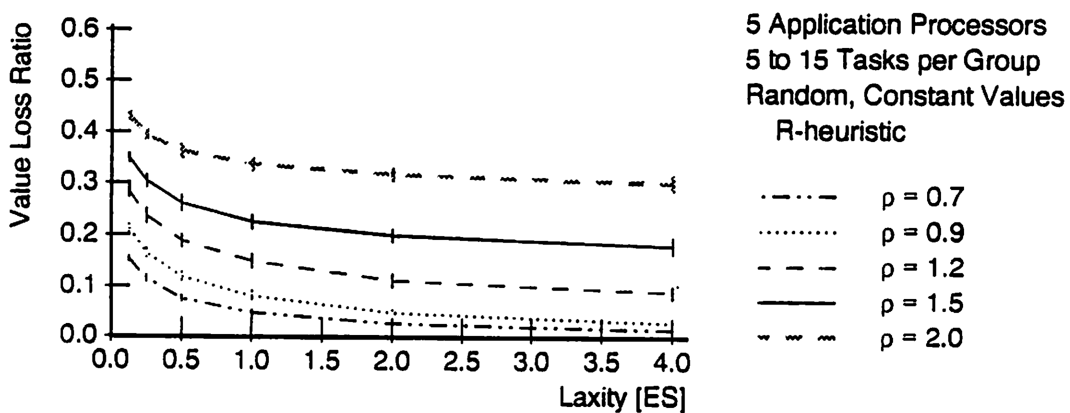

Figure 5.3 Value Loss Ratios for Groups with Random Values, R-heuristic.

Table 5.4 Groups with Random Values: I-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	Both are Comparable		I-heuristic is Better		
Difference [%]	0	-1 to 2	-1 to 6	2 to 6	1 to 9
Laxity Range	All	Low-Hi	Low-Hi	Low-Hi	Low-Hi

Table 5.5 Groups with Random Values: R-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	R-heuristic is Better				
Difference [%]	3	4 to 7	5 to 15	7 to 15	6 to 16
Laxity Range	All	Low-Hi	Low-Hi	Low-Hi	Low-Hi

On the other hand, Table 5.5 shows that systems with R-heuristic outperform systems with G-heuristic by a significantly larger margin under all loads. The difference in performance is especially pronounced under overloads, where systems that use reflective value densities perform up to 15% to 16% better than systems that use group value densities.

By comparing the performance of systems with R-heuristic and systems with I-heuristic, it is clear that systems with reflective value densities perform significantly better. Under overloads, the difference is up to 7%; while under moderate and heavy loads, the difference is about 3% in favor of the R-heuristic.

5.4.1.2 *Groups with Linear Value Assignment*

Let's now look at the system performance under two extreme assignments of contributing values. First, let's look at the system where contributing values are

assigned in a decreasing order, that is, where the source tasks are assigned the highest contributing values, and the terminal tasks are assigned the lowest contributing values. These task groups are so-called top-heavy task groups. In these groups, the contributing values are assigned using the exponential function to assign the values in a range of 100% to 1.8%, i.e., from e^0 to e^{-4} times the highest contributing value within a given group. The difference in value between two consecutive tasks is determined by this interval and by the number of tasks in the group. (For example, if a group has n tasks, the first task, i.e. one of the source tasks, will be assigned contributing value of $e^{(-4+\frac{4}{n}\times n)} \times value$, and the next task, determined on a level by level basis, will be assigned contributing value of $e^{(-4+\frac{4}{n}\times(n-1))} \times value$. Where *value* is a parameter randomly assigned to each group.)

Besides top-heavy task groups, we also look at systems where contributing values are assigned in the reverse order. That is, we analyze the performance of systems with so-called bottom-heavy task groups. In bottom-heavy task groups, the terminal tasks are assigned the largest contributing values, and the source tasks are assigned the lowest contributing values.

These two cases are chosen for the following reasons:

- The top-heavy task groups should, intuitively, favor the I-heuristic because no task in a given group has more valuable successors than the task itself.

- The bottom-heavy task groups should, on the other hand, favor the use of G-heuristic, because a group delivers the highest value only if the terminal tasks are executed; furthermore, this group type does not favor early severing of task groups because of the potentially high value loss. (Remember that the contributing values are assigned using the exponential distribution, ranging from 100% to 1.8% of the largest value within a group.)

Regarding the R-heuristic, the performance results for these two extreme cases should indicate whether systems with R-heuristic are applicable to a wide range of out-tree task groups or not.

First, let's analyze the systems with top-heavy task groups. Figure 5.4, Figure 5.5, and Figure 5.6 present the value loss ratios for systems with I-heuristic, G-heuristic, and R-heuristic, respectively.

The differences in performance, the overall trend, and the laxity range are summarized in Table 5.6 and Table 5.7. Table 5.6 compares results for systems with I-heuristic and systems with G-heuristic. Table 5.7, on the other hand, compares the results for systems with R-heuristic and systems with G-heuristic.

From the first table, it is clear that systems that use initial value densities are up to about 4% better than systems that use group value densities. However, this trend holds only under overloads. Under moderate and heavy loads, both systems perform approximately the same.

From the second table, almost identical difference in performance between systems that use R-heuristic and systems that use G-heuristic can be observed. In this case, systems with R-heuristic outperform systems with G-heuristic by up to 3%, but only under overloads. Under moderate and heavy loads they behave almost the same.

As expected, these results that for top-heavy task groups show that the systems with R-heuristic deliver almost the same performance as the best performing I-heuristic or G-heuristic.

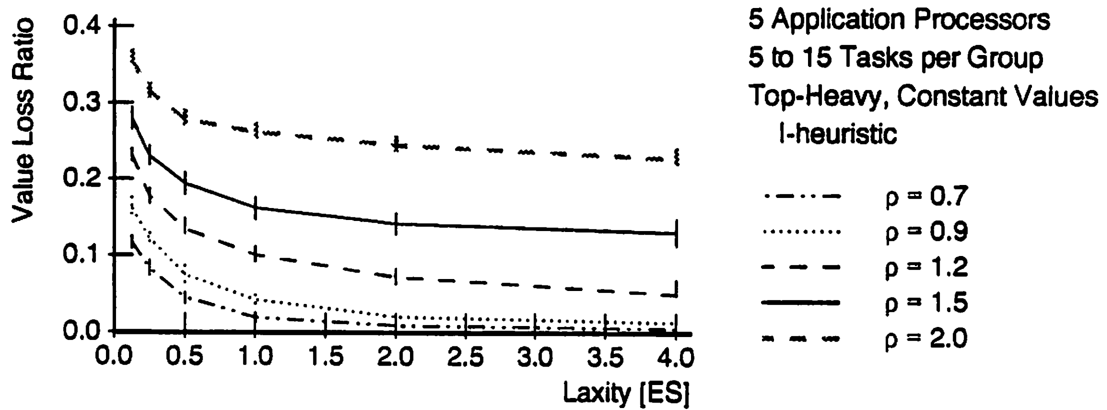

Figure 5.4 Value Loss Ratios for Groups with Top-Heavy Values, I-heuristic.

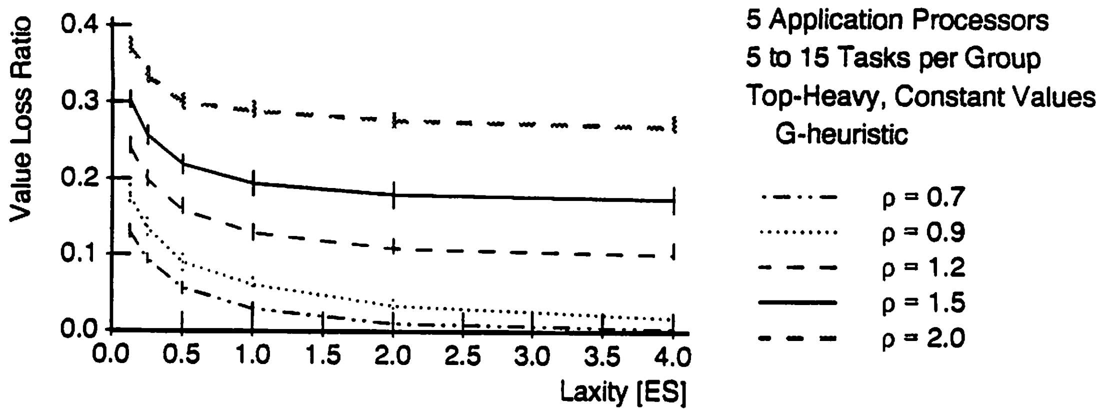

Figure 5.5 Value Loss Ratios for Groups with Top-Heavy Values, G-heuristic.

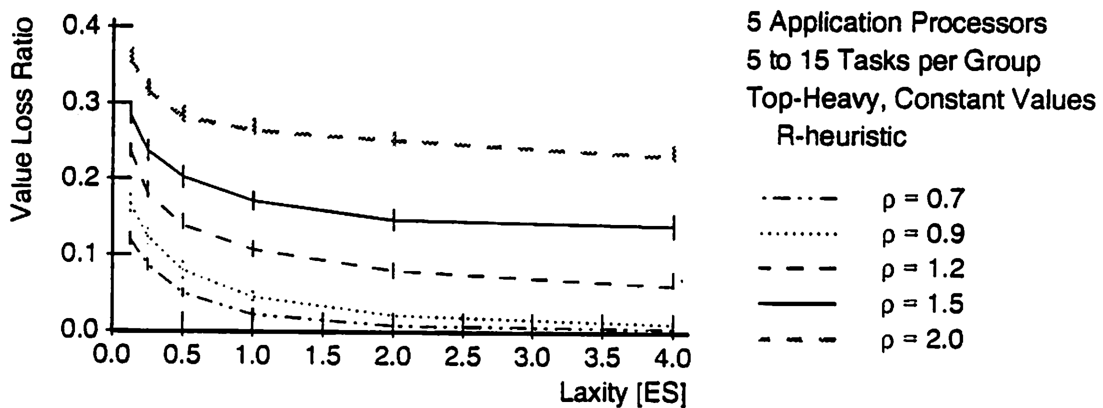

Figure 5.6 Value Loss Ratios for Groups with Top-Heavy Values, R-heuristic.

Table 5.6 Groups with Top-Heavy Values: I-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	Both are Comparable		I-heuristic is Better		
Difference [%]	2 to 0	1 to 0	1 to 5	2 to 4	2 to 4
Laxity Range	Low-Hi	Low-Hi	Low-Hi	Low-Hi	Low-Hi

Table 5.7 Groups with Top-Heavy Values: R-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	Both are Comparable		R-heuristic is Better		
Difference [%]	1 to 0	1 to 0	1 to 4	2 to 3	2 to 3
Laxity Range	Low-Hi	Low-Hi	Low-Hi	Low-Hi	Low-Hi

Consider bottom-heavy task groups. Figure 5.7, Figure 5.8, and Figure 5.9 plot the value loss ratios for systems that use I-heuristic, G-heuristic, and R-heuristic, respectively.

The differences in performance between the systems with I-, G-, and R-heuristic are summarized in two tables. Table 5.8 compares systems with I-heuristic to systems with G-heuristic; and Table 5.9 compares systems with R-heuristic to systems with G-heuristic. (The actual value loss ratios are plotted in Figure 5.7, Figure 5.8, and Figure 5.9 for respective systems with I-, G-, and R-heuristic.)

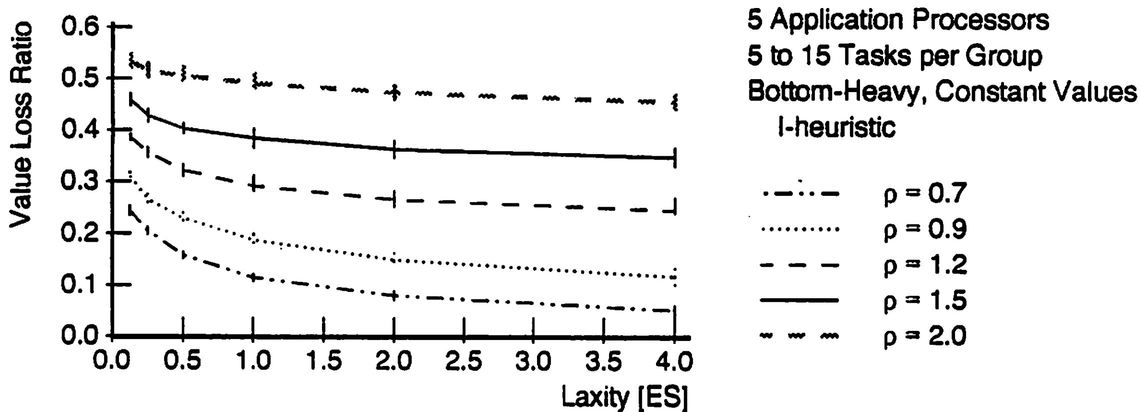

Figure 5.7 Value Loss Ratios for Groups with Bottom-Heavy Values, I-heuristic.

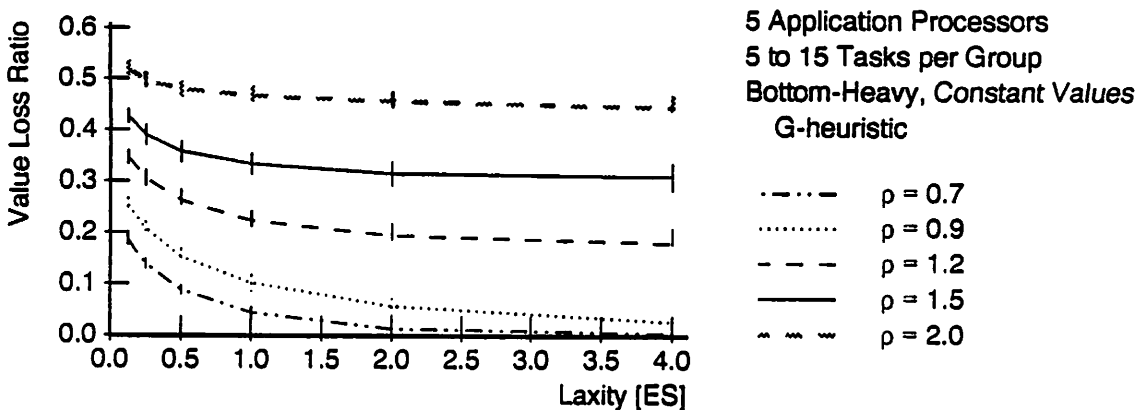

Figure 5.8 Value Loss Ratios for Groups with Bottom-Heavy Values, G-heuristic.

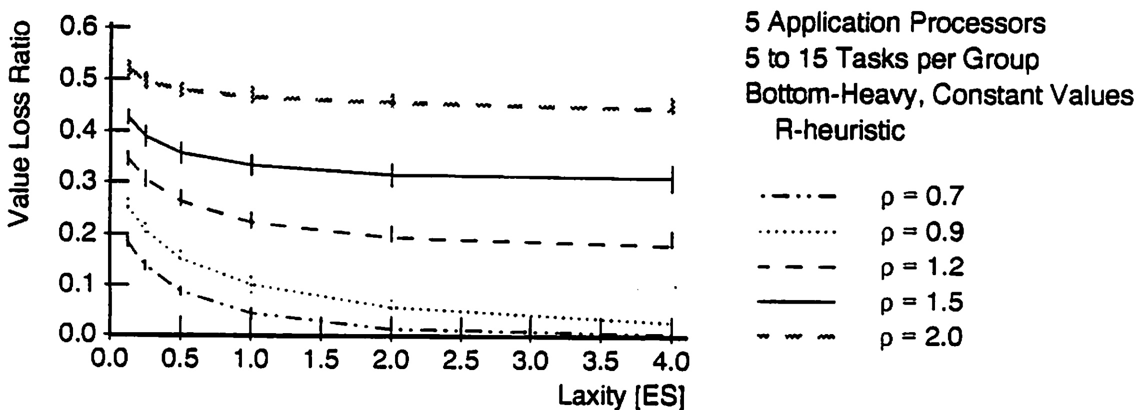

Figure 5.9 Value Loss Ratios for Groups with Bottom-Heavy Values, R-heuristic.

Table 5.8 Groups with Bottom-Heavy Values: I-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	G-heuristic is Better				
Difference [%]	-6 to -7	-6 to -9	-4 to -7	-3 to -5	-2
Laxity Range	All	Low-Hi	Low-Hi	Low-Hi	All

Table 5.9 Groups with Bottom-Heavy Values: R-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	Both are Comparable				
Laxity Range	All				

From both tables, it can be observed that systems with R-heuristic, just as in the case of top-heavy task groups, perform as good as the best performing system with either I- or G-heuristic. Specifically, Table 5.8 shows that systems with G-heuristic perform significantly better than systems with I-heuristic. (Remember, the difference in performance is given in percentages obtained by subtracting the value loss of the system with G-heuristic from the value loss of the system with I-heuristic. Therefore, if the first system outperforms the second system, the difference is positive; and if the first system has higher value loss than the second system, the difference is negative.)

Furthermore, Table 5.9 indicates that systems that use reflective value densities and systems that use group value densities perform almost identically. Thus, systems with R-heuristic perform significantly better than systems with I-heuristic.

In conclusion, the experimental results for the extreme cases, namely top-heavy and bottom-heavy task groups, indicate that systems with R-heuristic provides consistently good performance. That is, in both cases the performance of systems with R-heuristic envelopes the performance of other two systems. Thus, off-line value density determination is recommended in all cases presented above.

5.4.1.3 Large Groups with Random Values

Let's next look at systems with large task groups that are made up of 20 to 30 tasks. These task groups are on the average twice as large as the task groups analyzed in previous sections. Regarding all other parameters, this case study is the same as the case study presented in Section 5.4.1.1.

The performance difference between systems with I-heuristic and systems with G-heuristic is summarized in Table 5.10. On the other hand, Table 5.11 summarizes the performance difference for systems with R-heuristic and systems with G-heuristic.

Table 5.10 Large Groups with Random Values: I-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	G-heuristic is Better				
Difference [%]	-7 to -5	-6	-6 to -4	-4 to -2	-3 to 1
Laxity Range	Low-Hi	All	Low-Hi	Low-Hi	Low-Hi

From the first table, it is clear that for large task groups the systems with G-heuristic outperform the systems with I-heuristic under all tested loads and laxities. This result is the reverse of the results obtained in a similar system with smaller task groups, where systems with I-heuristic perform better but only under overloads.

Table 5.11 Large Groups with Random Values: R-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	Both are Comparable		R-heuristic is Better		
Difference [%]	2 to -1	3 to 0	3 to 6	5 to 9	5 to 11
Laxity Range	Low-Hi	Low-Hi	Low-Hi	Low-Hi	Low-Hi

The fluctuation observed is roughly about ±5% in both cases. On the other hand, the systems with R-heuristic perform consistently better even in the case of large task groups. For example, the difference in performance, presented in Table 5.11, ranges up to 11% under heavy overloads. Under moderate and heavy loads, the systems with R-heuristic provide almost the same performance as the systems with G-heuristic.

Clearly, the performance difference between the systems with R-heuristic and the systems with I-heuristic is even greater in this case than in the case of smaller task groups. This indicates that the systems with the R-heuristic is the most stable system of all three, while the I-heuristic fluctuates the most.

5.4.1.4 Groups with Diminishing Contributing Values

To measure how early the most valuable tasks are scheduled, we analyze the performance of systems where tasks' contributing values linearly diminish with time. In these systems, if a task completes at its earliest possible time, assuming an ideal case with no resource and no processor conflicts, the 100% of initially assigned value is contributed to the system. At the other extreme, if the task finishes at its latest possible time, only 50% of its initial value is contributed to the system. In this case study, all other parameters are kept the same as in case study (1), Section 5.4.1.1.

Table 5.12 shows the difference in performance between systems with I-heuristic and systems with G-heuristic. The I-heuristic performs better up to 9% than the systems with G-heuristic. This is an expected behavior since the systems with I-heuristic favor the current most valuable task. By favoring the most valuable task, the delays in executing very valuable tasks are shortened. The result is reflected in a larger accrued system value.

By comparing systems with R-heuristic and systems with G-heuristic, Table 5.13, it is observed that the systems with R-heuristic perform better. Under overloads, for example, the systems with R-heuristic outperform systems with G-heuristic by up to 13%, and under moderate and heavy loads the difference between 5% to 9% in favor of systems with R-heuristic.

Table 5.12 Groups with Diminishing Values: I-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	I-heuristic is Better				
Difference [%]	1	1 to 4	2 to 7	4 to 8	4 to 9
Laxity Range	All	Low-Hi	Low-Hi	Low-Hi	Low-Hi

Table 5.13 Groups with Diminishing Values: R-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	R-heuristic is Better				
Difference [%]	5	5 to 9	4 to 13	8 to 13	8 to 13
Laxity Range	All	Low-Hi	Low-Hi	Low-Hi	Low-Hi

Combining the results from both tables, it is clear that if the fast response to very valuable tasks is required—that is, if the contributing values are diminishing linearly with time—the R-heuristic is recommended; it outperforms both the G-heuristic and the I-heuristic.

5.4.1.5 Groups with Random Values: A Two Processor System

Overall performance of multiprocessor systems is affected by the number of available application processors. The smaller the number, the less parallelism can be obtained, and thus, the worse the overall performance is. This is especially notable when the laxities are short.

The analysis of systems with only two application processors is presented in this section. The value loss ratios for systems with I-, G-, and R-heuristic are plotted in Figure 5.10, Figure 5.11, and Figure 5.12, respectively. Note that the value loss ratios in all three systems are significantly larger than in the similar 5 processor system in Section 5.4.1.1.

In spite of the overall performance difference, the smaller number of processors does not have any significant impact on the earlier observed performance trends for systems with I-, G-, and R-heuristic. Table 5.14 shows the difference in performance between systems with I-heuristic and systems with G-heuristic. Besides the slight variation in performance, both systems perform almost the same.

On the other hand, Table 5.15 shows the difference in performance between systems with R-heuristic and systems with G-heuristic. This table indicates that R-heuristic performs significantly better than other two systems. The largest difference is obtained under overloads, and it ranges from 4% for low laxities up to 10% for large laxities.

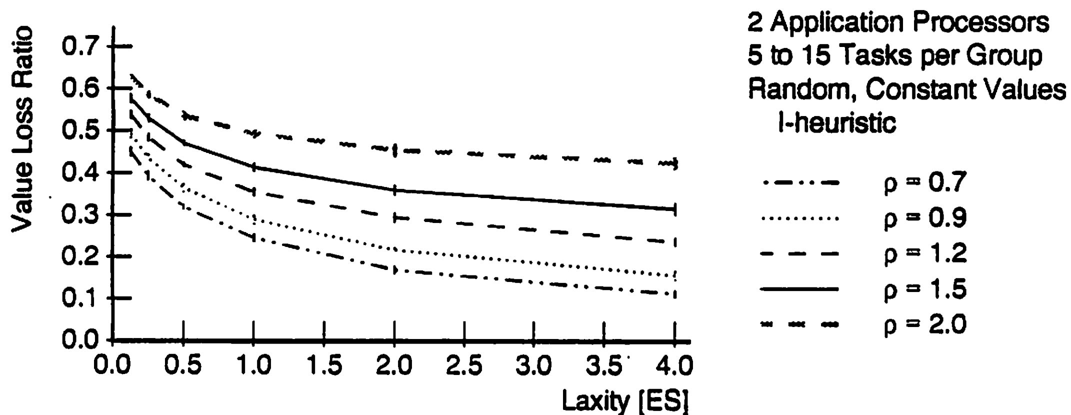

Figure 5.10 Value Loss Ratios for Two Processor Systems, I-heuristic.

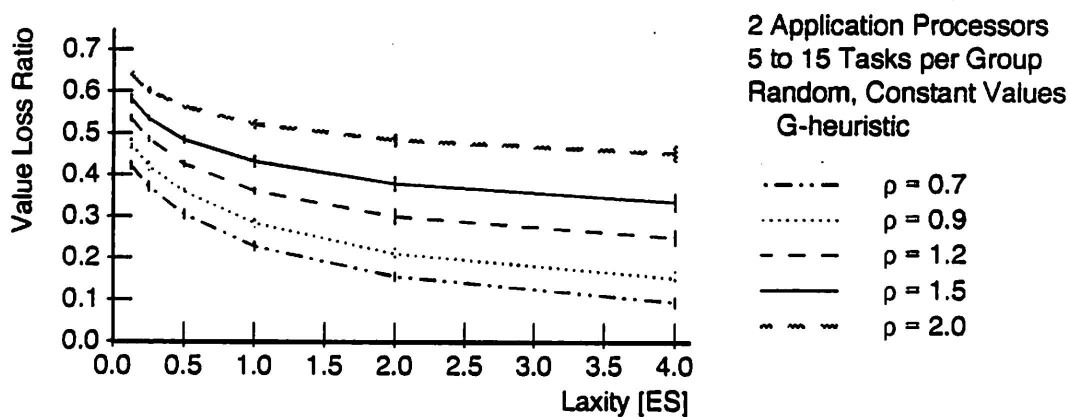

Figure 5.11 Value Loss Ratios for Two Processor Systems, G-heuristic.

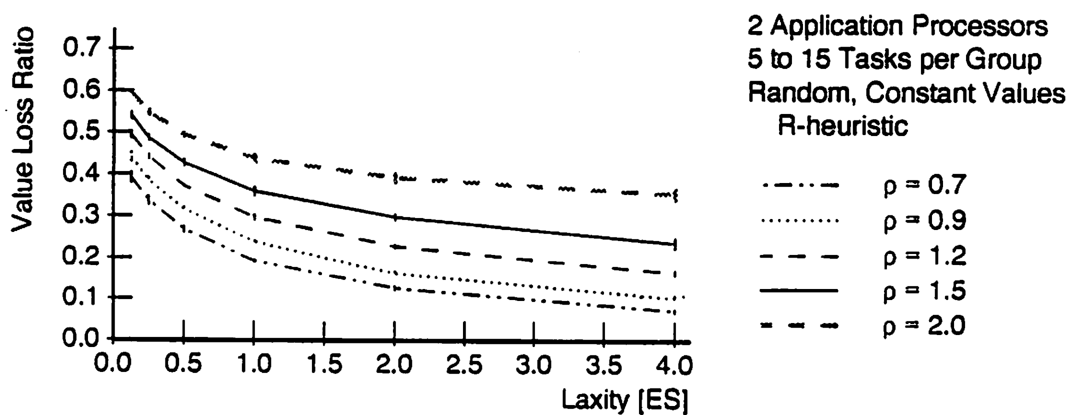

Figure 5.12 Value Loss Ratios for Two Processor Systems, R-heuristic.

Table 5.14 Two Processor Systems: I-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	G-heuristic is Better		I-heuristic is Better		
Difference [%]	-2	-2 to 0	0 to 1	0 to 2	0 to 3
Laxity Range	All	Low-Hi	Low-Hi	Low-Hi	Low-Hi

Table 5.15 Two Processor Systems: R-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	R-heuristic is Better				
Difference [%]	3	3 to 5	4 to 8	4 to 10	4 to 10
Laxity Range	All	Low-Hi	Low-Hi	Low-Hi	Low-Hi

Therefore, even in the case of small multiprocessor systems, the on-line schedulers should use R-heuristic for the best performance.

With the analysis of two processor systems, we conclude the analysis of systems with out-tree task groups. From the presented analysis, it is clear that systems with reflective parameters, i.e., systems with R-heuristic provide consistently high performance in all cases studied.

In the next section, the in-tree task groups are analyzed. This analysis is then followed by the analysis of the task groups with arbitrary precedence constraints.

5.4.2 Analysis of In-Tree Task Groups

The task groups analyzed in this section are the task groups with in-tree precedence constraints. This analysis focuses only on two cases:

1. The analysis of five processor system with task groups made up of 5 to 15 tasks. The initially assigned contributing values remain fixed within given timing constraints. These values are assigned using the uniform distribution.

2. The analysis of systems the same as above but with linearly diminishing contributing values. If a task completes at its earliest possible time, the contributed value is the same as its initial value; while if a task completes at its latest possible time, the contributed value is only 50% of its initial value.

5.4.2.1 *Groups with Random Values*

The differences in performance between the systems with I-heuristic and G-heuristic, and the systems with R-heuristic and G-heuristic are presented in Table 5.16 and Table 5.17, respectively.

Table 5.16 Groups with Random Values: I-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	Both are Comparable		I-heuristic is Better		
Difference [%]	-2	-2 to -1	-1 to 3	0 to 7	2 to 10
Laxity Range	All	Low-Hi	Low-Hi	Low-Hi	Low-Hi

The first table indicates that the I-heuristic preforms better under overloads; while under moderate and heavy loads, both systems perform almost the same. This result corresponds to the result from the analysis of task groups with out-tree precedence constraints.

However, the second table indicates that in the case of in-tree task groups, the R-heuristic is better only under overloads, and not under all loads like in the case of

Table 5.17 Groups with Random Values: R-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	Both are Comparable		R-heuristic is Better		
Difference [%]	-1	-2 to 0	-1 to 5	0 to 9	2 to 12
Laxity Range	All	Low-Hi	Low-Hi	Low-Hi	Low-Hi

out-tree task groups. Furthermore, by comparing the results from both tables, it is clear that the R-heuristic actually performs the almost the same as the I-heuristic for in-tree task groups.

Therefore, in the case of in-tree task groups with randomly assigned contributing values, the initial value densities provide sufficiently good performance. Thus, the task group preprocessing, in this case, is not recommended. The use of initial parameters assigned to individual tasks, i.e., the use of I-heuristic is recommended, due to its simplicity.

5.4.2.2 *Groups with Diminishing, Random Values*

To observe the trend in performance when the penalty is paid if the task is not executed as early as possible, we analyze the in-tree task groups with diminishing values.

Table 5.18 and Table 5.19 present the performance differences and overall trends. Table 5.18 compares the performance of systems with I-heuristic to the performance of systems with G-heuristic. This table indicates that the I-heuristic provides better performance under overloads. However, under moderate and heavy loads, both systems perform almost the same.

When these results are compared to the results for the out-tree task groups, no significant difference can be reported—the performance is almost the same, that is, the difference in performance is within the confidence intervals.

Table 5.18 Groups with Diminishing Values: I-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	Both are Comparable		I-heuristic is Better		
Difference [%]	0	1	2 to 6	3 to 8	4 to 11
Laxity Range	All	All	Low-Hi	Low-Hi	Low-Hi

Table 5.19 Groups with Diminishing Values: R-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	Both are Comparable		R-heuristic is Better		
Difference [%]	0	1 to 3	2 to 8	4 to 10	5 to 13
Laxity Range	All	Low-Hi	Low-Hi	Low-Hi	Low-Hi

Table 5.19 compares the performance of systems with R-heuristic and systems with G-heuristic. This table, on the other hand, shows that the R-heuristic provides better performance only under overloads. Under moderate and heavy overloads both systems perform within the confidence intervals, thus, no significant difference is observed. This is somewhat different from the observation in the case of out-tree task groups, where the R-heuristic is better over the entire range of loads.

Furthermore, when comparing the results from both tables, Table 5.18 and Table 5.19, it is evident that the R-heuristic is better than other two. However,

the difference in performance between systems with R-heuristic and systems with I-heuristic is only about 2% in favor of R-heuristic. This difference is not sufficient to recommend the use of off-line preprocessing for the case with diminishing values.

Therefore, for in-tree task groups, the recommended scheduling heuristic is I-heuristic. This heuristic provides sufficiently good performance over the entire tested range, and it is the simplest heuristic with a widest applicability range.

5.4.3 Analysis of Arbitrary Task Groups

The analysis of the applicability of VDP-G algorithm is concluded by the discussion of five processor systems with arbitrary task groups. In this section, we present:

1. The analysis of systems that schedule task groups with arbitrary precedence constraints made up of 5 to 15 tasks. The contributing values in this case study are fixed within the given timing constraints.

2. The analysis of systems with large task groups, the task groups with 20 to 30 tasks. The contributing values are fixed within the given timing constraints.

3. The analysis of systems with task groups make up of 5 to 15 tasks, where contributing values diminish to 50% of the initial values, if tasks are scheduled as late as possible.

4. The analysis of systems with large task groups made up of 20 to 30 tasks, and with diminishing contributing values.

5.4.3.1 Groups with Random Values

The overall value loss ratios for systems with I-, G-, and R-heuristic are presented in Figure 5.13, Figure 5.14, and Figure 5.15, respectively.

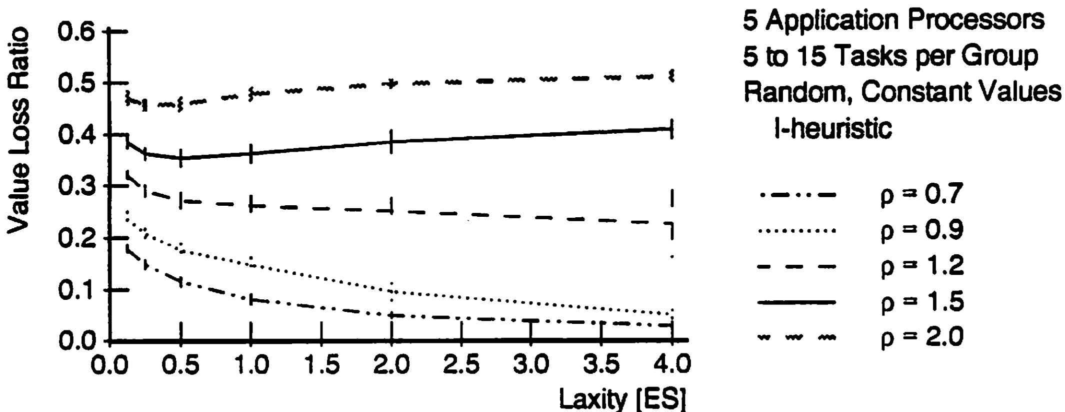

Figure 5.13 Value Loss Ratios for Groups with Random Values, I-heuristic.

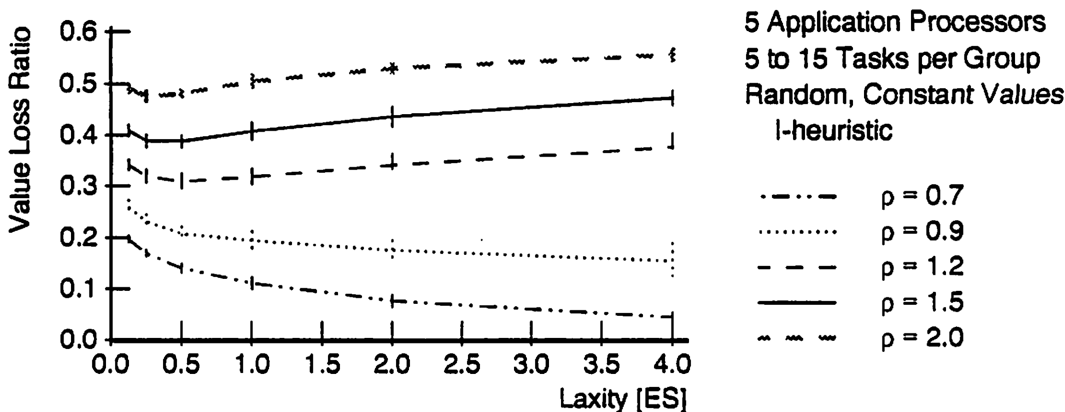

Figure 5.14 Value Loss Ratios for Groups with Random Values, G-heuristic.

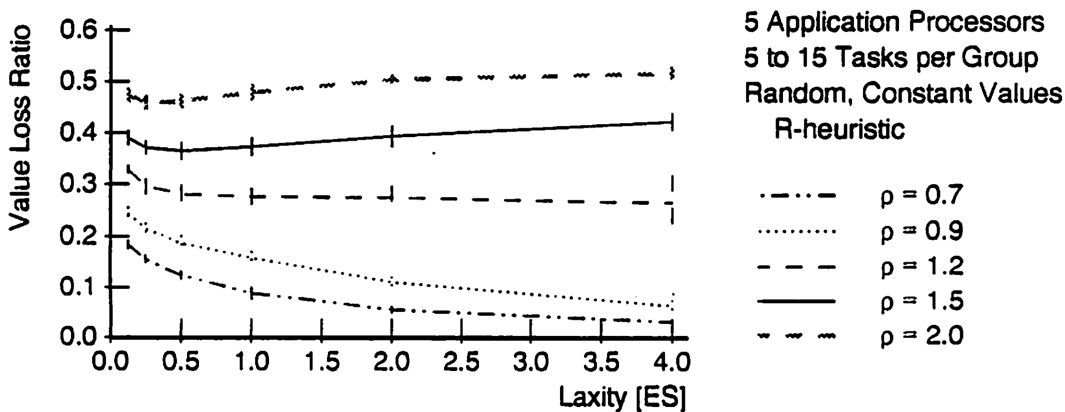

Figure 5.15 Value Loss Ratios for Groups with Random Values, R-heuristic.

These figures show that task groups with arbitrary precedence constraints perform differently from the in-tree and out-tree task groups. Under overloads, the value loss ratio of arbitrary task groups tends to increase with the increase in laxities. Corresponding in-tree and out-tree task groups exhibit more stable value loss. That is, their performance has non-increasing value loss ratio over all laxities.

This difference indicates that the arbitrary task groups are more difficult to schedule under overloads. Their random structure does not allow easy isolation of less valuable parts of task groups, the parts that can be severed if required by arrivals of more valuable task groups.

The difference in performance of systems with I-heuristic and systems with G-heuristic is presented in Table 5.20. This table shows that systems with I-heuristic have overall better performance than systems with G-heuristic. The largest difference in performance is around load $\rho = 1$. For example, for $\rho = 0.9$, the difference is up to 10%, and for $\rho = 1.2$, the difference is up to 15% at large laxities.

Table 5.20 Groups with Random Values: I-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	I-heuristic is Better				
Difference [%]	2 to 3	3 to 10	2 to 15	2 to 7	2 to 4
Laxity Range	Low-Hi	Low-Hi	Low-Hi	Low-Hi	Low-Hi

Almost identical difference in performance is observed in Table 5.21. This table compares systems with R-heuristic and systems with G-heuristic; and it shows that systems with R-heuristic perform better than systems with G-heuristic, over the entire range of loads and laxities. The largest difference is, again, around load

Table 5.21 Groups with Random Values: R-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	R-heuristic is Better				
Difference [%]	1 to 2	2 to 9	2 to 11	1 to 5	2 to 4
Laxity Range	Low-Hi	Low-Hi	Low-Hi	Low-Hi	Low-Hi

$\rho = 1$. For load $\rho = 0.9$, the difference is up to 9%, and for load $\rho = 1.2$, the difference is up to 11%.

At the same time, both tables indicate that systems with I-heuristic and systems with R-heuristic perform almost the same. The difference between these systems is within the confidence intervals, that is, there is no significant difference. However, both systems perform significantly better than systems with G-heuristic.

Even though systems with I-heuristic and R-heuristic perform almost identical, the systems with I-heuristic are recommended for use. They have better performance, they are simpler and they have wider applicability range.

5.4.3.2 Large Groups with Random Values

Somewhat different value loss ratio is observed for large task groups with arbitrary precedence constraints. In this case, the value loss ratios resembled the value loss ratios of in-tree and out-tree task groups. Even under overloads, the value loss ratios are pretty stable, i.e., they do not increase with increase in laxities, as in the case for smaller task groups. Figure 5.16, Figure 5.17, and Figure 5.18 present value loss ratios for systems with I-heuristic, G-heuristic, and R-heuristic, respectively.

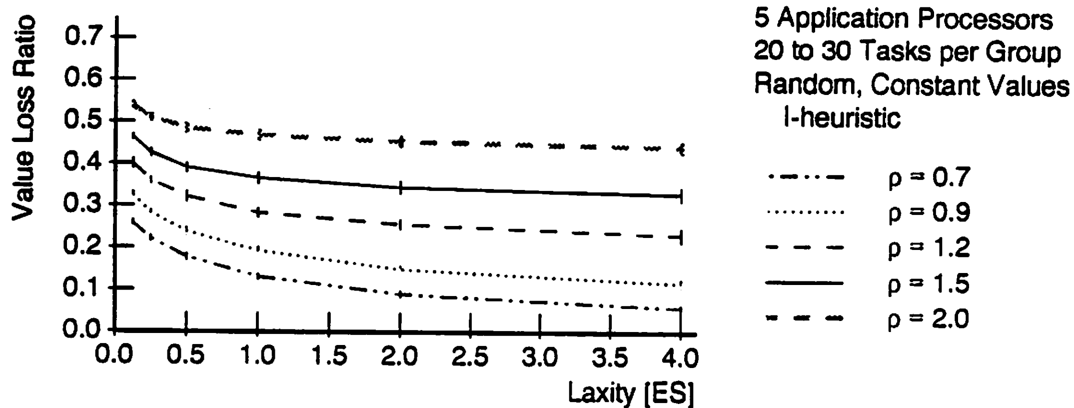

Figure 5.16 Value Loss Ratios for Large Groups with Random Values, I-heuristic.

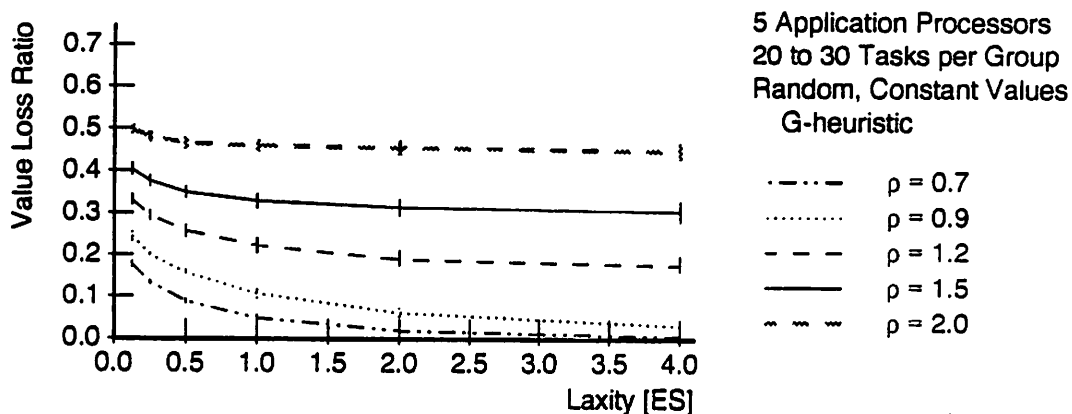

Figure 5.17 Value Loss Ratios for Large Groups with Random Values, G-heuristic.

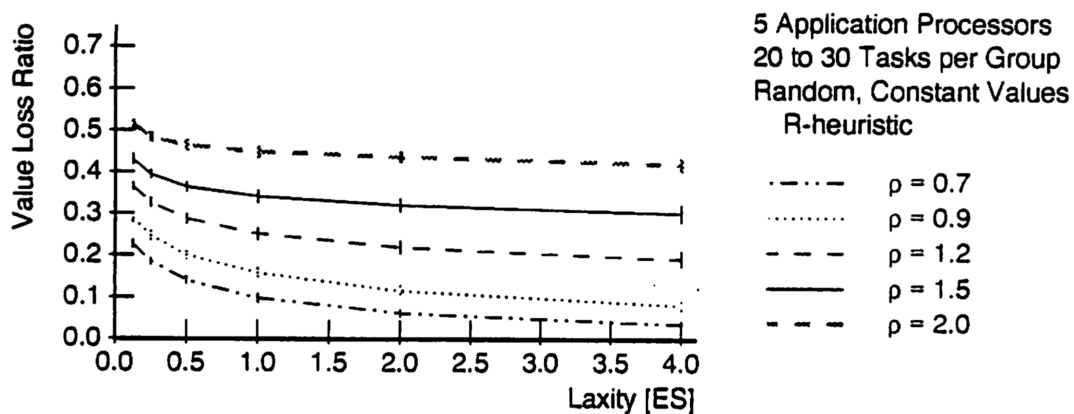

Figure 5.18 Value Loss Ratios for Large Groups with Random Values, R-heuristic.

The observed change in a performance trend can be attributed to the larger number of branches that can be isolated as low value branches. By isolating low value branches with the larger number of tasks, the more efficient task group severing procedure at the rejection time is achieved. This, in turn, produces better overall performance.

The difference in performance between systems with I-heuristic and systems with G-heuristic is presented in Table 5.22. From this table, it is clear that in this case the systems with G-heuristic outperform earlier recommended systems with I-heuristic. The largest difference in performance is noticed for moderate loads. For example, for load $\rho = 0.7$, systems with G-heuristic outperform systems with I-heuristic by up to 6% to 9% over all loads. The difference in performance is smaller under high overload. For small laxities and for load $\rho = 2.0$, the difference is 4%, whereas for large laxities and for the same load, both systems perform the same.

Table 5.22 Large Groups with Random Values: I-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	G-heuristic is Better				
Difference [%]	-9 to -6	-8	-6	-6 to -2	-4 to 0
Laxity Range	Low-Hi	All	All	Low-Hi	Low-Hi

The difference in performance between systems with R-heuristic and systems with G-heuristic, presented in Table 5.23, indicates that systems with G-heuristic perform better than systems with R-heuristic. In this case, the difference is much smaller. For example, for load $\rho = 0.7$ and $\rho = 0.9$, the difference is only about 5% in favor of systems with I-heuristic, over all laxities. This difference continuously

Table 5.23 Large Groups with Random Values: R-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	G-heuristic is Better				Both are Comparable
Difference [%]	-5	-5	-3	-3 to 0	-2 to 2
Laxity Range	All	All	All	Low-Hi	Low-Hi

decreases with the increase in load. Under overloads, it reduces to insignificantly small difference, that is, the difference is within the confidence intervals.

The above observations require reconsideration of the previously made recommendation. If the task groups are small, I-heuristic gives the best performance, but if the task groups are large G-heuristic is better. However, for small to very large task groups, the most consistent performance is obtained if R-heuristic is used. Thus, for more general systems with arbitrary precedence constraints R-heuristic is recommended.

5.4.3.3 Groups with Diminishing, Random Values

Let's now observe the difference in performance for systems that have diminishing contributing values.

Table 5.24 indicates the difference between systems with I-heuristic and systems with G-heuristic. The difference between systems with R–heuristic and systems with G-heuristic is illustrated in Table 5.25.

Comparing the differences from these tables with the differences for systems with random contributing values, Table 5.20 and Table 5.21, it can be concluded that both systems exhibit almost the same performance trend. Therefore, observations and derived conclusions are the same in both cases.

Table 5.24 Groups with Diminishing Values: I-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	I-heuristic is Better				
Difference [%]	2 to 3	3 to 10	2 to 15	2 to 7	2 to 4
Laxity Range	Low-Hi	Low-Hi	Low-Hi	Low-Hi	Low-Hi

Table 5.25 Groups with Diminishing Values: R-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	R-heuristic is Better				
Difference [%]	1 to 2	2 to 9	2 to 13	1 to 5	2 to 4
Laxity Range	Low-Hi	Low-Hi	Low-Hi	Low-Hi	Low-Hi

5.4.3.4 Large Groups with Diminishing, Random Values

The difference in performance between systems with I-heuristic and systems with G-heuristic, for large task groups with diminishing contributing values, is illustrated in Table 5.26, whereas, the difference between systems with R-heuristic and systems with G-heuristic is illustrated in Table 5.27.

The first table indicates that I-heuristic is better under overloads, and that G-heuristic is better under loads less than $\rho = 1$. The difference between the two systems under loads less than $\rho = 1$ is insignificantly small, while the difference between two systems is significantly big under overloads and large laxities. Specifically, the difference under overloads is up to 12% in favor of systems with I-heuristic.

The second table shows that systems with R-heuristic perform better than systems with G-heuristic under all loads. For loads less than $\rho = 1$, the difference

is insignificant; while for overloads, the significant difference is observed. The difference for overloads is up to 14% in favor of systems with R-heuristic.

Table 5.26 Large Groups with Diminishing Values: I-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	G-heuristic is Better		I-heuristic is Better		
Difference [%]	-3	-2	-1 to 10	0 to 12	1 to 9
Laxity Range	All	All	Low-Hi	Low-Hi	Low-Hi

Table 5.27 Large Groups with Diminishing Values: R-heuristic vs. G-heuristic.

System Load	0.7	0.9	1.2	1.5	2.0
Performance Trend	Both are Comparable	R-heuristic is Better			
Difference [%]	-1	2	1 to 13	3 to 14	3 to 13
Laxity Range	All	All	Low-Hi	Low-Hi	Low-Hi

These observations lead to the conclusion that R-heuristic performs consistently well. Their performance is either the best, or it is within the confidence intervals of the best performing heuristic. Therefore, for consistently high performance, low sensitivity to various system parameters, and for arbitrary precedence constraints, the use of R-heuristic is recommended. That is, the least value loss is obtained when the task groups are off-line preprocessed, and when the reflective value densities are used to assist on-line schedulers.

5.5 Summary

To determine the applicability range, the quality of overall system performance, as well as deeper understanding of the issues involved with dynamic scheduling of non-atomic task groups with contributing values assigned to individual tasks, this chapter presented a comparative analysis of three heuristics used for on-line scheduling of complex real-time systems with arbitrary task groups.

Specifically, this chapter presents the analysis of three heuristic functions that integrate deadline, earliest possible start time, and either:

- initial value densities (I-heuristic),
- group value densities (G-heuristic), or
- reflective value densities (R-heuristic).

The performance comparison of these heuristics is conducted for the large number of different systems. For example, all heuristic functions are applied to scheduling of three different task groups:

- out-tree task groups,
- in-tree task groups, and
- arbitrary task groups.

Besides this, the experimental test are performed for a large range of group laxities, system loads, task group sizes, number of application processors, assignment of contributing values for tasks within a group, and for two time value functions, the step function and the linear decreasing function.

The difference in performance between heuristics for out-tree task groups is are summarized in Table 5.28. Similarly, the results for in-tree task groups and arbitrary task groups are summarized in Table 5.29 and Table 5.30, respectively. In

Table 5.28 Summary of Performance Trends for Out-Tree Task Groups.

Random Values:	Overloads:	I outperforms G by up to 9% R outperforms G by up to 16%
	$\rho < 1$:	I performs the same as G R outperforms G by up to 7%
Top-Heavy:	Overloads:	I outperforms G by up to 4% R outperforms G by up to 3%
	$\rho < 1$:	I, G, and R perform the same
Bottom-Heavy:	All loads:	G outperforms I by up to 9% R performs the same as G
Large Groups:	All loads:	G outperforms I by up to 7%
	Overloads:	R outperforms G by up to 11%
	$\rho < 1$:	R performs the same as G
Diminishing Values:	All loads:	I outperforms G by up to 9% R outperforms G by up to 13%
2 Processor System:	Overloads:	I outperforms G by up to 3%
	$\rho < 1$:	G outperforms I by up to 2%
	All loads:	R outperforms G by up to 10%

these tables, the first field gives the most distinguishing characteristic of the tested system. For example:

Random Values refers to systems with uniform distribution of contributing values assigned to individual tasks within a given group.

Top-Heavy refers to systems with task groups where contributing values are assigned in decreasing order with source tasks having the highest values.

Bottom-Heavy refers to systems with task groups where contributing values are assigned in decreasing order with terminal tasks having the highest values.

Large Groups refers to systems with task groups made up of 20 to 30 tasks, as opposed to the default group size of 5 to 15 tasks.

Diminishing Values refers to systems where contributing value are linearly decreases from 100% to 50% of the initial value.

2 Processor System refers to a system with two application processors, as opposed to 5 processor system used in all other simulations.

Large Random refers to systems with uniform distribution of contributing values within task groups made up of 20 to 30 tasks.

Large Diminishing refers to systems with task groups made up of 20 to 30 tasks, where contributing value linearly decreases with time.

The second field specifies the load range for which the differences in performance between I-, G-, and R-heuristic, given in the third field, apply to. In these tables, I-heuristic, G-heuristic, and R-heuristic are abbreviated to I, G, and R, respectively.

The table that presents performance trends for systems with out-tree task groups, Table 5.28, indicates following. R-heuristic provides the best overall performance. In the case with bottom-heavy task groups, R-heuristic performs the same as G-heuristic, under all loads; while in the case of top-heavy task groups, bottom-heavy task groups, and large task groups, for loads $\rho < 1$, R-heuristic and G-heuristic perform approximately the same.

When R-heuristic is compared to I-heuristic, there is only one case where both heuristics perform approximately the same, the case with top-heavy task groups. In all other cases R-heuristic performs better than I-heuristic.

The results for systems with in-tree task groups, summarized in Table 5.29, show that under overloads R-heuristic and I-heuristic perform approximately the same, while both heuristics outperform G-heuristic. On the other hand, for loads $\rho = 1$, all three heuristics perform the same.

Table 5.29 Summary of Performance Trends for In-Tree Task Groups.

Random Values:	Overloads:	I outperforms G by up to 10%
		R outperforms G by up to 12%
	$\rho < 1$:	I, G, and R perform the same
Diminishing Values:	Overloads:	I outperforms G by up to 11%
		R outperforms G by up to 13%
	$\rho < 1$:	I, G, and R perform the same

Table 5.30 Summary of Performance Trends for Arbitrary Task Groups.

Random Values:	All loads:	I outperforms G by up to 15%
		R outperforms G by up to 11%
Large Random:	All loads:	G outperforms G by up to 9%
		G outperforms R by up to 5%
Diminishing Values:	All loads:	I outperforms G by up to 15%
		R outperforms G by up to 13%
Large Diminishing:	Overloads:	I outperforms G by up to 10%
		R outperforms G by up to 14%
	$\rho < 1$:	G outperforms I by up to 14%
		I, G, and R perform the same

Finally, the simulations of systems with arbitrary task groups, summarized in Table 5.30, show the following trends. R-heuristic is better than G-heuristic in all cases but in: (1) systems with large task groups, where G-heuristic performs better under all loads, and (2) systems with large task groups with diminishing values, where all three heuristics perform the same for $\rho < 1$.

I-heuristic, on the other hand, outperforms other two in systems with small task groups with and without diminishing values. Even though R-heuristic is

outperformed in most systems with arbitrary task groups, its performance is always very close to the performance of the best heuristic for a particular system.

From the performed analysis, it can be concluded that R-heuristic has the overall best performance. It performs the best or very close to the best of the three studied heuristics. Thus, R-heuristic is recommended for general use.

In systems where types of task groups are limited to cases where either I-heuristic of G-heuristic performs the same or better than R-heuristic, we recommend the use of the simpler heuristics. They do not require off-line preprocessing, and thus, they have wider applicability range.

CHAPTER 6

OBSERVATIONS AND CONCLUSIONS

With the advances in technology, the range of real-time applications is continuously expanding. New systems are more complex, more independent, and more demanding. To cope with the dynamic, non-deterministic environments, these systems require more comprehensive scheduling approaches that offer higher flexibility, predictability, and adaptability.

In this dissertation, we presented and analyzed novel approaches to scheduling problems involved with multiprocessor scheduling of very complex and demanding real-time environments. Specifically, in the first part of this dissertation, we developed and analyzed the so-called Well-Timed Scheduling approach. This approach is designed for dynamic, non-deterministic real-time systems that require on-line scheduling of complex independent tasks, where tasks are described by their timing constraints, resource constraints, and contributing values. The main component of Well-Timed Scheduling is the analytically derived control parameter, called the punctual point. The punctual point is designed to prevent too early scheduling of "irrelevant" tasks, the tasks that are ready-to-execute but that do not have any impact on the tasks at the top of a schedule.

Well-Timed scheduling is not yet another scheduling algorithm, it is rather a framework for real-time schedulers. In other words, it lends itself to real-time schedulers specifically designed and tuned for the best performance of a given system.

In the second half of this dissertation, we presented and discussed two scheduling approaches to the problem of scheduling tasks within the same dynamic environment

and the same requirements as above, but with the addition of precedence constraints. The developed and analyzed scheduling algorithms assume that individual tasks in a group are assigned different contributing values; the value is contributed to the system upon the completion of individual tasks rather than upon the completion of the entire group; and, the task groups are non-atomic (severable) task groups.

The overall goal of this dissertation is to provide effective and efficient on-line scheduling of complex real-time tasks in a very demanding non-deterministic environment, where the best-effort algorithms are used to maximize the accrued system value.

6.1 Contributions

The summary of the contributions of this dissertation are presented in the following two sections.

6.1.1 Well-Timed Scheduling

The benefits of our Well-Timed Scheduling approach are:

- Well-Timed Scheduling provides a methodical approach to schedulability analysis through the use of the analytically derived punctual points—the points at which a task becomes schedulable.
- Well-Timed Scheduling is applicable to a wide range of $M/G/c$ real-time systems.
- For ideal cost and FCFS scheduling policy, the probabilistic guarantee that the tasks will meet given timing constraints can be obtained.
- For all but ideal systems, where guarantee ratio cannot be analytically derived, the punctual points can be used to prevent scheduling/rescheduling of "irrelevant" tasks—resulting in a much smaller number of schedulable tasks and

lower scheduling overhead than in traditional systems that always schedule all ready-to-run tasks.

- Well-Timed Scheduling outperforms the traditional approach by a wide margin, especially in overloads, for all but zero cost systems.
- In Well-Timed Scheduling *all* arrived tasks are considered for scheduling—something that approaches with fixed size buffer cannot guarantee. For example, while waiting for an empty slot in a fixed size buffer, a task might miss its deadline, and thus, it might never get a chance to get scheduled.
- Being a framework for real-time schedulers, Well-Timed Scheduling supports the use of different scheduling algorithms; that is, it lends itself to different real- time schedulers.
- In contrast to scheduling-at-dispatch time, Well-Timed Scheduling preserves the notion of early warning, thus, providing the lead time necessary to schedule contingency tasks, if needed.
- Well-Timed Scheduling is powerful to be used at run-time and as a design tool.

In summary, Well-Timed Scheduling provides highly efficient, low overhead scheduling with excellent performance in overloads. The tremendous savings in scheduling cost and the high performance capabilities—especially for high overloads, large laxities, and with a large number of processors—is experimentally demonstrated in this dissertation.

As a final remark, the centralized scheduling approach, with and without the punctual points, faces the obvious limitations when used in multiprocessor systems with large number of processors. As the number of processors increases, the number of pending ready-to-execute tasks increases (assuming the same system load), and consequently, the scheduling overhead increases—leading to the total loss of control.

In spite of the fact that all schedulers have a point of total loss of control, Well-Timed Scheduling is very robust and the breakpoint comes significantly later than in a traditional approach that always schedules all ready-to-execute tasks.

To move the breakpoint even further, we suggest a multilevel scheduling approach, where clusters of smaller number of processors are scheduled separately.

6.1.2 *Scheduling with Precedence Constraints*

The benefits of off-line value density preprocessing algorithms, developed and analyzed in the second half of this dissertation are:

- The value density preprocessing is applicable to task groups with arbitrary precedence constraints.
- By off-line assigning reflective parameters to individual tasks—that is, the parameters that indicate how valuable individual tasks and their successors are— the on-line scheduler has to consider only eligible tasks when making a scheduling decision, thus resulting in a low scheduling overhead.
- Through the use of reflective parameters, the domain of individual task schedulers is extended to schedule tasks with precedence constraints, thus, creating a possibility to use existing on-line schedulers for independent tasks that have been demonstrated as very efficient and effective.
- Due to separation of value-density preprocessing from deadline preprocessing, our method is applicable to both real-time and non real-time systems.

6.2 Extensions

The design of more complex real-time systems applicable in dynamic non-deterministic environments is a very difficult, broad, and still open problem. This dissertation

scratches just the surface of this large problem by introducing a number of approaches for multiprocessor scheduling of complex real-time tasks.

Some of the possible extensions of our research are:

- The focus on centralized multiprocessor scheduling presets just one dimension along which the idea of Well-Timed Scheduling can be applied. Furthermore, our research indicates that the centralized approach has serious limitation with respect to the number of processors that can be handled in the presence of realistic scheduling costs. For these reasons, it would be very interesting to explore the possibilities of applying the concept of the punctual point and Well-Timed Scheduling to systems with multiple levels of scheduling. For example, cooperating clusters are one of the interesting architectures that would benefit from this research.

- The analytically derived punctual points are used as excellent approximations in systems with a large number of imposed constraints. For systems with resource constraints, for example, it is not possible to analytically derive the punctual points. The only solution is to experimentally determine the pattern of fine tuning of the analytically derived punctual points. Another interesting and very useful fine tuning method would be: the tuning of the punctual points for different scheduling costs. This method will be of great help during the design phase.

- This dissertation focuses mainly on the impact of Well-Timed Scheduling on the run-time environment, while the analysis of its applicability to a design stage got much less attention. Therefore, due to the high potential and its practical value, it would be highly recommended to continue the research in this direction, and to recognize and specify the full capabilities of Well-Timed Scheduling as a design tool.

- Finally, the complexity of real systems require that our analysis of Well-Timed Scheduling should be extended by addition of precedence constraints. The use of the punctual points in presence of task groups would be a significant improvement to our work.

To recognize some other future directions that this research can take, it is important to understand that real-time systems are not simply systems that perform fast, or use one of the real-time scheduling algorithms. More importantly, real-time systems are systems that are aware of time and the timing requirements of the application and the external environment.

APPENDIX A

NOTATION

λ – Arrival rate

ES – Mean service time

$\rho = \lambda ES/c$ – Traffic intensity

$F(t)$ – General probability distribution function

$F_e(t)$ – Equilibrium distribution function given as $F_e(t) = \frac{1}{ES}\int_0^t (1 - F(x))\,dx$

W_q – Steady–state expected waiting time of a customer in the queue (excluding service time)

$V(t)$ – the waiting time distribution for the delayed customer where

$$V(t) = 1 - P\{W_q > t | W_q > 0\}, \qquad t \geq 0.$$

APPENDIX B

Waiting Time Distribution for an M/G/c System

Originally, the approximation formula for the waiting time distribution of an $M/G/c$ system, with its numerical solution, was proposed in [60], while a more complete description was given in [59]. The approximation is shown to be accurate, easy to implement, and very practical.

In the general form, the approximation for the waiting time distribution is given in a form of the defective renewal equation, i.e., it is given as:

$$\overline{V}(t) = (1-\rho)\{1-(1-F_e(t))^c\} + \lambda \int_0^t \overline{V}(x)\{1-F(c(t-x))\}\,dx, \qquad t \geq 0. \quad \text{(B.1)}$$

Equation B.1 has the form of Volterra equation of the second kind for which several numerical solutions are readily available. Details on how to solve Volterra equations can be found in classical textbooks, such as, chapter 11 of [15] and chapter 5 of [14]. However, by exploiting the specific form of equation B.1, more condensed and abbreviated numerical solution is given in [60]. This solution is based on repeated Simpson's rule and Day's starting procedure.

APPENDIX C

Rejection Probabilities for an $M/G/c/K$ System

The $M/G/c/K$ system is a system with finite capacity queue where only K ($\geq c$) customers are allowed in the system at any time—the ones in the waiting area plus the ones in service. The distinct characteristic of this system is its state dependent arrival rate, that is, if an arriving customer finds no space in the waiting area it gets immediately rejected.

The tractable approximation algorithm, proposed in [57], is an excellent and very useful approximation. It uses the following stable recursive algorithm to compute approximations $\overline{p}_j$, $j = 0, 1, ..., K$, for the state probabilities p_j.

$$\overline{p}_n = \begin{cases} \frac{(c\rho)^n}{n!}\overline{p}_0, & 0 \leq n \leq c-1, \\ \lambda \overline{p}_{c-1}\alpha_{n-c} + \lambda \sum_{j=c}^{n} \overline{p}_j \beta_{n-j}, & c \leq n \leq K-1, \end{cases} \tag{C.1}$$

$$\overline{p}_K = \rho \overline{p}_{c-1} - (1-\rho) \sum_{j=c}^{K-1} \overline{p}_j. \tag{C.2}$$

α_r and β_r are defined as follows:

$$\alpha_r = \int_0^\infty (1 - F_e(t))^{c-1}(1 - F(t))e^{-\lambda t}\frac{(\lambda t)^r}{r!}\, dt, \qquad r = 0, 1, \ldots \tag{C.3}$$

$$\beta_r = \int_0^\infty (1 - F(ct))e^{-\lambda t}\frac{(\lambda t)^r}{r!}\, dt, \qquad r = 0, 1, \ldots \tag{C.4}$$

To find the rejection probability, equations C.1 and C.2 should be solved in terms of $\overline{p}_n/\overline{p}_0$ ratios, for $n = 1, ..., N$; then, the normalization equation $\sum_{j=0}^{K} \overline{p}_j = 1$ should be used to obtain the value for $\overline{p}_0$.

This simple algorithm can be implemented very efficiently. However, in its general form the algorithm does not address some specific, but very important, issues that one might come across when computing a rejection probability for some given service distribution time. For that reason, we present a case study when the service time is given as an Erlang-k distribution.

APPENDIX D

$M/E_k/c/K$: A CASE STUDY

The expected service time for general Erlang-k distribution is defined as $ES = k/\mu$, and the traffic intensity is given as $\rho = k\lambda/c\mu$. Probability distribution is given as:

$$F(t) = 1 - \sum_{j=0}^{k-1} e^{-\mu t} \frac{(\mu t)^j}{j!}, \tag{D.1}$$

while the equilibrium distribution is given as:

$$F_e(t) = \frac{1}{ES} \int_0^t \sum_{j=0}^{k-1} e^{-\mu x} \frac{(\mu x)^j}{j!}\, dx. \tag{D.2}$$

In order to complete the computation of the rejection probability for the general Erlang-k distribution of service time the equations C.3 and C.4 for α_r and β_r coefficients must be solved first.

Using the integral tables, the solution for the β_r integral can be obtained as:

$$\beta_r = \sum_{j=0}^{k-1} \binom{r+j}{j} \frac{(c\mu)^j \lambda^r}{(\lambda + c\mu)^{r+j+1}}. \tag{D.3}$$

On the other hand, by a straightforward substitution of $F_e(t)$ and $F(t)$, the α_r integral can be reduced to the following form:

$$\alpha_r = \int_0^\infty \left(\sum_{j=0}^{k-1} \frac{k-j}{k} \frac{(\mu t)^j}{j!} \right)^{c-1} \sum_{j=0}^{k-1} \frac{(\mu t)^j}{j!} \frac{(\lambda t)^r}{r!} e^{-(\lambda + c\mu)t}\, dt \tag{D.4}$$

This form is particularly suitable for direct application of Gauss–Laguerre quadrature. Generally, Gauss–Laguerre quadrature is used when the numerical integration has the following general form:

$$\int_0^\infty F(z)\, e^{-\theta z}\, dz \doteq \sum_{l=0}^{a} \frac{w_l}{\theta} F\left(\frac{z_l}{\theta}\right) = \sum_{l=0}^{a} e^{\ln(w_l) - ln(\theta) + \ln(F(\frac{z_l}{\theta}))}, \tag{D.5}$$

where w_l denotes the weight factors, or Cristoffel numbers, corresponding to the l-th Laguerre polynomial $L_l(z)$, and z_l denotes the zeroes of $L_l(z)$. For the theory

behind this method refer to [21]; for the description of the method with examples and tables for smaller a ($a = 1, 2, 3, 4, 5, 9, 14$) see [10].

The logarithmic form of numerical integration is more suitable for large a—where w_l tends to get very small while z_l gets very large. The table for Gauss–Laguerre quadrature of order $a = 64$ can be found in [59]. These tabulated values for $\ln(w_l)$ and z_l are obtained by using the numerical procedures from Numal library [20]. Due to large a, this table is suitable for the wide range of problems. For example, when solving for α_r integral, the restriction on table's applicability is expressed by the following inequality:

$$(k-1)c + r < 2a - 1 = 127, \tag{D.6}$$

where k is a shape parameter for Erlang-k distribution, c is the number of processors, r is equal to $K - c - 1$ (for a given buffer size K), and a is the number of zeroes and weights for the 64 point formula.

BIBLIOGRAPHY

[1] Adolphson, D. and Hu, T. C. Optimal linear ordering. *SIAM J. of Appl. Math.*, 25(3), November 1973.

[2] Adolphson, D. L. Single machine job sequencing with precedence constraints. *SIAM J. of Comput.*, 6(1), March 1977.

[3] Alighieri, D. *The Divine Comedy.* Translated by L.G. White. Pantheon Books Inc., New York, 1948.

[4] Associated Press. *Dante the Robot Inches into Volcano.* The Boston Globe., January 2 1993.

[5] Baker, K. R. and Merten, A. G. Scheduling with parallel processors and linear dealy costs. *Naval Res. Logist. Quart.*, 20, 1973.

[6] Bansal, S. P. Single machine scheduling to minimize weighted sum of complition times with secondary criterion—a branch and bound approach. *European Journal of Operational Research*, 5, 1980.

[7] Biyabani, S. R., Stankovic, J. A., and Ramamrithm, K. The integration of deadline and criticalness in hard real-time scheduling. *Real-Time Systems Workshop*, May 1988.

[8] Bruno, J., Coffman Jr., E. G., and Sethi, R. Scheduling independent tasks to reduce mean finishing time. *Communications of the Association for Computing Machinery*, 17, 1974.

[9] Burns, R. N. Scheduling to minimize the weighted sum of completion times with secondry criteria. *Naval Res. Logist. Quart.*, 23(1), 1976.

[10] Carnahan, B., Luther, H. A., and Wilkes, J. O. *Applied Numerical Methods.* John Wiley & Sons, Inc., 1969.

[11] Cheng, S. C. *Dynamic Scheduling Algorithms for Distributed Hard Real-Time Systems.* PhD thesis, University of Massachusetts, Amherst, Department of Electrical and Computer Engineering, May 1987.

[12] Chetto, H., Silly, M., and Bouchentouf, T. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems Journal*, 2(3), September 1990.

[13] Conway, R. W., Maxwell, W. L., and Miller, L. W. *Theory of Scheduling.* Addison-Wesley, 1967.

[14] Delves, L. M. and Mohamed, J. L. *Computatonal methods for integral equations.* Cambridge Univ. Press, Cambridge, 1985.

[15] Delves, L. M. and Walsh, J., editors. *Numerical Solution of Integral Equations.* Clarendon Press, Oxford, 1974.

[16] Emmons, H. A note on a scheduling problem with dual criteria. *Naval Res. Logist. Quart.*, 22, 1975.

[17] Garey, M. R. Optimal task sequencing with precedence constraints. *Discrete Mathematics*, 4, 1973.

[18] Goli, P., Kurose, J., and Towsley, D. Approximate minimum laxity scheduling algorithms for real-time systems. Technical Report COINS 90-88, University of Massachusetts, Amherst, Department of Computer and Information Science, 1990.

[19] Haritsa, J. R., Livny, M., and Carey, M. J. Earliest deadline scheduling for real-time database systmes. In *Proceedings of Real-Time Systems Symposium*, December 1991.

[20] Hemker., P. W. *Numal, Numerical Procedures in Algol*, volume 47. MC Syllabus, 1981.

[21] Hildebrand, F. B. *Introduction to Numerical Analysis.* McGraw-Hill Book Company, Inc., 1956.

[22] Hong, J., Tan, X., and Towsley, D. A performance analysis of minimum laxity and earliest deadline scheduling in a real-time systems. *IEEE Transactions on Computers*, C-38(12), December 1989.

[23] Horn, W. A. Single-machine job sequencing with treelike precedence ordering and linear delay penalties. *SIAM J. of Appl. Math.*, 23(2), September 1972.

[24] Horowitz, E. and Sahni, S. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the Association for Computing Machinery*, 23, 1976.

[25] Huang, J., Stankovic, J. A., Towsley, D., and Ramamritham, K. Real-time processing: Design, implementation and performance evaluation. Technical Report COINS 90-43, University of Massachusetts, Amherst, Department of Computer and Information Science, May 1990.

[26] Lann, G. L. Designing real-time dependable distributed systems. Technical Report 1425, INRIA-Rocquencourt, April 1991.

[27] Lawler, E. L. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. *Annals of Discrete Mathematics*, 2, 1978.

[28] Lawler, E. L. Preemptive scheduling of precedence-constrained jobs on parallel machines. In et al., M. A. H. D., editor, *Deterministic and Stochastic Scheduling*. D. Reidel Publishing Company, 1982.

[29] Lenstra, J. K. and Rinnooy Kan, A. H. G. Complexity of scheduling under precidence constraints. *Operations Research*, 26(1), January–February 1978.

[30] Lenstra, J. K., Rinnooy Kan, A. H. G., and Brucker, P. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1, 1977.

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

[32] Locke, C. D. *Best-Effort Decision Making for Real-Time Scheduling.* PhD thesis, Carnegie-Mellon University, Computer Science Dept., 1986.

[33] Mcnaughton, R. Scheduling with deadlines and loss functions. *Management Science*, 6(1), October 1959.

[34] Möhring, R. H. and Radermacher, F. J. Generalized results on the polinomiality of certain weighted sum scheduling problems. *Mathematics of Operations Research*, 1984.

[35] Mok, A. K. and Sutanthavibul, S. Modeling and scheduling of dataflow real-time systems. *Real-Time Systems Symposium*, December 1985.

[36] Panwar, S. S. and Towsley, D. On the optimality of the ste rue for multiple server queues that serve customers with deadlines. Technical Report COINS 88-81, University of Massachusetts, Amherst, Department of Computer and Information Science, July 1988.

[37] Panwar, S. S., Towsley, D., and Wolf, J. K. Optimal scheduling policies for a class of queues with customer deadlines until the beginning of service. *Journal of the Association for Computing Machinery*, 35(4), October 1988.

[38] Paulin, P. G. and Knight, J. P. Force-directed scheduling for the behavioral synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design*, 8(6), June 1989.

[39] Peng, D. T. and Shin, K. G. Static allocation of periodic tasks with precedence constraints in distributed real-time systems. In *Proc. of the International Conference on Distributed Computing*, June 1989.

[40] Posner, M. E. The deadline constrained weighted completion time problem: Analysis of a heuristic. *Operations Research*, 36(5), September-October 1988.

[41] Ramamritham, K. Allocation and scheduling complex periodic tasks. *submitted to 10th International Conference on Distributed Computer Systems*, 1989.

[42] Ramamritham, K. and Stankovic, J. A. Dynamic task scheduling in distributed hard real-time systems. *IEEE Software*, 1(3), July 1984.

[43] Ramamritham, K. and Stankovic, J. A. Scheduling strategies adopted in spring: An overview. In van Tilborg, A. and Koob, G. M., editors, *Foundations of Real-Time computing: Scheduling and Resource Management*. Kluwer Academic Publishers, Boston, 1991.

[44] Ramamritham, K., Stankovic, J. A., and Shiah, P. F. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2), April 1990.

[45] Rinnooy Kan, A. H. G. *Machine Scheduling Problems: Classification, Complexity and Computations*. Martinus Nijhoff, The Hague, Holand, 1976.

[46] Sethi, R. On the complexity of mean flow time scheduling. *Mathematics of Operations Research*, 2, 1977.

[47] Sha, L., Rajkumar, R., Lehoczky, J., and Ramamritham, K. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems Journal*, 1(3), December 1989.

[48] Shen, C. *An Integrated Approach to Real-Time Task and Resource Management in Multiprocessor Systems*. PhD thesis, University of Massachusetts, Amherst, Department of Computer and Information Science, September 1992.

[49] Shen, C., Ramamritham, K., and Stankovic, J. A. Resource reclaiming in multiprocessor real-time systems. *IEEE Transactions on Parallel and Distributed Systems*, 1992. To Appear.

[50] Sidney, J. B. Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs. *Operations Research*, 22, 1975.

[51] Sidney, J. B. A decomposition algorithm for sequencing with general precedence constraints. *Mathematics of Operations Research*, 6(2), May 1981.

[52] Sidney, J. B. and Steiner, G. Optimal sequencing by modular decomposition: Polynomial algorithms. *Operations Research*, 34(4), July–August 1986.

[53] Smith, W. E. Various optimizers for single-stage production. *Naval Res. Logist. Quart.*, 3, 1956.

[54] Stankovic, J. A. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer*, October 1988.

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

[56] Tijms, H. C. *Stochastic Modelling and Analysis: A Computational Approach.* John Wiley & Sons, Chichester, 1986.

[57] Tijms, H. C. and van Hoorn, M. H. Computational methods for single server and multi-server queues with markovian input and general service time. In Disney, R. L. and Ott, T. J., editors, *Applied Probability—Computer Science, The Interface*, volume 2. Birkhauser, Boston, 1982.

[58] Valdes, J., Tarjan, R. E., and Lawler, E. L. The recognition of series parallel digraphs. *SIAM J. of Comput.*, 11(2), May 1982.

[59] van Hoorn, M. H. *Algorithms and approximations for queueing systems.* CWI Tract No. 8. CWI, Amsterdam, 1984.

[60] van Hoorn, M. H. and Tijms, H. C. Approximation for the waiting time distribution of the M/G/c queue. *Performance Evaluation*, 2, 1982.

[61] Varghese, G. and Lauck, T. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. *ACM SIGOP Operating Systems Review*, 21(5), 1987. Proceedings of the Eleventh ACM Symp. on Operating System Principles.

[62] Wang, F. *Dynamic Scheduling in Real-Time Systems—Algorithms and Analysis.* PhD thesis, University of Massachusetts, Amherst, Department of Computer and Information Science, 1992. To Appear.

[63] Xu, J. and Parnas, D. L. Scheduling with release times, deadlines, precedence, and exclusion relations. *IEEE Transactions on Software Engineering*, 16(3), March 1990.

[64] Zhao, W. and Ramamritham, K. Simple and integrated heuristic algorithms for scheduling tasks with time and resource constraints. *Journal of Systems and Software*, 1987.

[65] Zhao, W. and Stankovic, J. A. Performance analysis of FCFS and improved FCFS scheduling algorithms for dynamic real-time computer systems. *Real-Time Systems Symposium*, December 1989.